# Using as

The GNU Assembler

(GNU Binutils)

Version 2.41

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of **as** for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

Dean Elsner, Jay Fenlason & friends

Using as Edited by Cygnus Support

Copyright © 1991-2023 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

# Table of Contents

1	Ο	verview1
	1.1	Structure of this Manual
	1.2	The GNU Assembler
	1.3	Object File Formats
	1.4	Command Line
	1.5	Input Files
	1.6	Output (Object) File   23
	1.7	Error and Warning Messages
<b>2</b>	С	ommand-Line Options 25
	2.1	Enable Listings: -a[cdghlns]
	2.1 2.2	alternate
	$\frac{2.2}{2.3}$	-D
	$\frac{2.3}{2.4}$	Work Faster: -f
	$2.4 \\ 2.5$	.include Search Path: -I path
	$\frac{2.5}{2.6}$	Difference Tables: -K
	2.0 2.7	Include Local Symbols: -L
	$\frac{2.7}{2.8}$	Configuring listing output:listing
	$2.0 \\ 2.9$	Assemble in MRI Compatibility Mode: -M
	2.9 2.10	Dependency Tracking:MD
	2.10 2.11	Output Section Padding
	2.11 2.12	Name the Object File: -o
	2.12 2.13	Join Data and Text Sections: -R
	2.13 2.14	
	2.14 2.15	Display Assembly Statistics:statistics
	2.15	Announce Version: -v
	2.10 2.17	Control Warnings: -W,warn,no-warn,fatal-warnings. 29
	2.17	Generate Object File in Spite of Errors: -Z
	2.10	Generate Object File in Spite of Errors2
3	Sy	yntax
	3.1	Preprocessing
	3.2	Whitespace
	3.3	Comments
	3.4	Symbols
	3.5	Statements
	3.6	Constants
	3	.6.1 Character Constants
		3.6.1.1 Strings
		3.6.1.2 Characters
	3	.6.2 Number Constants
		3.6.2.1 Integers
		3.6.2.2 Bignums
		3.6.2.3 Flonums

4 S	ections and Relocation 3	37
4.1	Background	37
4.2	Linker Sections	
4.3	Assembler Internal Sections	
4.4	Sub-Sections	
4.5	bss Section	
1.0		-0
$\mathbf{S}$	ymbols	3
5.1	Labels	43
5.2	Giving Symbols Other Values	
5.3	Symbol Names	
5.4	The Special Dot Symbol	
5.5	Symbol Attributes	
	5.5.1 Value	
	5.5.2 Type	
	5.5.3 Symbol Attributes: a.out	
	5.5.3.1 Descriptor	
	5.5.3.2 Other	
1	5.5.4 Symbol Attributes for COFF	
,	5.5.4.1 Primary Attributes	
	5.5.4.2 Auxiliary Attributes	
	5.5.5 Symbol Attributes for SOM	
	expressions	
6.1	Empty Expressions	
6.2	Integer Expressions	
	6.2.1 Arguments	
	5.2.2 Operators	
	3.2.3 Prefix Operator	
(	5.2.4   Infix Operators	48
Δ	Assembler Directives5	1
7.1	.abort	
7.2	.ABORT (COFF)	
7.3	.align [abs-expr[, abs-expr[, abs-expr]]]	
7.4	.altmacro	
7.5	ascii "string"	
7.6	asciz "string"	
7.0	-	
7.8	.attach_to_group name	
7.8 7.9	<pre>.balign[wl] [abs-expr[, abs-expr[, abs-expr]]]</pre>	
	.bss subsection	
7.10		
	7.10.1 .bundle_align_mode abs-expr	
	7.10.2 .bundle_lock and .bundle_unlock	
7.11	.byte expressions	b4

7.12	CFI directives	54
7.1	2.1 .cfi_sections section_list	54
7.1	2.2 .cfi_startproc [simple]	55
7.1	2.3 .cfi_endproc	55
7.1	2.4 .cfi_personality encoding [, exp]	55
7.1	2.5 .cfi_personality_id <i>id</i>	55
7.1	2.6 .cfi_fde_data [opcode1 [,]]	55
7.1	2.7 .cfi_lsda encoding [, exp]	
7.1	2.8 .cfi_inline_lsda [ <i>align</i> ]	55
7.1	2.9 .cfi_def_cfa register, offset	56
	2.10 .cfi_def_cfa_register register	56
7.1	2.11 .cfi_def_cfa_offset offset	56
7.1	2.12 .cfi_adjust_cfa_offset offset	56
7.1	2.13 .cfi_offset register, offset	56
7.1	2.14 .cfi_val_offset register, offset	56
7.1	2.15 .cfi_rel_offset register, offset	56
7.1	2.16 .cfi_register register1, register2	56
7.1	2.17 .cfi_restore register	56
7.1	2.18 .cfi_undefined register	56
7.1	2.19 .cfi_same_value register	56
7.1	2.20 .cfi_remember_state and .cfi_restore_state	57
7.1	2.21 .cfi_return_column register	57
7.1	2.22 .cfi_signal_frame	57
7.1	2.23 .cfi_window_save	57
7.1	2.24 .cfi_escape <i>expression</i> [,]	57
7.1	$2.25$ .cfi_val_encoded_addr register, encoding, label	58
7.13	.comm symbol , length	58
7.14	.data subsection	58
7.15		
1.15	.dc[size] expressions	58
$7.13 \\ 7.16$	.dcb[size] number [,fill]	59
	.dcb[size] number [,fill]ds[size] number [,fill]	$59\\59$
7.16	.dcb[size] number [,fill]	59 59 60
7.16 7.17 7.18 7.19	.dcb[size] number [,fill]ds[size] number [,fill]	59 59 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20$	.dcb[size] number [,fill]	59 59 60 60
7.16 7.17 7.18 7.19 7.20 7.21	.dcb[size] number [,fill]	59 59 60 60 60 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22$	.dcb[size] number [,fill] .ds[size] number [,fill] .def name .desc symbol, abs-expression .dim .double flonums .eject	59 59 60 60 60 60 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22 \\ 7.23$	.dcb[size] number [,fill] .ds[size] number [,fill] .def name. .desc symbol, abs-expression .dim .double flonums. .eject .else.	59 59 60 60 60 60 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22 \\ 7.23 \\ 7.24$	<pre>.dcb[size] number [,fill] .ds[size] number [,fill] .def name .desc symbol, abs-expression .dim .double flonums .eject .else .elseif.</pre>	59 59 60 60 60 60 60 60 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22 \\ 7.23 \\ 7.24 \\ 7.25$	<pre>.dcb[size] number [,fill] .ds[size] number [,fill] .def name .desc symbol, abs-expression .dim .double flonums .eject .else .elseif .end</pre>	59 59 60 60 60 60 60 60 60
7.167.177.187.197.207.217.227.237.247.257.26	<pre>.dcb[size] number [,fill] .ds[size] number [,fill] .def name. .desc symbol, abs-expression .dim .double flonums. .eject .else .else .elseif .end .endef.</pre>	59 59 60 60 60 60 60 60 60 60
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22 \\ 7.23 \\ 7.24 \\ 7.25 \\ 7.26 \\ 7.27 $	<pre>.dcb[size] number [,fill] .ds[size] number [,fill] .def name .desc symbol, abs-expression .dim .double flonums .eject .else .else .elseif .end .endef .endfunc</pre>	<ol> <li>59</li> <li>59</li> <li>60</li> <li>61</li> </ol>
$7.16 \\ 7.17 \\ 7.18 \\ 7.19 \\ 7.20 \\ 7.21 \\ 7.22 \\ 7.23 \\ 7.24 \\ 7.25 \\ 7.26 \\ 7.27 \\ 7.28 $	<pre>.dcb[size] number [,fill] .ds[size] number [,fill] .def name .desc symbol, abs-expression .dim .double flonums .eject .else .else .elseif .end .endef .endfunc .endif</pre>	$59 \\ 59 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 6$
$\begin{array}{c} 7.16\\ 7.17\\ 7.18\\ 7.19\\ 7.20\\ 7.21\\ 7.22\\ 7.23\\ 7.24\\ 7.25\\ 7.26\\ 7.26\\ 7.27\\ 7.28\\ 7.29\end{array}$	.dcb[size] number [,fill].ds[size] number [,fill].def namedef symbol, abs-expressiondim.double flonumsejectelseelseifendendfendifequ symbol, expression.	$59 \\ 59 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 6$
$\begin{array}{c} 7.16\\ 7.17\\ 7.18\\ 7.19\\ 7.20\\ 7.21\\ 7.22\\ 7.23\\ 7.24\\ 7.25\\ 7.26\\ 7.26\\ 7.27\\ 7.28\\ 7.29\\ 7.30\\ \end{array}$	.dcb[size] number [,fill].ds[size] number [,fill].def name.desc symbol, abs-expression.dim.double flonums.eject.else.elseif.endef.endfunc.endif.equ symbol, expression.equiv symbol, expression	$59 \\ 59 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 60 \\ 6$
$\begin{array}{c} 7.16\\ 7.17\\ 7.18\\ 7.19\\ 7.20\\ 7.21\\ 7.22\\ 7.23\\ 7.24\\ 7.25\\ 7.26\\ 7.27\\ 7.28\\ 7.29\\ 7.30\\ 7.31\\ \end{array}$	.dcb[size] number [,fill].ds[size] number [,fill].def name.desc symbol, abs-expression.dim.double flonums.eject.else.elseif.end.endfinc.endif.equiv symbol, expression.eqv symbol, expression	$\begin{array}{c} 59\\ 59\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 61\\ 61\\ 61\\ 61\\ 61\\ 61\\ 61\\ 61\\ \end{array}$
$\begin{array}{c} 7.16\\ 7.17\\ 7.18\\ 7.19\\ 7.20\\ 7.21\\ 7.22\\ 7.23\\ 7.24\\ 7.25\\ 7.26\\ 7.26\\ 7.27\\ 7.28\\ 7.29\\ 7.30\\ \end{array}$	.dcb[size] number [,fill].ds[size] number [,fill].def name.desc symbol, abs-expression.dim.double flonums.eject.else.elseif.endef.endfunc.endif.equ symbol, expression.equiv symbol, expression	$\begin{array}{c} 59\\ 59\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 60\\ 61\\ 61\\ 61\\ 61\\ 61\\ 61\\ 61\\ \end{array}$

7.34	.exitm	62
7.35	.extern	62
7.36	.fail expression	. 62
7.37	.file	62
7.38	.fill repeat , size , value	. 63
7.39	.float flonums	63
7.40	.func name[,label]	63
7.41	.global symbol, .globl symbol	. 63
7.42	.gnu_attribute tag, value	
7.43	.hidden names	64
7.44	.hword expressions	. 64
7.45	.ident	
7.46	.if absolute expression	64
7.47	.incbin "file"[,skip[,count]]	. 65
7.48	.include "file"	65
7.49	.int expressions	. 66
7.50	.internal names	66
7.51	.irp symbol, values	66
7.52	.irpc symbol, values	
7.53	.lcomm symbol , length	. 67
7.54	.lflags	
7.55	.line line-number	
7.56	.linkonce [type]	
7.57	.list	
7.58	.ln line-number	
7.59	.loc fileno lineno [column] [options]	
7.60	.loc_mark_labels enable	
7.61	.local names	
7.62	.long expressions	
7.63	.macro	
7.64	.mri val	
7.65	.noaltmacro	
7.66	.nolist	
7.67	.nop [size]	
7.68	.nops size[, control]	
7.69	.octa bignums	
7.70	.offset loc	
7.71	.org new-lc , fill	
7.72	.p2align[w1] [abs-expr[, abs-expr[, abs-expr]]]	
7.73	.popsection	
7.74	.previous	
7.75	.print string	
7.76	.protected names	
7.77	.psize lines, columns	
7.78	.purgem name	70
7.79	.pushsection name [, subsection] [,	76
"I1	ags"[, @type[,arguments]]]	70

	7.80	.quad expressions	76
	7.81	.reloc offset, reloc_name[, expression]	76
	7.82	.rept count	77
	7.83	.sbttl "subheading"	. 77
	7.84	.scl class	77
	7.85	.section name	. 77
	7.86	.set symbol, expression	. 81
	7.87	.short expressions	82
	7.88	.single flonums	. 82
	7.89	.size	. 82
	7.90	.skip size [,fill]	82
	7.91	.sleb128 expressions	. 83
	7.92	.space size [,fill]	83
	7.93	.stabd, .stabn, .stabs	. 83
	7.94	.string "str", .string8 "str", .string16	. 84
	7.95	.struct expression	84
	7.96	.subsection name	84
	7.97	.symver	. 84
	7.98	.tag structname	. 85
	7.99	.text subsection	85
	7.100	.title "heading"	86
	7.101	.tls_common symbol, length[, alignment]	. 86
	7.102	.type	86
	7.103	.uleb128 expressions	. 87
	7.104	.val addr	. 87
	7.105	.version "string"	
	7.106	.vtable_entry table, offset	. 87
	7.107	.vtable_inherit child, parent	
	7.108	.warning "string"	87
	7.109	.weak names	
	7.110	.weakref alias, target	. 88
	7.111	.word expressions	88
	7.112	.zero size	
	7.113	.2byte expression [, expression]*	
	7.114	.4byte expression [, expression]*	
	7.115	.8byte expression [, expression]*	
	7.116	Deprecated Directives	89
0		• • • • • • • •	0.1
8		ject Attributes	
		NU Object Attributes	
	8.1		
	8.1		
	8.1		
	8.1		
	8.1		
	8.1	.6 MSP430 Attributes	93

	8.2 Defi	ning New Object Attributes	. 93
9	Mach	nine Dependent Features	95
	9.1 AA1	rch64 Dependent Features	. 96
	9.1.1	Options	
	9.1.2	Architecture Extensions	
	9.1.3	Syntax	
		.3.1 Special Characters	
		.3.2 Register Names	
	9.1	.3.3 Relocations	
	9.1.4	Floating Point	
	9.1.5	AArch64 Machine Directives	
	9.1.6	Opcodes	
	9.1.7	Mapping Symbols	
	9.2 Alp	ha Dependent Features	102
	9.2.1	Notes	102
	9.2.2	Options	102
	9.2.3	Syntax	103
	9.2	3.1 Special Characters	103
	9.2	3.2 Register Names	103
	9.2	.3.3 Relocations	103
	9.2.4	Floating Point	105
	9.2.5	Alpha Assembler Directives	105
	9.2.6	Opcodes	108
	9.3 ARC	C Dependent Features	109
	9.3.1	Options	109
	9.3.2	Syntax	110
	9.3	2.2.1 Special Characters	110
	9.3	2.2.2 Register Names	
	9.3.3	ARC Machine Directives	
	9.3.4	ARC Assembler Modifiers	
	9.3.5	ARC Pre-defined Symbols	
	9.3.6	Opcodes	
		M Dependent Features	
	9.4.1	Options	
	9.4.2	Syntax	
		.2.1 Instruction Set Syntax	
		.2.2 Special Characters	
		.2.3 Register Names	
		.2.4 ARM relocation generation	
		.2.5 NEON Alignment Specifiers	
	9.4.3	Floating Point	
	9.4.4	ARM Machine Directives	
	9.4.5	Opcodes	
	9.4.6	Mapping Symbols	
	9.4.7	Unwinding	131

	•
37	
	1

9.5 AVI	R Dependent Features	
9.5.1	Options	134
9.5.2	Syntax	136
9.5	5.2.1 Special Characters	136
9.5	5.2.2 Register Names	
9.5	5.2.3 Relocatable Expression Modifiers	136
9.5.3	Opcodes	
9.5.4	Pseudo Instructions	
	ckfin Dependent Features	
9.6.1	Options	
9.6.2	Syntax	
9.6.3	Directives	
9.7 BPI	F Dependent Features	
9.7.1	Options	
9.7.2	Syntax	
9.7	7.2.1 Special Characters	
9.7	7.2.2 Register Names	
9.7	7.2.3 Pseudo Maps	
9.7.3	Machine Directives	
9.7.4	Opcodes	
	7.4.1 Arithmetic instructions	
9.7	7.4.2 32-bit arithmetic instructions	
9.7	7.4.3 Endianness conversion instructions	
9.7	7.4.4 64-bit load and pseudo maps	
9.7	7.4.5 Load instructions for socket filters	
9.7	7.4.6 Generic load/store instructions	
	7.4.7 Jump instructions	
9.7	7.4.8 Atomic instructions	151
9.7.5		
9.7	7.5.1 Pseudo-C Register Names	
	7.5.2 Arithmetic instructions	
9.7	7.5.3 Endianness conversion instructions	
9.7	7.5.4 64-bit load and pseudo maps	
	7.5.5 Load instructions for socket filters	
	7.5.6 Generic load/store instructions	
	7.5.7 Jump instructions	
	7.5.8 Atomic instructions	
	16 Dependent Features	
9.8.1	CR16 Operand Qualifiers	
9.8.2	CR16 Syntax	
	3.2.1 Special Characters	
	IS Dependent Features	
9.9.1	Command-line Options	
9.9.2	Instruction expansion	
9.9.3	Symbols	
9.9.4	Syntax	
9.9	0.4.1 Special Characters	159

9.9.4	1.2 Symbols in position-independent code	159
9.9.4	1.3 Register names	160
9.9.4	I.4 Assembler Directives	160
9.10 C-SI	XY Dependent Features	162
9.10.1	Options	162
9.10.2	Syntax	163
9.11 D10	V Dependent Features	
9.11.1	D10V Options	
9.11.2	Syntax	
9.11		
9.11		
9.11	1	
9.11	0	
9.11	0	
9.11		
9.11.3	0	
9.11.4	Opcodes	
	V Dependent Features	
9.12.1	D30V Options	
9.12.2	Syntax	
9.12		
9.12		
9.12	1	
9.12		
9.12	0	
9.12	0	
9.12.3	Floating Point	
9.12.4	Opcodes	
	Ontinue Optimies	
9.13.1	Options	
9.13.2	Epiphany Syntax	
	2.1 Special Characters	
	300 Dependent Features      Options	
9.14.1 9.14.2	Syntax	
9.14.2		
9.14	1	
9.14 9.14	0	
9.14.3	Floating Point	
9.14.3	H8/300 Machine Directives	
9.14.5	Opcodes	
	PA Dependent Features	
9.15	Notes	
9.15.1 9.15.2	Options	
9.15.2 9.15.3	Syntax	
9.15.4	Floating Point	
	HPPA Assembler Directives	

9.15.6 Opcodes	180
9.16 80386 Dependent Features	181
9.16.1 Options	181
9.16.2 x86 specific Directives	186
9.16.3 i386 Syntactical Considerations	188
9.16.3.1 AT&T Syntax versus Intel Syntax	189
9.16.3.2 Special Characters	189
9.16.4 i386-Mnemonics	
9.16.4.1 Instruction Naming	
9.16.4.2 AT&T Mnemonic versus Intel Mnemonic	
9.16.5 Register Naming	
9.16.6 Instruction Prefixes	
9.16.7 Memory References	
9.16.8 Handling of Jump Instructions	
9.16.9 Floating Point	
9.16.10 Intel's MMX and AMD's 3DNow! SIMD Operations $\ldots$	
9.16.11 AMD's Lightweight Profiling Instructions	
9.16.12 Bit Manipulation Instructions	
9.16.13 AMD's Trailing Bit Manipulation Instructions	
9.16.14 Writing 16-bit Code	
9.16.15 Specifying CPU Architecture	
9.16.16 AMD64 ISA vs. Intel64 ISA	
9.16.17 AT&T Syntax bugs	
9.16.18 Notes	
9.17 IA-64 Dependent Features	
9.17.1 Options	
9.17.2 Syntax	
9.17.2.1 Special Characters	
9.17.2.2 Register Names	
9.17.2.3 IA-64 Processor-Status-Register (PSR) Bit Names	
9.17.2.4 Relocations	
9.17.3 Opcodes 9.18 IP2K Dependent Features	
9.18.1         IP2K Options           9.18.2         IP2K Syntax	
U U	
9.18.2.1 Special Characters 9.19 LM32 Dependent Features	
9.19 LM32 Dependent Features	
9.19.2 Syntax	
9.19.2 Syntax	
9.19.2.2 Relocatable Expression Modifiers	
9.19.2.3 Special Characters	
_	200
9.20 M32C Dependent Features	200
9.20.1 M32C Options	207
9.20.2 M32C Syntax	207
9.20.2.1 Symbolic Operand Modifiers	

9.20.2	2.2 Special Characters	208
9.21 M321	R Dependent Features	209
9.21.1	M32R Options	209
9.21.2	M32R Directives	210
9.21.3	M32R Warnings	211
9.22 M680	0x0 Dependent Features	213
	M680x0 Options	
	Syntax	
9.22.3	Motorola Syntax	217
	Floating Point	
9.22.5	680x0 Machine Directives	218
9.22.6	Opcodes	219
9.22.0	6.1 Branch Improvement	219
9.22.0	6.2 Special Characters	220
9.23 M681	HC11 and M68HC12 Dependent Features	221
9.23.1	M68HC11 and M68HC12 Options	221
9.23.2	Syntax	222
9.23.3	Symbolic Operand Modifiers	224
9.23.4	Assembler Directives	224
9.23.5	Floating Point	225
9.23.6	Opcodes	225
9.23.	6.1 Branch Improvement	225
9.24 S12Z	Dependent Features	227
9.24.1	S12Z Options	227
9.24.2	Syntax	227
9.24.2	2.1 Overview	227
9.24.2	2.2 Addressing Modes	228
9.24.2	2.3 Register Notation	229
9.25 Meta	Dependent Features	231
9.25.1	Options	231
9.25.2	Syntax	231
9.25.2	2.1 Special Characters	231
9.25.2	2.2 Register Names	231
9.26 Micro	oBlaze Dependent Features	232
9.26.1	Directives	232
9.26.2	Syntax for the MicroBlaze	232
9.26.2	2.1 Special Characters	232
9.27 MIPS	S Dependent Features	233
	Assembler options	
	High-level assembly macros	
	Directives to override the size of symbols	
	Controlling the use of small data accesses	
	Directives to override the ISA level	
	Directives to control code generation	
9.27.7	Directives for extending MIPS 16 bit instructions	244
	Directive to mark data as an instruction	
9.27.9	Directives to control the FP ABI	244

9.27.9.1 History of FP ABIs	244
9.27.9.2 Supported FP ABIs	245
9.27.9.3 Automatic selection of FP ABI	245
9.27.9.4 Linking different FP ABI variants	246
9.27.10 Directives to record which NaN encoding is being used	246
9.27.11 Directives to save and restore options	246
9.27.12 Directives to control generation of MIPS ASE instruction	ns247
9.27.13 Directives to override floating-point options	248
9.27.14 Syntactical considerations for the MIPS assembler	248
9.27.14.1 Special Characters	248
9.28 MMIX Dependent Features	249
9.28.1 Command-line Options	249
9.28.2 Instruction expansion	250
9.28.3 Syntax	250
9.28.3.1 Special Characters	250
9.28.3.2 Symbols	
9.28.3.3 Register names	
9.28.3.4 Assembler Directives	
9.28.4 Differences to mmixal	254
9.29 MSP 430 Dependent Features	256
9.29.1 Options	256
9.29.2 Syntax	257
9.29.2.1 Macros	257
9.29.2.2 Special Characters	257
9.29.2.3 Register Names	
9.29.2.4 Assembler Extensions	258
9.29.3 Floating Point	259
9.29.4 MSP 430 Machine Directives	259
9.29.5 Opcodes	259
9.29.6 Profiling Capability	
9.30 NDS32 Dependent Features	
9.30.1 NDS32 Options	262
9.30.2 Syntax	263
9.30.2.1 Special Characters	263
9.30.2.2 Register Names	263
9.30.2.3 Pseudo Instructions	264
9.31 Nios II Dependent Features	267
9.31.1 Options	267
9.31.2 Syntax	267
9.31.2.1 Special Characters	267
9.31.3 Nios II Machine Relocations	267
9.31.4 Nios II Machine Directives	268
9.31.5 Opcodes	269
9.32 NS32K Dependent Features	
9.32.1 Syntax	270
9.32.1.1 Special Characters	
9.33 OPENRISC Dependent Features	271

9.33.1	OpenRISC Syntax	$\dots 271$
9.33	1.1 Special Characters	$\dots 271$
9.33	1.2 Register Names	$\dots 271$
9.33	1.3 Relocations	$\dots 271$
9.33.2	Floating Point	$\dots 273$
9.33.3	OpenRISC Machine Directives	$\dots 273$
9.33.4	Opcodes	274
9.34 PDI	P-11 Dependent Features	275
9.34.1	Options	275
9.34	.1.1 Code Generation Options	$\dots 275$
9.34	.1.2 Instruction Set Extension Options	275
9.34	.1.3 CPU Model Options	$\dots 276$
9.34	.1.4 Machine Model Options	276
9.34.2	Assembler Directives	277
9.34.3	PDP-11 Assembly Language Syntax	277
9.34.4	Instruction Naming	277
9.34.5	Synthetic Instructions	278
9.35 pico	Java Dependent Features	279
9.35.1	Options	279
9.35.2	PJ Syntax	279
9.35	.2.1 Special Characters	$\dots 279$
9.36 Pow	verPC Dependent Features	280
9.36.1	Options	
9.36.2	PowerPC Assembler Directives	282
9.36.3	PowerPC Syntax	282
9.36	.3.1 Special Characters	$\dots 282$
9.37 PRU	U Dependent Features	283
9.37.1	Options	283
9.37.2	Syntax	
9.37	1.2.1 Special Characters	$\dots 283$
9.37.3	PRU Machine Relocations	283
9.37.4	PRU Machine Directives	283
9.37.5	Opcodes	
9.38 RIS	C-V Dependent Features	
9.38.1	RISC-V Options	285
9.38.2	RISC-V Directives	
9.38.3	RISC-V Assembler Modifiers	
9.38.4	RISC-V Floating Point	
9.38.5	RISC-V Instruction Formats	
9.38.6	RISC-V Object Attribute	
9.38.7	RISC-V Custom (Vendor-Defined) Extensions	
	'8 Dependent Features	
9.39.1	RL78 Options	
9.39.2	Symbolic Operand Modifiers	
9.39.3	Assembler Directives	
9.39.4	Syntax for the RL78	
9.39	.4.1 Special Characters	299

9.40 RX Dependent Features	. 300
9.40.1 RX Options	. 300
9.40.2 Symbolic Operand Modifiers	. 301
9.40.3 Assembler Directives	. 302
9.40.4 Floating Point	. 302
9.40.5 Syntax for the RX	. 302
9.40.5.1 Special Characters	. 302
9.41 IBM S/390 Dependent Features	. 303
9.41.1 Options	
9.41.2 Special Characters	. 304
9.41.3 Instruction syntax	. 304
9.41.3.1 Register naming	
9.41.3.2 Instruction Mnemonics	. 304
9.41.3.3 Instruction Operands	. 306
9.41.3.4 Instruction Formats	. 307
9.41.3.5 Instruction Aliases	. 311
9.41.3.6 Instruction Operand Modifier	. 315
9.41.3.7 Instruction Marker	. 316
9.41.3.8 Literal Pool Entries	. 316
9.41.4 Assembler Directives	. 317
9.41.5 Floating Point	. 319
9.42 SCORE Dependent Features	. 320
9.42.1 Options	
9.42.2 SCORE Assembler Directives	. 320
9.42.3 SCORE Syntax	. 321
9.42.3.1 Special Characters	. 321
9.43 Renesas / SuperH SH Dependent Features	. 322
9.43.1 Options	
9.43.2 Syntax	. 322
9.43.2.1 Special Characters	
9.43.2.2 Register Names	. 323
9.43.2.3 Addressing Modes	
9.43.3 Floating Point	
9.43.4 SH Machine Directives	
9.43.5 Opcodes	
9.44 SPARC Dependent Features	
9.44.1 Options	
9.44.2 Enforcing aligned data	
9.44.3 Sparc Syntax	
9.44.3.1 Special Characters	
9.44.3.2 Register Names	
9.44.3.3 Constants	
9.44.3.4 Relocations	
9.44.3.5 Size Translations	
9.44.4 Floating Point	
9.44.5 Sparc Machine Directives	
9.45 TIC54X Dependent Features	. 336

9.45.1 Options	336
9.45.2 Blocking	336
9.45.3 Environment Settings	. 336
9.45.4 Constants Syntax	336
9.45.5 String Substitution	336
9.45.6 Local Labels	337
9.45.7 Math Builtins	337
9.45.8 Extended Addressing	339
9.45.9 Directives	339
9.45.10 Macros	. 344
9.45.11 Memory-mapped Registers	345
9.45.12 TIC54X Syntax	345
9.45.12.1 Special Characters	. 345
9.46 TIC6X Dependent Features	346
9.46.1 TIC6X Options	346
9.46.2 TIC6X Syntax	346
9.46.3 TIC6X Directives	. 347
9.47 TILE-Gx Dependent Features	349
9.47.1 Options	349
9.47.2 Syntax	349
9.47.2.1 Opcode Names	349
9.47.2.2 Register Names	. 350
9.47.2.3 Symbolic Operand Modifiers	350
9.47.3 TILE-Gx Directives	352
9.48 TILEPro Dependent Features	354
9.48.1 Options	354
9.48.2 Syntax	354
9.48.2.1 Opcode Names	354
9.48.2.2 Register Names	. 354
9.48.2.3 Symbolic Operand Modifiers	355
9.48.3 TILEPro Directives	357
9.49 v850 Dependent Features	. 357
9.49.1 Options	357
9.49.2 Syntax	359
9.49.2.1 Special Characters	. 359
9.49.2.2 Register Names	. 359
9.49.3 Floating Point	. 362
9.49.4 V850 Machine Directives	362
9.49.5 Opcodes	362
9.50 VAX Dependent Features	364
9.50.1 VAX Command-Line Options	. 365
9.50.2 VAX Floating Point	
9.50.3 Vax Machine Directives	. 366
9.50.4 VAX Opcodes	366
9.50.5 VAX Branch Improvement	. 366
9.50.6 VAX Operands	
9.50.7 Not Supported on VAX	. 368

9.50.8	VAX Syntax	368
9.50	.8.1 Special Characters	369
9.51 Visi	um Dependent Features	370
9.51.1	Options	370
9.51.2	Syntax	370
9.51	.2.1 Special Characters	370
9.51	.2.2 Register Names	370
9.51.3	Opcodes	370
9.52 Web	Assembly Dependent Features	371
9.52.1	Notes	371
9.52.2	Syntax	371
9.52	.2.1 Special Characters	371
9.52	.2.2 Relocations	371
9.52	.2.3 Signatures	371
9.52.3	Floating Point	371
9.52.4	Regular Opcodes	371
9.52.5	WebAssembly Module Layout	371
9.53 XGA	ATE Dependent Features	
9.53.1	XGATE Options	373
9.53.2	Syntax	373
9.53.3	Assembler Directives	374
9.53.4	Floating Point	374
9.53.5	Opcodes	375
9.54 XSt	ormy16 Dependent Features	375
9.54.1	Syntax	375
9.54	.1.1 Special Characters	375
9.54.2	XStormy16 Machine Directives	375
9.54.3	XStormy16 Pseudo-Opcodes	375
9.55 Xter	nsa Dependent Features	376
9.55.1	Command-line Options	376
9.55.2	Assembler Syntax	377
9.55	.2.1 Opcode Names	378
9.55	.2.2 Register Names	378
9.55.3	Xtensa Optimizations	379
9.55	.3.1 Using Density Instructions	379
9.55	.3.2 Automatic Instruction Alignment	379
9.55.4	Xtensa Relaxation	380
9.55	.4.1 Conditional Branch Relaxation	380
9.55	.4.2 Function Call Relaxation	380
9.55	.4.3 Jump Relaxation	381
9.55	.4.4 Other Immediate Field Relaxation	382
9.55.5	Directives	383
9.55	.5.1 schedule	383
9.55	.5.2 longcalls	384
9.55	.5.3 transform	384
9.55		
9.55	.5.5 literal_position	385

	9.55.	.5.6 literal_prefix	386
	9.55.	.5.7 absolute-literals	386
9.56	Z80	Dependent Features	
9.8	56.1	Command-line Options	387
9.8	56.2	Syntax	388
	9.56.	.2.1 Special Characters	388
	9.56.	.2.2 Register Names	388
	9.56.	.2.3 Case Sensitivity	388
	9.56.	.2.4 Labels	388
9.8	56.3	Floating Point	388
9.8	56.4	Z80 Assembler Directives	389
9.8	56.5	Opcodes	390
9.57	Z800	00 Dependent Features	391
9.8	57.1	Options	391
9.8	57.2	Syntax	391
	9.57.	.2.1 Special Characters	391
	9.57.	.2.2 Register Names	391
		.2.3 Addressing Modes	
9.8	57.3	Assembler Directives for the Z8000	392
9.5	57.4	Opcodes	393
10 F	lepo	orting Bugs	$\dots 395$
10.1	_	e You Found a Bug?	
$10.1 \\ 10.2$		v to Report Bugs	
10.2	1100		000
11 A		nowledgements	300
	ICULI		000
<b>A</b>			401
Appe	naix	<b>KA</b> GNU Free Documentation Lice	ense401
ACT	,		100
AS In	Idex	ξ	409

# 1 Overview

This manual is a user guide to the GNU assembler as.

Here is a brief summary of how to invoke **as**. For details, see Chapter 2 [Command-Line Options], page 25.

as [-a[cdghlns][=file]] [-alternate] [-compress-debug-sections] [-nocompress-debug-sections] [-D] [-dump-config] [-debug-prefix-map old=new] [-defsym sym=val] [-elf-stt-common=[no|yes]] [-emulation=name] [-f] [-g] [-gstabs] [-gstabs+] [-gdwarf-<N>] [-gdwarf-sections] [-gdwarf-cie-version=VERSION] [-generate-missing-build-notes=[no|yes]] [-gsframe] [-hash-size=N] [-help] [-target-help] [-I dir] [-J] [-K] [-keep-locals] [-L] [-listing-lhs-width=NUM] [-listing-lhs-width2=NUM] [-listing-rhs-width=NUM] [-listing-cont-lines=NUM] [-multibyte-handling=[allow | warn | warn-sym-only]] [-no-pad-sections] [-o objfile] [-R] [-sectname-subst] [-size-check=[error|warning]] [-statistics] [-v] [-version] [-version] [-W] [-warn] [-fatal-warnings] [-w] [-x] [-Z] [@FILE] [target-options] [-|files ...] Target AArch64 options: [-EB|-EL] [-mabi=ABI] Target Alpha options: [-mcpu] [-mdebug | -no-mdebug] [-replace | -noreplace] [-relax] [-g] [-Gsize] [-F] [-32addr] Target ARC options: [-mcpu=cpu] [-mA6|-mARC600|-mARC601|-mA7|-mARC700|-mEM|-mHS]

```
[-mcode-density]
   [-mrelax]
   [-EB|-EL]
Target ARM options:
   [-mcpu=processor[+extension...]]
   [-march=architecture[+extension...]]
   [-mfpu=floating-point-format]
   [-mfloat-abi=abi]
   [-meabi=ver]
   [-mthumb]
   [-EB|-EL]
   [-mapcs-32|-mapcs-26|-mapcs-float|
    -mapcs-reentrant]
   [-mthumb-interwork] [-k]
Target Blackfin options:
   [-mcpu=processor[-sirevision]]
   [-mfdpic]
   [-mno-fdpic]
   [-mnopic]
Target BPF options:
   [-EL] [-EB]
Target CRIS options:
   [-underscore | -no-underscore]
   [-pic] [-N]
   [-emulation=criself | -emulation=crisaout]
   [-march=v0_v10 \mid -march=v10 \mid -march=v32 \mid -march=common_v10_v32]
Target C-SKY options:
   [-march=arch] [-mcpu=cpu]
   [-EL] [-mlittle-endian] [-EB] [-mbig-endian]
   [-fpic] [-pic]
   [-mljump] [-mno-ljump]
   [-force2bsr] [-mforce2bsr] [-no-force2bsr] [-mno-force2bsr]
   [-jsri2bsr] [-mjsri2bsr] [-no-jsri2bsr] [-mno-jsri2bsr]
   [-mnolrw] [-mno-lrw]
   [-melrw] [-mno-elrw]
   [-mlaf] [-mliterals-after-func]
   [-mno-laf] [-mno-literals-after-func]
   [-mlabr] [-mliterals-after-br]
   [-mno-labr] [-mnoliterals-after-br]
   [-mistack] [-mno-istack]
   [-mhard-float] [-mmp] [-mcp] [-mcache]
   [-msecurity] [-mtrust]
   [-mdsp] [-medsp] [-mvdsp]
Target D10V options:
   [-0]
Target D30V options:
   [-0|-n|-N]
Target EPIPHANY options:
   [-mepiphany|-mepiphany16]
```

```
Target H8/300 options:
   [-h-tick-hex]
Target i386 options:
   [-32|-x32|-64] [-n]
   [-march=CPU[+EXTENSION...]] [-mtune=CPU]
Target IA-64 options:
   [-mconstant-gp|-mauto-pic]
   [-milp32|-milp64|-mlp64|-mp64]
   [-mle|mbe]
   [-mtune=itanium1|-mtune=itanium2]
   [-munwind-check=warning|-munwind-check=error]
   [-mhint.b=ok|-mhint.b=warning|-mhint.b=error]
   [-x|-xexplicit] [-xauto] [-xdebug]
Target IP2K options:
   [-mip2022|-mip2022ext]
Target M32C options:
   [-m32c|-m16c] [-relax] [-h-tick-hex]
Target M32R options:
   [-m32rx|-[no-]warn-explicit-parallel-conflicts|
   -W[n]p]
Target M680X0 options:
   [-l] [-m68000|-m68010|-m68020|...]
Target M68HC11 options:
   [-m68hc11|-m68hc12|-m68hcs12|-mm9s12x|-mm9s12xg]
   [-mshort|-mlong]
   [-mshort-double|-mlong-double]
   [-force-long-branches] [-short-branches]
   [-strict-direct-mode] [-print-insn-syntax]
   [-print-opcodes] [-generate-example]
Target MCORE options:
   [-jsri2bsr] [-sifilter] [-relax]
   [-mcpu=[210|340]]
Target Meta options:
   [-mcpu=cpu] [-mfpu=cpu] [-mdsp=cpu]
Target MICROBLAZE options:
Target MIPS options:
   [-nocpp] [-EL] [-EB] [-O[optimization level]]
   [-g[debug level]] [-G num] [-KPIC] [-call_shared]
   [-non_shared] [-xgot [-mvxworks-pic]
   [-mabi=ABI] [-32] [-n32] [-64] [-mfp32] [-mgp32]
   [-mfp64] [-mgp64] [-mfpxx]
   [-modd-spreg] [-mno-odd-spreg]
   [-march=CPU] [-mtune=CPU] [-mips1] [-mips2]
   [-mips3] [-mips4] [-mips5] [-mips32] [-mips32r2]
   [-mips32r3] [-mips32r5] [-mips32r6] [-mips64] [-mips64r2]
   [-mips64r3] [-mips64r5] [-mips64r6]
   [-construct-floats] [-no-construct-floats]
   [-mignore-branch-isa] [-mno-ignore-branch-isa]
```

[-mnan=encoding] [-trap] [-no-break] [-break] [-no-trap] [-mips16] [-no-mips16] [-mmips16e2] [-mno-mips16e2] [-mmicromips] [-mno-micromips] [-msmartmips] [-mno-smartmips] [-mips3d] [-no-mips3d] [-mdmx] [-no-mdmx] [-mdsp] [-mno-dsp] [-mdspr2] [-mno-dspr2] [-mdspr3] [-mno-dspr3] [-mmsa] [-mno-msa] [-mxpa] [-mno-xpa] [-mmt] [-mno-mt] [-mmcu] [-mno-mcu] [-mcrc] [-mno-crc] [-mginv] [-mno-ginv] [-mloongson-mmi] [-mno-loongson-mmi] [-mloongson-cam] [-mno-loongson-cam] [-mloongson-ext] [-mno-loongson-ext] [-mloongson-ext2] [-mno-loongson-ext2] [-minsn32] [-mno-insn32] [-mfix7000] [-mno-fix7000] [-mfix-rm7000] [-mno-fix-rm7000] [-mfix-vr4120] [-mno-fix-vr4120] [-mfix-vr4130] [-mno-fix-vr4130] [-mfix-r5900] [-mno-fix-r5900] [-mdebug] [-no-mdebug] [-mpdr] [-mno-pdr] Target MMIX options: [-fixed-special-register-names] [-globalize-symbols] [-gnu-syntax] [-relax] [-no-predefined-symbols] [-no-expand] [-no-merge-gregs] [-x] [-linker-allocated-gregs] Target Nios II options: [-relax-all] [-relax-section] [-no-relax] [-EB] [-EL] Target NDS32 options: [-EL] [-EB] [-O] [-Os] [-mcpu=cpu] [-misa=isa] [-mabi=abi] [-mall-ext] [-m[no-]16-bit] [-m[no-]perf-ext] [-m[no-]perf2-ext] [-m[no-]audio-isa-ext] [-m[no-]fpu-sp-ext] [-m[no-]fpu-dp-ext] [-m[no-]fpu-fma] [-mfpu-freg=FREG] [-mreduced-regs] [-mfull-regs] [-m[no-]dx-regs] [-mpic] [-mno-relax] [-mb2bb] Target PDP11 options: [-mpic|-mno-pic] [-mall] [-mno-extensions] [-mextension|-mno-extension] [-mcpu] [-mmachine] Target picoJava options: [-mb|-me]

4

```
Target PowerPC options:
            [-a32|-a64]
           [-mpwrx | -mpwr2 | -mpwr | -m601 | -mppc | -mppc32 | -m603 | -m604 | -m403 | -m405 |
               -m440 \left| -m464 \right| -m476 \left| -m7400 \right| -m7410 \left| -m7450 \right| -m7455 \left| -m750cl \right| -mgekko \right|
               -mbroadway \, | \, -mppc64 \, | \, -m620 \, | \, -me500 \, | \, -e500 x 2 \, | \, -me500 mc \, | \, -me500 mc 64 \, | \, -me5500 \, | \, -me5500 \, | \, -me500 mc \, | \, -me500 mc \, | \, -me500 \, | \, -me50
               -me6500|-mppc64bridge|-mbooke|-mpower4|-mpwr4|-mpower5|-mpwr5x|
               -mpower6|-mpwr6|-mpwr7|-mpwr7|-mpower8|-mpwr8|-mpwr9-ma2|
               -mcell|-mspe|-mspe2|-mtitan|-me300|-mcom]
             [-many] [-maltivec|-mvsx|-mhtm|-mvle]
            [-mregnames|-mno-regnames]
           [-mrelocatable|-mrelocatable-lib|-K PIC] [-memb]
           [-mlittle|-mlittle-endian|-le|-mbig|-mbig-endian|-be]
           [-msolaris|-mno-solaris]
           [-nops=count]
Target PRU options:
           [-link-relax]
           [-mnolink-relax]
           [-mno-warn-regname-label]
Target RISC-V options:
           [-fpic|-fPIC|-fno-pic]
           [-march=ISA]
           [-mabi=ABI]
           [-mlittle-endian|-mbig-endian]
Target RL78 options:
           [-mg10]
           [-m32bit-doubles]-m64bit-doubles]
Target RX options:
           [-mlittle-endian|-mbig-endian]
           [-m32bit-doubles|-m64bit-doubles]
           [-muse-conventional-section-names]
           [-msmall-data-limit]
           [-mpid]
           [-mrelax]
           [-mint-register=number]
           [-mgcc-abi|-mrx-abi]
Target s390 options:
           [-m31|-m64] [-mesa|-mzarch] [-march=CPU]
           [-mregnames|-mno-regnames]
           [-mwarn-areg-zero]
Target SCORE options:
             [-EB] [-EL] [-FIXDD] [-NWARN]
           [-SCORE5] [-SCORE5U] [-SCORE7] [-SCORE3]
            [-march=score7] [-march=score3]
           [-USE_R1] [-KPIC] [-O0] [-G num] [-V]
Target SPARC options:
            [-Av6|-Av7|-Av8|-Aleon|-Asparclet|-Asparclite
               -Av8plus|-Av8plusa|-Av8plusb|-Av8plusc|-Av8plusd
               -Av8plusv|-Av8plusm|-Av9|-Av9a|-Av9b|-Av9c
               -Av9d|-Av9e|-Av9v|-Av9m|-Asparc|-Asparcvis
               -Asparcvis2|-Asparcfmaf|-Asparcvis3
               -Asparcvisr |-Asparc5]
```

```
[-xarch=v8plus|-xarch=v8plusa] |-xarch=v8plusb|-xarch=v8plusc
-xarch=v8plusd|-xarch=v8plusv|-xarch=v8plusm|-xarch=v9
-xarch=v9a|-xarch=v9b|-xarch=v9c|-xarch=v9d|-xarch=v9e
-xarch=v9v|-xarch=v9m|-xarch=sparc|-xarch=sparcvis
-xarch=sparcvis2|-xarch=sparcfmaf|-xarch=sparcima
-xarch=sparcvis3|-xarch=sparcvisr|-xarch=sparc5
-bump]
[-32|-64]
[-enforce-aligned-data] [-dcti-couples-detect]
```

```
Target TIC54X options:
```

```
[-mcpu=54[123589]|-mcpu=54[56]lp] [-mfar-mode|-mf]
[-merrors-to-file <filename>|-me <filename>]
```

```
Target TIC6X options:
[-march=arch] [-mbig-endian|-mlittle-endian]
[-mdsbt|-mno-dsbt] [-mpid=no|-mpid=near|-mpid=far]
[-mpic|-mno-pic]
```

```
Target TILE-Gx options:
[-m32|-m64][-EB][-EL]
```

```
Target Visium options: [-mtune=arch]
```

```
Target Xtensa options:

[-[no-]text-section-literals] [-[no-]auto-litpools]

[-[no-]absolute-literals]

[-[no-]target-align] [-[no-]longcalls]

[-[no-]transform]

[-rename-section oldname=newname]

[-[no-]trampolines]

[-abi-windowed|-abi-call0]
```

```
Target Z80 options:
[-march=CPU[-EXT]]+EXT]]
[-local-prefix=PREFIX]
[-colonless]
[-sdcc]
[-fp-s=FORMAT]
[-fp-d=FORMAT]
```

**@file** Read command-line options from *file*. The options read are inserted in place of the original @*file* option. If *file* does not exist, or cannot be read, then the option will be treated literally, and not removed.

Options in file are separated by whitespace. A whitespace character may be included in an option by surrounding the entire option in either single or double quotes. Any character (including a backslash) may be included by prefixing the character to be included with a backslash. The *file* may itself contain additional @*file* options; any such options will be processed recursively.

-a[cdghlmns]

Turn on listings, in any of a variety of ways:

-ac omit false conditionals

-ad	$\operatorname{omit}$	debugging	directives
-----	-----------------------	-----------	------------

- -ag include general information, like as version and options passed
- -ah include high-level source
- -al include assembly
- -am include macro expansions
- -an omit forms processing
- -as include symbols
- =file set the name of the listing file

You may combine these options; for example, use '-aln' for assembly listing without forms processing. The '=file' option, if used, must be the last one. By itself, '-a' defaults to '-ahls'.

--alternate

Begin in alternate macro mode. See Section 7.4 [.altmacro], page 52.

#### --compress-debug-sections

Compress DWARF debug sections using zlib with SHF\_COMPRESSED from the ELF ABI. The resulting object file may not be compatible with older linkers and object file utilities. Note if compression would make a given section *larger* then it is not compressed.

- --compress-debug-sections=none
- --compress-debug-sections=zlib
- --compress-debug-sections=zlib-gnu
- --compress-debug-sections=zlib-gabi
- --compress-debug-sections=zstd

These options control how DWARF debug sections are compressed. --compress-debug-sections=none is equivalent to --nocompress-debugsections. --compress-debug-sections=zlib and --compress-debugsections=zlib-gabi are equivalent to --compress-debug-sections. --compress-debug-sections=zlib-gnu compresses DWARF debug sections using the obsoleted zlib-gnu format. The debug sections are renamed to begin with '.zdebug'. --compress-debug-sections=zstd compresses DWARF debug sections using zstd. Note - if compression would actually make a section *larger*, then it is not compressed nor renamed.

#### --nocompress-debug-sections

Do not compress DWARF debug sections. This is usually the default for all targets except the  $x86/x86_64$ , but a configure time option can be used to override this.

-D Enable denugging in target specific backends, if supported. Otherwise ignored. Even if ignored, this option is accepted for script compatibility with calls to other assemblers.

# --debug-prefix-map old=new

When assembling files in directory old, record debugging information describing them as in **new** instead.

#### --defsym sym=value

Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading '0x' indicates a hexadecimal value, and a leading '0' indicates an octal value. The value of the symbol can be overridden inside a source file via the use of a .set pseudo-op.

#### --dump-config

Displays how the assembler is configured and then exits.

#### --elf-stt-common=no

#### --elf-stt-common=yes

These options control whether the ELF assembler should generate common symbols with the STT\_COMMON type. The default can be controlled by a configure option --enable-elf-stt-common.

#### --emulation=name

If the assembler is configured to support multiple different target configurations then this option can be used to select the desired form.

#### -f

"fast"—skip whitespace and comment preprocessing (assume source is compiler output).

#### -g

#### --gen-debug

Generate debugging information for each assembler source line using whichever debug format is preferred by the target. This currently means either STABS, ECOFF or DWARF2. When the debug format is DWARF then a .debug\_info and .debug\_line section is only emitted when the assembly file doesn't generate one itself.

- --gstabs Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
- --gstabs+

Generate stabs debugging information for each assembler line, with GNU extensions that probably only gdb can handle, and that could make other debuggers crash or refuse to read your program. This may help debugging assembler code. Currently the only GNU extension is the location of the current working directory at assembling time.

--gdwarf-2

Generate DWARF2 debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it. Note—this option is only supported by some targets, not all of them.

--gdwarf-3

This option is the same as the --gdwarf-2 option, except that it allows for the possibility of the generation of extra debug information as per version 3 of the DWARF specification. Note - enabling this option does not guarantee the generation of any extra information, the choice to do so is on a per target basis.

#### --gdwarf-4

This option is the same as the --gdwarf-2 option, except that it allows for the possibility of the generation of extra debug information as per version 4 of the DWARF specification. Note - enabling this option does not guarantee the generation of any extra information, the choice to do so is on a per target basis.

#### --gdwarf-5

This option is the same as the --gdwarf-2 option, except that it allows for the possibility of the generation of extra debug information as per version 5 of the DWARF specification. Note - enabling this option does not guarantee the generation of any extra information, the choice to do so is on a per target basis.

#### --gdwarf-sections

Instead of creating a .debug\_line section, create a series of .debug\_line.foo sections where foo is the name of the corresponding code section. For example a code section called .text.func will have its dwarf line number information placed into a section called .debug\_line.text.func. If the code section is just called .text then debug line section will still be called just .debug\_line without any suffix.

#### --gdwarf-cie-version=version

Control which version of DWARF Common Information Entries (CIEs) are produced. When this flag is not specificed the default is version 1, though some targets can modify this default. Other possible values for version are 3 or 4.

# --generate-missing-build-notes=yes

# --generate-missing-build-notes=no

These options control whether the ELF assembler should generate GNU Build attribute notes if none are present in the input sources. The default can be controlled by the --enable-generate-build-notes configure option.

# --gsframe

# --gsframe

Create *.sframe* section from CFI directives.

--hash-size N

Ignored. Supported for command line compatibility with other assemblers.

--help Print a summary of the command-line options and exit.

#### --target-help

Print a summary of all target specific options and exit.

- -I dir Add directory dir to the search list for .include directives.
- -J Don't warn about signed overflow.
- -K Issue warnings when difference tables altered for long displacements.
- -L

# --keep-locals

Keep (in the symbol table) local symbols. These symbols start with systemspecific local label prefixes, typically '.L' for ELF systems or 'L' for traditional a.out systems. See Section 5.3 [Symbol Names], page 43.

```
--listing-lhs-width=number
```

Set the maximum width, in words, of the output data column for an assembler listing to *number*.

# --listing-lhs-width2=number

Set the maximum width, in words, of the output data column for continuation lines in an assembler listing to *number*.

#### --listing-rhs-width=number

Set the maximum width of an input source line, as displayed in a listing, to number bytes.

#### --listing-cont-lines=number

Set the maximum number of lines printed in a listing for a single line of input to number + 1.

#### --multibyte-handling=allow

--multibyte-handling=warn

# --multibyte-handling=warn-sym-only

# --multibyte-handling=warn\_sym\_only

Controls how the assembler handles multibyte characters in the input. The default (which can be restored by using the **allow** argument) is to allow such characters without complaint. Using the **warn** argument will make the assembler generate a warning message whenever any multibyte character is encountered. Using the **warn-sym-only** argument will only cause a warning to be generated when a symbol is defined with a name that contains multibyte characters. (References to undefined symbols will not generate a warning).

#### --no-pad-sections

Stop the assembler for padding the ends of output sections to the alignment of that section. The default is to pad the sections, but this can waste space which might be needed on targets which have tight memory constraints.

-o objfile

-R

Name the object-file output from as objfile.

Fold the data section into the text section.

# --reduce-memory-overheads

Ignored. Supported for compatibility with tools that apss the same option to both the assembler and the linker.

# --sectname-subst

Honor substitution sequences in section names. See [.section name], page 78.

```
--size-check=error
```

# --size-check=warning

Issue an error or warning for invalid ELF .size directive.

#### --statistics

Print the maximum space (in bytes) and total time (in seconds) used by assembly.

#### --strip-local-absolute

Remove local absolute symbols from the outgoing symbol table.

```
-v
          Print the as version.
-version
--version
           Print the as version and exit.
-W
--no-warn
           Suppress warning messages.
--fatal-warnings
           Treat warnings as errors.
           Don't suppress warning messages or treat them as errors.
--warn
           Ignored.
-w
           Ignored.
-x
-7.
           Generate an object file even after errors.
-- | files ...
           Standard input, or source files to assemble.
   See Section 9.1.1 [AArch64 Options], page 96, for the options available when as is
configured for the 64-bit mode of the ARM Architecture (AArch64).
   See Section 9.2.2 [Alpha Options], page 102, for the options available when as is configured
for an Alpha processor.
   The following options are available when as is configured for an ARC processor.
-mcpu=cpu
           This option selects the core processor variant.
-EB | -EL Select either big-endian (-EB) or little-endian (-EL) output.
-mcode-density
           Enable Code Density extension instructions.
   The following options are available when as is configured for the ARM processor family.
-mcpu=processor[+extension...]
           Specify which ARM processor variant is the target.
-march=architecture[+extension...]
           Specify which ARM architecture variant is used by the target.
-mfpu=floating-point-format
           Select which Floating Point architecture is the target.
-mfloat-abi=abi
           Select which floating point ABI is in use.
           Enable Thumb only instruction decoding.
-mthumb
-mapcs-32 | -mapcs-26 | -mapcs-float | -mapcs-reentrant
           Select which procedure calling convention is in use.
-EB | -EL Select either big-endian (-EB) or little-endian (-EL) output.
```

-mthumb-interwork

Specify that the code has been generated with interworking between Thumb and ARM code in mind.

-mccs Turns on CodeComposer Studio assembly syntax compatibility mode.

-k Specify that PIC code has been generated.

See Section 9.6.1 [Blackfin Options], page 142, for the options available when as is configured for the Blackfin processor family.

See Section 9.7.1 [BPF Options], page 146, for the options available when as is configured for the Linux kernel BPF processor family.

See the info pages for documentation of the CRIS-specific options.

See Section 9.10.1 [C-SKY Options], page 162, for the options available when as is configured for the C-SKY processor family.

The following options are available when as is configured for a D10V processor.

-0 Optimize output by parallelizing instructions.

The following options are available when as is configured for a D30V processor.

-0 Optimize output by parallelizing instructions.

-n Warn when nops are generated.

-N Warn when a nop after a 32-bit multiply instruction is generated.

The following options are available when as is configured for the Adapteva EPIPHANY series.

See Section 9.13.1 [Epiphany Options], page 172, for the options available when as is configured for an Epiphany processor.

See Section 9.16.1 [i386-Options], page 181, for the options available when as is configured for an i386 processor.

The following options are available when as is configured for the Ubicom IP2K series.

-mip2022ext

Specifies that the extended IP2022 instructions are allowed.

-mip2022 Restores the default behaviour, which restricts the permitted instructions to just the basic IP2022 ones.

The following options are available when as is configured for the Renesas M32C and M16C processors.

-m32c Assemble M32C instructions.

-m16c Assemble M16C instructions (the default).

-relax Enable support for link-time relaxations.

-h-tick-hex

Support H'00 style hex constants in addition to 0x00 style.

The following options are available when as is configured for the Renesas M32R (formerly Mitsubishi M32R) series.

- --m32rx Specify which processor in the M32R family is the target. The default is normally the M32R, but this option changes it to the M32RX.
- --warn-explicit-parallel-conflicts or --Wp

Produce warning messages when questionable parallel constructs are encountered.

--no-warn-explicit-parallel-conflicts or --Wnp Do not produce warning messages when questionable parallel constructs are encountered.

The following options are available when as is configured for the Motorola 68000 series.

-1 Shorten references to undefined symbols, to one word instead of two.

-m68000 | -m68008 | -m68010 | -m68020 | -m68030

| -m68040 | -m68060 | -m68302 | -m68331 | -m68332

| -m68333 | -m68340 | -mcpu32 | -m5200

Specify what processor in the 68000 family is the target. The default is normally the 68020, but this can be changed at configuration time.

```
-m68881 | -m68882 | -mno-68881 | -mno-68882
```

The target machine does (or does not) have a floating-point coprocessor. The default is to assume a coprocessor for 68020, 68030, and cpu32. Although the basic 68000 is not compatible with the 68881, a combination of the two can be specified, since it's possible to do emulation of the coprocessor instructions with the main processor.

```
-m68851 | -mno-68851
```

The target machine does (or does not) have a memory-management unit coprocessor. The default is to assume an MMU for 68020 and up.

See Section 9.31.1 [Nios II Options], page 267, for the options available when as is configured for an Altera Nios II processor.

For details about the PDP-11 machine dependent features options, see Section 9.34.1 [PDP-11-Options], page 275.

```
-mpic | -mno-pic
```

Generate position-independent (or position-dependent) code. The default is -mpic.

```
-mall
```

-mall-extensions

Enable all instruction set extensions. This is the default.

```
-mno-extensions
```

Disable all instruction set extensions.

# -mextension | -mno-extension

Enable (or disable) a particular instruction set extension.

-mcpu	Enable the instruction set extensions supported by a particular CPU, and disable
	all other extensions.

-mmachine

Enable the instruction set extensions supported by a particular machine model, and disable all other extensions.

The following options are available when as is configured for a picoJava processor.

-mb Generate "big endian" format output.

-ml Generate "little endian" format output.

See Section 9.37.1 [PRU Options], page 283, for the options available when as is configured for a PRU processor.

The following options are available when as is configured for the Motorola 68HC11 or 68HC12 series.

-m68hc11 | -m68hc12 | -m68hcs12 | -mm9s12x | -mm9s12xg

Specify what processor is the target. The default is defined by the configuration option when building the assembler.

#### --xgate-ramoffset

Instruct the linker to offset RAM addresses from S12X address space into XGATE address space.

-mshort Specify to use the 16-bit integer ABI.

-mlong Specify to use the 32-bit integer ABI.

-mshort-double

Specify to use the 32-bit double ABI.

-mlong-double

Specify to use the 64-bit double ABI.

# --force-long-branches

Relative branches are turned into absolute ones. This concerns conditional branches, unconditional branches and branches to a sub routine.

# -S | --short-branches

Do not turn relative branches into absolute ones when the offset is out of range.

#### --strict-direct-mode

Do not turn the direct addressing mode into extended addressing mode when the instruction does not support direct addressing mode.

#### --print-insn-syntax

Print the syntax of instruction in case of error.

#### --print-opcodes

Print the list of instructions with syntax and then exit.

#### --generate-example

Print an example of instruction for each possible instruction and then exit. This option is only useful for testing as.

The following options are available when **as** is configured for the SPARC architecture:

```
-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite
-Av8plus | -Av8plusa | -Av9 | -Av9a
```

Explicitly select a variant of the SPARC architecture.

'-Av8plus' and '-Av8plusa' select a 32 bit environment. '-Av9' and '-Av9a' select a 64 bit environment.

'-Av8plusa' and '-Av9a' enable the SPARC V9 instruction set with UltraSPARC extensions.

-xarch=v8plus | -xarch=v8plusa

For compatibility with the Solaris v9 assembler. These options are equivalent to -Av8plus and -Av8plusa, respectively.

-bump Warn when the assembler switches to another architecture.

The following options are available when as is configured for the 'c54x architecture.

#### -mfar-mode

Enable extended addressing mode. All addresses and relocations will assume extended addressing (usually 23 bits).

# -mcpu=CPU\_VERSION

Sets the CPU version being compiled for.

#### -merrors-to-file FILENAME

Redirect error output to a file, for broken systems which don't support such behaviour in the shell.

The following options are available when as is configured for a MIPS processor.

- -G num This option sets the largest size of an object that can be referenced implicitly with the gp register. It is only accepted for targets that use ECOFF format, such as a DECstation running Ultrix. The default value is 8.
- -EB Generate "big endian" format output.
- -EL Generate "little endian" format output.

```
-mips1
-mips2
-mips3
-mips4
-mips5
-mips32r2
-mips32r3
-mips32r5
-mips64
-mips64r2
-mips64r3
-mips64r5
-mips64r6
```

Generate code for a particular MIPS Instruction Set Architecture level. '-mips1' is an alias for '-march=r3000', '-mips2' is an alias for '-march=r6000', '-mips3' is an alias for '-march=r4000' and '-mips4' is an alias for '-march=r8000'. '-mips5', '-mips32', '-mips32r2', '-mips32r3', '-mips32r5', '-mips32r6', '-mips64', '-mips64r2', '-mips64r3', '-mips64r5', and '-mips64r6' correspond to generic MIPS V, MIPS32, MIPS32 Release 2, MIPS32 Release 3, MIPS32 Release 5, MIPS32 Release 6, MIPS64 Release 6 ISA processors, respectively.

#### -march=cpu

Generate code for a particular MIPS CPU.

#### -mtune=cpu

Schedule and tune for a particular MIPS CPU.

#### -mfix7000

#### -mno-fix7000

Cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

# -mfix-rm7000

# -mno-fix-rm7000

Cause nops to be inserted if a dmult or dmultu instruction is followed by a load instruction.

#### -mfix-r5900

# -mno-fix-r5900

Do not attempt to schedule the preceding instruction into the delay slot of a branch instruction placed at the end of a short loop of six instructions or fewer and always schedule a **nop** instruction there instead. The short loop bug under certain conditions causes loops to execute only once or twice, due to a hardware bug in the R5900 chip.

-mdebug		
-no-mdebu	g Cause stabs-style debugging output to go into an ECOFF-style .mdebug section instead of the standard ELF .stabs sections.	
-mpdr		
-mno-pdr	Control generation of .pdr sections.	
-mgp32		
-mfp32	The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 32 bits wide at all times. '-mgp32' controls the size of general-purpose registers and '-mfp32' controls the size of floating-point registers.	
-mgp64		
-mfp64	The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 64 bits wide at all times. '-mgp64' controls the size of general-purpose registers and '-mfp64' controls the size of floating-point registers.	
-mfpxx	The register sizes are normally inferred from the ISA and ABI, but using this flag in combination with '-mabi=32' enables an ABI variant which will operate correctly with floating-point registers which are 32 or 64 bits wide.	
-modd-spr	eg	
-mno-odd-spreg		
	Enable use of floating-point operations on odd-numbered single-precision registers when supported by the ISA. '-mfpxx' implies '-mno-odd-spreg', otherwise the default is '-modd-spreg'.	
-mips16		
-no-mips1		
	Generate code for the MIPS 16 processor. This is equivalent to putting .module mips16 at the start of the assembly file. '-no-mips16' turns off this option.	
-mmips16e	2	
-mno-mips		
	Enable the use of MIPS16e2 instructions in MIPS16 mode. This is equivalent to putting .module mips16e2 at the start of the assembly file. '-mno-mips16e2' turns off this option.	
-mmicromi	ps	
-mno-micr	-	
	Generate code for the microMIPS processor. This is equivalent to putting .module micromips at the start of the assembly file. '-mno-micromips' turns off this option. This is equivalent to putting .module nomicromips at the start of the assembly file.	
-msmartmi	ps	
-mno-smar	tmips	

Enables the SmartMIPS extension to the MIPS32 instruction set. This is equivalent to putting .module smartmips at the start of the assembly file. '-mno-smartmips' turns off this option.

-mips3d	
-no-mips3d	1
	Generate code for the MIPS-3D Application Specific Extension. This tells the assembler to accept MIPS-3D instructions. '-no-mips3d' turns off this option.
-mdmx	
-no-mdmx	Generate code for the MDMX Application Specific Extension. This tells the assembler to accept MDMX instructions. '-no-mdmx' turns off this option.
-mdsp	
-mno-dsp	Generate code for the DSP Release 1 Application Specific Extension. This tells the assembler to accept DSP Release 1 instructions. '-mno-dsp' turns off this option.
-mdspr2	
-mno-dspr2	2
-	Generate code for the DSP Release 2 Application Specific Extension. This option implies '-mdsp'. This tells the assembler to accept DSP Release 2 instructions. '-mno-dspr2' turns off this option.
-mdspr3	
-mno-dspr3	3
	Generate code for the DSP Release 3 Application Specific Extension. This option implies '-mdsp' and '-mdspr2'. This tells the assembler to accept DSP Release 3 instructions. '-mno-dspr3' turns off this option.
-mmsa	
-mno-msa	Generate code for the MIPS SIMD Architecture Extension. This tells the assembler to accept MSA instructions. '-mno-msa' turns off this option.
-mxpa	
-mno-xpa	Generate code for the MIPS eXtended Physical Address (XPA) Extension. This tells the assembler to accept XPA instructions. '-mno-xpa' turns off this option.
-mmt	
-mno-mt	Generate code for the MT Application Specific Extension. This tells the assembler to accept MT instructions. '-mno-mt' turns off this option.
-mmcu	
-mno-mcu	Generate code for the MCU Application Specific Extension. This tells the assembler to accept MCU instructions. '-mno-mcu' turns off this option.
-mcrc	
-mno-crc	Generate code for the MIPS cyclic redundancy check (CRC) Application Specific Extension. This tells the assembler to accept CRC instructions. '-mno-crc' turns off this option.
-mginv	
-mno-ginv	
	Generate code for the Global INValidate (GINV) Application Specific Extension. This tells the assembler to accept GINV instructions. '-mno-ginv' turns off this option.

### -mloongson-mmi

-mno-loongson-mmi

Generate code for the Loongson MultiMedia extensions Instructions (MMI) Application Specific Extension. This tells the assembler to accept MMI instructions. '-mno-loongson-mmi' turns off this option.

### -mloongson-cam

### -mno-loongson-cam

Generate code for the Loongson Content Address Memory (CAM) instructions. This tells the assembler to accept Loongson CAM instructions. '-mno-loongson-cam' turns off this option.

### -mloongson-ext

### -mno-loongson-ext

Generate code for the Loongson EXTensions (EXT) instructions. This tells the assembler to accept Loongson EXT instructions. '-mno-loongson-ext' turns off this option.

### -mloongson-ext2

### -mno-loongson-ext2

Generate code for the Loongson EXTensions R2 (EXT2) instructions. This option implies '-mloongson-ext'. This tells the assembler to accept Loongson EXT2 instructions. '-mno-loongson-ext2' turns off this option.

### -minsn32

### -mno-insn32

Only use 32-bit instruction encodings when generating code for the microMIPS processor. This option inhibits the use of any 16-bit instructions. This is equivalent to putting .set insn32 at the start of the assembly file. '-mno-insn32' turns off this option. This is equivalent to putting .set noinsn32 at the start of the assembly file. By default '-mno-insn32' is selected, allowing all instructions to be used.

### --construct-floats

### --no-construct-floats

The '--no-construct-floats' option disables the construction of double width floating point constants by loading the two halves of the value into the two single width floating point registers that make up the double width register. By default '--construct-floats' is selected, allowing construction of these floating point constants.

#### --relax-branch

### --no-relax-branch

The '--relax-branch' option enables the relaxation of out-of-range branches. By default '--no-relax-branch' is selected, causing any out-of-range branches to produce an error.

### -mignore-branch-isa

### -mno-ignore-branch-isa

Ignore branch checks for invalid transitions between ISA modes. The semantics of branches does not provide for an ISA mode switch, so in most cases the ISA mode a branch has been encoded for has to be the same as the ISA mode of the branch's target label. Therefore GAS has checks implemented that verify in branch assembly that the two ISA modes match. '-mignore-branch-isa' disables these checks. By default '-mno-ignore-branch-isa' is selected, causing any invalid branch requiring a transition between ISA modes to produce an error.

#### -mnan=encoding

Select between the IEEE 754-2008 (-mnan=2008) or the legacy (-mnan=legacy) NaN encoding format. The latter is the default.

--emulation=name

This option was formerly used to switch between ELF and ECOFF output on targets like IRIX 5 that supported both. MIPS ECOFF support was removed in GAS 2.24, so the option now serves little purpose. It is retained for backwards compatibility.

The available configuration names are: 'mipself', 'mipslelf' and 'mipsbelf'. Choosing 'mipself' now has no effect, since the output is always ELF. 'mipslelf' and 'mipsbelf' select little- and big-endian output respectively, but '-EL' and '-EB' are now the preferred options instead.

**-nocpp** as ignores this option. It is accepted for compatibility with the native tools.

```
--trap
```

### --no-trap

--break

--no-break

Control how to deal with multiplication overflow and division by zero. '--trap' or '--no-break' (which are synonyms) take a trap exception (and only work for Instruction Set Architecture level 2 and higher); '--break' or '--no-trap' (also synonyms, and the default) take a break exception.

-n When this option is used, as will issue a warning every time it generates a nop instruction from a macro.

The following options are available when as is configured for an MCore processor.

#### -jsri2bsr

```
-nojsri2bsr
```

Enable or disable the JSRI to BSR transformation. By default this is enabled. The command-line option '-nojsri2bsr' can be used to disable it.

## -sifilter

```
-nosifilter
```

Enable or disable the silicon filter behaviour. By default this is disabled. The default can be overridden by the '-sifilter' command-line option.

-relax Alter jump instructions for long displacements.

### -mcpu=[210|340]

Select the cpu type on the target hardware. This controls which instructions can be assembled.

-EB Assemble for a big endian target.

-EL Assemble for a little endian target.

See Section 9.25.1 [Meta Options], page 231, for the options available when as is configured for a Meta processor.

See the info pages for documentation of the MMIX-specific options.

See Section 9.30.1 [NDS32 Options], page 262, for the options available when as is configured for a NDS32 processor.

See Section 9.36.1 [PowerPC-Opts], page 280, for the options available when as is configured for a PowerPC processor.

See Section 9.38.1 [RISC-V-Options], page 285, for the options available when as is configured for a RISC-V processor.

See the info pages for documentation of the RX-specific options.

The following options are available when as is configured for the s390 processor family.

-m31

-m64 Select the word size, either 31/32 bits or 64 bits.

-mesa

-mzarch Select the architecture mode, either the Enterprise System Architecture (esa) or the z/Architecture mode (zarch).

### -march=processor

Specify which s390 processor variant is the target, 'g5' (or 'arch3'), 'g6', 'z900' (or 'arch5'), 'z990' (or 'arch6'), 'z9-109', 'z9-ec' (or 'arch7'), 'z10' (or 'arch8'), 'z196' (or 'arch9'), 'zEC12' (or 'arch10'), 'z13' (or 'arch11'), 'z14' (or 'arch12'), 'z15' (or 'arch13'), or 'z16' (or 'arch14').

#### -mregnames

-mno-regnames

Allow or disallow symbolic names for registers.

### -mwarn-areg-zero

Warn whenever the operand for a base or index register has been specified but evaluates to zero.

See Section 9.46.1 [TIC6X Options], page 346, for the options available when as is configured for a TMS320C6000 processor.

See Section 9.47.1 [TILE-Gx Options], page 349, for the options available when as is configured for a TILE-Gx processor.

See Section 9.51.1 [Visium Options], page 370, for the options available when as is configured for a Visium processor.

See Section 9.55.1 [Xtensa Options], page 376, for the options available when as is configured for an Xtensa processor.

See Section 9.56.1 [Z80 Options], page 387, for the options available when as is configured for an Z80 processor.

## 1.1 Structure of this Manual

This manual is intended to describe what you need to know to use GNU as. We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that as understands; and of course how to invoke as.

This manual also describes some of the machine-dependent features of various flavors of the assembler.

On the other hand, this manual is *not* intended as an introduction to programming in assembly language—let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do *not* describe the instruction set, standard mnemonics, registers or addressing modes that are standard to a particular architecture. You may want to consult the manufacturer's machine architecture manual for this information.

## 1.2 The GNU Assembler

GNU **as** is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

as is primarily intended to assemble the output of the GNU C compiler gcc for use by the linker 1d. Nevertheless, we've tried to make as assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see Chapter 9 [Machine Dependencies], page 95). This doesn't mean as always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, **as** is designed to assemble a source program in one pass of the source file. This has a subtle impact on the .org directive (see Section 7.71 [.org], page 73).

## **1.3 Object File Formats**

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See Section 5.5 [Symbol Attributes], page 45.

## 1.4 Command Line

After the program name **as**, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

-- (two hyphens) by itself names the standard input file explicitly, as one of the files for as to assemble.

Except for '--' any command-line argument that begins with a hyphen ('-') is an option. Each option changes the behavior of as. No option changes the way another option works. An option is a '-' followed by one or more letters; the case of the letter is important. All options are optional. Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

# 1.5 Input Files

We use the phrase *source program*, abbreviated *source*, to describe the program input to one run of **as**. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run **as** it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give **as** a command line that has zero or more input file names. The input files are read (from left file name to right). A command-line argument (in any position) that has no special meaning is taken to be an input file name.

If you give as no file names it attempts to read one input file from the as standard input, which is normally your terminal. You may have to type ctl-D to tell as there is no more program to assemble.

Use '--' if you need to explicitly name the standard input file in your command line.

If the source is empty, as produces a small, empty object file.

## Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See Section 1.7 [Error and Warning Messages], page 24.

Physical files are those files named in the command line given to as.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when as source is itself synthesized from other files. as understands the '#' directives emitted by the gcc preprocessor. See also Section 7.37 [.file], page 62.

# 1.6 Output (Object) File

Every time you run as it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is a.out. You can give it another name by using the -o option. Conventionally, object file names end with .o. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the a.out format.)

The object file is meant for input to the linker ld. It contains assembled program code, information to help ld integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

## 1.7 Error and Warning Messages

**as** may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs **as** automatically. Warnings report an assumption made so that **as** could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

file\_name:NNN:Warning Message Text

(where **NNN** is a line number). If both a logical file name (see Section 7.37 [.file], page 62) and a logical line number (see Section 7.55 [.line], page 67) have been given then they will be used, otherwise the file name and line number in the current assembler source file will be used. The message text is intended to be self explanatory (in the grand Unix tradition).

Note the file name must be set via the logical version of the .file directive, not the DWARF2 version of the .file directive. For example:

```
.file 2 "bar.c"
    error_assembler_source
.file "foo.c"
.line 30
    error_c_source
```

produces this output:

Assembler messages: asm.s:2: Error: no such instruction: 'error\_assembler\_source' foo.c:31: Error: no such instruction: 'error\_c\_source'

Error messages have the format

file\_name:NNN:FATAL:Error Message Text

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

# 2 Command-Line Options

This chapter describes command-line options available in *all* versions of the GNU assembler; see Chapter 9 [Machine Dependencies], page 95, for options specific to particular machine architectures.

If you are invoking **as** via the GNU C compiler, you can use the '-Wa' option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the '-Wa') by commas. For example:

gcc -c -g -O -Wa,-alh,-L file.c

This passes two options to the assembler: '-alh' (emit a listing to standard output with high-level and assembly source) and '-L' (retain local symbols in the symbol table).

Usually you do not need to use this '-Wa' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the '-v' option to see precisely what options it passes to each compilation pass, including the assembler.)

# 2.1 Enable Listings: -a[cdghlns]

These options enable listing output from the assembler. By itself, '-a' requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: '-ah' requests a high-level language listing, '-al' requests an output-program assembly listing, and '-as' requests a symbol table listing. High-level listings require that a compiler debugging option like '-g' be used, and that assembly listings ('-al') be requested also.

Use the '-ag' option to print a first section with general assembly information, like as version, switches passed, or time stamp.

Use the '-ac' option to omit false conditionals from a listing. Any lines which are not assembled because of a false .if (or .ifdef, or any other conditional), or a true .if followed by an .else, will be omitted from the listing.

Use the '-ad' option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives .list, .nolist, .psize, .eject, .title, and .sbttl. The '-an' option turns off all forms processing. If you do not request listing output with one of the '-a' options, the listing-control directives have no effect.

The letters after '-a' may be combined into one option, e.g., '-aln'.

Note if the assembler source is coming from the standard input (e.g., because it is being created by gcc and the '-pipe' command-line switch is being used) then the listing will not contain any comments or preprocessor directives. This is because the listing code buffers input source lines from stdin only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient.

## 2.2 --alternate

Begin in alternate macro mode, see Section 7.4 [.altmacro], page 52.

# 2.3 -D

This option enables debugging, if it is supported by the assembler's configuration. Otherwise it does nothing as is ignored. This allows scripts designed to work with other assemblers to also work with GAS. as.

# 2.4 Work Faster: -f

'-f' should only be used when assembling programs written by a (trusted) compiler. '-f' stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See Section 3.1 [Preprocessing], page 31.

*Warning:* if you use '-f' when the files actually need to be preprocessed (if they contain comments, for example), **as** does not work correctly.

# 2.5 .include Search Path: -I path

Use this option to add a *path* to the list of directories **as** searches for files specified in .include directives (see Section 7.48 [.include], page 65). You may use -I as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, **as** searches any '-I' directories in the same order as they were specified (left to right) on the command line.

# 2.6 Difference Tables: -K

as sometimes alters the code emitted for directives of the form '.word sym1-sym2'. See Section 7.111 [.word], page 88. You can use the '-K' option if you want a warning issued when this is done.

# 2.7 Include Local Symbols: -L

Symbols beginning with system-specific local label prefixes, typically '.L' for ELF systems or 'L' for traditional a.out systems, are called *local symbols*. See Section 5.3 [Symbol Names], page 43. Normally you do not see such symbols when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both **as** and **ld** discard such symbols, so you do not normally debug with them.

This option tells **as** to retain those local symbols in the object file. Usually if you do this you also tell the linker 1d to preserve those symbols.

# 2.8 Configuring listing output: --listing

The listing feature of the assembler can be enabled via the command-line switch '-a' (see Section 2.1 [a], page 25). This feature combines the input source file(s) with a hex dump of the corresponding locations in the output object file, and displays them as a listing file. The format of this listing can be controlled by directives inside the assembler source (i.e., .list (see Section 7.57 [List], page 68), .title (see Section 7.100 [Title], page 86), .sbttl (see Section 7.83 [Sbttl], page 77), .psize (see Section 7.77 [Psize], page 76), and .eject (see Section 7.22 [Eject], page 60) and also by the following switches:

```
--listing-lhs-width='number'
```

Sets the maximum width, in words, of the first line of the hex byte dump. This dump appears on the left hand side of the listing output.

```
--listing-lhs-width2='number'
```

Sets the maximum width, in words, of any further lines of the hex byte dump for a given input source line. If this value is not specified, it defaults to being the same as the value specified for '--listing-lhs-width'. If neither switch is used the default is to one.

```
--listing-rhs-width='number'
```

Sets the maximum width, in characters, of the source line that is displayed alongside the hex dump. The default value for this parameter is 100. The source line is displayed on the right hand side of the listing output.

### --listing-cont-lines='number'

Sets the maximum number of continuation lines of hex dump that will be displayed for a given single line of source input. The default value is 4.

## 2.9 Assemble in MRI Compatibility Mode: -M

The -M or --mri option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of as to make it compatible with the ASM68K assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using as.

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. Supporting these would require enhancing each object file format individually. These are:

• global symbols in common section

The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. **as** handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.

• complex relocations

The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.

• END pseudo-op specifying start address

The MRI END pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the -e option to the linker, or in a linker script.

• IDNT, .ident and NAME pseudo-ops

The MRI IDNT, .ident and NAME pseudo-ops assign a module name to the output file. This is not supported by other object file formats.

• ORG pseudo-op

The m68k MRI ORG pseudo-op begins an absolute section at a given address. This differs from the usual **as**.org pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by **as**, typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

• EBCDIC strings

EBCDIC strings are not supported.

- packed binary coded decimal Packed binary coded decimal is not supported. This means that the DC.P and DCB.P pseudo-ops are not supported.
- FEQU pseudo-op

The m68k FEQU pseudo-op is not supported.

• NOOBJ pseudo-op

The m68k NOOBJ pseudo-op is not supported.

• OPT branch control options

The m68k OPT branch control options—B, BRS, BRB, BRL, and BRW—are ignored. as automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.

- OPT list control options
   The following m68k OPT list control options are ignored: C, CEX, CL, CRE, E, G, I, M, MEX, MC, MD, X.
- other OPT options

The following m68k OPT options are ignored: NEST, O, OLD, OP, P, PCO, PCR, PCS, R.

• OPT D option is default

The m68k  $\tt OPT \ \tt D$  option is the default, unlike the MRI as sembler.  $\tt OPT \ \tt NOD$  may be used to turn it off.

• XREF pseudo-op.

The m68k XREF pseudo-op is ignored.

# 2.10 Dependency Tracking: --MD

as can generate a dependency file for the file it creates. This file consists of a single rule suitable for make describing the dependencies of the main source file.

The rule is written to the file named in its argument.

This feature is used in the automatic updating of makefiles.

# 2.11 Output Section Padding

Normally the assembler will pad the end of each output section up to its alignment boundary. But this can waste space, which can be significant on memory constrained targets. So the --no-pad-sections option will disable this behaviour.

## 2.12 Name the Object File: -o

There is always one object file output when you run **as**. By default it has the name **a.out**. You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, as overwrites any existing file of the same name.

## 2.13 Join Data and Text Sections: -R

-R tells as to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See Chapter 4 [Sections and Relocation], page 37.)

When you specify -R it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of **as**. In future, -R may work this way.

When as is configured for COFF or ELF output, this option is only useful if you use sections named '.text' and '.data'.

-R is not supported for any of the HPPA targets. Using -R generates a warning from as.

## 2.14 Display Assembly Statistics: --statistics

Use '--statistics' to display two statistics about the resources used by as: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

## 2.15 Compatible Output: --traditional-format

For some targets, the output of **as** is different in some ways from the output of some existing assembler. This switch requests **as** to use the traditional format instead.

For example, it disables the exception frame optimizations which as normally does by default on gcc output.

## 2.16 Announce Version: -v

You can find out what version of as is running by including the option '-v' (which you can also spell as '-version') on the command line.

# 2.17 Control Warnings: -W, --warn, --no-warn, --fatalwarnings

**as** should never give a warning or error message when assembling compiler output. But programs written by people often cause **as** to give a warning that a particular assumption was made. All such warnings are directed to the standard error file.

If you use the -W and --no-warn options, no warnings are issued. This only affects the warning messages: it does not change any particular of how as assembles your file. Errors, which stop the assembly, are still reported.

If you use the **--fatal-warnings** option, **as** considers files that generate warnings to be in error.

You can switch these options off again by specifying --warn, which causes warnings to be output as usual.

# 2.18 Generate Object File in Spite of Errors: -Z

After an error message, as normally produces no output. If for some reason you are interested in object file output even after as gives an error message on your program, use the '-Z' option. If there are any errors, as continues anyways, and writes an object file after a final warning message of the form 'n errors, m warnings, generating bad object file.'

# 3 Syntax

This chapter describes the machine-independent syntax allowed in a source file. **as** syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that **as** does not assemble Vax bit-fields.

# 3.1 Preprocessing

The **as** internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the .include directive (see Section 7.48 [.include], page 65). You can use the GNU C compiler driver to get other "CPP" style preprocessing by giving the input file a '.S' suffix. See the 'Options Controlling the Kind of Output' section of the GCC manual for more details (https:// gcc.gnu.org/onlinedocs/gcc/Overall-Options.html#Overall-Options)

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is #NO\_APP or if you use the '-f' option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the file by putting a line that says #APP before the text that may contain whitespace or comments, and putting a line that says #NO\_APP after this text. This feature is mainly intended to support asm statements in compilers whose output is otherwise free of comments and whitespace.

# 3.2 Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see Section 3.6.1 [Character Constants], page 33), any whitespace means the same as exactly one space.

# 3.3 Comments

There are two ways of rendering comments to **as**. In both cases the comment is equivalent to one space.

Anything from '/\*' through the next '\*/' is a comment. This means you may not nest these comments.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/
```

#### /\* This sort of comment does not nest. \*/

Anything from a *line comment* character up to the next newline is considered a comment and is ignored. The line comment character is target specific, and some targets support multiple comment characters. Some targets also have line comment characters that only work if they are the first character on a line. Some targets use a sequence of two characters to introduce a line comment. Some targets can also change their line comment characters depending upon command-line options that have been used. For more details see the *Syntax* section in the documentation for individual targets.

If the line comment character is the hash sign ('#') then it still has the special ability to enable and disable preprocessing (see Section 3.1 [Preprocessing], page 31) and to specify logical line numbers:

To be compatible with past assemblers, lines that begin with '#' have a special interpretation. Following the '#' should be an absolute expression (see Chapter 6 [Expressions], page 47): the logical line number of the *next* line. Then a string (see Section 3.6.1.1 [Strings], page 33) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
# This is an ordinary comment.
# 42-6 "new_file_name" # New logical file name
# This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of as.

## 3.4 Symbols

A symbol is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters '\_.\$'. On most machines, you can also use \$ in symbol names; exceptions are noted in Chapter 9 [Machine Dependencies], page 95. No symbol may begin with a digit. Case is significant. There is no length limit; all characters are significant. Multibyte characters are supported, but note that the setting of the --multibyte-handling option might prevent their use. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See Chapter 5 [Symbols], page 43.

Symbol names may also be enclosed in double quote " characters. In such cases any characters are allowed, except for the NUL character. If a double quote character is to be included in the symbol name it must be preceded by a backslash \ character.

## 3.5 Statements

A statement ends at a newline character ('n') or a line separator character. The line separator character is target specific and described in the Syntax section of each target's documentation. Not all targets support a line separator character. The newline or line separator character is considered to be part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline. An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot '.' then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction. Different versions of **as** for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (:). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See Section 5.1 [Labels], page 43.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero. This also implies that only one label may be defined on each line.

label: .directive	followed by something	
another_label:	# This is an empty statement.	
instruction	operand_1, operand_2,	

# 3.6 Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

## 3.6.1 Character Constants

There are two kinds of character constants. A *character* stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

### **3.6.1.1** Strings

A string is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to escape these characters: precede them with a backslash '\' character. For example '\\' represents one backslash: the first  $\$  is an escape which tells **as** to interpret the second character literally as a backslash (which prevents **as** from recognizing the second  $\$  as an escape character). The complete list of escapes follows.

\b Mnemonic for backspace; for ASCII this is octal code 010.

backslash-f

Mnemonic for FormFeed; for ASCII this is octal code 014.

- n Mnemonic for newline; for ASCII this is octal code 012.
- \r Mnemonic for carriage-Return; for ASCII this is octal code 015.

\t Mnemonic for horizontal Tab; for ASCII this is octal code 011.

### \ digit digit digit

An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, 008 has the value 010, and 009 the value 011.

### \x hex-digits...

A hex character code. All trailing hex digits are combined. Either upper or lower case  ${\tt x}$  works.

 $\$  Represents one '\' character.

\" Represents one '"' character. Needed in strings to represent this character, because an unescaped '"' would end the string.

### \ anything-else

Any other character when escaped by  $\$  gives a warning, but assembles as if the  $\$  was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However **as** has no other interpretation, so **as** knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

### 3.6.1.2 Characters

A single character may be written as a single quote immediately followed by that character. Some backslash escapes apply to characters, b, f, n, r, t, and " with the same meaning as for strings, plus  $\prime$  for a single quote. So if you want to write the character backslash, you must write ' $\$  where the first  $\$  escapes the second  $\.$ . As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. **as** assumes your character code is ASCII: 'A means 65, 'B means 66, and so on.

### 3.6.2 Number Constants

as distinguishes three kinds of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an int in the C language. *Bignums* are integers, but they are stored in more than 32 bits. *Flonums* are floating point numbers, described below.

### 3.6.2.1 Integers

A binary integer is '0b' or '0B' followed by zero or more of the binary digits '01'.

An octal integer is '0' followed by zero or more of the octal digits ('01234567').

A decimal integer starts with a non-zero digit followed by zero or more digits ('0123456789').

A hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits chosen from '0123456789abcdefABCDEF'.

Integers have the usual values. To denote a negative integer, use the prefix operator '-' discussed under expressions (see Section 6.2.3 [Prefix Operators], page 47).

## 3.6.2.2 Bignums

A bignum has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

## 3.6.2.3 Flonums

A flonum represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by **as** to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of **as** specialized to that computer.

A florum is written by writing (in order)

- The digit '0'. ('0' is optional on the HPPA.)
- A letter, to tell **as** the rest of the number is a flonum. *e* is recommended. Case is not important.

On the H8/300 and Renesas / SuperH SH architectures, the letter must be one of the letters 'DFPRSX' (in upper or lower case).

On the ARC, the letter must be one of the letters 'DFRS' (in upper or lower case).

On the HPPA architecture, the letter must be 'E' (upper case only).

- An optional sign: either '+' or '-'.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: '.' followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An 'E' or 'e'.
  - Optional sign: either '+' or '-'.
  - One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

**as** does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running **as**.

# 4 Sections and Relocation

## 4.1 Background

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker 1d reads many object files (partial programs) and combines their contents to form a runnable program. When **as** emits an object file, the partial program is assumed to start at address 0. 1d assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how **as** uses sections.

1d moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300, and for the Renessa / SuperH SH, as pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by **as** has at least three sections, any of which may be empty. These are named *text*, *data* and *bss* sections.

When it generates COFF or ELF output, as can also generate whatever other named sections you specify using the '.section' directive (see Section 7.85 [.section], page 77). If you do not use any directives that place output in the '.text' or '.data' sections, these sections still exist, but are empty.

When as generates SOM or ELF output for the HPPA, as can also generate whatever other named sections you specify using the '.space' and '.subspace' directives. See HP9000 Series 800 Assembly Language Reference Manual (HP 92432-90001) for details on the '.space' and '.subspace' assembler directives.

Additionally, as uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the '\$CODE\$' section, data into '\$DATA\$', and BSS into '\$BSS\$'.

Within the object file, the text section starts at address 0, the data section follows, and the bss section follows the data section.

When generating either SOM or ELF output files on the HPPA, the text section starts at address 0, the data section at address 0x4000000, and the bss section follows the data section.

To let 1d know which data changes when the sections are relocated, and how to change that data, as also writes to the object file details of the relocation needed. To perform relocation 1d must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of (address) (start-address of section)?

• Is the reference to an address "Program-Counter relative"?

In fact, every address as ever uses is expressed as

(section) + (offset into section)

Further, most expressions **as** computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

In this manual we use the notation  $\{secname N\}$  to mean "offset N into section secname."

Apart from text, data and bss sections you need to know about the *absolute* section. When 1d mixes partial programs, addresses in the absolute section remain unchanged. For example, address {absolute 0} is "relocated" to run-time address 0 by 1d. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address {absolute 239} in one part of a program is always the same address when the program is running as address {absolute 239} in any other part of the program.

The idea of sections is extended to the undefined section. Any address whose section is unknown at assembly time is by definition rendered {undefined U}—where U is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section undefined.

By analogy the word *section* is used to describe groups of sections in the linked program. Id puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by 1d; others are invented for use of as and have no meaning except during assembly.

## 4.2 Linker Sections

1d deals with just four kinds of sections, summarized below.

### named sections text section data section

These sections hold your program. as and 1d treat them as separate but equal sections. Anything you can say of one section is true of another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable: for example, C variables would be stored in the data section.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

### absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that 1d must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

### undefined section

This "section" is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The example uses the traditional section names '.text' and '.data'. Memory addresses are on the horizontal axis.

Partial program #1: text data bss ttttt dddd 00 Partial program #2: text data bss TTT DDDD 000 linked program: text data bss TTT ttttt dddd DDDD 00000

addresses:

0...

# 4.3 Assembler Internal Sections

These sections are meant only for the internal use of **as**. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in **as** warning messages, so it might be helpful to have an idea of their meanings to **as**. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

### ASSEMBLER-INTERNAL-LOGIC-ERROR!

An internal assembler logic error has been found. This means there is a bug in the assembler.

### expr section

The assembler stores complex expressions internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the expr section.

## 4.4 Sub-Sections

Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. **as** allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file

together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a '.text 0' before each section of code being output, and a '.text 1' before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Subsections may be padded a different amount on different flavors of as.)

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; 1d and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a '.text expression' or a '.data expression' statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: '.section name, expression'. When generating ELF output, you can also use the .subsection directive (see Section 7.96 [SubSection], page 84) to specify a subsection: '.subsection expression'. Expression should be an absolute expression (see Chapter 6 [Expressions], page 47). If you just say '.text' then '.text 0' is assumed. Likewise '.data' means '.data 0'. Assembly begins in text 0. For instance:

```
.text 0  # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to **as** there is no concept of a subsection location counter. There is no way to directly manipulate a location counter—but the .align directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

### 4.5 bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

The .lcomm pseudo-op defines a symbol in the bss section; see Section 7.53 [.lcomm], page 67.

The .comm pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see Section 7.13 [.comm], page 58.

When assembling for a target which supports multiple sections, such as ELF or COFF, you may switch into the .bss section and define symbols as usual; see Section 7.85 [.section], page 77. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and .skip directives (see Section 7.90 [.skip], page 82).

# 5 Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

*Warning:* as does not place symbols in the object file in the same order they were declared. This may break some debuggers.

## 5.1 Labels

A *label* is written as a symbol immediately followed by a colon ':'. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of **as** also provides a special directive .label for defining labels more flexibly.

# 5.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '=', followed by an expression (see Chapter 6 [Expressions], page 47). This is equivalent to using the .set directive. See Section 7.86 [.set], page 81. In the same way, using a double equals sign '=''=' here represents an equivalent of the .eqv directive. See Section 7.31 [.eqv], page 61.

Blackfin does not support symbol assignment with '='.

# 5.3 Symbol Names

Symbol names begin with a letter or with one of '.\_'. On most machines, you can also use in symbol names; exceptions are noted in Chapter 9 [Machine Dependencies], page 95. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted for a particular target machine), and underscores. These restrictions do not apply when quoting symbol names by '"', which is permitted for most targets. Escaping characters in quoted symbol names with '\' generally extends only to '\' itself and '"', at the time of writing.

Case of letters is significant: foo is a different symbol name than Foo.

Symbol names do not start with a digit. An exception to this rule is made for Local Labels. See below.

Multibyte characters are supported, but note that the setting of the multibyte-handling option might prevent their use. To generate a symbol name containing multibyte characters enclose it within double quotes and use escape codes. cf See Section 3.6.1.1 [Strings], page 33. Generating a multibyte symbol name from a label is not currently supported.

Since multibyte symbol names are unusual, and could possibly be used maliciously, as provides a command line option (--multibyte-handling=warn-sym-only) which can be used to generate a warning message whenever a symbol name containing multibyte characters is defined.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

### Local Symbol Names

A local symbol is any symbol beginning with certain local label prefixes. By default, the local label prefix is '.L' for ELF systems or 'L' for traditional a.out systems, but each target may have its own set of local label prefixes. On the HPPA local symbols begin with 'L\$'.

Local symbols are defined and used within the assembler, but they are normally not saved in object files. Thus, they are not visible when debugging. You may use the '-L' option (see Section 2.7 [Include Local Symbols], page 26) to retain the local symbols in the object files.

### Local Labels

Local labels are different from local symbols. Local labels help compilers and programmers use names temporarily. They create symbols which are guaranteed to be unique over the entire scope of the input source code and which can be referred to by a simple notation. To define a local label, write a label of the form 'N:' (where N represents any non-negative integer). To refer to the most recent previous definition of that label write 'Nb', using the same number as when you defined the label. To refer to the next definition of a local label, write 'Nf'. The 'b' stands for "backwards" and the 'f' stands for "forwards".

There is no restriction on how you can use these labels, and you can reuse them too. So that it is possible to repeatedly define the same local label (using the same number 'N'), although you can only refer to the most recently defined local label of that number (for a backwards reference) or the next definition of a specific local label for a forward reference. It is also worth noting that the first 10 local labels ('0:'...'9:') are implemented in a slightly more efficient manner than the others.

Here is an example:

1:	branch	1f
2:	branch	1b
1:	branch	2f
2:	branch	1b

Which is the equivalent of:

label_1:	branch	label_3
label_2:	branch	label_1
label_3:	branch	label_4
label_4:	branch	label_3

Local label names are only a notational device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names are stored in the symbol table, appear in error messages, and are optionally emitted to the object file. The names are constructed using these parts:

### local label prefix

All local symbols begin with the system-specific local label prefix. Normally both **as** and **ld** forget symbols that start with the local label prefix. These labels are used for symbols you are never intended to see. If you use the '-L' option then **as** retains these symbols in the object file. If you also instruct **ld** to retain these symbols, you may use them in debugging.

- number This is the number that was used in the local label definition. So if the label is written '55:' then the number is '55'.
- *C-B* This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value of '\002' (control-B).

### ordinal number

This is a serial number to keep the labels distinct. The first definition of '0:' gets the number '1'. The 15th definition of '0:' gets the number '15', and so on. Likewise the first definition of '1:' gets the number '1' and its 15th definition gets '15' as well.

So for example, the first 1: may be named .L1C-B1, and the 44th 3: may be named .L3C-B44.

### **Dollar Local Labels**

On some targets **as** also supports an even more local form of local labels called dollar labels. These labels go out of scope (i.e., they become undefined) as soon as a non-local label is defined. Thus they remain valid for only a small region of the input source code. Normal local labels, by contrast, remain in scope for the entire file, or until they are redefined by another occurrence of the same local label.

Dollar labels are defined in exactly the same way as ordinary local labels, except that they have a dollar sign suffix to their numeric value, e.g., '55\$:'.

They can also be distinguished from ordinary local labels by their transformed names which use ASCII character '\001' (control-A) as the magic character to distinguish them from ordinary labels. For example, the fifth definition of '6\$' may be named '.L6C-A5'.

## 5.4 The Special Dot Symbol

The special symbol '.' refers to the current address that **as** is assembling into. Thus, the expression 'melvin: .long .' defines melvin to contain its own address. Assigning a value to . is treated the same as a .org directive. Thus, the expression '.=.+4' is the same as saying '.space 4'.

## 5.5 Symbol Attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, **as** assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

### 5.5.1 Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as 1d changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and 1d tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a .comm common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

# 5.5.2 Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

## 5.5.3 Symbol Attributes: a.out

## 5.5.3.1 Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a .desc statement (see Section 7.19 [.desc], page 60). A descriptor value means nothing to as.

## 5.5.3.2 Other

This is an arbitrary 8-bit value. It means nothing to as.

## 5.5.4 Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between .def and .endef directives.

## 5.5.4.1 Primary Attributes

The symbol name is set with .def; the value and type, respectively, with .val and .type.

## 5.5.4.2 Auxiliary Attributes

The as directives .dim, .line, .scl, .size, .tag, and .weak can generate auxiliary symbol table information for COFF.

## 5.5.5 Symbol Attributes for SOM

The SOM format for the HPPA supports a multitude of symbol attributes set with the .EXPORT and .IMPORT directives.

The attributes are described in *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) under the IMPORT and EXPORT assembler directive documentation.

# 6 Expressions

An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when **as** sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression—but the second pass is currently not implemented. **as** aborts with an error message in this situation.

## 6.1 Empty Expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and **as** assumes a value of (absolute) 0. This is compatible with other assemblers.

# 6.2 Integer Expressions

An integer expression is one or more arguments delimited by operators.

## 6.2.1 Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called "arithmetic operands". In this manual, to avoid confusing them with the "instruction operands" of the machine language, we use the term "argument" to refer to parts of expressions only, reserving the word "operand" to refer only to machine instruction operands.

Symbols are evaluated to yield {section NNN} where section is one of text, data, bss, absolute, or undefined. NNN is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and **as** pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis '(' followed by an integer expression, followed by a right parenthesis ')'; or a prefix operator followed by an argument.

### 6.2.2 Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

## 6.2.3 Prefix Operator

as has the following prefix operators. They each take one argument, which must be absolute.

- Negation. Two's complement negation.
- ~ Complementation. Bitwise not.

# 6.2.4 Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

*	Multiplication.	
*	munipilcation.	

- / Division. Truncation is the same as the C operator '/'
- % Remainder.
- << Shift Left. Same as the C operator '<<'.
- >> Shift Right. Same as the C operator '>>'.
- 2. Intermediate precedence
  - Т

Bitwise Inclusive Or.

- & Bitwise And.
- *<sup>^</sup>* Bitwise Exclusive Or.
- ! Bitwise Or Not.
- 3. Low Precedence
  - + Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.
  - Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.
  - == Is Equal To
  - <>
  - != Is Not Equal To
  - < Is Less Than
  - > Is Greater Than
  - >= Is Greater Than Or Equal To
  - <= Is Less Than Or Equal To

The comparison operators can be used as infix operators. A true result has a value of -1 whereas a false result has a value of 0. Note, these operators perform signed comparisons.

- 4. Lowest Precedence
  - && Logical And.

Logical Or.
 These two logical operations can be used to combine the results of sub expressions. Note, unlike the comparison operators a true result returns a value of 1 but a false result does still return 0. Also note that the logical or operator has a slightly lower precedence than logical and.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

# 7 Assembler Directives

All assembler directives have names that begin with a period ('.'). The names are case insensitive for most targets, and usually written in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 9 [Machine Dependencies], page 95.

## 7.1 .abort

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells **as** to quit also. One day .abort will not be supported.

# 7.2 . ABORT (COFF)

When producing COFF output, as accepts this directive as a synonym for '.abort'.

# 7.3 .align [abs-expr[, abs-expr[, abs-expr]]]

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below. If this expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the arc, hppa, i386 using ELF, iq2000, m68k, or1k, s390, sparc, tic4x and xtensa, the first expression is the alignment request in bytes. For example '.align 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. For the tic54x, the first expression is the alignment request in words.

For other systems, including ppc, i386 using a out format, arm and strongarm, it is the number of low-order zero bits the location counter must have after advancement. For example '.align 3' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides .balign and .p2align directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

## 7.4 .altmacro

Enable alternate macro mode, enabling:

### LOCAL name [ , ... ]

One additional directive, LOCAL, is available. It is used to generate a string replacement for each of the *name* arguments, and replace any instances of *name* in each macro expansion. The replacement string is unique in the assembly, and different for each separate macro expansion. LOCAL allows you to write macros that define symbols, without fear of conflict between separate macro expansions.

### String delimiters

You can write strings delimited in these other ways besides "string":

'string' You can delimit strings with single-quote characters.

*<string>* You can delimit strings with matching angle brackets.

### single-character string escape

To include any single character literally in a string (even if the character would otherwise have some special meaning), you can prefix the character with '!' (an exclamation mark). For example, you can write '<4.3 !> 5.4!!>' to get the literal text '4.3 > 5.4!'.

### Expression results as strings

You can write '*kexpr*' to evaluate the expression *expr* and use the result as a string.

## 7.5 .ascii "string"...

.ascii expects zero or more string literals (see Section 3.6.1.1 [Strings], page 33) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

## 7.6 .asciz "string"...

.asciz is just like .ascii, but each string is followed by a zero byte. The "z" in '.asciz' stands for "zero". Note that multiple string arguments not separated by commas will be concatenated together and only one final zero byte will be stored.

## 7.7 .attach\_to\_group name

Attaches the current section to the named group. This is like declaring the section with the G attribute, but can be done after the section has been created. Note if the group section does not exist at the point that this directive is used then it will be created.

# 7.8 .balign[wl] [abs-expr[, abs-expr[, abs-expr]]]

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example '.balign 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. If the expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The .balignw and .balignl directives are variants of the .balign directive. The .balignw directive treats the fill pattern as a two byte word value. The .balignl directives treats the fill pattern as a four byte longword value. For example, .balignw 4,0x368d will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

### 7.9 .bss subsection

.bss tells as to assemble the following statements onto the end of the bss section. For ELF based targets an optional *subsection* expression (which must evaluate to a positive integer) can be provided. In this case the statements are appended to the end of the indicated bss subsection.

## 7.10 Bundle directives

### 7.10.1 .bundle\_align\_mode abs-expr

.bundle\_align\_mode enables or disables aligned instruction bundle mode. In this mode, sequences of adjacent instructions are grouped into fixed-sized bundles. If the argument is zero, this mode is disabled (which is the default state). If the argument it not zero, it gives the size of an instruction bundle as a power of two (as for the .p2align directive, see Section 7.72 [P2align], page 74).

For some targets, it's an ABI requirement that no instruction may span a certain aligned boundary. A *bundle* is simply a sequence of instructions that starts on an aligned boundary. For example, if *abs-expr* is 5 then the bundle size is 32, so each aligned chunk of 32 bytes is a bundle. When aligned instruction bundle mode is in effect, no single instruction may span a boundary between bundles. If an instruction would start too close to the end of a bundle for the length of that particular instruction to fit within the bundle, then the space at the end of that bundle is filled with no-op instructions so the instruction starts in the next bundle. As a corollary, it's an error if any single instruction's encoding is longer than the bundle size.

### 7.10.2 .bundle\_lock and .bundle\_unlock

The .bundle\_lock and directive .bundle\_unlock directives allow explicit control over instruction bundle padding. These directives are only valid when .bundle\_align\_mode has been used to enable aligned instruction bundle mode. It's an error if they appear when .bundle\_align\_mode has not been used at all, or when the last directive was .bundle\_align\_mode 0.

For some targets, it's an ABI requirement that certain instructions may appear only as part of specified permissible sequences of multiple instructions, all within the same bundle. A pair of .bundle\_lock and .bundle\_unlock directives define a *bundle-locked* instruction sequence. For purposes of aligned instruction bundle mode, a sequence starting with .bundle\_lock and ending with .bundle\_unlock is treated as a single instruction. That is, the entire sequence must fit into a single bundle and may not span a bundle boundary. If necessary, no-op instructions will be inserted before the first instruction of the sequence so that the whole sequence starts on an aligned bundle boundary. It's an error if the sequence is longer than the bundle size.

For convenience when using .bundle\_lock and .bundle\_unlock inside assembler macros (see Section 7.63 [Macro], page 69), bundle-locked sequences may be nested. That is, a second .bundle\_lock directive before the next .bundle\_unlock directive has no effect except that it must be matched by another closing .bundle\_unlock so that there is the same number of .bundle\_lock and .bundle\_unlock directives.

### 7.11 .byte expressions

.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

Note - this directive is not intended for encoding instructions, and it will not trigger effects like DWARF line number generation. Instead some targets support special directives for encoding arbitrary binary sequences as instructions such as .insn or .inst.

## 7.12 CFI directives

### 7.12.1 .cfi\_sections section\_list

.cfi\_sections may be used to specify whether CFI directives should emit .eh\_frame section, .debug\_frame section and/or .sframe section. If section\_list contains .eh\_frame, .eh\_frame is emitted, if section\_list contains .debug\_frame, .debug\_frame is emitted, and finally, if section\_list contains .sframe, .sframe is emitted. To emit multiple sections, specify them together in a list. For example, to emit both .eh\_frame and .debug\_frame, use .eh\_frame, .debug\_frame. The default if this directive is not used is .cfi\_sections .eh\_frame.

On targets that support compact unwinding tables these can be generated by specifying .eh\_frame\_entry instead of .eh\_frame.

Some targets may support an additional name, such as .c6xabi.exidx which is used by the target.

The .cfi\_sections directive can be repeated, with the same or different arguments, provided that CFI generation has not yet started. Once CFI generation has started however the section list is fixed and any attempts to redefine it will result in an error.

# 7.12.2 .cfi\_startproc [simple]

.cfi\_startproc is used at the beginning of each function that should have an entry in .eh\_frame. It initializes some internal data structures. Don't forget to close the function by .cfi\_endproc.

Unless .cfi\_startproc is used along with parameter simple it also emits some architecture dependent initial CFI instructions.

## 7.12.3 .cfi\_endproc

.cfi\_endproc is used at the end of a function where it closes its unwind entry previously opened by .cfi\_startproc, and emits it to .eh\_frame.

# 7.12.4 .cfi\_personality encoding [, exp]

.cfi\_personality defines personality routine and its encoding. encoding must be a constant determining how the personality should be encoded. If it is 255 (DW\_EH\_PE\_omit), second argument is not present, otherwise second argument should be a constant or a symbol name. When using indirect encodings, the symbol provided should be the location where personality can be loaded from, not the personality routine itself. The default after .cfi\_startproc is .cfi\_personality 0xff, no personality routine.

## 7.12.5 .cfi\_personality\_id id

cfi\_personality\_id defines a personality routine by its index as defined in a compact unwinding format. Only valid when generating compact EH frames (i.e. with .cfi\_sections eh\_frame\_entry.

## 7.12.6 .cfi\_fde\_data [opcode1 [, ...]]

cfi\_fde\_data is used to describe the compact unwind opcodes to be used for the current function. These are emitted inline in the .eh\_frame\_entry section if small enough and there is no LSDA, or in the .gnu.extab section otherwise. Only valid when generating compact EH frames (i.e. with .cfi\_sections eh\_frame\_entry.

# 7.12.7 .cfi\_lsda encoding [, exp]

.cfi\_lsda defines LSDA and its encoding. *encoding* must be a constant determining how the LSDA should be encoded. If it is 255 (DW\_EH\_PE\_omit), the second argument is not present, otherwise the second argument should be a constant or a symbol name. The default after .cfi\_startproc is .cfi\_lsda 0xff, meaning that no LSDA is present.

## 7.12.8 .cfi\_inline\_lsda [align]

.cfi\_inline\_lsda marks the start of a LSDA data section and switches to the corresponding .gnu.extab section. Must be preceded by a CFI block containing a .cfi\_lsda directive. Only valid when generating compact EH frames (i.e. with .cfi\_sections eh\_frame\_entry.

The table header and unwinding opcodes will be generated at this point, so that they are immediately followed by the LSDA data. The symbol referenced by the .cfi\_lsda directive

should still be defined in case a fallback FDE based encoding is used. The LSDA data is terminated by a section directive.

The optional *align* argument specifies the alignment required. The alignment is specified as a power of two, as with the .p2align directive.

## 7.12.9 .cfi\_def\_cfa register, offset

.cfi\_def\_cfa defines a rule for computing CFA as: take address from register and add offset to it.

## 7.12.10 .cfi\_def\_cfa\_register register

.cfi\_def\_cfa\_register modifies a rule for computing CFA. From now on *register* will be used instead of the old one. Offset remains the same.

## 7.12.11 .cfi\_def\_cfa\_offset offset

.cfi\_def\_cfa\_offset modifies a rule for computing CFA. Register remains the same, but offset is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address.

### 7.12.12 .cfi\_adjust\_cfa\_offset offset

Same as .cfi\_def\_cfa\_offset but offset is a relative value that is added/subtracted from the previous offset.

## 7.12.13 .cfi\_offset register, offset

Previous value of register is saved at offset offset from CFA.

### 7.12.14 .cfi\_val\_offset register, offset

Previous value of register is CFA + offset.

## 7.12.15 .cfi\_rel\_offset register, offset

Previous value of *register* is saved at offset offset from the current CFA register. This is transformed to .cfi\_offset using the known displacement of the CFA register from the CFA. This is often easier to use, because the number will match the code it's annotating.

## 7.12.16 .cfi\_register register1, register2

Previous value of register1 is saved in register register2.

### 7.12.17 .cfi\_restore register

.cfi\_restore says that the rule for *register* is now the same as it was at the beginning of the function, after all initial instruction added by .cfi\_startproc were executed.

## 7.12.18 .cfi\_undefined register

From now on the previous value of *register* can't be restored anymore.

### 7.12.19 .cfi\_same\_value register

Current value of register is the same like in the previous frame, i.e. no restoration needed.

## 7.12.20 .cfi\_remember\_state and .cfi\_restore\_state

.cfi\_remember\_state pushes the set of rules for every register onto an implicit stack, while .cfi\_restore\_state pops them off the stack and places them in the current row. This is useful for situations where you have multiple .cfi\_\* directives that need to be undone due to the control flow of the program. For example, we could have something like this (assuming the CFA is the value of rbp):

```
je label
popq %rbx
.cfi_restore %rbx
popq %r12
.cfi_restore %r12
popq %rbp
.cfi_restore %rbp
.cfi_def_cfa %rsp, 8
ret
label:
/* Do something else */
```

Here, we want the .cfi directives to affect only the rows corresponding to the instructions before label. This means we'd have to add multiple .cfi directives after label to recreate the original save locations of the registers, as well as setting the CFA back to the value of rbp. This would be clumsy, and result in a larger binary size. Instead, we can write:

```
je label
popq %rbx
.cfi_remember_state
.cfi_restore %rbx
popq %r12
.cfi_restore %r12
popq %rbp
.cfi_restore %rbp
.cfi_def_cfa %rsp, 8
ret
label:
.cfi_restore_state
/* Do something else */
```

That way, the rules for the instructions after label will be the same as before the first .cfi\_restore without having to use multiple .cfi directives.

## 7.12.21 .cfi\_return\_column register

Change return column *register*, i.e. the return address is either directly in *register* or can be accessed by rules for *register*.

### 7.12.22 .cfi\_signal\_frame

Mark current function as signal trampoline.

### 7.12.23 .cfi\_window\_save

SPARC register window has been saved.

## 7.12.24 .cfi\_escape expression[, ...]

Allows the user to add arbitrary bytes to the unwind info. One might use this to add OS-specific CFI opcodes, or generic CFI opcodes that GAS does not yet support.

## 7.12.25 .cfi\_val\_encoded\_addr register, encoding, label

The current value of *register* is *label*. The value of *label* will be encoded in the output file according to *encoding*; see the description of .cfi\_personality for details on this encoding.

The usefulness of equating a register to a fixed label is probably limited to the return address register. Here, it can be useful to mark a code segment that has only one return address which is reached by a direct branch and no copy of the return address exists in memory or another register.

## 7.13 .comm symbol , length

.comm declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If 1d does not see a definition for the symbol-just one or more common symbols-then it will allocate *length* bytes of uninitialized memory. *length* must be an absolute expression. If 1d sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF or (as a GNU extension) PE, the .comm directive takes an optional third argument. This is the desired alignment of the symbol, specified for ELF as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero), and for PE as a power of two (for example, an alignment of 5 means aligned to a 32-byte boundary). The alignment must be an absolute expression, and it must be a power of two. If 1d allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, as will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 16 on ELF, or the default section alignment of 4 on PE<sup>1</sup>.

The syntax for .comm differs slightly on the HPPA. The syntax is 'symbol .comm, length'; symbol is optional.

## 7.14 .data subsection

.data tells as to assemble the following statements onto the end of the data subsection numbered *subsection* (which is an absolute expression). If *subsection* is omitted, it defaults to zero.

# 7.15 .dc[size] expressions

The .dc directive expects zero or more *expressions* separated by commas. These expressions are evaluated and their values inserted into the current section. The size of the emitted value depends upon the suffix to the .dc directive:

- '.a' Emits N-bit values, where N is the size of an address on the target system.
- '.b' Emits 8-bit values.
- '.d' Emits double precision floating-point values.

<sup>&</sup>lt;sup>1</sup> This is not the same as the executable image file alignment controlled by ld's '--section-alignment' option; image file sections in PE are aligned to multiples of 4096, which is far too large an alignment for ordinary variables. It is rather the default alignment for (non-debug) sections within object ('\*.o') files, which are less strictly aligned.

'.l' E	mits 32-bit value	es.
--------	-------------------	-----

- **'.s'** Emits single precision floating-point values.
- '.w' Emits 16-bit values. Note this is true even on targets where the .word directive would emit 32-bit values.
- '.x' Emits long double precision floating-point values.

If no suffix is used then  $\cdot .w'$  is assumed.

The byte ordering is target dependent, as is the size and format of floating point values.

Note - these directives are not intended for encoding instructions, and they will not trigger effects like DWARF line number generation. Instead some targets support special directives for encoding arbitrary binary sequences as instructions such as .insn or .inst.

## 7.16 .dcb[size] number [,fill]

This directive emits *number* copies of *fill*, each of *size* bytes. Both *number* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. The *size* suffix, if present, must be one of:

- **'.b'** Emits single byte values.
- '.d' Emits double-precision floating point values.
- '.1' Emits 4-byte values.
- **'.s'** Emits single-precision floating point values.
- '.w' Emits 2-byte values.
- '.x' Emits long double-precision floating point values.

If the size suffix is omitted then '.w' is assumed.

The byte ordering is target dependent, as is the size and format of floating point values.

# 7.17 .ds[size] number [,fill]

This directive emits *number* copies of *fill*, each of *size* bytes. Both *number* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. The *size* suffix, if present, must be one of:

- **'.b'** Emits single byte values.
- '.d' Emits 8-byte values.
- '.1' Emits 4-byte values.
- '.p' Emits values with size matching packed-decimal floating-point ones.
- **'.s'** Emits 4-byte values.
- '.w' Emits 2-byte values.

'.x' Emits values with size matching long double precision floating-point ones.

Note - unlike the .dcb directive the '.d', '.s' and '.x' suffixes do not indicate that floating-point values are to be inserted.

If the size suffix is omitted then '.w' is assumed.

The byte ordering is target dependent.

# 7.18 .def name

Begin defining debugging information for a symbol *name*; the definition extends until the .endef directive is encountered.

# 7.19 .desc symbol, abs-expression

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 45) to the low 16 bits of an absolute expression.

The '.desc' directive is not available when as is configured for COFF output; it is only for a.out or b.out object format. For the sake of compatibility, as accepts it, but produces no output, when configured for COFF.

# 7.20 .dim

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def/.endef pairs.

# 7.21 .double flonums

.double expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how as is configured. See Chapter 9 [Machine Dependencies], page 95.

# 7.22 .eject

Force a page break at this point, when generating assembly listings.

# 7.23 .else

.else is part of the as support for conditional assembly; see Section 7.46 [.if], page 64. It marks the beginning of a section of code to be assembled if the condition for the preceding .if was false.

# 7.24 .elseif

.elseif is part of the as support for conditional assembly; see Section 7.46 [.if], page 64. It is shorthand for beginning a new .if block that would otherwise fill the entire .else section.

# 7.25 .end

.end marks the end of the assembly file. as does not process anything in the file past the .end directive.

# 7.26 .endef

This directive flags the end of a symbol definition begun with .def.

# 7.27 .endfunc

.endfunc marks the end of a function specified with .func.

# 7.28 .endif

.endif is part of the as support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See Section 7.46 [.if], page 64.

## 7.29 .equ symbol, expression

This directive sets the value of symbol to expression. It is synonymous with '.set'; see Section 7.86 [.set], page 81.

The syntax for equ on the HPPA is 'symbol .equ expression'.

The syntax for equ on the Z80 is 'symbol equ expression'. On the Z80 it is an error if symbol is already defined, but the symbol is not protected from later redefinition. Compare Section 7.30 [Equiv], page 61.

## 7.30 .equiv symbol, expression

The .equiv directive is like .equ and .set, except that the assembler will signal an error if *symbol* is already defined. Note a symbol which has been referenced but not actually defined is considered to be undefined.

Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

plus it protects the symbol from later redefinition.

## 7.31 .eqv symbol, expression

The .eqv directive is like .equiv, but no attempt is made to evaluate the expression or any part of it immediately. Instead each time the resulting symbol is used in an expression, a snapshot of its current value is taken.

## 7.32 .err

If as assembles a .err directive, it will print an error message and, unless the -Z option was used, it will not generate an object file. This can be used to signal an error in conditionally compiled code.

## 7.33 .error "string"

Similarly to .err, this directive emits an error, but you can specify a string that will be emitted as the error message. If you don't specify the message, it defaults to ".error directive invoked in source file". See Section 1.7 [Error and Warning Messages], page 24.

.error "This code has not been assembled and tested."

# 7.34 .exitm

Exit early from the current macro definition. See Section 7.63 [Macro], page 69.

# 7.35 .extern

.extern is accepted in the source program—for compatibility with other assemblers—but it is ignored. as treats all undefined symbols as external.

# 7.36 .fail expression

Generates an error or a warning. If the value of the *expression* is 500 or more, **as** will print a warning message. If the value is less than 500, **as** will print an error message. The message will include the value of *expression*. This can occasionally be useful inside complex nested macros or conditional assembly.

# 7.37 .file

There are two different versions of the .file directive. Targets that support DWARF2 line number information use the DWARF2 version of .file. Other targets use the default version.

# **Default Version**

This version of the .file directive tells as that we are about to start a new logical file. The syntax is:

.file string

string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes '"'; but if you wish to specify an empty file name, you must give the quotes-"". This statement may go away in future: it is only recognized to be compatible with old **as** programs.

# **DWARF2** Version

When emitting DWARF2 line number information, .file assigns filenames to the .debug\_line file name table. The syntax is:

## .file fileno filename

The *fileno* operand should be a unique positive integer to use as the index of the entry in the table. The *filename* operand is a C string literal enclosed in double quotes. The *filename* can include directory elements. If it does, then the directory will be added to the directory table and the basename will be added to the file table.

The detail of filename indices is exposed to the user because the filename table is shared with the .debug\_info section of the DWARF2 debugging information, and thus the user must know the exact indices that table entries will have.

If DWARF5 support has been enabled via the <code>-gdwarf-5</code> option then an extended version of .file is also allowed:

.file fileno [dirname] filename [md5 value]

With this version a separate directory name is allowed, although if this is used then *filename* should not contain any directory component, except for *fileno* equal to 0: in this

case, dirname is expected to be the current directory and filename the currently processed file, and the latter need not be located in the former. In addition an MD5 hash value of the contents of filename can be provided. This will be stored in the the file table as well, and can be used by tools reading the debug information to verify that the contents of the source file match the contents of the compiled file.

## 7.38 .fill repeat , size , value

repeat, size and value are absolute expressions. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer **as** is assembling for. Each size bytes in a repetition is taken from the lowest order size bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and value are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.

## 7.39 .float flonums

This directive assembles zero or more flonums, separated by commas. It has the same effect as .single. The exact kind of floating point numbers emitted depends on how as is configured. See Chapter 9 [Machine Dependencies], page 95.

## 7.40 .func name[,label]

.func emits debugging information to denote function *name*, and is ignored unless the file is assembled with debugging enabled. Only '--gstabs[+]' is currently supported. *label* is the entry point of the function and if omitted *name* prepended with the 'leading char' is used. 'leading char' is usually \_ or nothing, depending on the target. All functions are currently defined to have void return type. The function must be terminated with .endfunc.

## 7.41 .global symbol, .globl symbol

.global makes the symbol visible to ld. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings ('.globl' and '.global') are accepted, for compatibility with other assemblers.

On the HPPA, .global is not always enough to make it accessible to other partial programs. You may need the HPPA-only .EXPORT directive as well. See Section 9.15.5 [HPPA Assembler Directives], page 176.

## 7.42 .gnu\_attribute tag, value

Record a GNU object attribute for this file. See Chapter 8 [Object Attributes], page 91.

## 7.43 .hidden names

This is one of the ELF visibility directives. The other two are .internal (see Section 7.50 [.internal], page 66) and .protected (see Section 7.76 [.protected], page 75).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to hidden which means that the symbols are not visible to other components. Such symbols are always considered to be protected as well.

## 7.44 .hword expressions

This expects zero or more expressions, and emits a 16 bit number for each.

This directive is a synonym for '.short'; depending on the target architecture, it may also be a synonym for '.word'.

## 7.45 .ident

This directive is used by some assemblers to place tags in object files. The behavior of this directive varies depending on the target. When using the a.out object file format, as simply accepts the directive for source-file compatibility with existing assemblers, but does not emit anything for it. When using COFF, comments are emitted to the .comment or .rdata section, depending on the target. When using ELF, comments are emitted to the .comment section.

## 7.46 .if absolute expression

.if marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an *absolute expression*) is non-zero. The end of the conditional section of code must be marked by .endif (see Section 7.28 [.endif], page 61); optionally, you may include code for the alternative condition, flagged by .else (see Section 7.23 [.else], page 60). If you have several conditions to check, .elseif may be used to avoid nesting blocks if/else within each subsequent .else block.

The following variants of .if are also supported:

.ifdef symbol

Assembles the following section of code if the specified *symbol* has been defined. Note a symbol which has been referenced but not yet defined is considered to be undefined.

.ifb text Assembles the following section of code if the operand is blank (empty).

.ifc string1, string2

Assembles the following section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

### .ifeq absolute expression

Assembles the following section of code if the argument is zero.

### .ifeqs string1, string2

Another form of .ifc. The strings must be quoted using double quotes.

### .ifge absolute expression

Assembles the following section of code if the argument is greater than or equal to zero.

### .ifgt absolute expression

Assembles the following section of code if the argument is greater than zero.

#### .ifle absolute expression

Assembles the following section of code if the argument is less than or equal to zero.

### .iflt absolute expression

Assembles the following section of code if the argument is less than zero.

### .ifnb text

Like .ifb, but the sense of the test is reversed: this assembles the following section of code if the operand is non-blank (non-empty).

### .ifnc string1, string2.

Like .ifc, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

### .ifndef symbol

### .ifnotdef symbol

Assembles the following section of code if the specified *symbol* has not been defined. Both spelling variants are equivalent. Note a symbol which has been referenced but not yet defined is considered to be undefined.

### .ifne absolute expression

Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to .if).

### .ifnes string1, string2

Like .ifeqs, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

# 7.47 .incbin "file"[, skip[, count]]

The incbin directive includes file verbatim at the current location. You can control the search paths used with the '-I' command-line option (see Chapter 2 [Command-Line Options], page 25). Quotation marks are required around file.

The *skip* argument skips a number of bytes from the start of the *file*. The *count* argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the **incbin** directive.

## 7.48 .include "file"

This directive provides a way to include supporting files at specified points in your source program. The code from *file* is assembled as if it followed the point of the .include; when

the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the '-I' command-line option (see Chapter 2 [Command-Line Options], page 25). Quotation marks are required around *file*.

## 7.49 .int expressions

Expect zero or more *expressions*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

Note - this directive is not intended for encoding instructions, and it will not trigger effects like DWARF line number generation. Instead some targets support special directives for encoding arbitrary binary sequences as instructions such as eg .insn or .inst.

### 7.50 .internal names

This is one of the ELF visibility directives. The other two are .hidden (see Section 7.43 [.hidden], page 64) and .protected (see Section 7.76 [.protected], page 75).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to **internal** which means that the symbols are considered to be **hidden** (i.e., not visible to other components), and that some extra, processor specific processing must also be performed upon the symbols as well.

## 7.51 .irp symbol, values...

Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the .irp directive, and is terminated by an .endr directive. For each value, symbol is set to value, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use \symbol.

For example, assembling

```
.irp param,1,2,3
move d\param,sp@-
.endr
```

is equivalent to assembling

```
move d1,sp@-
move d2,sp@-
move d3,sp@-
```

For some caveats with the spelling of symbol, see also Section 7.63 [Macro], page 69.

## 7.52 .irpc symbol,values...

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the .irpc directive, and is terminated by an .endr directive. For each character in *value*, *symbol* is set to the character, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use \*symbol*.

For example, assembling

.irpc param,123 move d\param,sp@-.endr

is equivalent to assembling

move d1,sp@move d2,sp@move d3,sp@-

For some caveats with the spelling of symbol, see also the discussion at See Section 7.63 [Macro], page 69.

## 7.53 .lcomm symbol , length

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 7.41 [.global], page 63), so is normally not visible to 1d.

Some targets permit a third argument to be used with .lcomm. This argument specifies the desired alignment of the symbol in the bss section.

The syntax for .lcomm differs slightly on the HPPA. The syntax is 'symbol .lcomm, length'; symbol is optional.

## 7.54 .lflags

as accepts this directive, for compatibility with other assemblers, but ignores it.

### 7.55 .line line-number

Change the logical line number. *line-number* must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number *line-number* -1. One day **as** will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

Even though this is a directive associated with the a.out or b.out object-code formats, as still recognizes it when producing COFF output, and treats '.line' as though it were the COFF '.ln' *if* it is found outside a .def/.endef pair.

Inside a .def, '.line' is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

## 7.56 .linkonce [type]

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The .linkonce pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique. This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The *type* argument is optional. If specified, it must be one of the following strings. For example:

.linkonce same\_size

Not all types may be supported on all object file formats.

discard Silently discard duplicate sections. This is the default.

one\_only Warn if there are duplicate sections, but still keep only one copy.

same\_size

Warn if any of the duplicates have different sizes.

#### same\_contents

Warn if any of the duplicates do not have exactly the same contents.

## 7.57 .list

Control (in conjunction with the .nolist directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). .list increments the counter, and .nolist decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the '-a' command-line option; see Chapter 2 [Command-Line Options], page 25), the initial value of the listing counter is one.

## 7.58 .ln line-number

'.ln' is a synonym for '.line'.

## 7.59 .loc fileno lineno [column] [options]

When emitting DWARF2 line number information, the .loc directive will add a row to the .debug\_line line number matrix corresponding to the immediately following assembly instruction. The *fileno*, *lineno*, and optional *column* arguments will be applied to the .debug\_line state machine before the row is added. It is an error for the input assembly file to generate a non-empty .debug\_line and also use loc directives.

The options are a sequence of the following tokens in any order:

basic\_block

This option will set the **basic\_block** register in the .debug\_line state machine to true.

### prologue\_end

This option will set the prologue\_end register in the .debug\_line state machine to true.

### epilogue\_begin

This option will set the epilogue\_begin register in the .debug\_line state machine to true.

is\_stmt value

This option will set the is\_stmt register in the .debug\_line state machine to value, which must be either 0 or 1.

isa value This directive will set the isa register in the .debug\_line state machine to value, which must be an unsigned integer.

discriminator value

This directive will set the discriminator register in the .debug\_line state machine to value, which must be an unsigned integer.

view value

This option causes a row to be added to .debug\_line in reference to the current address (which might not be the same as that of the following assembly instruction), and to associate value with the view register in the .debug\_line state machine. If value is a label, both the view register and the label are set to the number of prior .loc directives at the same program location. If value is the literal 0, the view register is set to zero, and the assembler asserts that there aren't any prior .loc directives at the same program location. If value is the literal -0, the assembler arrange for the view register to be reset in this row, even if there are prior .loc directives at the same program location.

## 7.60 .loc\_mark\_labels enable

When emitting DWARF2 line number information, the .loc\_mark\_labels directive makes the assembler emit an entry to the .debug\_line line number matrix with the basic\_block register in the state machine set whenever a code label is seen. The *enable* argument should be either 1 or 0, to enable or disable this function respectively.

## 7.61 .local names

This directive, which is available for ELF targets, marks each symbol in the comma-separated list of **names** as a local symbol so that it will not be externally visible. If the symbols do not already exist, they will be created.

For targets where the .lcomm directive (see Section 7.53 [Lcomm], page 67) does not accept an alignment argument, which is the case for most ELF targets, the .local directive can be used in combination with .comm (see Section 7.13 [Comm], page 58) to define aligned local common data.

### 7.62 .long expressions

.long is the same as '.int'. See Section 7.49 [.int], page 66.

## 7.63 .macro

The commands .macro and .endm allow you to define macros that generate assembly output. For example, this definition specifies a macro sum that puts a sequence of numbers into memory:

```
.macro sum from=0, to=5
.long \from
```

```
.if \to-\from
sum "(\from+1)",\to
.endif
.endm
```

With that definition, 'SUM 0,5' is equivalent to this assembly input:

.long 0 .long 1 .long 2 .long 3 .long 4 .long 5

#### .macro macname

### .macro macname macargs ...

Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can qualify the macro argument to indicate whether all invocations must specify a non-blank value (through ':req'), or whether it takes all of the remaining arguments (through ':vararg'). You can supply a default value for any macro argument by following the name with '=deflt'. You cannot define two macros with the same *macname* unless it has been subject to the .purgem directive (see Section 7.78 [Purgem], page 76) between the two definitions. For example, these are all valid .macro statements:

.macro comm

Begin the definition of a macro called **comm**, which takes no arguments.

```
.macro plus1 p, p1
```

```
.macro plus1 p p1
```

Either statement begins the definition of a macro called plus1, which takes two arguments; within the macro definition, write '\p' or '\p1' to evaluate the arguments.

### .macro reserve\_str p1=0 p2

Begin the definition of a macro called **reserve\_str**, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as 'reserve\_str a, b' (with '\p1' evaluating to a and '\p2' evaluating to b), or as 'reserve\_str ,b' (with '\p1' evaluating as the default, in this case '0', and '\p2' evaluating to b).

### .macro m p1:req, p2=0, p3:vararg

Begin the definition of a macro called m, with at least three arguments. The first argument must always have a value specified, but not the second, which instead has a default value. The third formal will get assigned all remaining arguments specified at invocation time.

When you call a macro, you can specify the argument values either by position, or by keyword. For example, 'sum 9,17' is equivalent to 'sum to=17, from=9'.

Note that since each of the *macargs* can be an identifier exactly as any other one permitted by the target architecture, there may be occasional problems if the target hand-crafts special meanings to certain characters when they occur in a special position. For example, if the colon (:) is generally permitted to be part of a symbol name, but the architecture specific code special-cases it when occurring as the final character of a symbol (to denote a label), then the macro parameter replacement code will have no way of knowing that and consider the whole construct (including the colon) an identifier, and check only this identifier for being the subject to parameter substitution. So for example this macro definition:

```
.macro label l
\l:
.endm
```

might not work as expected. Invoking 'label foo' might not create a label called 'foo' but instead just insert the text '\l:' into the assembler source, probably generating an error about an unrecognised identifier.

Similarly problems might occur with the period character ('.') which is often allowed inside opcode names (and hence identifier names). So for example constructing a macro to build an opcode from a base name and a length specifier like this:

```
.macro opcode base length 
\base.\length
```

.endm

and invoking it as 'opcode store 1' will not create a 'store.1' instruction but instead generate some kind of error as the assembler tries to interpret the text '\base.\length'.

There are several possible ways around this problem:

### Insert white space

If it is possible to use white space characters then this is the simplest solution. eg:

```
.macro label l
\l :
.endm
```

Use '\()' The string '\()' can be used to separate the end of a macro argument from the following text. eg:

.macro opcode base length
 \base\().\length

.endm

### Use the alternate macro syntax mode

In the alternative macro syntax mode the ampersand character ('&') can be used as a separator. eg:

```
.altmacro
.macro label l
l&:
.endm
```

Note: this problem of correctly identifying string parameters to pseudo ops also applies to the identifiers used in .irp (see Section 7.51 [Irp], page 66) and .irpc (see Section 7.52 [Irpc], page 66) as well.

Another issue can occur with the actual arguments passed during macro invocation: Multiple arguments can be separated by blanks or commas. To have arguments actually contain blanks or commas (or potentially other non-alphanumeric characters), individual arguments will need to be enclosed in either parentheses (), square brackets [], or double quote " characters. The latter may be the only viable option in certain situations, as only double quotes are actually stripped while establishing arguments. It may be important to be aware of two escaping models used when processing such quoted argument strings: For one two adjacent double quotes represent a single double quote in the resulting argument, going along the lines of the stripping of the enclosing quotes. But then double quotes can also be escaped by a backslash  $\$ , but this backslash will not be retained in the resulting actual argument as then seen / used while expanding the macro.

As a consequence to the first of these escaping mechanisms two string literals intended to be representing separate macro arguments need to be separated by white space (or, better yet, by a comma). To state it differently, such adjacent string literals - even if separated only by a blank - will not be concatenated when determining macro arguments, even if they're only separated by white space. This is unlike certain other pseudo ops, e.g. .ascii.

- .endm Mark the end of a macro definition.
- .exitm Exit early from the current macro definition.
- \**@** as maintains a counter of how many macros it has executed in this pseudovariable; you can copy that number to your output with '\**@**', but only within a macro definition.
- LOCAL name [, ...] Warning: LOCAL is only available if you select "alternate macro syntax" with '--alternate' or .altmacro. See Section 7.4 [.altmacro], page 52.

## 7.64 .mri val

If val is non-zero, this tells as to enter MRI mode. If val is zero, this tells as to exit MRI mode. This change affects code assembled until the next .mri directive, or until the end of the file. See Section 2.9 [MRI mode], page 27.

## 7.65 .noaltmacro

Disable alternate macro mode. See Section 7.4 [Altmacro], page 52.

# 7.66 .nolist

Control (in conjunction with the .list directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). .list increments the counter, and .nolist decrements it. Assembly listings are generated whenever the counter is greater than zero.

# 7.67 .nop [size]

This directive emits no-op instructions. It is provided on all architectures, allowing the creation of architecture neutral tests involving actual code. The size of the generated instruction is target specific, but if the optional *size* argument is given and resolves to an absolute positive value at that point in assembly (no forward expressions allowed) then the fewest no-op instructions are emitted that equal or exceed a total *size* in bytes. .nop does affect the generation of DWARF debug line information. Some targets do not support using .nop with *size*.

# 7.68 .nops size[, control]

This directive emits no-op instructions. It is specific to the Intel 80386 and AMD x86-64 targets. It takes a size argument and generates size bytes of no-op instructions. size must be absolute and positive. These bytes do not affect the generation of DWARF debug line information.

The optional *control* argument specifies a size limit for a single no-op instruction. If not provided then a value of 0 is assumed. The valid values of *control* are between 0 and 4 in 16-bit mode, between 0 and 7 when tuning for older processors in 32-bit mode, between 0 and 11 in 64-bit mode or when tuning for newer processors in 32-bit mode. When 0 is used, the no-op instruction size limit is set to the maximum supported size.

## 7.69 .octa bignums

This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer.

The term "octa" comes from contexts in which a "word" is two bytes; hence *octa*-word for 16 bytes.

## 7.70 .offset loc

Set the location counter to *loc* in the absolute section. *loc* must be an absolute expression. This directive may be useful for defining symbols with absolute values. Do not confuse it with the .org directive.

# 7.71 .org new-lc , fill

Advance the location counter of the current section to *new-lc*. *new-lc* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use .org to cross sections: if *new-lc* has the wrong section, the .org directive is ignored. To be compatible with former assemblers, if the section of *new-lc* is absolute, as issues a warning, then pretends the section of *new-lc* is the same as the current subsection.

.org may only increase the location counter, or leave it unchanged; you cannot use .org to move the location counter backwards.

Because **as** tries to assemble programs in one pass, *new-lc* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *fill* which should be an absolute expression. If the comma and *fill* are omitted, *fill* defaults to zero.

# 7.72 .p2align[wl] [abs-expr[, abs-expr[, abs-expr]]]

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example '.p2align 3' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. If the expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The .p2alignw and .p2align1 directives are variants of the .p2align directive. The .p2alignw directive treats the fill pattern as a two byte word value. The .p2align1 directives treats the fill pattern as a four byte longword value. For example, .p2alignw 2,0x368d will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

## 7.73 .popsection

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.85 [Section], page 77), .subsection (see Section 7.96 [SubSection], page 84), .pushsection (see Section 7.79 [PushSection], page 76), and .previous (see Section 7.74 [Previous], page 75).

This directive replaces the current section (and subsection) with the top section (and subsection) on the section stack. This section is popped off the stack.

## 7.74 .previous

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.85 [Section], page 77), .subsection (see Section 7.96 [SubSection], page 84), .pushsection (see Section 7.79 [PushSection], page 76), and .popsection (see Section 7.73 [PopSection], page 74).

This directive swaps the current section (and subsection) with most recently referenced section/subsection pair prior to this one. Multiple .previous directives in a row will flip between two sections (and their subsections). For example:

```
.section A
.subsection 1
.word 0x1234
.subsection 2
.word 0x5678
.previous
.word 0x9abc
```

Will place 0x1234 and 0x9abc into subsection 1 and 0x5678 into subsection 2 of section A. Whilst:

```
.section A
.subsection 1
    # Now in section A subsection 1
    .word 0x1234
.section B
.subsection 0
    # Now in section B subsection 0
    .word 0x5678
.subsection 1
    # Now in section B subsection 1
    .word 0x9abc
.previous
    # Now in section B subsection 0
    .word 0xdef0
```

Will place 0x1234 into section A, 0x5678 and 0xdef0 into subsection 0 of section B and 0x9abc into subsection 1 of section B.

In terms of the section stack, this directive swaps the current section with the top section on the section stack.

## 7.75 .print string

**as** will print *string* on the standard output during assembly. You must put *string* in double quotes.

## 7.76 .protected names

This is one of the ELF visibility directives. The other two are .hidden (see Section 7.43 [Hidden], page 64) and .internal (see Section 7.50 [Internal], page 66).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to **protected** which means that any references to the symbols from within the components that defines them must be resolved to the definition in that component, even if a definition in another component would normally preempt this.

## 7.77 .psize lines , columns

Use this directive to declare the number of lines—and, optionally, the number of columns—to use for each page, when generating listings.

If you do not use .psize, listings use a default line-count of 60. You may omit the comma and *columns* specification; the default width is 200 columns.

as generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using .eject).

If you specify *lines* as 0, no formfeeds are generated save those explicitly specified with .eject.

## 7.78 .purgem name

Undefine the macro *name*, so that later uses of the string will not be expanded. See Section 7.63 [Macro], page 69.

# 7.79 .pushsection name [, subsection] [, "flags"[, @type[,arguments]]]

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.85 [Section], page 77), .subsection (see Section 7.96 [SubSection], page 84), .popsection (see Section 7.73 [PopSection], page 74), and .previous (see Section 7.74 [Previous], page 75).

This directive pushes the current section (and subsection) onto the top of the section stack, and then replaces the current section and subsection with name and subsection. The optional flags, type and arguments are treated the same as in the .section (see Section 7.85 [Section], page 77) directive.

## 7.80 .quad expressions

For 64-bit architectures, or more generally with any GAS configured to support 64-bit target virtual addresses, this is like '.int', but emitting 64-bit quantities. Otherwise .quad expects zero or more bignums, separated by commas. For each item, it emits an 8-byte integer. If a bignum won't fit in 8 bytes, a warning message is printed and just the lowest order 8 bytes of the bignum are taken.

The term "quad" comes from contexts in which a "word" is two bytes; hence quad-word for 8 bytes.

Note - this directive is not intended for encoding instructions, and it will not trigger effects like DWARF line number generation. Instead some targets support special directives for encoding arbitrary binary sequences as instructions such as .insn or .inst.

## 7.81 .reloc offset, reloc\_name[, expression]

Generate a relocation at offset of type reloc\_name with value expression. If offset is a number, the relocation is generated in the current section. If offset is an expression that resolves to a symbol plus offset, the relocation is generated in the given symbol's section. expression, if present, must resolve to a symbol plus addend or to an absolute value, but

note that not all targets support an addend. e.g. ELF REL targets such as i386 store an addend in the section contents rather than in the relocation. This low level interface does not support addends stored in the section.

# 7.82 .rept count

Repeat the sequence of lines between the .rept directive and the next .endr directive *count* times.

For example, assembling

.rept 3 .long 0 .endr is equivalent to assembling

> .long 0 .long 0

> > .long 0

A count of zero is allowed, but nothing is generated. Negative counts are not allowed and if encountered will be treated as if they were zero.

# 7.83 .sbttl "subheading"

Use subheading as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

# 7.84 .scl class

Set the storage-class value for a symbol. This directive may only be used inside a .def/.endef pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

## 7.85 .section name

Use the .section directive to assemble the following code into a section named name.

This directive is only supported for targets that actually support arbitrarily named sections; on **a.out** targets, for example, it is not accepted, even with a standard **a.out** section name.

# **COFF** Version

For COFF targets, the .section directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsection]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

b bss section (uninitialized data)

n	section is not loaded
W	writable section
d	data section
е	exclude section from linking
r	read-only section
x	executable section
s	shared section (meaningful for PE targets)
a	ignored. (For compatibility with the ELF version)
у	section is not readable (meaningful for PE targets)
0-9	single-digit power-of-two section alignment (GNU extension)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable. Note the n and w flags remove attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all.

If the optional argument to the .section directive is not quoted, it is taken as a subsection number (see Section 4.4 [Sub-Sections], page 39).

## ELF Version

This is one of the ELF section stack manipulation directives. The others are .subsection (see Section 7.96 [SubSection], page 84), .pushsection (see Section 7.79 [PushSection], page 76), .popsection (see Section 7.73 [PopSection], page 74), and .previous (see Section 7.74 [Previous], page 75).

For ELF targets, the .section directive is used like this:

.section name [, "flags"[, @type[,flag\_specific\_arguments]]]

If the '--sectname-subst' command-line option is provided, the *name* argument may contain a substitution sequence. Only %S is supported at the moment, and substitutes the current section name. For example:

```
.macro exception_code
.section %S.exception
[exception code here]
.previous
.endm
.text
[code]
exception_code
[...]
.section .init
[init code]
exception_code
[...]
```

The two exception\_code invocations above would create the .text.exception and .init.exception sections respectively. This is useful e.g. to discriminate between ancillary

sections that are tied to setup code to be discarded after use from ancillary sections that need to stay resident without having to define multiple exception\_code macros just for that purpose.

The optional *flags* argument is a quoted string which may contain any combination of the following characters:

a	section is allocatable
d	section is a GNU_MBIND section
е	section is excluded from executable and shared library.
0	section references a symbol defined in another section (the linked-to section) in the same file.
W	section is writable
x	section is executable
М	section is mergeable
S	section contains zero terminated strings
G	section is a member of a section group
Т	section is used for thread-local-storage
?	section is a member of the previously-current section's group, if any
R	retained section (apply SHF_GNU_RETAIN to prevent linker garbage collection, GNU ELF extension)
<number></number>	a numeric value indicating the bits to be set in the ELF section header's flags field. Note - if one or more of the alphabetic characters described above is also

included in the flags field, their bit values will be ORed into the resulting value.

### <target specific>

some targets extend this list with their own flag characters

Note - once a section's flags have been set they cannot be changed. There are a few exceptions to this rule however. Processor and application specific flags can be added to an already defined section. The .interp, .strtab and .symtab sections can have the allocate flag (a) set after they are initially defined, and the .note-GNU-stack section may have the executable (x) flag added. Also note that the .attach\_to\_group directive can be used to add a section to a group even if the section was not originally declared to be part of that group.

The optional type argument may contain one of the following constants:

## 0progbits

section contains data

<b>Cnobits</b> section does not contain data (i.e., section only occupies s	s space)
---	----------

**@note** section contains data which is used by things other than the program

### @init\_array

section contains an array of pointers to init functions

#### @fini\_array

section contains an array of pointers to finish functions

#### @preinit\_array

section contains an array of pointers to pre-init functions

#### @<number>

a numeric value to be set as the ELF section header's type field.

#### @<target specific>

some targets extend this list with their own types

Many targets only support the first three section types. The type may be enclosed in double quotes if necessary.

Note on targets where the @ character is the start of a comment (eg ARM) then another character is used instead. For example the ARM port uses the % character.

Note - some sections, eg .text and .data are considered to be special and have fixed types. Any attempt to declare them with a different type will generate an error from the assembler.

If flags contains the M symbol then the type argument must be specified as well as an extra argument—entsize—like this:

.section name , "flags"M, @type, entsize

Sections with the M flag but not S flag must contain fixed size constants, each *entsize* octets long. Sections with both M and S must contain zero terminated strings where each character is *entsize* bytes long. The linker may remove duplicates within sections with the same name, same entity size and same flags. *entsize* must be an absolute expression. For sections with both M and S, a string which is a suffix of a larger string is considered a duplicate. Thus "def" will be merged with "abcdef"; A reference to the first "def" will be changed to a reference to "abcdef"+3.

If flags contains the o flag, then the type argument must be present along with an additional field like this:

.section name, "flags"o, @type, SymbolName | SectionIndex

The SymbolName field specifies the symbol name which the section references. Alternatively a numeric SectionIndex can be provided. This is not generally a good idea as section indicies are rarely known at assembly time, but the facility is provided for testing purposes. An index of zero is allowed. It indicates that the linked-to section has already been discarded.

Note: If both the M and o flags are present, then the fields for the Merge flag should come first, like this:

.section name, "flags"Mo, @type, entsize, SymbolName

If flags contains the G symbol then the type argument must be present along with an additional field like this:

.section name , "flags"G, @type, GroupName[, linkage]

The *GroupName* field specifies the name of the section group to which this particular section belongs. The optional linkage field can contain:

comdat indicates that only one copy of this section should be retained

#### .gnu.linkonce

an alias for comdat

Note: if both the M and G flags are present then the fields for the Merge flag should come first, like this:

.section name , "flags"MG, @type, entsize, GroupName[, linkage]

If both o flag and G flag are present, then the *SymbolName* field for o comes first, like this:

.section name, "flags"oG, @type, SymbolName, GroupName[, linkage]

If flags contains the ? symbol then it may not also contain the G symbol and the GroupName or linkage fields should not be present. Instead, ? says to consider the section that's current before this directive. If that section used G, then the new section will use G with those same GroupName and linkage fields implicitly. If not, then the ? symbol has no effect.

The optional *unique*,<**number**> argument must come last. It assigns <**number**> as a unique section ID to distinguish different sections with the same section name like these:

.section name,"flags",@type,unique,<number>
.section name,"flags"G,@type,GroupName,[linkage],unique,<number>
.section name,"flags"MG,@type,entsize,GroupName[,linkage],unique,<number>

The valid values of <number> are between 0 and 4294967295.

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

For SPARC ELF targets, the assembler supports another type of .section directive for compatibility with the Solaris assembler:

.section "name"[, flags...]

Note that the section name is quoted. There may be a sequence of comma separated flags:

**#alloc** section is allocatable

**#write** section is writable

#execinstr

section is executable

**#exclude** section is excluded from executable and shared library.

**#tls** section is used for thread local storage

This directive replaces the current section and subsection. See the contents of the gas testsuite directory gas/testsuite/gas/elf for some examples of how this directive and the other section stack directives work.

### 7.86 .set symbol, expression

Set the value of symbol to expression. This changes symbol's value and type to conform to expression. If symbol was flagged as external, it remains flagged (see Section 5.5 [Symbol Attributes], page 45).

You may .set a symbol many times in the same assembly provided that the values given to the symbol are constants. Values that are based on expressions involving other symbols are allowed, but some targets may restrict this to only being done once per assembly. This is because those targets do not set the addresses of symbols at assembly time, but rather delay the assignment until a final link is performed. This allows the linker a chance to change the code in the files, changing the location of, and the relative distance between, various different symbols.

If you .set a global symbol, the value stored in the object file is the last value stored into it.

On Z80 set is a real instruction, use .set or 'symbol defl expression' instead.

## 7.87 .short expressions

.short is normally the same as '.word'. See Section 7.111 [.word], page 88.

In some configurations, however, .short and .word generate numbers of different lengths. See Chapter 9 [Machine Dependencies], page 95.

Note - this directive is not intended for encoding instructions, and it will not trigger effects like DWARF line number generation. Instead some targets support special directives for encoding arbitrary binary sequences as instructions such as .insn or .inst.

### 7.88 .single flonums

This directive assembles zero or more flonums, separated by commas. It has the same effect as .float. The exact kind of floating point numbers emitted depends on how as is configured. See Chapter 9 [Machine Dependencies], page 95.

## 7.89 .size

This directive is used to set the size associated with a symbol.

## **COFF** Version

For COFF targets, the .size directive is only permitted inside .def/.endef pairs. It is used like this:

.size expression

## ELF Version

For ELF targets, the .size directive is used like this:

```
.size name , expression
```

This directive sets the size associated with a symbol *name*. The size in bytes is computed from *expression* which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

## 7.90 .skip size [,fill]

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as '.space'.

## 7.91 .sleb128 expressions

*sleb128* stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.103 [.uleb128], page 87.

# 7.92 .space size [,fill]

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. This is the same as '.skip'.

*Warning:* .space has a completely different meaning for HPPA targets; use .block as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the .space directive. See Section 9.15.5 [HPPA Assembler Directives], page 176, for a summary.

## 7.93 .stabd, .stabn, .stabs

There are three directives that begin '.stab'. All emit symbols (see Chapter 5 [Symbols], page 43), for use by symbolic debuggers. The symbols are not entered in the as hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string	This is the symbol's name. It may contain any character except '\000', so
	is more general than ordinary symbol names. Some debuggers used to code
	arbitrarily complex structures into symbol names using this field.
turno	An absolute expression. The symbol's type is set to the law 8 bits of this

- type An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but 1d and debuggers choke on silly bit patterns.
- other An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.
- desc An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value An absolute expression which becomes the symbol's value.

If a warning is detected while reading a .stabd, .stabn, or .stabs statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

### .stabd type , other , desc

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the .stabd was assembled.

```
.stabn type , other , desc , value
```

The name of the symbol is set to the empty string "".

```
.stabs string , type , other , desc , value All five fields are specified.
```

## 7.94 .string "str", .string8 "str", .string16

"str", .string32 "str", .string64 "str"

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in Section 3.6.1.1 [Strings], page 33.

The variants string16, string32 and string64 differ from the string pseudo opcode in that each 8-bit character from *str* is copied and expanded to 16, 32 or 64 bits respectively. The expanded characters are stored in target endianness byte order.

Example:

```
.string32 "BYE"
expands to:
.string "B\0\0\0Y\0\0\0E\0\0" /* On little endian targets. */
.string "\0\0\0B\0\0\0Y\0\0\0E" /* On big endian targets. */
```

## 7.95 .struct expression

Switch to the absolute section, and set the section offset to *expression*, which must be an absolute expression. You might use this as follows:

```
.struct 0
field1:
.struct field1 + 4
field2:
.struct field2 + 4
field3:
```

This would define the symbol field1 to have the value 0, the symbol field2 to have the value 4, and the symbol field3 to have the value 8. Assembly would be left in the absolute section, and you would need to use a .section directive of some sort to change to some other section before further assembly.

## 7.96 .subsection name

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.85 [Section], page 77), .pushsection (see Section 7.79 [PushSection], page 76), .popsection (see Section 7.73 [PopSection], page 74), and .previous (see Section 7.74 [Previous], page 75).

This directive replaces the current subsection with name. The current section is not changed. The replaced subsection is put onto the section stack in place of the then current top of stack subsection.

## 7.97 .symver

Use the .symver directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. There are cases where it may make sense to use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library.

For ELF targets, the .symver directive can be used like this:

.symver name, name2@nodename[ ,visibility]

If the original symbol name is defined within the file being assembled, the .symver directive effectively creates a symbol alias with the name name2@nodename, and in fact the main reason that we just don't try and create a regular alias is that the @ character isn't permitted in symbol names. The *name2* part of the name is the actual name of the symbol by which it will be externally referenced. The name name itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The *nodename* portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, then *nodename* should correspond to the nodename of the symbol you are trying to override. The optional argument visibility updates the visibility of the original symbol. The valid visibilities are local, hidden, and remove. The local visibility makes the original symbol a local symbol (see Section 7.61 [Local], page 69). The hidden visibility sets the visibility of the original symbol to hidden (see Section 7.43 [Hidden], page 64). The remove visibility removes the original symbol from the symbol table. If visibility isn't specified, the original symbol is unchanged.

If the symbol name is not defined within the file being assembled, all references to name will be changed to name2@nodename. If no reference to name is made, name2@nodename will be removed from the symbol table.

Another usage of the .symver directive is:

.symver name, name200nodename

In this case, the symbol name must exist and be defined within the file being assembled. It is similar to name2@nodename. The difference is name2@@nodename will also be used to resolve references to name2 by the linker.

The third usage of the .symver directive is:

.symver name, name2000nodename

When name is not defined within the file being assembled, it is treated as name2@nodename. When name is defined within the file being assembled, the symbol name, name, will be changed to name2@@nodename.

### 7.98 .tag structname

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def/.endef pairs. Tags are used to link structure definitions in the symbol table with instances of those structures.

## 7.99 .text subsection

Tells **as** to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.

# 7.100 .title "heading"

Use *heading* as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

# 7.101 .tls\_common symbol, length[, alignment]

This directive behaves in the same way as the .comm directive (see Section 7.13 [Comm], page 58) except that symbol has type of STT\_TLS instead of STT\_OBJECT.

## 7.102 .type

This directive is used to set the type of a symbol.

# **COFF** Version

For COFF targets, this directive is permitted only within .def/.endef pairs. It is used like this:

.type int

This records the integer *int* as the type attribute of a symbol table entry.

## ELF Version

For ELF targets, the .type directive is used like this:

.type name , type description

This sets the type of symbol name to be either a function symbol or an object symbol. There are five different syntaxes supported for the *type description* field, in order to provide compatibility with various other assemblers.

Because some of the characters used in these syntaxes (such as '@' and '#') are comment characters for some architectures, some of the syntaxes below do not work on all architectures. The first variant will be accepted by the GNU assembler on all architectures so that variant should be used for maximum portability, if you do not need to assemble your code with other assemblers.

The syntaxes supported are:

```
.type <name> STT_<TYPE_IN_UPPER_CASE>
.type <name>,#<type>
.type <name>,@<type>
.type <name>,%<type>
.type <name>,%<type>
```

The types supported are:

### STT\_FUNC

function Mark the symbol as being a function name.

#### STT\_GNU\_IFUNC

### gnu\_indirect\_function

Mark the symbol as an indirect function when evaluated during reloc processing. (This is only supported on assemblers targeting GNU systems). STT\_OBJECT object Mark the symbol as being a data object. STT\_TLS tls\_object Mark the symbol as being a thread-local data object. STT\_COMMON Mark the symbol as being a common data object. common STT\_NOTYPE Does not mark the symbol in any way. It is supported just for completeness. notype gnu\_unique\_object Marks the symbol as being a globally unique data object. The dynamic linker will make sure that in the entire process there is just one symbol with this name and type in use. (This is only supported on assemblers targeting GNU systems). Changing between incompatible types other than from/to STT\_NOTYPE will result in a

diagnostic. An intermediate change to STT\_NOTYPE will silence this.

Note: Some targets support extra types in addition to those listed above.

# 7.103 .uleb128 expressions

*uleb128* stands for "unsigned little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.91 [.sleb128], page 83.

## 7.104 .val addr

This directive, permitted only within .def/.endef pairs, records the address *addr* as the value attribute of a symbol table entry.

# 7.105 .version "string"

This directive creates a .note section and places into it an ELF formatted note of type NT\_VERSION. The note's name is set to string.

## 7.106 .vtable\_entry table, offset

This directive finds or creates a symbol table and creates a VTABLE\_ENTRY relocation for it with an addend of offset.

## 7.107 .vtable\_inherit child, parent

This directive finds the symbol child and finds or creates the symbol parent and then creates a VTABLE\_INHERIT relocation for the parent whose addend is the value of the child symbol. As a special case the parent name of 0 is treated as referring to the \*ABS\* section.

## 7.108 .warning "string"

Similar to the directive .error (see Section 7.33 [.error "string"], page 61), but just emits a warning.

### 7.109 .weak names

This directive sets the weak attribute on the comma separated list of symbol names. If the symbols do not already exist, they will be created.

On COFF targets other than PE, weak symbols are a GNU extension. This directive sets the weak attribute on the comma separated list of symbol names. If the symbols do not already exist, they will be created.

On the PE target, weak symbols are supported natively as weak aliases. When a weak symbol is created that is not an alias, GAS creates an alternate symbol to hold the default value.

## 7.110 .weakref alias, target

This directive creates an alias to the target symbol that enables the symbol to be referenced with weak-symbol semantics, but without actually making it weak. If direct references or definitions of the symbol are present, then the symbol will not be weak, but if all references to it are through weak references, the symbol will be marked as weak in the symbol table.

The effect is equivalent to moving all references to the alias to a separate assembly source file, renaming the alias to the symbol in it, declaring the symbol as weak there, and running a reloadable link to merge the object files resulting from the assembly of the new source file and the old source file that had the references to the alias removed.

The alias itself never makes to the symbol table, and is entirely handled within the assembler.

## 7.111 .word expressions

This directive expects zero or more *expressions*, of any section, separated by commas.

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

### Warning: Special Treatment to support Compilers

Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see Chapter 9 [Machine Dependencies], page 95), you can ignore this issue.

In order to assemble compiler output into something that works, as occasionally does strange things to '.word' directives. Directives of the form '.word sym1-sym2' are often emitted by compilers as part of jump tables. Therefore, when as assembles a directive of the form '.word sym1-sym2', and the difference between sym1 and sym2 does not fit in 16 bits, as creates a secondary jump table, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to sym2. The original '.word' contains sym1 minus the address of the long-jump to sym2.

If there were several occurrences of '.word sym1-sym2' before the secondary jump table, all of them are adjusted. If there was a '.word sym3-sym4', that also did not fit in sixteen bits, a long-jump to sym4 is included in the secondary jump table, and the .word directives are adjusted to contain sym3 minus the address of the long-jump to sym4; and so on, for as many entries in the original jump table as necessary.

# 7.112 .zero size

This directive emits size 0-valued bytes. size must be an absolute expression. This directive is actually an alias for the '.skip' directive so it can take an optional second argument of the value to store in the bytes instead of zero. Using '.zero' in this way would be confusing however.

# 7.113 .2byte expression [, expression]\*

This directive expects zero or more expressions, separated by commas. If there are no expressions then the directive does nothing. Otherwise each expression is evaluated in turn and placed in the next two bytes of the current output section, using the endian model of the target. If an expression will not fit in two bytes, a warning message is displayed and the least significant two bytes of the expression's value are used. If an expression cannot be evaluated at assembly time then relocations will be generated in order to compute the value at link time.

This directive does not apply any alignment before or after inserting the values. As a result of this, if relocations are generated, they may be different from those used for inserting values with a guaranteed alignment.

# 7.114 .4byte expression [, expression]\*

Like the .2byte directive, except that it inserts unaligned, four byte long values into the output.

# 7.115 .8byte expression [, expression]\*

For 64-bit architectures, or more generally with any GAS configured to support 64-bit target virtual addresses, this is like the .2byte directive, except that it inserts unaligned, eight byte long values into the output. Otherwise, like Section 7.80 [.quad expressions], page 76, it expects zero or more bignums, separated by commas.

# 7.116 Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

```
.abort
```

.line

## 8 Object Attributes

**as** assembles source files written for a specific architecture into object files for that architecture. But not all object files are alike. Many architectures support incompatible variations. For instance, floating point arguments might be passed in floating point registers if the object file requires hardware floating point support—or floating point arguments might be passed in integer registers if the object file supports processors with no hardware floating point unit. Or, if two objects are built for different generations of the same architecture, the combination may require the newer generation at run-time.

This information is useful during and after linking. At link time, 1d can warn about incompatible object files. After link time, tools like gdb can use it to process the linked file correctly.

Compatibility information is recorded as a series of object attributes. Each attribute has a vendor, tag, and value. The vendor is a string, and indicates who sets the meaning of the tag. The tag is an integer, and indicates what property the attribute describes. The value may be a string or an integer, and indicates how the property affects this object. Missing attributes are the same as attributes with a zero value or empty string value.

Object attributes were developed as part of the ABI for the ARM Architecture. The file format is documented in *ELF* for the ARM Architecture.

## 8.1 GNU Object Attributes

The .gnu\_attribute directive records an object attribute with vendor 'gnu'.

Except for 'Tag\_compatibility', which has both an integer and a string for its value, GNU attributes have a string value if the tag number is odd and an integer value if the tag number is even. The second bit (tag & 2 is set for architecture-independent attributes and clear for architecture-dependent ones.

#### 8.1.1 Common GNU attributes

These attributes are valid on all architectures.

 $Tag_compatibility (32)$ 

The compatibility attribute takes an integer flag value and a vendor name. If the flag value is 0, the file is compatible with other toolchains. If it is 1, then the file is only compatible with the named toolchain. If it is greater than 1, the file can only be processed by other toolchains under some private arrangement indicated by the flag value and the vendor name.

## 8.1.2 M680x0 Attributes

#### $Tag_GNU_M68K_ABI_FP$ (4)

The floating-point ABI used by this object file. The value will be:

- 0 for files not affected by the floating-point ABI.
- 1 for files using double-precision hardware floating-point ABI.
- 2 for files using the software floating-point ABI.

## 8.1.3 MIPS Attributes

## Tag\_GNU\_MIPS\_ABI\_FP (4)

The floating-point ABI used by this object file. The value will be:

- 0 for files not affected by the floating-point ABI.
- 1 for files using the hardware floating-point ABI with a standard double-precision FPU.
- 2 for files using the hardware floating-point ABI with a single-precision FPU.
- 3 for files using the software floating-point ABI.
- 4 for files using the deprecated hardware floating-point ABI which used 64-bit floating-point registers, 32-bit general-purpose registers and increased the number of callee-saved floating-point registers.
- 5 for files using the hardware floating-point ABI with a double-precision FPU with either 32-bit or 64-bit floating-point registers and 32-bit general-purpose registers.
- 6 for files using the hardware floating-point ABI with 64-bit floating-point registers and 32-bit general-purpose registers.
- 7 for files using the hardware floating-point ABI with 64-bit floating-point registers, 32-bit general-purpose registers and a rule that forbids the direct use of odd-numbered single-precision floating-point registers.

## Tag\_GNU\_MIPS\_ABI\_MSA (8)

The MIPS SIMD Architecture (MSA) ABI used by this object file. The value will be:

- 0 for files not affected by the MSA ABI.
- 1 for files using the 128-bit MSA ABI.

## 8.1.4 PowerPC Attributes

## Tag\_GNU\_Power\_ABI\_FP (4)

The floating-point ABI used by this object file. The value will be:

- 0 for files not affected by the floating-point ABI.
- 1 for files using double-precision hardware floating-point ABI.
- 2 for files using the software floating-point ABI.
- 3 for files using single-precision hardware floating-point ABI.

## $Tag_GNU_Power_ABI_Vector$ (8)

- The vector ABI used by this object file. The value will be:
  - 0 for files not affected by the vector ABI.
  - 1 for files using general purpose registers to pass vectors.
  - 2 for files using AltiVec registers to pass vectors.
  - 3 for files using SPE registers to pass vectors.

## 8.1.5 IBM z Systems Attributes

 $Tag_GNU_S390_ABI_Vector$  (8)

The vector ABI used by this object file. The value will be:

- 0 for files not affected by the vector ABI.
- 1 for files using software vector ABI.
- 2 for files using hardware vector ABI.

## 8.1.6 MSP430 Attributes

Tag\_GNU\_MSP430\_Data\_Region (4)

- The data region used by this object file. The value will be:
  - 0 for files not using the large memory model.
  - 1 for files which have been compiled with the condition that all data is in the lower memory region, i.e. below address 0x10000.
  - 2 for files which allow data to be placed in the full 20-bit memory range.

## 8.2 Defining New Object Attributes

If you want to define a new GNU object attribute, here are the places you will need to modify. New attributes should be discussed on the 'binutils' mailing list.

- This manual, which is the official register of attributes.
- The header for your architecture include/elf, to define the tag.
- The **bfd** support file for your architecture, to merge the attribute and issue any appropriate link warnings.
- Test cases in ld/testsuite for merging and link warnings.
- **binutils/readelf.c** to display your attribute.
- GCC, if you want the compiler to mark the attribute automatically.

# 9 Machine Dependent Features

The machine instruction sets are (almost by definition) different on each machine where **as** runs. Floating point representations vary as well, and **as** often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of **as** support special pseudo-instructions for branch optimization.

This chapter discusses most of these differences, though it does not include details on any machine's instruction set. For details on that subject, see the hardware manufacturer's manual.

## 9.1 AArch64 Dependent Features

### 9.1.1 Options

- -EB This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.
- -EL This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor.

-mabi=abi

Specify which ABI the source code uses. The recognized arguments are: i1p32 and 1p64, which decides the generated object file in ELF32 and ELF64 format respectively. The default is 1p64.

#### -mcpu=processor[+extension...]

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: cortex-a34, cortex-a35, cortex-a53, cortex-a55, cortex-a57, cortex-a65. cortex-a65ae, cortex-a72, cortex-a73, cortex-a75, cortex-a76. cortex-a76ae. cortex-a77, cortex-a78, cortex-a78ae. cortex-a78c, cortex-a510, cortex-a710, ares, exynos-m1, falkor, neoverse-n1, neoverse-n2, neoverse-e1, neoverse-v1, qdf24xx, saphira, thunderx, vulcan, xgene1 xgene2, cortex-r82, cortex-x1, and cortex-x2. The special name **all** may be used to allow the assembler to accept instructions valid for any supported processor, including all optional extensions.

In addition to the basic instruction set, the assembler can be told to accept, or restrict, various extension mnemonics that extend the processor. See Section 9.1.2 [AArch64 Extensions], page 97.

If some implementations of a particular processor can have an extension, then then those extensions are automatically enabled. Consequently, you will not normally have to specify any additional extensions.

#### -march=architecture[+extension...]

This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: armv8-a, armv8.1-a, armv8.2-a, armv8.3-a, armv8.4-a armv8.5-a, armv8.6-a, armv8.7-a, armv8.8-a, armv8-r, armv9-a, armv9.1-a, armv9.2-a, and armv9.3-a.

If both -mcpu and -march are specified, the assembler will use the setting for -mcpu. If neither are specified, the assembler will default to -mcpu=all.

The architecture option can be extended with the same instruction set extension options as the -mcpu option. Unlike -mcpu, extensions are not always enabled by default, See Section 9.1.2 [AArch64 Extensions], page 97.

#### -mverbose-error

This option enables verbose error messages for AArch64 gas. This option is enabled by default. -mno-verbose-error

This option disables verbose error messages in AArch64 gas.

## 9.1.2 Architecture Extensions

The table below lists the permitted architecture extensions that are supported by the assembler and the conditions under which they are automatically enabled.

Multiple extensions may be specified, separated by a +. Extension mnemonics may also be removed from those the assembler accepts. This is done by prepending **no** to the option that adds the extension. Extensions that are removed must be listed after all extensions that have been added.

Enabling an extension that requires other extensions will automatically cause those extensions to be enabled. Similarly, disabling an extension that is required by other extensions will automatically cause those extensions to be disabled.

Extension	Minimum Architecture	Enabled by default	Description
aes	ARMv8-A	No	Enable the AES cryptographic extensions. This implies <b>fp</b> and <b>simd</b> .
bf16	ARMv8.2-A	ARMv8.6-A or later	Enable BFloat16 extension.
compnum	ARMv8.2-A	ARMv8.3-A or later	Enable the complex number SIMD extensions. This implies fp16 and simd.
crc	ARMv8-A	ARMv8.1-A or later	Enable CRC instructions.
crypto	ARMv8-A	No	Enable cryptographic extensions. This implies fp, simd, aes and sha2.
dotprod	ARMv8.2-A	ARMv8.4-A or later	Enable the Dot Product extension. This implies simd.
f32mm	ARMv8.2-A	No	Enable F32 Matrix Multiply extension. This implies sve.
f64mm	ARMv8.2-A	No	Enable F64 Matrix Multiply extension. This implies sve.
flagm	ARMv8-A	ARMv8.4-A or later	Enable Flag Manipulation instructions.
fp16fml	ARMv8.2-A	ARMv8.4-A or later	Enable ARMv8.2 16-bit floating-point multipli- cation variant support. This implies fp and fp16.
fp16	ARMv8.2-A	ARMv8.2-A or later	Enable ARMv8.2 16-bit floating-point support. This implies fp.
fp	ARMv8-A	ARMv8-A or later	Enable floating-point extensions.
hbc		Armv8.8-A or later	Enable Armv8.8-A hinted conditional branch instructions
CSSC		Armv8.7-A or later	Enable Armv8.9-A Common Short Sequence Compression instructions.

i8mm	ARMv8.2-A	ARMv8.6-A or later	Enable Int8 Matrix Multiply extension.			
lor	ARMv8-A	ARMv8.1-A or later	Enable Limited Ordering Regions extensions.			
ls64	ARMv8.6-A	ARMv8.7-A or later	Enable 64 Byte Loads/Stores.			
lse	ARMv8-A	ARMv8.1-A or later	Enable Large System extensions.			
memtag	ARMv8.5-A	No	Enable ARMv8.5-A Memory Tagging Extensions.			
mops		Armv8.8-A or later	Enable Armv8.8-A memcpy and memset acceleration instructions			
pan	ARMv8-A	ARMv8.1-A or later	Enable Privileged Access Never support.			
pauth	ARMv8-A	No	Enable Pointer Authentication.			
predres	ARMv8-A	ARMv8.5-A or later	Enable the Execution and Data and Prediction instructions.			
profile	ARMv8.2-A	No	Enable statistical profiling extensions.			
ras	ARMv8-A	ARMv8.2-A	Enable the Reliability, Availability and Service-			
		or later	ability extension.			
rcpc	ARMv8.2-A	ARMv8.3-A or later	Enable the weak release consistency extension.			
rdma	ARMv8-A	ARMv8.1-A or later	Enable ARMv8.1 Advanced SIMD extensions. This implies simd.			
rng	ARMv8.5-A	No	Enable ARMv8.5-A random number instructions.			
sb	ARMv8-A	ARMv8.5-A or later	Enable the speculation barrier instruction sb.			
sha2	ARMv8-A	No	Enable the SHA2 cryptographic extensions. This implies fp and simd.			
sha3	ARMv8.2-A	No	Enable the ARMv8.2-A SHA2 and SHA3 cryptographic extensions. This implies fp, simd and sha2.			
simd	ARMv8-A	ARMv8-A or later	Enable Advanced SIMD extensions. This implies fp.			
sm4	ARMv8.2-A	No	Enable the ARMv8.2-A SM3 and SM4 crypto- graphic extensions. This implies fp and simd.			
sme	Armv9-A	No	Enable SME Extension.			
sme-f64f64		No	Enable SME F64F64 Extension.			
sme-i16i64		No	Enable SME 16464 Extension.			
sme 110104 sme2	Armv9-A	No	Enable SME2. This implies sme.			
ssbs	ARMv8-A	ARMv8.5-A	Enable Speculative Store Bypassing Safe state			
6066	1111111 V O-M	or later	read and write.			
sve	ARMv8.2-A	Armv9-A or later	Enable the Scalable Vector Extensions. This implies fp16, simd and compnum.			

sve2	ARMv8-A	Armv9-A or later	Enable the SVE2 Extension. This implies ${\tt sve}.$
sve2-aes	ARMv8-A	No	Enable SVE2 AES Extension. This also enables the .Q->.B form of the pmullt and pmullb in- structions. This implies aes and sve2.
sve2-bitper ${ m ARMv8-A}$		No	Enable SVE2 BITPERM Extension.
sve2-sha3	ARMv8-A	No	Enable SVE2 SHA3 Extension. This implies sha3 and sve2.
sve2-sm4	ARMv8-A	No	Enable SVE2 SM4 Extension. This implies $sm4$ and $sve2$ .
tme	ARMv8-A	No	Enable Transactional Memory Extensions.

## 9.1.3 Syntax

#### 9.1.3.1 Special Characters

The presence of a '//' on a line indicates the start of a comment that extends to the end of the current line. If a '#' appears as the first character of a line, the whole line is treated as a comment.

The ';' character can be used instead of a newline to separate statements.

The '#' can be optionally used to indicate immediate operands.

#### 9.1.3.2 Register Names

Please refer to the section '4.4 Register Names' of 'ARMv8 Instruction Set Overview', which is available at http://infocenter.arm.com.

#### 9.1.3.3 Relocations

Relocations for 'MOVZ' and 'MOVK' instructions can be generated by prefixing the label with '#:abs\_g2:' etc. For example to load the 48-bit absolute address of *foo* into x0:

movz x0, #:abs\_g2:foo // bits 32-47, overflow check
movk x0, #:abs\_g1\_nc:foo // bits 16-31, no overflow check
movk x0, #:abs\_g0\_nc:foo // bits 0-15, no overflow check

Relocations for 'ADRP', and 'ADD', 'LDR' or 'STR' instructions can be generated by prefixing the label with ':pg\_hi21:' and '#:lo12:' respectively.

For example to use 33-bit (+/-4GB) pc-relative addressing to load the address of foo into x0:

adrp x0, :pg\_hi21:foo
add x0, x0, #:lo12:foo

Or to load the value of foo into x0:

adrp x0, :pg\_hi21:foo ldr x0, [x0, #:lo12:foo]

Note that ':pg\_hi21:' is optional.

adrp x0, foo

is equivalent to

adrp x0, :pg\_hi21:foo

## 9.1.4 Floating Point

The AArch64 architecture uses IEEE floating-point numbers.

## 9.1.5 AArch64 Machine Directives

.arch name

Select the target architecture. Valid values for *name* are the same as for the **-march** command-line option.

Specifying .arch clears any previously selected architecture extensions.

#### .arch\_extension name

Add or remove an architecture extension to the target architecture. Valid values for *name* are the same as those accepted as architectural extensions by the **-mcpu** command-line option.

.arch\_extension may be used multiple times to add or remove extensions incrementally to the architecture being compiled for.

- .bss This directive switches to the .bss section.
- .cpu name Set the target processor. Valid values for name are the same as those accepted by the -mcpu= command-line option.
- .dword expressions

The .dword directive produces 64 bit values.

- .even The .even directive aligns the output on the next even byte boundary.
- .float16 value [,...,value\_n]

Place the half precision floating point representation of one or more floatingpoint values into the current section. The format used to encode the floating point values is always the IEEE 754-2008 half precision floating point format.

#### .inst expressions

Inserts the expressions into the output as if they were instructions, rather than data.

.ltorg This directive causes the current contents of the literal pool to be dumped into the current section (which is assumed to be the .text section) at the current location (aligned to a word boundary). GAS maintains a separate literal pool for each section and each sub-section. The .ltorg directive will only affect the literal pool of the current section and sub-section. At the end of assembly all remaining, un-empty literal pools will automatically be dumped.

> Note - older versions of GAS would dump the current literal pool any time a section change occurred. This is no longer done, since it prevents accurate control of the placement of literal pools.

.pool This is a synonym for .ltorg.

#### name .req register name

This creates an alias for register name called name. For example:

#### foo .req wO

ip0, ip1, lr and fp are automatically defined to alias to X16, X17, X30 and X29 respectively.

#### .tlsdescadd

Emits a TLSDESC\_ADD reloc on the next instruction.

#### .tlsdesccall

Emits a TLSDESC\_CALL reloc on the next instruction.

#### .tlsdescldr

Emits a TLSDESC\_LDR reloc on the next instruction.

#### .unreq alias-name

This undefines a register alias which was previously defined using the **req** directive. For example:

# foo .req w0 .unreq foo

An error occurs if the name is undefined. Note - this pseudo op can be used to delete builtin in register name aliases (eg 'w0'). This should only be done if it is really necessary.

#### .variant\_pcs symbol

This directive marks *symbol* referencing a function that may follow a variant procedure call standard with different register usage convention from the base procedure call standard.

#### .xword expressions

The .xword directive produces 64 bit values. This is the same as the .dword directive.

#### .cfi\_b\_key\_frame

The .cfi\_b\_key\_frame directive inserts a 'B' character into the CIE corresponding to the current frame's FDE, meaning that its return address has been signed with the B-key. If two frames are signed with differing keys then they will not share the same CIE. This information is intended to be used by the stack unwinder in order to properly authenticate return addresses.

## 9.1.6 Opcodes

GAS implements all the standard AArch64 opcodes. It also implements several pseudo opcodes, including several synthetic load instructions.

LDR =

ldr <register> , =<expression>

The constant expression will be placed into the nearest literal pool (if it not already there) and a PC-relative LDR instruction will be generated.

For more information on the AArch64 instruction set and assembly language notation, see 'ARMv8 Instruction Set Overview' available at http://infocenter.arm.com.

### 9.1.7 Mapping Symbols

The AArch64 ELF specification requires that special symbols be inserted into object files to mark certain features:

- **\$**x At the start of a region of code containing AArch64 instructions.
- **\$d** At the start of a region of data.

## 9.2 Alpha Dependent Features

## 9.2.1 Notes

The documentation here is primarily for the ELF object format. **as** also supports the ECOFF and EVAX formats, but features specific to these formats are not yet documented.

## 9.2.2 Options

-mcpu This option specifies the target processor. If an attempt is made to assemble an instruction which will not execute on the target processor, the assembler may either expand the instruction as a macro or issue an error message. This option is equivalent to the .arch directive.

The following processor names are recognized: 21064, 21064a, 21066, 21068, 21164, 21164a, 21164pc, 21264, 21264a, 21264b, ev4, ev5, lca45, ev5, ev56, pca56, ev6, ev67, ev68. The special name all may be used to allow the assembler to accept instructions valid for any Alpha processor.

In order to support existing practice in OSF/1 with respect to .arch, and existing practice within MILO (the Linux ARC bootloader), the numbered processor names (e.g. 21064) enable the processor-specific PALcode instructions, while the "electro-vlasic" names (e.g. ev4) do not.

#### -mdebug

-no-mdebug

Enables or disables the generation of .mdebug encapsulation for stabs directives and procedure descriptors. The default is to automatically enable .mdebug when the first stabs directive is seen.

-relax This option forces all relocations to be put into the object file, instead of saving space and resolving some relocations at assembly time. Note that this option does not propagate all symbol arithmetic into the object file, because not all symbol arithmetic can be represented. However, the option can still be useful in specific applications.

#### -replace

#### -noreplace

Enables or disables the optimization of procedure calls, both at assemblage and at link time. These options are only available for VMS targets and **-replace** is the default. See section 1.4.1 of the OpenVMS Linker Utility Manual.

- -g This option is used when the compiler generates debug information. When gcc is using mips-tfile to generate debug information for ECOFF, local labels must be passed through to the object file. Otherwise this option has no effect.
- -Gsize A local common symbol larger than size is placed in .bss, while smaller symbols are placed in .sbss.

#### -F

-32addr These options are ignored for backward compatibility.

## 9.2.3 Syntax

The assembler syntax closely follow the Alpha Reference Manual; assembler directives and general syntax closely follow the OSF/1 and OpenVMS syntax, with a few differences for ELF.

## 9.2.3.1 Special Characters

"#" is the line comment character. Note that if "#" is the first character on a line then it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

';' can be used instead of a newline to separate statements.

## 9.2.3.2 Register Names

The 32 integer registers are referred to as '\$n' or '\$rn'. In addition, registers 15, 28, 29, and 30 may be referred to by the symbols '\$fp', '\$at', '\$gp', and '\$sp' respectively.

The 32 floating-point registers are referred to as '\$fn'.

### 9.2.3.3 Relocations

Some of these relocations are available for ECOFF, but mostly only for ELF. They are modeled after the relocation format introduced in Digital Unix 4.0, but there are additions.

The format is '!tag' or '!tag!number' where tag is the name of the relocation. In some cases number is used to relate specific instructions.

The relocation is placed at the end of the instruction like so:

ldah	\$0,a(\$29)	!gprelhigh
lda	\$0,a(\$0)	gprellow!
ldq	\$1,b(\$29)	!literal!100
ldl	\$2,0(\$1)	<pre>!lituse_base!100</pre>

## !literal

```
!literal!N
```

Used with an ldq instruction to load the address of a symbol from the GOT.

A sequence number N is optional, and if present is used to pair lituse relocations with this literal relocation. The lituse relocations are used by the linker to optimize the code based on the final location of the symbol.

Note that these optimizations are dependent on the data flow of the program. Therefore, if *any* lituse is paired with a literal relocation, then *all* uses of the register set by the literal instruction must also be marked with lituse relocations. This is because the original literal instruction may be deleted or transformed into another instruction.

Also note that there may be a one-to-many relationship between literal and lituse, but not a many-to-one. That is, if there are two code paths that load up the same address and feed the value to a single use, then the use may not use a lituse relocation.

#### !lituse\_base!N

Used with any memory format instruction (e.g. 1d1) to indicate that the literal is used for an address load. The offset field of the instruction must be zero. During relaxation, the code may be altered to use a gp-relative load.

#### !lituse\_jsr!N

Used with a register branch format instruction (e.g. jsr) to indicate that the literal is used for a call. During relaxation, the code may be altered to use a direct branch (e.g. bsr).

#### !lituse\_jsrdirect!N

Similar to lituse\_jsr, but also that this call cannot be vectored through a PLT entry. This is useful for functions with special calling conventions which do not allow the normal call-clobbered registers to be clobbered.

#### !lituse\_bytoff!N

Used with a byte mask instruction (e.g. extbl) to indicate that only the low 3 bits of the address are relevant. During relaxation, the code may be altered to use an immediate instead of a register shift.

#### !lituse\_addr!N

Used with any other instruction to indicate that the original address is in fact used, and the original ldq instruction may not be altered or deleted. This is useful in conjunction with lituse\_jsr to test whether a weak symbol is defined.

ldq	\$27,foo(\$29)	!literal!1
beq	\$27,is_undef	<pre>!lituse_addr!1</pre>
jsr	\$26,(\$27),foo	!lituse_jsr!1

#### !lituse\_tlsgd!N

Used with a register branch format instruction to indicate that the literal is the call to \_\_tls\_get\_addr used to compute the address of the thread-local storage variable whose descriptor was loaded with !tlsgd!N.

#### !lituse\_tlsldm!N

Used with a register branch format instruction to indicate that the literal is the call to \_\_tls\_get\_addr used to compute the address of the base of the thread-local storage block for the current module. The descriptor for the module must have been loaded with !tlsldm!N.

#### !gpdisp!N

Used with ldah and lda to load the GP from the current address, a-la the ldgp macro. The source register for the ldah instruction must contain the address of the ldah instruction. There must be exactly one lda instruction paired with the ldah instruction, though it may appear anywhere in the instruction stream. The immediate operands must be zero.

bsr \$26,foo ldah \$29,0(\$26) !gpdisp!1 lda \$29,0(\$29) !gpdisp!1

#### !gprelhigh

Used with an ldah instruction to add the high 16 bits of a 32-bit displacement from the GP.

#### !gprellow

Used with any memory format instruction to add the low 16 bits of a 32-bit displacement from the GP.

- **!gprel** Used with any memory format instruction to add a 16-bit displacement from the GP.
- **!samegp** Used with any branch format instruction to skip the GP load at the target address. The referenced symbol must have the same GP as the source object file, and it must be declared to either not use \$27 or perform a standard GP load in the first two instructions via the .prologue directive.

#### !tlsgd

!tlsgd!N Used with an 1da instruction to load the address of a TLS descriptor for a symbol in the GOT.

The sequence number N is optional, and if present it used to pair the descriptor load with both the literal loading the address of the \_\_tls\_get\_addr function and the lituse\_tlsgd marking the call to that function.

For proper relaxation, both the tlsgd, literal and lituse relocations must be in the same extended basic block. That is, the relocation with the lowest address must be executed first at runtime.

## !tlsldm

#### !tlsldm!N

Used with an lda instruction to load the address of a TLS descriptor for the current module in the GOT.

Similar in other respects to tlsgd.

#### !gotdtprel

Used with an ldq instruction to load the offset of the TLS symbol within its module's thread-local storage block. Also known as the dynamic thread pointer offset or dtp-relative offset.

#### !dtprelhi

!dtprello

!dtprel Like gprel relocations except they compute dtp-relative offsets.

!gottprel

Used with an 1dq instruction to load the offset of the TLS symbol from the thread pointer. Also known as the tp-relative offset.

## !tprelhi

!tprello

!tprel Like gprel relocations except they compute tp-relative offsets.

### 9.2.4 Floating Point

The Alpha family uses both IEEE and VAX floating-point numbers.

## 9.2.5 Alpha Assembler Directives

as for the Alpha supports many additional directives for compatibility with the native assembler. This section describes them only briefly.

These are the additional directives in **as** for the Alpha:

- .arch cpu Specifies the target processor. This is equivalent to the -mcpu command-line option. See Section 9.2.2 [Alpha Options], page 102, for a list of values for cpu.
- .ent function[, n]

Mark the beginning of *function*. An optional number may follow for compatibility with the OSF/1 assembler, but is ignored. When generating .mdebug information, this will create a procedure descriptor for the function. In ELF, it will mark the symbol as a function a-la the generic .type directive.

#### .end function

Mark the end of *function*. In ELF, it will set the size of the symbol a-la the generic .size directive.

#### .mask mask, offset

Indicate which of the integer registers are saved in the current function's stack frame. mask is interpreted a bit mask in which bit n set indicates that register n is saved. The registers are saved in a block located offset bytes from the canonical frame address (CFA) which is the value of the stack pointer on entry to the function. The registers are saved sequentially, except that the return address register (normally \$26) is saved first.

This and the other directives that describe the stack frame are currently only used when generating .mdebug information. They may in the future be used to generate DWARF2 .debug\_frame unwind information for hand written assembly.

#### .fmask mask, offset

Indicate which of the floating-point registers are saved in the current stack frame. The mask and offset parameters are interpreted as with .mask.

#### .frame framereg, frameoffset, retreg[, argoffset]

Describes the shape of the stack frame. The frame pointer in use is framereg; normally this is either **\$fp** or **\$sp**. The frame pointer is frameoffset bytes below the CFA. The return address is initially located in *retreg* until it is saved as indicated in .mask. For compatibility with OSF/1 an optional argoffset parameter is accepted and ignored. It is believed to indicate the offset from the CFA to the saved argument registers.

#### .prologue n

Indicate that the stack frame is set up and all registers have been spilled. The argument n indicates whether and how the function uses the incoming procedure vector (the address of the called function) in \$27. 0 indicates that \$27 is not used; 1 indicates that the first two instructions of the function use \$27 to perform a load of the GP register; 2 indicates that \$27 is used in some non-standard way and so the linker cannot elide the load of the procedure vector during relaxation.

#### .usepv function, which

Used to indicate the use of the \$27 register, similar to .prologue, but without the other semantics of needing to be inside an open .ent/.end block.

The which argument should be either no, indicating that \$27 is not used, or std, indicating that the first two instructions of the function perform a GP load.

One might use this directive instead of .prologue if you are also using dwarf2 CFI directives.

#### .gprel32 expression

Computes the difference between the address in *expression* and the GP for the current object file, and stores it in 4 bytes. In addition to being smaller than a full 8 byte address, this also does not require a dynamic relocation when used in a shared library.

#### .t\_floating expression

Stores *expression* as an IEEE double precision value.

#### .s\_floating expression

Stores expression as an IEEE single precision value.

.f\_floating expression

Stores expression as a VAX F format value.

#### .g\_floating expression

Stores expression as a VAX G format value.

#### .d\_floating expression

Stores expression as a VAX D format value.

#### .set feature

Enables or disables various assembler features. Using the positive name of the feature enables while using 'nofeature' disables.

- at Indicates that macro expansions may clobber the assembler temporary (\$at or \$28) register. Some macros may not be expanded without this and will generate an error message if noat is in effect. When at is in effect, a warning will be generated if \$at is used by the programmer.
- macro Enables the expansion of macro instructions. Note that variants of real instructions, such as br label vs br \$31,label are considered alternate forms and not macros.

move
reorder
volatile These control whether and how the assembler may re-order instructions. Accepted for compatibility with the OSF/1 assembler, but
as does not do instruction scheduling, so these features are ignored.

The following directives are recognized for compatibility with the OSF/1 assembler but are ignored.

.proc	.aproc
.reguse	.livereg
.option	.aent
.ugen	.eflag
.alias	.noalias

## 9.2.6 Opcodes

For detailed information on the Alpha machine instruction set, see the Alpha Architecture Handbook located at

ftp://ftp.digital.com/pub/Digital/info/semiconductor/literature/alphaahb.pdf

## 9.3 ARC Dependent Features

## 9.3.1 Options

The following options control the type of CPU for which code is assembled, and generic constraints on the code generated:

-mcpu=*cpu* 

pu.						
	Set architecture type and register usage for $cpu$ . There are also shortcut alias options available for backward compatibility and convenience. Supported values for $cpu$ are					
	arc600	Assemble for ARC 600. Aliases: -mA6, -mARC600.				
	arc600_nor	Assemble for ARC 600 with norm instructions.				
	arc600_mu]	Assemble for ARC 600 with mul64 instructions.				
	arc600_mu]	L32x16 Assemble for ARC 600 with mul32x16 instructions.				
	arc601	Assemble for ARC 601. Alias: -mARC601.				
	arc601_nom	Assemble for ARC 601 with norm instructions.				
	arc601_mu]	arc601_mul64 Assemble for ARC 601 with mul64 instructions.				
	arc601_mu]	132x16 Assemble for ARC 601 with mul32x16 instructions.				
	arc700	Assemble for ARC 700. Aliases: -mA7, -mARC700.				
	arcem	Assemble for ARC EM. Aliases: -mEM				
	em	Assemble for ARC EM, identical as arcem variant.				
	em4	Assemble for ARC EM with code-density instructions.				
	em4_dmips					
		Assemble for ARC EM with code-density instructions.				
	em4_fpus	Assemble for ARC EM with code-density instructions.				
	em4_fpuda	Assemble for ARC EM with code-density, and double-precision assist instructions.				
	quarkse_en					
		Assemble for QuarkSE-EM cpu.				
	archs	Assemble for ARC HS. Aliases: -mHS, -mav2hs.				
	hs	Assemble for ARC HS.				
	hs34	Assemble for ARC HS34.				

hs38	Assemble	for	ARC	HS38.

hs38\_linux

Assemble for ARC HS38 with floating point support on.

nps400 Assemble for ARC 700 with NPS-400 extended instructions.

Note: the .cpu directive (see Section 9.3.3 [ARC Directives], page 112) can to be used to select a core variant from within assembly code.

- -EB This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.
- -EL This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor this is the default.

#### -mcode-density

This option turns on Code Density instructions. Only valid for ARC EM processors.

- -mrelax Enable support for assembly-time relaxation. The assembler will replace a longer version of an instruction with a shorter one, whenever it is possible.
- -mnps400 Enable support for NPS-400 extended instructions.
- -mspfp Enable support for single-precision floating point instructions.

-mdpfp Enable support for double-precision floating point instructions.

-mfpuda Enable support for double-precision assist floating point instructions. Only valid for ARC EM processors.

## 9.3.2 Syntax

## 9.3.2.1 Special Characters

- % A register name can optionally be prefixed by a '%' character. So register %r0 is equivalent to r0 in the assembly code.
- **#** The presence of a '**#**' character within a line (but not at the start of a line) indicates the start of a comment that extends to the end of the current line.

*Note:* if a line starts with a '#' character then it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

<sup>©</sup> Prefixing an operand with an '©' specifies that the operand is a symbol and not a register. This is how the assembler disambiguates the use of an ARC register name as a symbol. So the instruction

mov r0, @r0

moves the address of symbol r0 into register r0.

- " The "" (backtick) character is used to separate statements on a single line.
- Used as a separator to obtain a sequence of commands from a C preprocessor macro.

#### 9.3.2.2 Register Names

The ARC assembler uses the following register names for its core registers:

- r0-r31 The core general registers. Registers r26 through r31 have special functions, and are usually referred to by those synonyms.
- gp The global pointer and a synonym for r26.
- fp The frame pointer and a synonym for r27.
- sp The stack pointer and a synonym for r28.
- ilink1 For ARC 600 and ARC 700, the level 1 interrupt link register and a synonym for r29. Not supported for ARCv2.
- ilink For ARCv2, the interrupt link register and a synonym for r29. Not supported for ARC 600 and ARC 700.
- ilink2 For ARC 600 and ARC 700, the level 2 interrupt link register and a synonym for r30. Not supported for ARC v2.
- blink The link register and a synonym for r31.
- r32-r59 The extension core registers.
- lp\_count The loop count register.
- pcl The word aligned program counter.

In addition the ARC processor has a large number of *auxiliary registers*. The precise set depends on the extensions being supported, but the following baseline set are always defined:

identity	Processor	Identification	register.	Auxiliary	register	address	0x4.

- pc Program Counter. Auxiliary register address 0x6.
- status32 Status register. Auxiliary register address 0x0a.

bta Branch Target Address. Auxiliary register address 0x412.

- ecr Exception Cause Register. Auxiliary register address 0x403.
- int\_vector\_base

Interrupt Vector Base address. Auxiliary register address 0x25.

status32\_p0

Stored STATUS32 register on entry to level P0 interrupts. Auxiliary register address 0xb.

#### aux\_user\_sp

Saved User Stack Pointer. Auxiliary register address 0xd.

- eret Exception Return Address. Auxiliary register address 0x400.
- erbta BTA saved on exception entry. Auxiliary register address 0x401.
- erstatus STATUS32 saved on exception. Auxiliary register address 0x402.
- bcr\_ver Build Configuration Registers Version. Auxiliary register address 0x60.

```
bta_link_build
```

Build configuration for: BTA Registers. Auxiliary register address 0x63.

```
vecbase_ac_build
```

Build configuration for: Interrupts. Auxiliary register address 0x68.

rf\_build Build configuration for: Core Registers. Auxiliary register address 0x6e.

dccm\_build

DCCM RAM Configuration Register. Auxiliary register address 0xc1.

Additional auxiliary register names are defined according to the processor architecture version and extensions selected by the options.

## 9.3.3 ARC Machine Directives

The ARC version of **as** supports the following additional machine directives:

.lcomm symbol, length[, alignment]

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. Since *symbol* is not declared global, it is normally not visible to 1d. The optional third parameter, *alignment*, specifies the desired alignment of the symbol in the bss section, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be a power of two less than or equal to the size of the symbol, up to a maximum of 16.

### .lcommon symbol, length[, alignment]

The same as lcomm directive.

.cpu cpu The .cpu directive must be followed by the desired core version. Permitted values for CPU are:

ARC600 Assemble for the ARC600 instruction set.

arc600\_norm

Assemble for ARC 600 with norm instructions.

```
arc600_mul64
```

Assemble for ARC 600 with mul64 instructions.

```
arc600_mul32x16
```

Assemble for ARC 600 with mul32x16 instructions.

arc601 Assemble for ARC 601 instruction set.

```
arc601_norm
```

Assemble for ARC 601 with norm instructions.

arc601\_mul64

Assemble for ARC 601 with mul64 instructions.

```
arc601_mul32x16
```

Assemble for ARC 601 with mul32x16 instructions.

ARC700	Assemble for the ARC700 instruction set.
NPS400	Assemble for the NPS400 instruction set.
EM	Assemble for the ARC EM instruction set.
arcem	Assemble for ARC EM instruction set
em4	Assemble for ARC EM with code-density instructions.
em4_dmips	
_	Assemble for ARC EM with code-density instructions.
em4_fpus	Assemble for ARC EM with code-density instructions.
em4_fpuda	
	Assemble for ARC EM with code-density, and double-precision assist instructions.
quarkse_e	n
	Assemble for QuarkSE-EM instruction set.
HS	Assemble for the ARC HS instruction set.
archs	Assemble for ARC HS instruction set.
hs	Assemble for ARC HS instruction set.
hs34	Assemble for ARC HS34 instruction set.
hs38	Assemble for ARC HS38 instruction set.
hs38_linu	X
	Assemble for ARC HS38 with floating point support on.
Note: the	.cpu directive overrides the command-line option $-mcpu=cpu$ ; a

warning is emitted when the version is not consistent between the two.

#### .extAuxRegister name, addr, mode

Auxiliary registers can be defined in the assembler source code by using this directive. The first parameter, *name*, is the name of the new auxiliary register. The second parameter, *addr*, is address the of the auxiliary register. The third parameter, *mode*, specifies whether the register is readable and/or writable and is one of:

- r Read only;
- w Write only;
- r|w Read and write.

For example:

#### .extAuxRegister mulhi, 0x12, w

specifies a write only extension auxiliary register, mulhi at address 0x12.

#### .extCondCode suffix, val

ARC supports extensible condition codes. This directive defines a new condition code, to be known by the suffix, *suffix* and will depend on the value, *val* in the condition code.

For example:

.extCondCode is\_busy,0x14
add.is\_busy r1,r2,r3

will only execute the add instruction if the condition code value is 0x14.

#### .extCoreRegister name, regnum, mode, shortcut

Specifies an extension core register named *name* as a synonym for the register numbered *regnum*. The register number must be between 32 and 59. The third argument, *mode*, indicates whether the register is readable and/or writable and is one of:

r Read only;

w Write only;

**r**|**w** Read and write.

The final parameter, *shortcut* indicates whether the register has a short cut in the pipeline. The valid values are:

```
can_shortcut
```

The register has a short cut in the pipeline;

```
cannot_shortcut
```

The register does not have a short cut in the pipeline.

For example:

.extCoreRegister mlo, 57, r , can\_shortcut

defines a read only extension core register, mlo, which is register 57, and can short cut the pipeline.

#### .extInstruction name, opcode, subopcode, suffixclass, syntaxclass

ARC allows the user to specify extension instructions. These extension instructions are not macros; the assembler creates encodings for use of these instructions according to the specification by the user.

The first argument, name, gives the name of the instruction.

The second argument, *opcode*, is the opcode to be used (bits 31:27 in the encoding).

The third argument, *subopcode*, is the sub-opcode to be used, but the correct value also depends on the fifth argument, *syntaxclass* 

The fourth argument, *suffixclass*, determines the kinds of suffixes to be allowed. Valid values are:

#### SUFFIX\_NONE

No suffixes are permitted;

#### SUFFIX\_COND

Conditional suffixes are permitted;

#### SUFFIX\_FLAG

Flag setting suffixes are permitted.

#### SUFFIX\_COND|SUFFIX\_FLAG

Both conditional and flag setting suffices are permitted.

The fifth and final argument, *syntaxclass*, determines the syntax class for the instruction. It can have the following values:

#### SYNTAX\_20P

Two Operand Instruction;

#### SYNTAX\_30P

Three Operand Instruction.

#### SYNTAX\_10P

One Operand Instruction.

#### SYNTAX\_NOP

No Operand Instruction.

The syntax class may be followed by '|' and one of the following modifiers.

#### OP1\_MUST\_BE\_IMM

Modifies syntax class SYNTAX\_30P, specifying that the first operand of a three-operand instruction must be an immediate (i.e., the result is discarded). This is usually used to set the flags using specific instructions and not retain results.

#### OP1\_IMM\_IMPLIED

Modifies syntax class SYNTAX\_20P, specifying that there is an implied immediate destination operand which does not appear in the syntax.

For example, if the source code contains an instruction like:

#### inst r1,r2

the first argument is an implied immediate (that is, the result is discarded). This is the same as though the source code were: inst 0,r1,r2.

For example, defining a 64-bit multiplier with immediate operands:

.extInstruction mp64, 0x07, 0x2d, SUFFIX\_COND|SUFFIX\_FLAG, SYNTAX\_30P|OP1\_MUST\_BE\_IMM

which specifies an extension instruction named mp64 with 3 operands. It sets the flags and can be used with a condition code, for which the first operand is an immediate, i.e. equivalent to discarding the result of the operation.

A two operands instruction variant would be:

.extInstruction mul64, 0x07, 0x2d, SUFFIX\_COND, SYNTAX\_20P|0P1\_IMM\_IMPLIED

which describes a two operand instruction with an implicit first immediate operand. The result of this operation would be discarded.

#### .arc\_attribute tag, value

Set the ARC object attribute tag to value.

The tag is either an attribute number, or one of the following: Tag\_ARC\_PCS\_ config, Tag\_ARC\_CPU\_base, Tag\_ARC\_CPU\_variation, Tag\_ARC\_CPU\_name,

Tag\_ARC\_ABI\_rf16, Tag\_ARC\_ABI\_osver, Tag\_ARC\_ABI\_sda, Tag\_ARC\_ABI\_ pic, Tag\_ARC\_ABI\_tls, Tag\_ARC\_ABI\_enumsize, Tag\_ARC\_ABI\_exceptions, Tag\_ARC\_ABI\_double\_size, Tag\_ARC\_ISA\_config, Tag\_ARC\_ISA\_apex, Tag\_ARC\_ISA\_mpy\_option

The value is either a number, "string", or number, "string" depending on the tag.

## 9.3.4 ARC Assembler Modifiers

The following additional assembler modifiers have been added for position-independent code. These modifiers are available only with the ARC 700 and above processors and generate relocation entries, which are interpreted by the linker as follows:

#### @pcl(symbol)

Relative distance of symbol's from the current program counter location.

@gotpc(symbol)

Relative distance of *symbol*'s Global Offset Table entry from the current program counter location.

#### @gotoff(symbol)

Distance of symbol from the base of the Global Offset Table.

@plt(symbol)

Distance of *symbol*'s Procedure Linkage Table entry from the current program counter. This is valid only with branch and link instructions and PC-relative calls.

#### @sda(symbol)

Relative distance of symbol from the base of the Small Data Pointer.

## 9.3.5 ARC Pre-defined Symbols

The following assembler symbols will prove useful when developing position-independent code. These symbols are available only with the ARC 700 and above processors.

#### \_\_GLOBAL\_OFFSET\_TABLE\_\_

Symbol referring to the base of the Global Offset Table.

\_\_DYNAMIC\_\_

An alias for the Global Offset Table Base\_\_GLOBAL\_OFFSET\_TABLE\_\_. It can be used only with @gotpc modifiers.

## 9.3.6 Opcodes

For information on the ARC instruction set, see ARC Programmers Reference Manual, available where you download the processor IP library.

## 9.4 ARM Dependent Features

#### 9.4.1 Options

#### -mcpu=processor[+extension...]

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: arm1, arm2, arm250, arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m, arm7d, arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710t, arm720, arm720t, arm740t, arm710c, arm7100, arm7500, arm7500fe, arm7t, arm7tdmi, arm7tdmi-s, arm8, arm810, strongarm, strongarm1, strongarm110, strongarm1100, strongarm1110, arm9, arm920, arm920t, arm922t, arm940t, arm9tdmi, fa526 (Faraday FA526 processor), fa626 (Faraday FA626 processor), arm9e, arm926e, arm926ej-s, arm946e-r0, arm946e-s, arm966e-r0, arm966e, arm946e, arm966e-s, arm968e-s, arm10t, arm10tdmi, arm10e, arm1020, arm1020t, arm1020e, arm1022e, arm1026ej-s, fa606te (Faraday FA606TE processor), fa616te (Faraday FA616TE processor), fa626te (Faraday FA626TE processor), fmp626 (Faraday FMP626 processor), fa726te (Faraday FA726TE processor), arm1136j-s, arm1136jf-s, arm1156t2-s, arm1156t2f-s, arm1176jz-s, arm1176jzf-s, mpcore, mpcorenovfp, cortex-a5, cortex-a7, cortex-a8, cortex-a9, cortex-a15, cortex-a17, cortex-a32, cortex-a35, cortex-a53, cortex-a55, cortex-a57, cortex-a72, cortex-a73, cortex-a75, cortex-a76, cortex-a76ae, cortex-a77, cortex-a78, cortex-a78ae, cortex-a78c, cortex-a710, ares, cortex-r4, cortex-r4f, cortex-r5, cortex-r7, cortex-r8, cortex-r52, cortex-r52plus, cortex-m35p, cortex-m33, cortex-m23, cortex-m7, cortex-m4, cortex-m3, cortex-m1, cortex-m0, cortex-mOplus, cortex-x1, cortex-x1c, exynos-m1, marvell-pj4, marvell-whitney, neoverse-n1, neoverse-n2, neoverse-v1, xgene1, xgene2, ep9312 (ARM920 with Cirrus Maverick coprocessor), i80200 (Intel XScale processor) iwmmxt (Intel XScale processor with Wireless MMX technology coprocessor) and xscale. The special name all may be used to allow the assembler to accept instructions valid for any ARM processor.

In addition to the basic instruction set, the assembler can be told to accept various extension mnemonics that extend the processor using the co-processor instruction space. For example, -mcpu=arm920+maverick is equivalent to specifying -mcpu=ep9312.

Multiple extensions may be specified, separated by a +. The extensions should be specified in ascending alphabetical order.

Some extensions may be restricted to particular architectures; this is documented in the list of extensions below.

Extension mnemonics may also be removed from those the assembler accepts. This is done be prepending **no** to the option that adds the extension. Extensions that are removed should be listed after all extensions which have been added,

again in ascending alphabetical order. For example, -mcpu=ep9312+nomaverick is equivalent to specifying -mcpu=arm920.

The following extensions are currently supported: bf16 (BFloat16 extensions for v8.6-A architecture), i8mm (Int8 Matrix Multiply extensions for v8.6-A architecture), crc crypto (Cryptography Extensions for v8-A architecture, implies fp+simd), dotprod (Dot Product Extensions for v8.2-A architecture, implies fp+simd), fp (Floating Point Extensions for v8-A architecture), fp16 (FP16 Extensions for v8.2-A architecture, implies fp), fp16fml (FP16 Floating Point Multiplication Variant Extensions for v8.2-A architecture, implies fp16), idiv (Integer Divide Extensions for v7-A and v7-R architectures), iwmmxt, iwmmxt2, xscale, maverick, mp (Multiprocessing Extensions for v7-A and v7-R architectures), os (Operating System for v6M architecture), predres (Execution and Data Prediction Restriction Instruction for v8-A architectures, added by default from v8.5-A), sb (Speculation Barrier Instruction for v8-A architectures, added by default from v8.5-A), sec (Security Extensions for v6K and v7-A architectures), simd (Advanced SIMD Extensions for v8-A architecture, implies fp), virt (Virtualization Extensions for v7-A architecture, implies idiv), pan (Privileged Access Never Extensions for v8-A architecture), ras (Reliability, Availability and Serviceability extensions for v8-A architecture), rdma (ARMv8.1 Advanced SIMD extensions for v8-A architecture, implies simd) and xscale.

#### -march=architecture[+extension...]

This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: armv1, armv2, armv2a, armv2s, armv3, armv3m, armv4, armv4xm, armv4t, armv4txm, armv5, armv5t, armv5txm, armv5te, armv5texp, armv6, armv6j, armv6k, armv6z, armv6kz, armv6-m, armv6s-m, armv7, armv7-a, armv7ve, armv7-r, armv7-m, armv7e-m, armv8-a, armv8.1-a, armv8.2-a, armv8.3-a, armv8.4-a, armv8.5-a, armv8.8-a, armv9-a, iwmmxt, iwmmxt2 and xscale. If both -mcpu and -march are specified, the assembler will use the setting for -mcpu.

The architecture option can be extended with a set extension options. These extensions are context sensitive, i.e. the same extension may mean different things when used with different architectures. When used together with a -mfpu option, the union of both feature enablement is taken. See their availability and meaning below:

For armv5te, armv5texp, armv5tej, armv6, armv6j, armv6k, armv6z, armv6kz, armv6kz, armv6kt2 and armv6zt2:

+fp: Enables VFPv2 instructions. +nofp: Disables all FPU instrunctions.

For armv7:

+fp: Enables VFPv3 instructions with 16 double-word registers. +nofp: Disables all FPU instructions.

For armv7-a:

+fp: Enables VFPv3 instructions with 16 double-word registers. +vfpv3-d16: Alias for +fp. +vfpv3: Enables VFPv3 instructions with 32 double-word registers. +vfpv3-d16-fp16: Enables VFPv3 with half precision floating-point conversion instructions and 16 double-word registers. +vfpv3-fp16: Enables VFPv3 with half precision floating-point conversion instructions and 32 doubleword registers. +vfpv4-d16: Enables VFPv4 instructions with 16 double-word registers. +vfpv4-d16: Enables VFPv4 instructions with 32 double-word registers. +simd: Enables VFPv3 and NEONv1 instructions with 32 double-word registers. +simd: Enables VFPv3 and NEONv1 instructions with 32 double-word registers. +neon: Alias for +simd. +neon-vfpv3: Alias for +simd. +neon-fp16: Enables VFPv3, half precision floating-point conversion and NEONv1 instructions with 32 double-word registers. +neon-vfpv4: Enables VFPv4 and NEONv1 with Fused-MAC instructions and 32 double-word registers. +mp: Enables Multiprocessing Extensions. +sec: Enables Security Extensions. +nofp: Disables all FPU and NEON instructions. +nosimd: Disables all NEON instructions.

#### For armv7ve:

+fp: Enables VFPv4 instructions with 16 double-word registers. +vfpv4-d16: Alias for +fp. +vfpv3-d16: Enables VFPv3 instructions with 16 double-word registers. +vfpv3: Enables VFPv3 instructions with 32 double-word registers. +vfpv3-d16-fp16: Enables VFPv3 with half precision floating-point conversion instructions and 16 double-word registers. +vfpv3-fp16: Enables VFPv3 with half precision floating-point conversion instructions and 32 double-word registers. +vfpv4: Enables VFPv4 instructions with 32 double-word registers. +vfpv4: Enables VFPv4 instructions with 32 double-word registers. +simd: Enables VFPv4 and NEONv1 with Fused-MAC instructions and 32 doubleword registers. +neon-vfpv4: Alias for +simd. +neon: Enables VFPv3 and NEONv1 instructions with 32 double-word registers. +neon-vfpv3: Alias for +neon. +neon-fp16: Enables VFPv3, half precision floating-point conversion and NEONv1 instructions with 32 double-word registers. double-word registers. +nofp: Disables all FPU and NEON instructions. +nosimd: Disables all NEON instructions.

### For armv7-r:

+fp.sp: Enables single-precision only VFPv3 instructions with 16 double-word registers. +vfpv3xd: Alias for +fp.sp. +fp: Enables VFPv3 instructions with 16 double-word registers. +vfpv3-d16: Alias for +fp. +vfpv3xd-fp16: Enables single-precision only VFPv3 and half floating-point conversion instructions with 16 double-word registers. +vfpv3-d16-fp16: Enables VFPv3 and half precision floating-point conversion instructions with 16 double-word registers. +idiv: Enables integer division instructions in ARM mode. +nofp: Disables all FPU instructions.

#### For armv7e-m:

+fp: Enables single-precision only VFPv4 instructions with 16 double-word registers. +vfpvf4-sp-d16: Alias for +fp. +fpv5: Enables single-precision only VFPv5 instructions with 16 double-word registers. +fp.dp: Enables VFPv5 instructions with 16 double-word registers. +fpv5-d16": Alias for +fp.dp. +nofp: Disables all FPU instructions.

For armv8-m.main:

+dsp: Enables DSP Extension. +fp: Enables single-precision only VFPv5 instructions with 16 double-word registers. +fp.dp: Enables VFPv5 instructions with 16 double-word registers. +cdecp0 (CDE extensions for v8-m architecture with coprocessor 0), +cdecp1 (CDE extensions for v8-m architecture with coprocessor 1), +cdecp2 (CDE extensions for v8-m architecture with coprocessor 2), +cdecp3 (CDE extensions for v8-m architecture with coprocessor 3), +cdecp4 (CDE extensions for v8-m architecture with coprocessor 3), +cdecp4 (CDE extensions for v8-m architecture with coprocessor 4), +cdecp5 (CDE extensions for v8-m architecture with coprocessor 5), +cdecp6 (CDE extensions for v8-m architecture with coprocessor 5), +cdecp6 (CDE extensions for v8-m architecture with coprocessor 6), +cdecp7 (CDE extensions for v8-m architecture with coprocessor 7), +nofp: Disables all FPU instructions. +nodsp: Disables DSP Extension.

For armv8.1-m.main:

+dsp: Enables DSP Extension. +fp: Enables single and half precision scalar Floating Point Extensions for Armv8.1-M Mainline with 16 double-word registers. +fp.dp: Enables double precision scalar Floating Point Extensions for Armv8.1-M Mainline, implies +fp. +mve: Enables integer only M-profile Vector Extension for Armv8.1-M Mainline, implies +dsp. +mve.fp: Enables Floating Point Mprofile Vector Extension for Armv8.1-M Mainline, implies +mve and +fp. +nofp: Disables all FPU instructions. +nodsp: Disables DSP Extension. +nomve: Disables all M-profile Vector Extensions.

For armv8-a:

+crc: Enables CRC32 Extension. +simd: Enables VFP and NEON for Armv8-A.
A. +crypto: Enables Cryptography Extensions for Armv8-A, implies +simd.
+sb: Enables Speculation Barrier Instruction for Armv8-A. +predres: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. +nofp: Disables all FPU, NEON and Cryptography Extensions. +nocrypto: Disables Cryptography Extensions.

For armv8.1-a:

+simd: Enables VFP and NEON for Armv8.1-A. +crypto: Enables Cryptography Extensions for Armv8-A, implies +simd. +sb: Enables Speculation Barrier Instruction for Armv8-A. +predres: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. +nofp: Disables all FPU, NEON and Cryptography Extensions. +nocrypto: Disables Cryptography Extensions.

For armv8.2-a and armv8.3-a:

+simd: Enables VFP and NEON for Armv8.1-A. +fp16: Enables FP16 Extension for Armv8.2-A, implies +simd. +fp16fm1: Enables FP16 Floating Point Multiplication Variant Extensions for Armv8.2-A, implies +fp16. +crypto: Enables Cryptography Extensions for Armv8-A, implies +simd. +dotprod: Enables Dot Product Extensions for Armv8.2-A, implies +simd. +sb: Enables Speculation Barrier Instruction for Armv8-A. +predres: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. +nofp: Disables all FPU, NEON, Cryptography and Dot Product Extensions. +nocrypto: Disables Cryptography Extensions.

For armv8.4-a:

+simd: Enables VFP and NEON for Armv8.1-A and Dot Product Extensions for Armv8.2-A. +fp16: Enables FP16 Floating Point and Floating Point Multiplication Variant Extensions for Armv8.2-A, implies +simd. +crypto: Enables Cryptography Extensions for Armv8-A, implies +simd. +sb: Enables Speculation Barrier Instruction for Armv8-A. +predres: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. +nofp: Disables all FPU, NEON, Cryptography and Dot Product Extensions. +nocryptp: Disables Cryptography Extensions.

#### For armv8.5-a:

+simd: Enables VFP and NEON for Armv8.1-A and Dot Product Extensions for Armv8.2-A. +fp16: Enables FP16 Floating Point and Floating Point Multiplication Variant Extensions for Armv8.2-A, implies +simd. +crypto: Enables Cryptography Extensions for Armv8-A, implies +simd. +nofp: Disables all FPU, NEON, Cryptography and Dot Product Extensions. +nocryptp: Disables Cryptography Extensions.

#### -mfpu=floating-point-format

This option specifies the floating point format to assemble for. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target floating point unit. The following format options are recognized: softfpa, fpe, fpe2, fpe3, fpa, fpa10, fpa11, arm7500fe, softvfp, softvfp+vfp, vfp, vfp10, vfp10-r0, vfp9, vfpxd, vfpv2, vfpv3, vfpv3-fp16, vfpv3-d16, vfpv3-d16-fp16, vfpv3xd, vfpv3xd-d16, vfpv4, vfpv4-d16, fpv4-sp-d16, fpv5-sp-d16, fpv5-d16, fp-armv8, arm1020t, arm1020e, arm1136jf-s, maverick, neon, neon-vfpv3, neon-fp16, neon-vfpv4, neon-fp-armv8, crypto-neon-fp-armv8, neon-fp-armv8.1 and crypto-neon-fp-armv8.1.

In addition to determining which instructions are assembled, this option also affects the way in which the .double assembler directive behaves when assembling little-endian code.

The default is dependent on the processor selected. For Architecture 5 or later, the default is to assemble for VFP instructions; for earlier architectures the default is to assemble for FPA instructions.

#### -mfp16-format=format

This option specifies the half-precision floating point format to use when assembling floating point numbers emitted by the .float16 directive. The following format options are recognized: ieee, alternative. If ieee is specified then the IEEE 754-2008 half-precision floating point format is used, if alternative is specified then the Arm alternative half-precision format is used. If this option is set on the command line then the format is fixed and cannot be changed with the float16\_format directive. If this value is not set then the IEEE 754-2008 format is used until the format is explicitly set with the float16\_format directive.

-mthumb This option specifies that the assembler should start assembling Thumb instructions; that is, it should behave as though the file starts with a .code 16 directive.

#### -mthumb-interwork

This option specifies that the output generated by the assembler should be marked as supporting interworking. It also affects the behaviour of the ADR and ADRL pseudo opcodes.

#### -mimplicit-it=never

```
-mimplicit-it=always
```

## -mimplicit-it=arm

-mimplicit-it=thumb

The -mimplicit-it option controls the behavior of the assembler when conditional instructions are not enclosed in IT blocks. There are four possible behaviors. If never is specified, such constructs cause a warning in ARM code and an error in Thumb-2 code. If always is specified, such constructs are accepted in both ARM and Thumb-2 code, where the IT instruction is added implicitly. If arm is specified, such constructs are accepted in ARM code and cause an error in Thumb-2 code. If thumb is specified, such constructs cause a warning in ARM code and are accepted in Thumb-2 code. If you omit this option, the behavior is equivalent to -mimplicit-it=arm.

#### -mapcs-26

-mapcs-32

These options specify that the output generated by the assembler should be marked as supporting the indicated version of the Arm Procedure. Calling Standard.

-matpcs This option specifies that the output generated by the assembler should be marked as supporting the Arm/Thumb Procedure Calling Standard. If enabled this option will cause the assembler to create an empty debugging section in the object file called .arm.atpcs. Debuggers can use this to determine the ABI being used by.

#### -mapcs-float

This indicates the floating point variant of the APCS should be used. In this variant floating point arguments are passed in FP registers rather than integer registers.

#### -mapcs-reentrant

This indicates that the reentrant variant of the APCS should be used. This variant supports position independent code.

-mfloat-abi=abi

This option specifies that the output generated by the assembler should be marked as using specified floating point ABI. The following values are recognized: soft, softfp and hard.

#### -meabi=ver

This option specifies which EABI version the produced object files should conform to. The following values are recognized: gnu, 4 and 5.

-EB This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.

Note: If a program is being built for a system with big-endian data and littleendian instructions then it should be assembled with the -EB option, (all of it, code and data) and then linked with the --be8 option. This will reverse the endianness of the instructions back to little-endian, but leave the data as big-endian.

- -EL This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor.
- -k This option specifies that the output of the assembler should be marked as position-independent code (PIC).
- --fix-v4bx

Allow BX instructions in ARMv4 code. This is intended for use with the linker option of the same name.

#### -mwarn-deprecated

```
-mno-warn-deprecated
```

Enable or disable warnings about using deprecated options or features. The default is to warn.

-mccs Turns on CodeComposer Studio assembly syntax compatibility mode.

#### -mwarn-syms

-mno-warn-syms

Enable or disable warnings about symbols that match the names of ARM instructions. The default is to warn.

## 9.4.2 Syntax

## 9.4.2.1 Instruction Set Syntax

Two slightly different syntaxes are support for ARM and THUMB instructions. The default, divided, uses the old style where ARM and THUMB instructions had their own, separate syntaxes. The new, unified syntax, which can be selected via the .syntax directive, and has the following main features:

- Immediate operands do not require a **#** prefix.
- The IT instruction may appear, and if it does it is validated against subsequent conditional affixes. In ARM mode it does not generate machine code, in THUMB mode it does.
- For ARM instructions the conditional affixes always appear at the end of the instruction. For THUMB instructions conditional affixes can be used, but only inside the scope of an IT instruction.
- All of the instructions new to the V6T2 architecture (and later) are available. (Only a few such instructions can be written in the divided syntax).
- The .N and .W suffixes are recognized and honored.
- All instructions set the flags if and only if they have an **s** affix.

## 9.4.2.2 Special Characters

The presence of a '@' anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used instead of a newline to separate statements.

Either '#' or '\$' can be used to indicate immediate operands.

\*TODO\* Explain about /data modifier on symbols.

## 9.4.2.3 Register Names

\*TODO\* Explain about ARM register naming, and the predefined names.

## 9.4.2.4 ARM relocation generation

Specific data relocations can be generated by putting the relocation name in parentheses after the symbol name. For example:

.word foo(TARGET1)

This will generate an 'R\_ARM\_TARGET1' relocation against the symbol foo. The following relocations are supported: GOT, GOTOFF, TARGET1, TARGET2, SBREL, TLSGD, TLSLDM, TLSLDO, TLSDESC, TLSCALL, GOTTPOFF, GOT\_PREL and TPOFF.

For compatibility with older toolchains the assembler also accepts (PLT) after branch targets. On legacy targets this will generate the deprecated 'R\_ARM\_PLT32' relocation. On EABI targets it will encode either the 'R\_ARM\_CALL' or 'R\_ARM\_JUMP24' relocation, as appropriate.

Relocations for 'MOVW' and 'MOVT' instructions can be generated by prefixing the value with '#:lower16:' and '#:upper16' respectively. For example to load the 32-bit address of foo into r0:

```
MOVW r0, #:lower16:foo
MOVT r0, #:upper16:foo
```

Relocations 'R\_ARM\_THM\_ALU\_ABS\_GO\_NC', 'R\_ARM\_THM\_ALU\_ABS\_G1\_NC', 'R\_ARM\_THM\_ALU\_ABS\_G2\_NC' and 'R\_ARM\_THM\_ALU\_ABS\_G3\_NC' can be generated by prefixing the value with '#:lower0\_7:#', '#:lower8\_15:#', '#:upper0\_7:#' and '#:upper8\_15:#' respectively. For example to load the 32-bit address of foo into r0:

```
MOVS r0, #:upper8_15:#foo
LSLS r0, r0, #8
ADDS r0, #:upper0_7:#foo
LSLS r0, r0, #8
ADDS r0, #:lower8_15:#foo
LSLS r0, r0, #8
ADDS r0, #:lower0_7:#foo
```

## 9.4.2.5 NEON Alignment Specifiers

Some NEON load/store instructions allow an optional address alignment qualifier. The ARM documentation specifies that this is indicated by '@ align'. However GAS already

interprets the '**C**' character as a "line comment" start, so ': **align**' is used instead. For example:

vld1.8 {q0}, [r0, :128]

## 9.4.3 Floating Point

The ARM family uses IEEE floating-point numbers.

## 9.4.4 ARM Machine Directives

#### .align expression [, expression]

This is the generic .*align* directive. For the ARM however if the first argument is zero (ie no alignment is needed) the assembler will behave as if the argument had been 2 (ie pad to the next four byte boundary). This is for compatibility with ARM's own assembler.

.arch name

Select the target architecture. Valid values for *name* are the same as for the **-march** command-line option without the instruction set extension.

Specifying .arch clears any previously selected architecture extensions.

.arch\_extension name

Add or remove an architecture extension to the target architecture. Valid values for *name* are the same as those accepted as architectural extensions by the **-mcpu** and **-march** command-line options.

.arch\_extension may be used multiple times to add or remove extensions incrementally to the architecture being compiled for.

- .arm This performs the same action as .code 32.
- .bss This directive switches to the .bss section.
- .cantunwind

Prevents unwinding through the current function. No personality routine or exception table data is required or permitted.

.code [16|32]

This directive selects the instruction set being generated. The value 16 selects Thumb, with the value 32 selecting ARM.

.cpu name Select the target processor. Valid values for name are the same as for the -mcpu command-line option without the instruction set extension.

Specifying .cpu clears any previously selected architecture extensions.

#### name .dn register name [.type] [[index]]

#### name .qn register name [.type] [[index]]

The dn and qn directives are used to create typed and/or indexed register aliases for use in Advanced SIMD Extension (Neon) instructions. The former should be used to create aliases of double-precision registers, and the latter to create aliases of quad-precision registers.

If these directives are used to create typed aliases, those aliases can be used in Neon instructions instead of writing types after the mnemonic or after each operand. For example: x .dn d2.f32
y .dn d3.f32
z .dn d4.f32[1]
vmul x,y,z

This is equivalent to writing the following:

vmul.f32 d2,d3,d4[1]

Aliases created using dn or qn can be destroyed using unreq.

.eabi\_attribute tag, value

Set the EABI object attribute tag to value.

The tag is either an attribute number, or one of the following: Tag\_ Tag\_CPU\_name, Tag\_CPU\_arch, Tag\_CPU\_arch\_profile, CPU\_raw\_name, Tag\_ARM\_ISA\_use, Tag\_THUMB\_ISA\_use, Tag\_FP\_arch, Tag\_WMMX\_ arch, Tag\_Advanced\_SIMD\_arch, Tag\_MVE\_arch, Tag\_PCS\_config, Tag\_ABI\_PCS\_R9\_use, Tag\_ABI\_PCS\_RW\_data, Tag\_ABI\_PCS\_RO\_data, Tag\_ABI\_PCS\_GOT\_use, Tag\_ABI\_PCS\_wchar\_t, Tag\_ABI\_FP\_rounding, Tag\_ ABI\_FP\_denormal, Tag\_ABI\_FP\_exceptions, Tag\_ABI\_FP\_user\_exceptions, Tag\_ABI\_align\_needed, Tag\_ABI\_FP\_number\_model, Tag\_ABI\_align\_ preserved, Tag\_ABI\_enum\_size, Tag\_ABI\_HardFP\_use, Tag\_ABI\_VFP\_args, Tag\_ABI\_WMMX\_args, Tag\_ABI\_optimization\_goals, Tag\_ABI\_FP\_ optimization\_goals, Tag\_compatibility, Tag\_CPU\_unaligned\_access, Tag\_ABI\_FP\_16bit\_format, Tag\_FP\_HP\_extension, Tag\_MPextension\_ use. Tag\_DIV\_use, Tag\_nodefaults, Tag\_also\_compatible\_with, Tag\_conformance, Tag\_T2EE\_use, Tag\_Virtualization\_use

The value is either a number, "string", or number, "string" depending on the tag.

Note - the following legacy values are also accepted by *tag*: Tag\_VFP\_arch, Tag\_ABI\_align8\_needed, Tag\_ABI\_align8\_preserved, Tag\_VFP\_HP\_extension,

.even This directive aligns to an even-numbered address.

#### .extend expression [, expression]\*

.ldouble expression [, expression]\*

These directives write 12byte long double floating-point values to the output section. These are not compatible with current ARM processors or ABIs.

.float16 value [,...,value\_n]

Place the half precision floating point representation of one or more floating-point values into the current section. The exact format of the encoding is specified by .float16\_format. If the format has not been explicitly set yet (either via the .float16\_format directive or the command line option) then the IEEE 754-2008 format is used.

#### .float16\_format format

Set the format to use when encoding float16 values emitted by the .float16 directive. Once the format has been set it cannot be changed. format should be one of the following: ieee (encode in the IEEE 754-2008 half precision format) or alternative (encode in the Arm alternative half precision format).

.fnend Marks the end of a function with an unwind table entry. The unwind index table entry is created when this directive is processed.If no personality routine has been specified then standard personality routine 0

or 1 will be used, depending on the number of unwind opcodes required.

- .fnstart Marks the start of a function with an unwind table entry.
- .force\_thumb

This directive forces the selection of Thumb instructions, even if the target processor does not support those instructions

.fpu name Select the floating-point unit to assemble for. Valid values for name are the same as for the -mfpu command-line option.

#### .handlerdata

Marks the end of the current function, and the start of the exception table entry for that function. Anything between this directive and the .fnend directive will be added to the exception table entry.

Must be preceded by a .personality or .personalityindex directive.

.inst opcode [, ...]

- .inst.n opcode [, ...]
- .inst.w opcode [ , ... ]

Generates the instruction corresponding to the numerical value *opcode*. .inst.n and .inst.w allow the Thumb instruction size to be specified explicitly, overriding the normal encoding rules.

### .ldouble expression [, expression]\* See .extend.

.ltorg This directive causes the current contents of the literal pool to be dumped into the current section (which is assumed to be the .text section) at the current location (aligned to a word boundary). GAS maintains a separate literal pool for each section and each sub-section. The .ltorg directive will only affect the literal pool of the current section and sub-section. At the end of assembly all remaining, un-empty literal pools will automatically be dumped.

Note - older versions of GAS would dump the current literal pool any time a section change occurred. This is no longer done, since it prevents accurate control of the placement of literal pools.

## .movsp reg [, #offset]

Tell the unwinder that *reg* contains an offset from the current stack pointer. If *offset* is not specified then it is assumed to be zero.

.object\_arch name

Override the architecture recorded in the EABI object attribute section. Valid values for *name* are the same as for the .arch directive. Typically this is useful when code uses runtime detection of CPU features.

## .packed expression [, expression]\*

This directive writes 12-byte packed floating-point values to the output section. These are not compatible with current ARM processors or ABIs.

#### .pacspval

Generate unwinder annotations to use effective vsp as modifier in PAC validation.

#### .pad #count

Generate unwinder annotations for a stack adjustment of *count* bytes. A positive value indicates the function prologue allocated stack space by decrementing the stack pointer.

### .personality name

Sets the personality routine for the current function to name.

#### .personalityindex index

Sets the personality routine for the current function to the EABI standard routine number *index* 

.pool This is a synonym for .ltorg.

#### name .req register name

This creates an alias for register name called name. For example:

foo .req r0

### .save reglist

Generate unwinder annotations to restore the registers in *reglist*. The format of *reglist* is the same as the corresponding store-multiple instruction.

```
core registers
        .save {r4, r5, r6, lr}
        stmfd sp!, {r4, r5, r6, lr}
FPA registers
        .save f4, 2
        sfmfd f4, 2, [sp]!
VFP registers
        .save {d8, d9, d10}
        fstmdx sp!, {d8, d9, d10}
iWMMXt registers
        .save {wr10, wr11}
        wstrd wr11, [sp, #-8]!
        wstrd wr10, [sp, #-8]!
      or
        .save wr11
        wstrd wr11, [sp, #-8]!
        .save wr10
        wstrd wr10, [sp, #-8]!
```

## .setfp fpreg, spreg [, #offset]

Make all unwinder annotations relative to a frame pointer. Without this the unwinder will use offsets from the stack pointer.

The syntax of this directive is the same as the add or mov instruction used to set the frame pointer. spreg must be either sp or mentioned in a previous .movsp directive.

```
.movsp ip
mov ip, sp
...
.setfp fp, ip, #4
add fp, ip, #4
```

```
.secrel32 expression [, expression]*
```

This directive emits relocations that evaluate to the section-relative offset of each expression's symbol. This directive is only supported for PE targets.

### .syntax [unified | divided]

This directive sets the Instruction Set Syntax as described in the Section 9.4.2.1 [ARM-Instruction-Set], page 123, section.

.thumb This performs the same action as .code 16.

### .thumb\_func

This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking between Arm and Thumb instructions and should be used even if interworking is not going to be performed. The presence of this directive also implies .thumb

This directive is not necessary when generating EABI objects. On these targets the encoding is implicit when generating Thumb code.

#### .thumb\_set

This performs the equivalent of a .set directive in that it creates a symbol which is an alias for another symbol (possibly not yet defined). This directive also has the added property in that it marks the aliased symbol as being a thumb function entry point, in the same way that the .thumb\_func directive does.

## .tlsdescseq tls-variable

This directive is used to annotate parts of an inlined TLS descriptor trampoline. Normally the trampoline is provided by the linker, and this directive is not needed.

### .unreq alias-name

This undefines a register alias which was previously defined using the req, dn or qn directives. For example:

foo .req r0 .unreq foo

An error occurs if the name is undefined. Note - this pseudo op can be used to delete builtin in register name aliases (eg 'r0'). This should only be done if it is really necessary.

### .unwind\_raw offset, byte1, ...

Insert one of more arbitrary unwind opcode bytes, which are known to adjust the stack pointer by *offset* bytes.

For example .unwind\_raw 4, 0xb1, 0x01 is equivalent to .save {r0}

### .vsave vfp-reglist

Generate unwinder annotations to restore the VFP registers in *vfp-reglist* using FLDMD. Also works for VFPv3 registers that are to be restored using VLDM. The format of *vfp-reglist* is the same as the corresponding store-multiple instruction.

 $VFP \ registers$ 

.vsave {d8, d9, d10} fstmdd sp!, {d8, d9, d10} VFPv3 registers .vsave {d15, d16, d17} vstm sp!, {d15, d16, d17}

Since FLDMX and FSTMX are now deprecated, this directive should be used in favour of .save for saving VFP registers for ARMv6 and above.

# 9.4.5 Opcodes

**as** implements all the standard ARM opcodes. It also implements several pseudo opcodes, including several synthetic load instructions.

NOP

#### nop

This pseudo op will always evaluate to a legal ARM instruction that does nothing. Currently it will evaluate to MOV r0, r0.

### LDR

ldr <register> , = <expression>

If expression evaluates to a numeric constant then a MOV or MVN instruction will be used in place of the LDR instruction, if the constant can be generated by either of these instructions. Otherwise the constant will be placed into the nearest literal pool (if it not already there) and a PC relative LDR instruction will be generated.

ADR

### adr <register> <label>

This instruction will load the address of *label* into the indicated register. The instruction will evaluate to a PC relative ADD or SUB instruction depending upon where the label is located. If the label is out of range, or if it is not defined in the same file (and section) as the ADR instruction, then an error will be generated. This instruction will not make use of the literal pool.

If *label* is a thumb function symbol, and thumb interworking has been enabled via the -mthumb-interwork option then the bottom bit of the value stored into *register* will be set. This allows the following sequence to work as expected:

adr r0, thumb\_function blx r0

ADRL

#### adrl <register> <label>

This instruction will load the address of *label* into the indicated register. The instruction will evaluate to one or two PC relative ADD or SUB instructions depending upon where the label is located. If a second instruction is not needed a NOP instruction will be generated in its place, so that this instruction is always 8 bytes long.

If the label is out of range, or if it is not defined in the same file (and section) as the ADRL instruction, then an error will be generated. This instruction will not make use of the literal pool.

If *label* is a thumb function symbol, and thumb interworking has been enabled via the -mthumb-interwork option then the bottom bit of the value stored into *register* will be set.

For information on the ARM or Thumb instruction sets, see ARM Software Development Toolkit Reference Manual, Advanced RISC Machines Ltd.

## 9.4.6 Mapping Symbols

The ARM ELF specification requires that special symbols be inserted into object files to mark certain features:

**\$a** At the start of a region of code containing ARM instructions.

**\$t** At the start of a region of code containing THUMB instructions.

**\$d** At the start of a region of data.

The assembler will automatically insert these symbols for you - there is no need to code them yourself. Support for tagging symbols (\$b, \$f, \$p and \$m) which is also mentioned in the current ARM ELF specification is not implemented. This is because they have been dropped from the new EABI and so tools cannot rely upon their presence.

## 9.4.7 Unwinding

The ABI for the ARM Architecture specifies a standard format for exception unwind information. This information is used when an exception is thrown to determine where control should be transferred. In particular, the unwind information is used to determine which function called the function that threw the exception, and which function called that one, and so forth. This information is also used to restore the values of callee-saved registers in the function catching the exception.

If you are writing functions in assembly code, and those functions call other functions that throw exceptions, you must use assembly pseudo ops to ensure that appropriate exception unwind information is generated. Otherwise, if one of the functions called by your assembly code throws an exception, the run-time library will be unable to unwind the stack through your assembly code and your program will not behave correctly.

To illustrate the use of these pseudo ops, we will examine the code that G++ generates for the following C++ input:

```
void callee (int *);
```

```
int
caller ()
{
    int i;
    callee (&i);
    return i;
}
```

This example does not show how to throw or catch an exception from assembly code. That is a much more complex operation and should always be done in a high-level language, such as C++, that directly supports exceptions. The code generated by one particular version of G++ when compiling the example above is:

```
_Z6callerv:
        .fnstart
.LFB2:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
                 sp!, {fp, lr}
        stmfd
        .save {fp, lr}
.LCFIO:
        .setfp fp, sp, #4
                 fp, sp, #4
        add
.LCFI1:
        .pad #8
        sub
                 sp, sp, #8
.LCFI2:
        sub
                 r3, fp, #8
                 r0, r3
        mov
                 _Z6calleePi
        bl
                 r3, [fp, #-8]
        ldr
                 r0, r3
        mov
        sub
                 sp, fp, #4
        ldmfd
                 sp!, {fp, lr}
        bx
                 lr
.LFE2:
        .fnend
```

Of course, the sequence of instructions varies based on the options you pass to GCC and on the version of GCC in use. The exact instructions are not important since we are focusing on the pseudo ops that are used to generate unwind information.

An important assumption made by the unwinder is that the stack frame does not change during the body of the function. In particular, since we assume that the assembly code does not itself throw an exception, the only point where an exception can be thrown is from a call, such as the bl instruction above. At each call site, the same saved registers (including lr, which indicates the return address) must be located in the same locations relative to the frame pointer.

The .fnstart (see [.fnstart pseudo op], page 127) pseudo op appears immediately before the first instruction of the function while the .fnend (see [.fnend pseudo op], page 126) pseudo op appears immediately after the last instruction of the function. These pseudo ops specify the range of the function.

Only the order of the other pseudos ops (e.g., .setfp or .pad) matters; their exact locations are irrelevant. In the example above, the compiler emits the pseudo ops with particular instructions. That makes it easier to understand the code, but it is not required for correctness. It would work just as well to emit all of the pseudo ops other than .fnend in the same order, but immediately after .fnstart.

The .save (see [.save pseudo op], page 128) pseudo op indicates registers that have been saved to the stack so that they can be restored before the function returns. The argument to the .save pseudo op is a list of registers to save. If a register is "callee-saved" (as specified by the ABI) and is modified by the function you are writing, then your code must save the value before it is modified and restore the original value before the function returns. If an exception is thrown, the run-time library restores the values of these registers from their locations on the stack before returning control to the exception handler. (Of course, if an exception is not thrown, the function that contains the .save pseudo op restores these registers in the function epilogue, as is done with the ldmfd instruction above.)

You do not have to save callee-saved registers at the very beginning of the function and you do not need to use the .save pseudo op immediately following the point at which the registers are saved. However, if you modify a callee-saved register, you must save it on the stack before modifying it and before calling any functions which might throw an exception. And, you must use the .save pseudo op to indicate that you have done so.

The .pad (see [.pad], page 128) pseudo op indicates a modification of the stack pointer that does not save any registers. The argument is the number of bytes (in decimal) that are subtracted from the stack pointer. (On ARM CPUs, the stack grows downwards, so subtracting from the stack pointer increases the size of the stack.)

The .setfp (see [.setfp pseudo op], page 128) pseudo op indicates the register that contains the frame pointer. The first argument is the register that is set, which is typically fp. The second argument indicates the register from which the frame pointer takes its value. The third argument, if present, is the value (in decimal) added to the register specified by the second argument to compute the value of the frame pointer. You should not modify the frame pointer in the body of the function.

If you do not use a frame pointer, then you should not use the .setfp pseudo op. If you do not use a frame pointer, then you should avoid modifying the stack pointer outside of the function prologue. Otherwise, the run-time library will be unable to find saved registers when it is unwinding the stack.

The pseudo ops described above are sufficient for writing assembly code that calls functions which may throw exceptions. If you need to know more about the object-file format used to represent unwind information, you may consult the *Exception Handling ABI* for the *ARM Architecture* available from http://infocenter.arm.com.

# 9.5 AVR Dependent Features

# 9.5.1 Options

### -mmcu=mcu

Specify ATMEL AVR instruction set or MCU type.

Instruction set avr1 is for the minimal AVR core, not supported by the C compiler, only for assembler programs (MCU types: at90s1200, attiny11, attiny12, attiny15, attiny28).

Instruction set avr2 (default) is for the classic AVR core with up to 8K program memory space (MCU types: at90s2313, at90s2323, at90s2333, at90s2343, attiny22, attiny26, at90s4414, at90s4433, at90s4434, at90s8515, at90c8534, at90s8535).

Instruction set avr25 is for the classic AVR core with up to 8K program memory space plus the MOVW instruction (MCU types: attiny13, attiny13a, attiny2313, attiny2313a, attiny24, attiny24a, attiny4313, attiny44, attiny44a, attiny84, attiny84a, attiny25, attiny45, attiny85, attiny261, attiny261a, attiny461, attiny461a, attiny861a, attiny861a, attiny87, attiny43u, attiny48, attiny88, attiny828, at86rf401, ata6289, ata5272).

Instruction set avr3 is for the classic AVR core with up to 128K program memory space (MCU types: at43usb355, at76c711).

Instruction set avr31 is for the classic AVR core with exactly 128K program memory space (MCU types: atmega103, at43usb320).

Instruction set avr35 is for classic AVR core plus MOVW, CALL, and JMP instructions (MCU types: attiny167, attiny1634, at90usb82, at90usb162, atmega8u2, atmega16u2, atmega32u2, ata5505).

Instruction set avr4 is for the enhanced AVR core with up to 8K program memory space (MCU types: atmega48, atmega48a, atmega48pa, atmega48p, atmega8, atmega8a, atmega8a, atmega88a, atmega88p, atmega88pa, atmega8515, atmega8535, atmega8hva, at90pwm1, at90pwm2, at90pwm2b, at90pwm3, at90pwm3b, at90pwm81, ata6285, ata6286).

Instruction set avr5 is for the enhanced AVR core with up to 128K program memory space (MCU types: at90pwm161, atmega16, atmega16a, atmega161, atmega162, atmega163, atmega164a, atmega164p, atmega164pa, atmega165, atmega165a, atmega165p, atmega165pa, atmega168, atmega168a, atmega168p, atmega168pa, atmega169, atmega169a, atmega169p, atmega169pa, atmega32, atmega323, atmega324a, atmega324p, atmega324pa, atmega325, atmega325a, atmega32, atmega32a, atmega323, atmega324a, atmega324p, atmega324pa, atmega325, atmega325a, atmega325p, atmega325pa, atmega3250a, atmega3250p, atmega3250, atmega3250pa, atmega328, atmega328p, atmega329a, atmega329a, atmega329pa, atmega329pa, atmega3290a, atmega3290p, atmega3290pa, atmega406, atmega64, atmega64a, atmega64rfr2, atmega644rfr2, atmega640, atmega644, atmega644a, atmega644p, atmega644pa, atmega645, atmega645a, atmega645p, atmega6450, atmega6450a, atmega6450p, atmega649, atmega649a, atmega649p, atmega6490, atmega6490a, atmega6490p, atmega16hva, atmega16hva2, atmega16hvb, atmega16hvbrevb, atmega32hvb, atmega32hvbrevb, atmega64hve, at90can32, at90can64, at90pwm161, at90pwm216, at90pwm316, atmega32c1, atmega64c1, atmega16m1, atmega32m1, atmega64m1, atmega16u4, atmega32u4, atmega32u6, at90usb646, at90usb647, at94k, at90scr100, ata5790, ata5795).

Instruction set avr51 is for the enhanced AVR core with exactly 128K program memory space (MCU types: atmega128, atmega128a, atmega1280, atmega1281, atmega1284, atmega1284p, atmega128rfa1, atmega128rfr2, atmega1284rfr2, at90can128, at90usb1286, at90usb1287, m3000).

Instruction set avr6 is for the enhanced AVR core with a 3-byte PC (MCU types: atmega2560, atmega2561, atmega256rfr2, atmega2564rfr2).

Instruction set avrxmega2 is for the XMEGA AVR core with 8K to 64K program memory space and less than 64K data space (MCU types: atxmega16a4, atxmega16a4u, atxmega16c4, atxmega16d4, atxmega16x1, atxmega32a4u, atxmega32a4u, atxmega32c4, atxmega32d4, atxmega16e5, atxmega8e5, atxmega32e5, atxmega32x1).

Instruction set avrxmega3 is for the XMEGA AVR core with up to 64K of combined program memory and RAM, and with program memory visible in the RAM address space (MCU types: attiny212, attiny214, attiny412, attiny414, attiny416, attiny417, attiny814, attiny816, attiny817, attiny1614, attiny1616, attiny1617, attiny3214, attiny3216, attiny3217).

Instruction set avrxmega4 is for the XMEGA AVR core with up to 64K program memory space and less than 64K data space (MCU types: atxmega64a3, atxmega64a3u, atxmega64a4u, atxmega64b1, atxmega64b3, atxmega64c3, atxmega64d3, atxmega64d4).

Instruction set avrxmega5 is for the XMEGA AVR core with up to 64K program memory space and greater than 64K data space (MCU types: atxmega64a1, atxmega64a1u).

Instruction set avrxmega6 is for the XMEGA AVR core with larger than 64K program memory space and less than 64K data space (MCU types: atxmega128a3, atxmega128a3u, atxmega128c3, atxmega128d4, atxmega192a3, atxmega192a3u, atxmega128b1, atxmega128b3, atxmega192c3, atxmega192d3, atxmega256a3, atxmega256a3u, atxmega256a3bu, atxmega256c3, atxmega256d3, atxmega256d3, atxmega256d3).

Instruction set avrxmega7 is for the XMEGA AVR core with larger than 64K program memory space and greater than 64K data space (MCU types: atxmega128a1, atxmega128a1u, atxmega128a4u).

Instruction set avrting is for the ATtiny4/5/9/10/20/40 microcontrollers.

#### -mall-opcodes

Accept all AVR opcodes, even if not supported by -mmcu.

#### -mno-skip-bug

This option disable warnings for skipping two-word instructions.

#### -mno-wrap

This option reject rjmp/rcall instructions with 8K wrap-around.

-mrmw Accept Read-Modify-Write (XCH, LAC, LAS, LAT) instructions.

### -mlink-relax

Enable support for link-time relaxation. This is now on by default and this flag no longer has any effect.

### -mno-link-relax

Disable support for link-time relaxation. The assembler will resolve relocations when it can, and may be able to better compress some debug information.

### -mgcc-isr

Enable the <u>\_\_gcc\_isr</u> pseudo instruction.

### -mno-dollar-line-separator

Do not treat the \$ character as a line separator character. This is for languages where \$ is valid character inside symbol names.

## 9.5.2 Syntax

## 9.5.2.1 Special Characters

The presence of a ';' anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The '\$' character can be used instead of a newline to separate statements. Note: the -mno-dollar-line-separator option disables this behaviour.

## 9.5.2.2 Register Names

The AVR has 32 x 8-bit general purpose working registers 'r0', 'r1', ... 'r31'. Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing. One of the these address pointers can also be used as an address pointer for look up tables in Flash program memory. These added function registers are the 16-bit 'X', 'Y' and 'Z' - registers.

```
X = r26:r27
Y = r28:r29
Z = r30:r31
```

## 9.5.2.3 Relocatable Expression Modifiers

The assembler supports several modifiers when using relocatable addresses in AVR instruction operands. The general syntax is the following:

```
modifier(relocatable-expression)
```

108

This modifier allows you to use bits 0 through 7 of an address expression as an 8 bit relocatable expression.

This modifier allows you to use bits 7 through 15 of an address expression as an 8 bit relocatable expression. This is useful with, for example, the AVR 'ldi' instruction and 'lo8' modifier.

For example

```
ldi r26, lo8(sym+10)
ldi r27, hi8(sym+10)
```

### hh8

This modifier allows you to use bits 16 through 23 of an address expression as an 8 bit relocatable expression. Also, can be useful for loading 32 bit constants.

### hlo8

Synonym of 'hh8'.

### hhi8

This modifier allows you to use bits 24 through 31 of an expression as an 8 bit expression. This is useful with, for example, the AVR 'ldi' instruction and 'lo8', 'hi8', 'hlo8', 'hhi8', modifier.

For example

ldi r26, lo8(285774925) ldi r27, hi8(285774925) ldi r28, hlo8(285774925) ldi r29, hhi8(285774925) ; r29,r28,r27,r26 = 285774925

### pm\_lo8

This modifier allows you to use bits 0 through 7 of an address expression as an 8 bit relocatable expression. This modifier is useful for addressing data or code from Flash/Program memory by two-byte words. The use of 'pm\_lo8' is similar to 'lo8'.

## pm\_hi8

This modifier allows you to use bits 8 through 15 of an address expression as an 8 bit relocatable expression. This modifier is useful for addressing data or code from Flash/Program memory by two-byte words.

For example, when setting the AVR 'Z' register with the 'ldi' instruction for subsequent use by the 'ijmp' instruction:

```
ldi r30, pm_lo8(sym)
ldi r31, pm_hi8(sym)
ijmp
```

## pm\_hh8

This modifier allows you to use bits 15 through 23 of an address expression as an 8 bit relocatable expression. This modifier is useful for addressing data or code from Flash/Program memory by two-byte words.

# 9.5.3 Opcodes

For detailed information on the AVR machine instruction set, see www.atmel.com/products/AVR.

**as** implements all the standard AVR opcodes. The following table summarizes the AVR opcodes, and their arguments.

Legend:

- r any register
- d 'ldi' register (r16-r31)
- v 'movw' even register (r0, r2, ..., r28, r30)
- a 'fmul' register (r16-r23)
- w 'adiw' register (r24,r26,r28,r30)
- e pointer registers (X,Y,Z)
- **b** base pointer register and displacement ([YZ]+disp)
- z Z pointer register (for [e]lpm Rd,Z[+])
- ${\tt M} \quad {\rm immediate \ value \ from \ 0 \ to \ 255}$
- ${\tt n}$  immediate value from 0 to 255 (  ${\tt n}=~{\tt \tilde{M}}$  ). Relocation impossible
- $\mathbf{s}$  immediate value from 0 to 7
- P Port address value from 0 to 63. (in, out)
- p Port address value from 0 to 31. (cbi, sbi, sbic, sbis)
- K immediate value from 0 to 63 (used in 'adiw', 'sbiw')
- i immediate value
- 1 signed pc relative offset from -64 to 63
- L signed pc relative offset from -2048 to 2047
- h absolute code address (call, jmp)
- **s** immediate value from 0 to 7 ( $S = s \ll 4$ )
- ? use this opcode entry if no parameters, else use next opcode entry

1001010010001000	clc	
1001010011011000	clh	
1001010011111000	cli	
1001010010101000	cln	
1001010011001000	cls	
1001010011101000	clt	
1001010010111000	clv	
1001010010011000	clz	
1001010000001000	sec	
1001010001011000	seh	
1001010001111000	sei	
1001010000101000	sen	
1001010001001000	ses	
1001010001101000	set	
1001010000111000	sev	
1001010000011000	sez	
100101001SSS1000	bclr	S
100101000SSS1000	bset	S
1001010100001001	icall	
1001010000001001	ijmp	
1001010111001000	lpm	?
1001000ddddd010+	lpm	r,z
1001010111011000	elpm	?
1001000ddddd011+	elpm	r,z
000000000000000000000000000000000000000	nop	
1001010100001000	ret	
1001010100011000	reti	
1001010110001000	sleep	
1001010110011000	break	
1001010110101000	wdr	
1001010111101000	$\mathtt{spm}$	
000111rdddddrrrr	adc	r,r
000011rdddddrrrr	add	r,r
001000rdddddrrrr	and	r,r

000101rdddddrrrr	cp	r,r
000001rdddddrrrr	cpc	r,r
000100rdddddrrrr	cpse	r,r
001001rdddddrrrr	eor	r,r
001011rdddddrrrr	mov	r,r
100111rdddddrrrr	mul	r,r
001010rdddddrrrr	or	r,r
000010rdddddrrrr	sbc	r,r
000110rdddddrrrr	sub	r,r
001001rdddddrrrr	clr	r
000011rdddddrrrr	lsl	r
000111rdddddrrrr	rol	r
001000rdddddrrrr	tst	r
0111KKKKddddKKKK	andi	d,M
0111KKKKddddKKKK	cbr	d,n
1110KKKKddddKKKK	ldi	d,M
11101111dddd1111	ser	d
0110KKKKddddKKKK	ori	d,M
0110KKKKddddKKKK	sbr	d,M
0011KKKKddddKKKK	cpi	d,M
0100KKKKddddKKKK	sbci	d,M
0101KKKKddddKKKK	subi	d,M
1111110rrrr0sss	sbrc	r,s
11111111rrrr0sss	sbrs	r,s
1111100ddddd0sss	bld	r,s
1111101dddddosss	bst	r,s
10110PPdddddPPPP	in	r,P
10111PPrrrrPPPP	out	P,r
10010110KKddKKKK	adiw	w,K
10010111KKddKKKK	sbiw	w,K
10011000ppppsss	cbi	p,s
10011010ppppsss	sbi sbic	p,s
10011001pppppsss 10011011pppppsss	sbic	p,s
1111011111111000	brcc	p,s l
111101111111000	brcs	1
1111001111111000	breg	1
1111001111111001	brge	1
1111011111111100	brhc	1
1111001111111101	brhs	1
11110011111111111	brid	1
1111001111111111111	brie	1
1111001111111000	brlo	1
11110011111111000	brlt	1
1111001111111010	brmi	1
1111011111111001	brne	1
1111011111111010	brpl	1
1111011111111000	brsh	1
1111011111111111	brtc	1
11110011111111110	brts	1
1111011111111011	brvc	1
1111001111111011	brvs	1
1111011111111sss	brbc	s,l
1111001111111sss	brbs	s,l
1101LLLLLLLLLLL	rcall	L
1100LLLLLLLLLLL	rjmp	L
1001010hhhhh111h	call	h
1001010hhhhh110h	jmp	h

1001010rrrrr0101	asr	r
1001010rrrr0000	com	r
1001010rrrr1010	dec	r
1001010rrrrr0011	inc	r
1001010rrrrr0110	lsr	r
1001010rrrrr0001	neg	r
1001000rrrr1111	pop	r
1001001rrrrr1111	push	r
1001010rrrrr0111	ror	r
1001010rrrrr0010	swap	r
00000001ddddrrrr	movw	v,v
00000010ddddrrrr	muls	d,d
000000110ddd0rrr	mulsu	a,a
000000110ddd1rrr	fmul	a,a
000000111ddd0rrr	fmuls	a,a
000000111ddd1rrr	fmulsu	a,a
1001001ddddd0000	sts	i,r
1001000ddddd0000	lds	r,i
10o0oo0dddddbooo	ldd	r,b
100!000dddddee-+	ld	r,e
10o0oo1rrrrbooo	std	b,r
100!001rrrrree-+	st	e,r
1001010100011001	eicall	
1001010000011001	eijmp	

# 9.5.4 Pseudo Instructions

The only available pseudo-instruction \_\_gcc\_isr can be activated by option -mgcc-isr.

\_\_gcc\_isr 1

Emit code chunk to be used in avr-gcc ISR prologue. It will expand to at most six 1-word instructions, all optional: push of tmp\_reg, push of SREG, push and clear of zero\_reg, push of Reg.

\_\_gcc\_isr 2

Emit code chunk to be used in an avr-gcc ISR epilogue. It will expand to at most five 1-word instructions, all optional: pop of *Reg*, pop of zero\_reg, pop of SREG, pop of tmp\_reg.

\_\_gcc\_isr 0, Reg

Finish avr-gcc ISR function. Scan code since the last prologue for usage of: SREG, tmp\_reg, zero\_reg. Prologue chunk and epilogue chunks will be replaced by appropriate code to save / restore SREG, tmp\_reg, zero\_reg and Reg.

Example input:

```
__vector1:

    __gcc_isr 1

    lds r24, var

    inc r24

    sts var, r24

    __gcc_isr 2

    reti

    __gcc_isr 0, r24

Example output:

    00000000 <__vector1>:
```

0:	8f 93	push	r24
2:	8f b7	in	r24, 0x3f
4:	8f 93	push	r24
6:	80 91 60 00	lds	r24, 0x0060 ; 0x800060 <var></var>
a:	83 95	inc	r24
c:	80 93 60 00	sts	0x0060, r24 ; 0x800060 <var></var>
10:	8f 91	pop	r24
12:	8f bf	out	0x3f, r24
14:	8f 91	pop	r24
16:	18 95	reti	

# 9.6 Blackfin Dependent Features

## 9.6.1 Options

## -mcpu=processor[-sirevision]

This option specifies the target processor. The optional *sirevision* is not used in assembler. It's here such that GCC can easily pass down its -mcpu= option. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: bf504, bf506, bf512, bf514, bf516, bf518, bf522, bf523, bf524, bf525, bf526, bf527, bf531, bf532, bf533, bf534, bf535 (not implemented yet), bf536, bf537, bf538, bf539, bf542, bf542m, bf544, bf544m, bf547, bf547m, bf548, bf548m, bf549, bf549m, bf561, and bf592.

-mfdpic Assemble for the FDPIC ABI.

-mno-fdpic -mnopic Disable -mfdpic.

## 9.6.2 Syntax

### Special Characters

Assembler input is free format and may appear anywhere on the line. One instruction may extend across multiple lines or more than one instruction may appear on the same line. White space (space, tab, comments or newline) may appear anywhere between tokens. A token must not have embedded spaces. Tokens include numbers, register names, keywords, user identifiers, and also some multicharacter special symbols like "+=", "/\*" or "||".

Comments are introduced by the '#' character and extend to the end of the current line. If the '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

### Instruction Delimiting

A semicolon must terminate every instruction. Sometimes a complete instruction will consist of more than one operation. There are two cases where this occurs. The first is when two general operations are combined. Normally a comma separates the different parts, as in

a0= r3.h \* r2.l, a1 = r3.l \* r2.h ;

The second case occurs when a general instruction is combined with one or two memory references for joint issue. The latter portions are set off by a "||" token.

```
a0 = r3.h * r2.l || r1 = [p3++] || r4 = [i2++];
```

Multiple instructions can occur on the same line. Each must be terminated by a semicolon character.

### Register Names

The assembler treats register names and instruction keywords in a case insensitive manner. User identifiers are case sensitive. Thus, R3.1, R3.L, r3.1 and r3.L are all equivalent input to the assembler.

Register names are reserved and may not be used as program identifiers.

Some operations (such as "Move Register") require a register pair. Register pairs are always data registers and are denoted using a colon, eg., R3:2. The larger number must be written firsts. Note that the hardware only supports odd-even pairs, eg., R7:6, R5:4, R3:2, and R1:0.

Some instructions (such as –SP (Push Multiple)) require a group of adjacent registers. Adjacent registers are denoted in the syntax by the range enclosed in parentheses and separated by a colon, eg., (R7:3). Again, the larger number appears first.

Portions of a particular register may be individually specified. This is written with a dot (".") following the register name and then a letter denoting the desired portion. For 32-bit registers, ".H" denotes the most significant ("High") portion. ".L" denotes the least-significant portion. The subdivisions of the 40-bit registers are described later.

### Accumulators

The set of 40-bit registers A1 and A0 that normally contain data that is being manipulated. Each accumulator can be accessed in four ways.

### one 40-bit register

The register will be referred to as A1 or A0.

#### one 32-bit register

The registers are designated as A1.W or A0.W.

### two 16-bit registers

The registers are designated as A1.H, A1.L, A0.H or A0.L.

### one 8-bit register

The registers are designated as A1.X or A0.X for the bits that extend beyond bit 31.

#### Data Registers

The set of 32-bit registers (R0, R1, R2, R3, R4, R5, R6 and R7) that normally contain data for manipulation. These are abbreviated as D-register or Dreg. Data registers can be accessed as 32-bit registers or as two independent 16-bit registers. The least significant 16 bits of each register is called the "low" half and is designated with ".L" following the register name. The most significant 16 bits are called the "high" half and is designated with ".H" following the name. R7.L, r2.h, r4.L, R0.H

#### Pointer Registers

The set of 32-bit registers (P0, P1, P2, P3, P4, P5, SP and FP) that normally contain byte addresses of data structures. These are abbreviated as P-register or Preg.

p2, p5, fp, sp

### Stack Pointer SP

The stack pointer contains the 32-bit address of the last occupied byte location in the stack. The stack grows by decrementing the stack pointer.

### Frame Pointer FP

The frame pointer contains the 32-bit address of the previous frame pointer in the stack. It is located at the top of a frame.

Loop Top LT0 and LT1. These registers contain the 32-bit address of the top of a zero overhead loop.

#### Loop Count

LC0 and LC1. These registers contain the 32-bit counter of the zero overhead loop executions.

### Loop Bottom

LB0 and LB1. These registers contain the 32-bit address of the bottom of a zero overhead loop.

### Index Registers

The set of 32-bit registers (I0, I1, I2, I3) that normally contain byte addresses of data structures. Abbreviated I-register or Ireg.

### Modify Registers

The set of 32-bit registers (M0, M1, M2, M3) that normally contain offset values that are added and subtracted to one of the index registers. Abbreviated as Mreg.

### Length Registers

The set of 32-bit registers (L0, L1, L2, L3) that normally contain the length in bytes of the circular buffer. Abbreviated as Lreg. Clear the Lreg to disable circular addressing for the corresponding Ireg.

### Base Registers

The set of 32-bit registers (B0, B1, B2, B3) that normally contain the base address in bytes of the circular buffer. Abbreviated as Breg.

### Floating Point

The Blackfin family has no hardware floating point but the .float directive generates ieee floating point numbers for use with software floating point libraries.

### Blackfin Opcodes

For detailed information on the Blackfin machine instruction set, see the Blackfin Processor Instruction Set Reference.

## 9.6.3 Directives

The following directives are provided for compatibility with the VDSP assembler.

- .byte2 Initializes a two byte data object. This maps to the .short directive.
  .byte4 Initializes a four byte data object.
  - This maps to the .int directive.

.db	Initializes a single byte data object.
	This directive is a synonym for .byte.
.dw	Initializes a two byte data object.
	This directive is a synonym for .byte2.
.dd	Initializes a four byte data object.
	This directive is a synonym for .byte4.
.var	Define and initialize a 32 bit data object.

# 9.7 BPF Dependent Features

# 9.7.1 Options

-EB This option specifies that the assembler should emit big-endian eBPF.

-EL This option specifies that the assembler should emit little-endian eBPF.

Note that if no endianness option is specified in the command line, the host endianness is used.

# 9.7.2 Syntax

# 9.7.2.1 Special Characters

The presence of a ';' on a line indicates the start of a comment that extends to the end of the current line. If a '#' appears as the first character of a line, the whole line is treated as a comment.

Statements and assembly directives are separated by newlines.

# 9.7.2.2 Register Names

The eBPF processor provides ten general-purpose 64-bit registers, which are read-write, and a read-only frame pointer register:

'%r0 .. %r9'

General-purpose registers.

"%r10" Frame pointer register.

Some registers have additional names, to reflect their role in the eBPF ABI:

"%a' This is "%r0'.

"%ctx' This is "%r6'.

""". This is "%r10".

# 9.7.2.3 Pseudo Maps

The 'LDDW' instruction can take a literal pseudo map file descriptor as its second argument. This uses the syntax '%map\_fd(N)' where 'N' is a signed number.

For example, to load the address of the pseudo map with file descriptor `2` in register `r1' we would do:

lddw %r1, %map\_fd(2)

# 9.7.3 Machine Directives

The BPF version of **as** supports the following additional machine directives:

.word The .half directive produces a 16 bit value.

.word The .word directive produces a 32 bit value.

.dword The .dword directive produces a 64 bit value.

## 9.7.4 Opcodes

In the instruction descriptions below the following field descriptors are used:

%d	Destination general-purpose register whose role is to be destination of an opera- tion.
%s	Source general-purpose register whose role is to be the source of an operation.
disp16	16-bit signed PC-relative offset, measured in number of 64-bit words, minus one.
disp32	32-bit signed PC-relative offset, measured in number of 64-bit words, minus one.
offset16	Signed 16-bit immediate.
imm32	Signed 32-bit immediate.
imm64	Signed 64-bit immediate.

## 9.7.4.1 Arithmetic instructions

The destination register in these instructions act like an accumulator.

add %d, (%s|imm32) 64-bit arithmetic addition. sub %d, (%s|imm32) 64-bit arithmetic subtraction. mul %d, (%s|imm32) 64-bit arithmetic multiplication. div %d, (%s|imm32) 64-bit arithmetic integer division. mod %d, (%s|imm32) 64-bit integer remainder. and %d, (%s|imm32) 64-bit bit-wise "and" operation. or %d, (%s|imm32) 64-bit bit-wise "or" operation. xor %d, (%s|imm32) 64-bit bit-wise exclusive-or operation. lsh %d, (%s|imm32) 64-bit left shift, by %s or imm32 bits. rsh %d, (%s|imm32) 64-bit right logical shift, by %s or imm32 bits. arsh %d, (%s|imm32) 64-bit right arithmetic shift, by %s or imm32 bits. neg %d 64-bit arithmetic negation. mov %d, (%s|imm32) Move the 64-bit value of %s in %d, or load imm32 in %d.

## 9.7.4.2 32-bit arithmetic instructions

The destination register in these instructions act as an accumulator.

- add32 %d, (%s|imm32) 32-bit arithmetic addition.
- sub32 %d, (%s|imm32)
  - 32-bit arithmetic subtraction.
- div32 %d, (%s|imm32) 32-bit arithmetic integer division.
- mod32 %d, (%s|imm32) 32-bit integer remainder.
- and32 %d, (%s|imm32) 32-bit bit-wise "and" operation.
- or32 %d, (%s|imm32) 32-bit bit-wise "or" operation.
- xor32 %d, (%s|imm32) 32-bit bit-wise exclusive-or operation.
- lsh32 %d, (%s|imm32) 32-bit left shift, by %s or imm32 bits.
- arsh32 %d, (%s|imm32) 32-bit right arithmetic shift, by %s or imm32 bits.
- neg32 %d 32-bit arithmetic negation.
- mov32 %d, (%s|imm32) Move the 32-bit value of %s in %d, or load imm32 in %d.

# 9.7.4.3 Endianness conversion instructions

endle %d, (16|32|64) Convert the 16-bit, 32-bit or 64-bit value in %d to little-endian. endbe %d, (16|32|64)

Convert the 16-bit, 32-bit or 64-bit value in %d to big-endian.

# 9.7.4.4 64-bit load and pseudo maps

lddw %d, imm64

Load the given signed 64-bit immediate, or pseudo map descriptor, to the destination register %d.

lddw %d, %map\_fd(N)

Load the address of the given pseudo map fd N to the destination register %d.

## 9.7.4.5 Load instructions for socket filters

The following instructions are intended to be used in socket filters, and are therefore not general-purpose: they make assumptions on the contents of several registers. See the file Documentation/networking/filter.txt in the Linux kernel source tree for more information.

Absolute loads:

ldabsdw imm32 Absolute 64-bit load. ldabsw imm32 Absolute 32-bit load. ldabsh imm32 Absolute 16-bit load. ldabsb imm32 Absolute 8-bit load. Indirect loads: ldinddw %s, imm32 Indirect 64-bit load. ldindw %s, imm32 Indirect 32-bit load. ldindh %s, imm32 Indirect 16-bit load. ldindb %s, imm32

Indirect 8-bit load.

## 9.7.4.6 Generic load/store instructions

General-purpose load and store instructions are provided for several word sizes.

Load to register instructions:

- ldxdw %d, [%s+offset16] Generic 64-bit load.
- ldxw %d, [%s+offset16] Generic 32-bit load.
- ldxh %d, [%s+offset16] Generic 16-bit load.
- ldxb %d, [%s+offset16] Generic 8-bit load.

Store from register instructions:

stxdw [%d+offset16], %s Generic 64-bit store. stxw [%d+offset16], %s Generic 32-bit store.

stxh [%d+offset16], %s Generic 16-bit store.

stxb [%d+offset16], %s Generic 8-bit store.

Store from immediates instructions:

- stdw [%d+offset16], imm32 Store immediate as 32-bit.

# 9.7.4.7 Jump instructions

eBPF provides the following compare-and-jump instructions, which compare the values of the two given registers, or the values of a register and an immediate, and perform a branch in case the comparison holds true.

- ja %d,(%s|imm32),disp16 Jump-always.
- jgt %d,(%s|imm32),disp16 Jump if greater, unsigned.
- jge %d,(%s|imm32),disp16
   Jump if greater or equal.
- jlt %d,(%s|imm32),disp16 Jump if lesser.
- jset %d, (%s|imm32), disp16 Jump if signed equal.
- jne %d,(%s|imm32),disp16 Jump if not equal.

jslt %d,(%s|imm32),disp16 Jump if signed lesser.

A call instruction is provided in order to perform calls to other eBPF functions, or to external kernel helpers:

call (disp32|imm32)

Jump and link to the offset *disp32*, or to the kernel helper function identified by *imm32*.

Finally:

exit Terminate the eBPF program.

## 9.7.4.8 Atomic instructions

Atomic exchange-and-add instructions are provided in two flavors: one for swapping 64-bit quantities and another for 32-bit quantities.

```
xadddw [%d+offset16],%s
```

Exchange-and-add a 64-bit value at the specified location.

xaddw [%d+offset16],%s

Exchange-and-add a 32-bit value at the specified location.

## 9.7.5 BPF Pseudo-C Syntax

This assembler supports another syntax to denote BPF instructions, which is an alternative to the normal looking syntax documented above. This alternatative syntax, which we call pseudo-C syntax, is supported by the LLVM/clang integrated assembler.

This syntax is very unconventional, but we need to support it in order to support inline assembly in existing BPF programs.

Note that the assembler is able to parse sources in which both syntaxes coexist: some instructions can use the usual assembly like syntax, whereas some other instructions in the same file can use the pseudo-C syntax.

## 9.7.5.1 Pseudo-C Register Names

All BPF registers are 64-bit long. However, in the Pseudo-C syntax registers can be referred using different names, which actually reflect the kind of instruction they appear on:

- 'r0..r9' General-purpose register in an instruction that operates on its value as if it was a 64-bit value.
- 'w0..w9' General-purpose register in an instruction that operates on its value as if it was a 32-bit value.

Note that in the Pseudo-C syntax register names are not preceded by % characters.

# 9.7.5.2 Arithmetic instructions

In all the instructions below, the operations are 64-bit or 32-bit depending on the names used to refer to the registers. For example r3 += r2 will perform 64-bit addition, whereas w3 += w2 will perform 32-bit addition. Mixing register prefixes is an error, for example r3 += w2.

dst\_reg += (imm32|src\_reg) Arithmetic addition. dst\_reg -= (imm32|src\_reg) Arithmetic subtraction. dst\_reg \*= (imm32|src\_reg) Arithmetic multiplication. dst\_reg /= (imm32|src\_reg) Arithmetic integer unsigned division. dst\_reg %= (imm32|src\_reg) Arithmetic integer unsigned remainder. dst\_reg &= (imm32|src\_reg) Bit-wise "and" operation. dst\_reg |= (imm32|src\_reg) Bit-wise "or" operation. dst\_reg ^= (imm32|src\_reg) Bit-wise exclusive-or operation. dst\_reg <<= (imm32|src\_reg)</pre> Left shift, by whatever specified number of bits. dst\_reg >>= (imm32|src\_reg) Right logical shift, by whatever specified number of bits. dst\_reg s>>= (imm32|src\_reg) Right arithmetic shift, by whatever specified number of bits. dst\_reg = (imm32|src\_reg) Move the value in imm32 or src\_reg in dst\_reg. dst\_reg = -dst\_reg Arithmetic negation. 9.7.5.3 Endianness conversion instructions dst\_reg = le16 src\_reg Convert the 16-bit value in src\_reg to little-endian. dst\_reg = le32 src\_reg Convert the 32-bit value in src\_reg to little-endian. dst\_reg = le64 src\_reg Convert the 64-bit value in src\_reg to little-endian.

- dst\_reg = be16 src\_reg Convert the 16-bit value in src\_reg to big-endian.
- dst\_reg = be32 src\_reg Convert the 32-bit value in src\_reg to big-endian.
- dst\_reg = be64 src\_reg Convert the 64-bit value in src\_reg to big-endian.

## 9.7.5.4 64-bit load and pseudo maps

dst\_reg = imm64 ll

Load the given signed 64-bit immediate, or pseudo map descriptor, to the destination register dst\_reg.

## 9.7.5.5 Load instructions for socket filters

r0 = \*(u8 \*)skb[imm32] Absolute 8-bit load.

r0 = \*(u16 \*)skb[imm32] Absolute 16-bit load.

r0 = \*(u32 \*)skb[imm32] Absolute 32-bit load.

r0 = \*(u64 \*)skb[imm32] Absolute 64-bit load.

- r0 = \*(u8 \*)skb[src\_reg + imm32] Indirect 8-bit load.
- r0 = \*(u16 \*)skb[src\_reg + imm32] Indirect 16-bit load.
- r0 = \*(u32 \*)skb[src\_reg + imm32] Indirect 32-bit load.
- r0 = \*(u64 \*)skb[src\_reg + imm32] Indirect 64-bit load.

## 9.7.5.6 Generic load/store instructions

dst\_reg = \*(u8 \*)(src\_reg + offset16)
 Generic 8-bit load.
dst\_reg = \*(u16 \*)(src\_reg + offset16)
 Generic 16-bit load.
dst\_reg = \*(u32 \*)(src\_reg + offset16)
 Generic 32-bit load.
dst\_reg = \*(u64 \*)(src\_reg + offset16)
 Generic 64-bit load.

\*(u8 \*)(dst\_reg + offset16) = src\_reg
Generic 8-bit store.

- \*(u16 \*)(dst\_reg + offset16) = src\_reg
  Generic 16-bit store.
- \*(u32 \*)(dst\_reg + offset16) = src\_reg Generic 32-bit store.
- \*(u64 \*)(dst\_reg + offset16) = src\_reg
  Generic 64-bit store.

## 9.7.5.7 Jump instructions

### goto disp16

Jump-always.

if dst\_reg == (imm32|src\_reg) goto disp16
 Jump if equal.

if dst\_reg & (imm32|src\_reg) goto disp16 Jump if signed equal.

if dst\_reg != (imm32|src\_reg) goto disp16
 Jump if not equal.

if dst\_reg > (imm32|src\_reg) goto disp16
 Jump if bigger, unsigned.

if dst\_reg < (imm32|src\_reg) goto disp16
 Jump if smaller, unsigned.</pre>

if dst\_reg >= (imm32|src\_reg) goto disp16
 Jump if bigger or equal, unsigned.

if dst\_reg <= (imm32|src\_reg) goto disp16
 Jump if smaller or equal, unsigned.</pre>

if dst\_reg s> (imm32|src\_reg) goto disp16 Jump if bigger, signed.

if dst\_reg s< (imm32|src\_reg) goto disp16 Jump if smaller, signed.

if dst\_reg s>= (imm32|src\_reg) goto disp16 Jump if bigger or equal, signed.

if dst\_reg s<= (imm32|src\_reg) goto disp16
 Jump if smaller or equal, signed.</pre>

## call imm32

Jump and link.

exit Terminate the eBPF program.

## 9.7.5.8 Atomic instructions

lock \*(u64 \*)(dst\_reg + offset16) += src\_reg
Exchange-and-add a 64-bit value at the specified location.

lock \*(u32 \*)(dst\_reg + offset16) += src\_reg
Exchange-and-add a 32-bit value at the specified location.

# 9.8 CR16 Dependent Features

## 9.8.1 CR16 Operand Qualifiers

The National Semiconductor CR16 target of **as** has a few machine dependent operand qualifiers.

Operand expression type qualifier is an optional field in the instruction operand, to determines the type of the expression field of an operand. The @ is required. CR16 architecture uses one of the following expression qualifiers:

- s Specifies expression operand type as small
- m Specifies expression operand type as medium
- 1 Specifies expression operand type as large
- c Specifies the CR16 Assembler generates a relocation entry for the operand, where pc has implied bit, the expression is adjusted accordingly. The linker uses the relocation entry to update the operand address at link time.
- got/GOT Specifies the CR16 Assembler generates a relocation entry for the operand, offset from Global Offset Table. The linker uses this relocation entry to update the operand address at link time
- cgot/cGOT

- Specifies the CompactRISC Assembler generates a relocation entry for the operand, where pc has implied bit, the expression is adjusted accordingly. The linker uses the relocation entry to update the operand address at link time.

CR16 target operand qualifiers and its size (in bits):

```
'Immediate Operand: s'
4 bits.
```

'Immediate Operand: m'

16 bits, for movb and movw instructions.

- 'Immediate Operand: m' 20 bits, movd instructions.
- 'Immediate Operand: 1' 32 bits.
- 'Absolute Operand: s' Illegal specifier for this operand.
- 'Absolute Operand: m' 20 bits, movd instructions.

'Displacement Operand: s' 8 bits.

'Displacement Operand: m' 16 bits.

```
'Displacement Operand: 1'
          24 bits.
  For example:
     1
         movw $_myfun@c,r1
         This loads the address of _myfun, shifted right by 1, into r1.
     2
         movd $_myfun@c,(r2,r1)
         This loads the address of _myfun, shifted right by 1, into register-
     pair r2-r1.
     3
         _myfun_ptr:
         .long _myfun@c
         loadd _myfun_ptr, (r1,r0)
         jal (r1,r0)
         This .long directive, the address of _myfunc, shifted right by 1 at link time.
     4
         loadd _data1@GOT(r12), (r1,r0)
         This loads the address of _data1, into global offset table (ie GOT) and its off-
     set value from GOT loads into register-pair r2-r1.
     5
         loadd _myfunc@cGOT(r12), (r1,r0)
         This loads the address of _myfun, shifted right by 1, into global off-
     set table (ie GOT) and its offset value from GOT loads into register-pair r1-
     r0.
```

# 9.8.2 CR16 Syntax

# 9.8.2.1 Special Characters

The presence of a '#' on a line indicates the start of a comment that extends to the end of the current line. If the '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

# 9.9 CRIS Dependent Features

## 9.9.1 Command-line Options

The CRIS version of as has these machine-dependent command-line options.

The format of the generated object files can be either ELF or a.out, specified by the command-line options --emulation=crisaout and --emulation=criself. The default is ELF (criself), unless as has been configured specifically for a.out by using the configuration name cris-axis-aout.

There are two different link-incompatible ELF object file variants for CRIS, for use in environments where symbols are expected to be prefixed by a leading '\_' character and for environments without such a symbol prefix. The variant used for GNU/Linux port has no symbol prefix. Which variant to produce is specified by either of the options --underscore and --no-underscore. The default is --underscore. Since symbols in CRIS a.out objects are expected to have a '\_' prefix, specifying --no-underscore when generating a.out objects is an error. Besides the object format difference, the effect of this option is to parse register names differently (see [crisnous], page 160). The --no-underscore option makes a '\$' register prefix mandatory.

The option --pic must be passed to as in order to recognize the symbol syntax used for ELF (SVR4 PIC) position-independent-code (see [crispic], page 159). This will also affect expansion of instructions. The expansion with --pic will use PC-relative rather than (slightly faster) absolute addresses in those expansions. This option is only valid when generating ELF format object files.

The option --march=architecture specifies the recognized instruction set and recognized register names. It also controls the architecture type of the object file. Valid values for architecture are:

- v0\_v10 All instructions and register names for any architecture variant in the set v0...v10 are recognized. This is the default if the target is configured as cris-\*.
- v10 Only instructions and register names for CRIS v10 (as found in ETRAX 100 LX) are recognized. This is the default if the target is configured as crisv10-\*.
- v32 Only instructions and register names for CRIS v32 (code name Guinness) are recognized. This is the default if the target is configured as crisv32-\*. This value implies --no-mul-bug-abort. (A subsequent --mul-bug-abort will turn it back on.)

```
common_v10_v32
```

Only instructions with register names and addressing modes with opcodes common to the v10 and v32 are recognized.

When -N is specified, as will emit a warning when a 16-bit branch instruction is expanded into a 32-bit multiple-instruction construct (see Section 9.9.2 [CRIS-Expand], page 158).

Some versions of the CRIS v10, for example in the Etrax 100 LX, contain a bug that causes destabilizing memory accesses when a multiply instruction is executed with certain values in the first operand just before a cache-miss. When the --mul-bug-abort command-line option is active (the default value), as will refuse to assemble a file containing a multiply instruction at a dangerous offset, one that could be the last on a cache-line, or is in a

section with insufficient alignment. This placement checking does not catch any case where the multiply instruction is dangerously placed because it is located in a delay-slot. The --mul-bug-abort command-line option turns off the checking.

### 9.9.2 Instruction expansion

as will silently choose an instruction that fits the operand size for '[register+constant]' operands. For example, the offset 127 in move.d [r3+127],r4 fits in an instruction using a signed-byte offset. Similarly, move.d [r2+32767],r1 will generate an instruction using a 16-bit offset. For symbolic expressions and constants that do not fit in 16 bits including the sign bit, a 32-bit offset is generated.

For branches, **as** will expand from a 16-bit branch instruction into a sequence of instructions that can reach a full 32-bit address. Since this does not correspond to a single instruction, such expansions can optionally be warned about. See Section 9.9.1 [CRIS-Opts], page 157.

If the operand is found to fit the range, a lapc mnemonic will translate to a lapcq instruction. Use lapc.d to force the 32-bit lapc instruction.

Similarly, the addo mnemonic will translate to the shortest fitting instruction of addoq, addo.w and addo.d, when used with a operand that is a constant known at assembly time.

## 9.9.3 Symbols

Some symbols are defined by the assembler. They're intended to be used in conditional assembly, for example:

```
.if ..asm.arch.cris.v32
code for CRIS v32
.elseif ..asm.arch.cris.common_v10_v32
code common to CRIS v32 and CRIS v10
.elseif ..asm.arch.cris.v10 | ..asm.arch.cris.any_v0_v10
code for v10
.else
.error "Code needs to be added here."
.endif
```

These symbols are defined in the assembler, reflecting command-line options, either when specified or the default. They are always defined, to 0 or 1.

```
..asm.arch.cris.any_v0_v10
```

This symbol is non-zero when --march=v0\_v10 is specified or the default.

```
..asm.arch.cris.common_v10_v32
Set according to the option --march=common_v10_v32.
```

```
..asm.arch.cris.v10
```

Reflects the option --march=v10.

..asm.arch.cris.v32

Corresponds to --march=v10.

Speaking of symbols, when a symbol is used in code, it can have a suffix modifying its value for use in position-independent code. See Section 9.9.4.2 [CRIS-Pic], page 159.

## 9.9.4 Syntax

There are different aspects of the CRIS assembly syntax.

## 9.9.4.1 Special Characters

The character '#' is a line comment character. It starts a comment if and only if it is placed at the beginning of a line.

A ';' character starts a comment anywhere on the line, causing all characters up to the end of the line to be ignored.

A '@' character is handled as a line separator equivalent to a logical new-line character (except in a comment), so separate instructions can be specified on a single line.

## 9.9.4.2 Symbols in position-independent code

When generating position-independent code (SVR4 PIC) for use in cris-axis-linux-gnu or crisv32-axis-linux-gnu shared libraries, symbol suffixes are used to specify what kind of run-time symbol lookup will be used, expressed in the object as different *relocation types*. Usually, all absolute symbol values must be located in a table, the *global offset table*, leaving the code position-independent; independent of values of global symbols and independent of the address of the code. The suffix modifies the value of the symbol, into for example an index into the global offset table where the real symbol value is entered, or a PC-relative value, or a value relative to the start of the global offset table. All symbol suffixes start with the character ':' (omitted in the list below). Every symbol use in code or a read-only section must therefore have a PIC suffix to enable a useful shared library to be created. Usually, these constructs must not be used with an additive constant offset as is usually allowed, i.e. no 4 as in symbol + 4 is allowed. This restriction is checked at link-time, not at assembly-time.

GOT

Attaching this suffix to a symbol in an instruction causes the symbol to be entered into the global offset table. The value is a 32-bit index for that symbol into the global offset table. The name of the corresponding relocation is 'R\_CRIS\_32\_GOT'. Example: move.d [\$r0+extsym:GOT],\$r9

GOT16

Same as for 'GOT', but the value is a 16-bit index into the global offset table. The corresponding relocation is 'R\_CRIS\_16\_GOT'. Example: move.d [\$r0+asymbol:GOT16],\$r10

PLT

This suffix is used for function symbols. It causes a *procedure linkage table*, an array of code stubs, to be created at the time the shared object is created or linked against, together with a global offset table entry. The value is a pc-relative offset to the corresponding stub code in the procedure linkage table. This arrangement causes the run-time symbol resolver to be called to look up and set the value of the symbol the first time the function is called (at latest; depending environment variables). It is only safe to leave the symbol unresolved this way if all references are function calls. The name of the relocation is 'R\_CRIS\_32\_PLT\_PCREL'. Example: add.d fnname:PLT,\$pc

PLTG

Like PLT, but the value is relative to the beginning of the global offset table. The relocation is 'R\_CRIS\_32\_PLT\_GOTREL'. Example: move.d fnname:PLTG,\$r3

## GOTPLT

Similar to 'PLT', but the value of the symbol is a 32-bit index into the global offset table. This is somewhat of a mix between the effect of the 'GOT' and the 'PLT' suffix; the difference to 'GOT' is that there will be a procedure linkage table entry created, and that the symbol is assumed to be a function entry and will be resolved by the run-time resolver as with 'PLT'. The relocation is 'R\_CRIS\_32\_GOTPLT'. Example: jsr [\$r0+fnname:GOTPLT]

# GOTPLT16

A variant of 'GOTPLT' giving a 16-bit value. Its relocation name is 'R\_CRIS\_16\_GOTPLT'. Example: jsr [\$r0+fnname:GOTPLT16]

## GOTOFF

This suffix must only be attached to a local symbol, but may be used in an expression adding an offset. The value is the address of the symbol relative to the start of the global offset table. The relocation name is 'R\_CRIS\_32\_GOTREL'. Example: move.d [\$r0+localsym:GOTOFF],r3

# 9.9.4.3 Register names

A '\$' character may always prefix a general or special register name in an instruction operand but is mandatory when the option --no-underscore is specified or when the .syntax register\_prefix directive is in effect (see [crisnous], page 160). Register names are case-insensitive.

# 9.9.4.4 Assembler Directives

There are a few CRIS-specific pseudo-directives in addition to the generic ones. See Chapter 7 [Pseudo Ops], page 51. Constants emitted by pseudo-directives are in little-endian order for CRIS. There is no support for floating-point-specific directives for CRIS.

## .dword EXPRESSIONS

The .dword directive is a synonym for .int, expecting zero or more EXPRES-SIONS, separated by commas. For each expression, a 32-bit little-endian constant is emitted.

## .syntax ARGUMENT

The  $\tt.syntax$  directive takes as ARGUMENT one of the following case-sensitive choices.

## no\_register\_prefix

The .syntax no\_register\_prefix directive makes a '\$' character prefix on all registers optional. It overrides a previous setting, including the corresponding effect of the option --no-underscore. If this directive is used when ordinary symbols do not have a '\_' character prefix, care must be taken to avoid ambiguities whether an operand is a register or a symbol; using symbols with names the same as general or special registers then invoke undefined behavior.

## register\_prefix

This directive makes a '\$' character prefix on all registers mandatory. It overrides a previous setting, including the corresponding effect of the option --underscore.

### leading\_underscore

This is an assertion directive, emitting an error if the --no-underscore option is in effect.

## no\_leading\_underscore

This is the opposite of the .syntax leading\_underscore directive and emits an error if the option --underscore is in effect.

### .arch ARGUMENT

This is an assertion directive, giving an error if the specified ARGUMENT is not the same as the specified or default value for the --march=architecture option (see [march-option], page 157).

# 9.10 C-SKY Dependent Features

# 9.10.1 Options

### -march=archname

Assemble for architecture archname. The --help option lists valid values for archname.

### -mcpu=cpuname

Assemble for architecture *cpuname*. The **--help** option lists valid values for *cpuname*.

## -EL

## -mlittle-endian

Generate little-endian output.

## -EB

### -mbig-endian

Generate big-endian output.

## -fpic

-pic Generate position-independent code.

## -mljump

-mno-ljump

Enable/disable transformation of the short branch instructions jbf, jbt, and jbr to jmpi. This option is for V2 processors only. It is ignored on CK801 and CK802 targets, which do not support the jmpi instruction, and is enabled by default for other processors.

## -mbranch-stub

-mno-branch-stub

Pass through <code>R\_CKCORE\_PCREL\_IMM26BY2</code> relocations for <code>bsr</code> instructions to the linker.

This option is only available for bare-metal C-SKY V2 ELF targets, where it is enabled by default. It cannot be used in code that will be dynamically linked against shared libraries.

## -force2bsr

## -mforce2bsr

```
-no-force2bsr
```

## -mno-force2bsr

Enable/disable transformation of jbsr instructions to bsr. This option is always enabled (and -mno-force2bsr is ignored) for CK801/CK802 targets. It is also always enabled when -mbranch-stub is in effect.

## -jsri2bsr

```
-mjsri2bsr
```

## -no-jsri2bsr

# -mno-jsri2bsr

Enable/disable transformation of jsri instructions to bsr. This option is enabled by default.

-mnolrw -mno-lrw	Enable/disable transformation of lrw instructions into a movih/ori pair.
-melrw -mno-elrw	
	Enable/disable extended $\tt lrw$ instructions. This option is enabled by default for CK800-series processors.
-mlaf	
-mliterals	s-after-func
-mno-laf	
-mno-liter	als-after-func
	Enable/disable placement of literal pools after each function.
-mlabr	
-mliterals	s-after-br
-mno-labr	
-mnolitera	ls-after-br
	Enable/disable placement of literal pools after unconditional branches. This option is enabled by default.

# -mistack

-mno-istack

Enable/disable interrupt stack instructions. This option is enabled by default on CK801, CK802, and CK802 processors.

The following options explicitly enable certain optional instructions. These features are also enabled implicitly by using -mcpu= to specify a processor that supports it.

### -mhard-float

Enable hard float instructions.

-mmp Enable multiprocessor instructions.

-mcp Enable coprocessor instructions.

-mcache Enable cache prefetch instruction.

-msecurity

Enable C-SKY security instructions.

-mtrust Enable C-SKY trust instructions.

-mdsp Enable DSP instructions.

-medsp Enable enhanced DSP instructions.

-mvdsp Enable vector DSP instructions.

# 9.10.2 Syntax

as implements the standard C-SKY assembler syntax documented in the C-SKY V2 CPU Applications Binary Interface Standards Manual.

# 9.11 D10V Dependent Features

# 9.11.1 D10V Options

The Mitsubishi D10V version of **as** has a few machine dependent options.

'-0' The D10V can often execute two sub-instructions in parallel. When this option is used, **as** will attempt to optimize its output by detecting when instructions can be executed in parallel.

'--nowarnswap'

To optimize execution performance, **as** will sometimes swap the order of instructions. Normally this generates a warning. When this option is used, no warning will be generated when instructions are swapped.

# '--gstabs-packing'

'--no-gstabs-packing'

as packs adjacent short instructions into a single packed instruction. '--no-gstabs-packing' turns instruction packing off if '--gstabs' is specified as well; '--gstabs-packing' (the default) turns instruction packing on even when '--gstabs' is specified.

# 9.11.2 Syntax

The D10V syntax is based on the syntax in Mitsubishi's D10V architecture manual. The differences are detailed below.

# 9.11.2.1 Size Modifiers

The D10V version of **as** uses the instruction names in the D10V Architecture Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? **as** will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either '.s' (short) or '.1' (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write '**bra.s foo**'. Objdump and GDB will always append '.s' or '.1' to instructions which have both short and long forms.

# 9.11.2.2 Sub-Instructions

The D10V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the next section.

## 9.11.2.3 Special Characters

A semicolon (';') can be used anywhere on a line to start a comment that extends to the end of the line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially. To specify the executing order, use the following symbols:

'->' Sequential with instruction on the left first.
---

'<-' Sequential with instruction on the right first.

## '||' Parallel

The D10V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. For example

abs a1 -> abs r0

Execute these sequentially. The instruction on the right is in the right container and is executed second.

abs r0 <- abs a1

Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

ld2w r2,@r8+ || mac a0,r0,r7

Execute these in parallel.

ld2w r2,@r8+ ||

mac a0,r0,r7

Two-line format. Execute these in parallel.

```
ld2w r2,@r8+
```

mac a0,r0,r7

Two-line format. Execute these sequentially. Assembler will put them in the proper containers.

### ld2w r2,@r8+ ->

### mac a0,r0,r7

Two-line format. Execute these sequentially. Same as above but second instruction will always go into right container.

Since '\$' has no special meaning, you may use it in symbol names.

# 9.11.2.4 Register Names

You can use the predefined symbols 'r0' through 'r15' to refer to the D10V registers. You can also use 'sp' as an alias for 'r15'. The accumulators are 'a0' and 'a1'. There are special register-pair names that may optionally be used in opcodes that require even-numbered registers. Register names are not case sensitive.

**Register** Pairs

r0-r1 r2-r3 r4-r5

r6-r7

r8-r9

r10-r11

r12-r13

```
r14-r15
```

The D10V also has predefined symbols for these control registers and status bits:

psw	Processor Status Word
bpsw	Backup Processor Status Word
рс	Program Counter
bpc	Backup Program Counter
rpt_c	Repeat Count
rpt_s	Repeat Start address
rpt_e	Repeat End address
mod_s	Modulo Start address
mod_e	Modulo End address
iba	Instruction Break Address
fO	Flag 0
f1	Flag 1
с	Carry flag

# 9.11.2.5 Addressing Modes

as understands the following addressing modes for the D10V. Rn in the following refers to any of the numbered registers, but *not* the control registers.

Rn	Register direct
@Rn	Register indirect
@Rn+	Register indirect with post-increment

@Rn-	Register indirect with post-decrement	
@-SP	Register indirect with pre-decrement	
@(disp, Rn)		
	Register indirect with displacement	
addr	PC relative address (for branch or rep).	
#imm	Immediate data (the ' <b>#</b> ' is optional and ignored)	

# 9.11.2.6 @WORD Modifier

Any symbol followed by **@word** will be replaced by the symbol's value shifted right by 2. This is used in situations such as loading a register with the address of a function (or any other code fragment). For example, if you want to load a register with the location of the function **main** then jump to that function, you could do it as follows:

ldi r2, main@word jmp r2

# 9.11.3 Floating Point

The D10V has no hardware floating point, but the .float and .double directives generates IEEE floating-point numbers for compatibility with other development tools.

# 9.11.4 Opcodes

For detailed information on the D10V machine instruction set, see D10V Architecture: A VLIW Microprocessor for Multimedia Applications (Mitsubishi Electric Corp.). as implements all the standard D10V opcodes. The only changes are those described in the section on size modifiers

# 9.12 D30V Dependent Features

# 9.12.1 D30V Options

The Mitsubishi D30V version of **as** has a few machine dependent options.

- '-0' The D30V can often execute two sub-instructions in parallel. When this option is used, **as** will attempt to optimize its output by detecting when instructions can be executed in parallel.
- '-n' When this option is used, as will issue a warning every time it adds a nop instruction.
- '-N' When this option is used, **as** will issue a warning if it needs to insert a nop after a 32-bit multiply before a load or 16-bit multiply instruction.

# 9.12.2 Syntax

The D30V syntax is based on the syntax in Mitsubishi's D30V architecture manual. The differences are detailed below.

# 9.12.2.1 Size Modifiers

The D30V version of **as** uses the instruction names in the D30V Architecture Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? **as** will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either '.s' (short) or '.1' (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write 'bra.s foo'. Objdump and GDB will always append '.s' or '.1' to instructions which have both short and long forms.

# 9.12.2.2 Sub-Instructions

The D30V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the next section.

# 9.12.2.3 Special Characters

A semicolon (';') can be used anywhere on a line to start a comment that extends to the end of the line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially unless you use the '-0' option.

To specify the executing order, use the following symbols:

'->' Sequential with instruction on the left first.	
---	--

'<-' Sequential with instruction on the right first.

## '||' Parallel

The D30V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. For example

## abs r2,r3 -> abs r4,r5

Execute these sequentially. The instruction on the right is in the right container and is executed second.

### abs r2,r3 <- abs r4,r5

Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

## abs r2,r3 || abs r4,r5

Execute these in parallel.

```
ldw r2,@(r3,r4) ||
```

mulx r6,r8,r9

Two-line format. Execute these in parallel.

## mulx a0,r8,r9

stw r2,@(r3,r4)

Two-line format. Execute these sequentially unless '-O' option is used. If the '-O' option is used, the assembler will determine if the instructions could be done in parallel (the above two instructions can be done in parallel), and if so, emit them as parallel instructions. The assembler will put them in the proper containers. In the above example, the assembler will put the 'stw' instruction in left container and the 'mulx' instruction in the right container.

## stw r2,@(r3,r4) ->

mulx a0,r8,r9

Two-line format. Execute the 'stw' instruction followed by the 'mulx' instruction sequentially. The first instruction goes in the left container and the second instruction goes into right container. The assembler will give an error if the machine ordering constraints are violated.

### stw r2,@(r3,r4) <-

### mulx a0,r8,r9

Same as previous example, except that the 'mulx' instruction is executed before the 'stw' instruction.

Since '\$' has no special meaning, you may use it in symbol names.

# 9.12.2.4 Guarded Execution

**as** supports the full range of guarded execution directives for each instruction. Just append the directive after the instruction proper. The directives are:

'/tx'	Execute the instruction if flag f0 is true.
'/fx'	Execute the instruction if flag f0 is false.
'/xt'	Execute the instruction if flag f1 is true.
'/xf'	Execute the instruction if flag f1 is false.
'/tt'	Execute the instruction if both flags f0 and f1 are true.
'/tf'	Execute the instruction if flag f0 is true and flag f1 is false.

# 9.12.2.5 Register Names

You can use the predefined symbols 'r0' through 'r63' to refer to the D30V registers. You can also use 'sp' as an alias for 'r63' and 'link' as an alias for 'r62'. The accumulators are 'a0' and 'a1'.

The D30V also has predefined symbols for these control registers and status bits:

psw	Processor Status Word
bpsw	Backup Processor Status Word
pc	Program Counter
bpc	Backup Program Counter
rpt_c	Repeat Count
rpt_s	Repeat Start address
rpt_e	Repeat End address
mod_s	Modulo Start address
mod_e	Modulo End address
iba	Instruction Break Address
fO	Flag 0
f1	Flag 1
f2	Flag 2
f3	Flag 3
f4	Flag 4
f5	Flag 5
f6	Flag 6
f7	Flag 7
S	Same as flag 4 (saturation flag)
v	Same as flag 5 (overflow flag)

va Same as flag 6 (sticky overflow flag)

c Same as flag 7 (carry/borrow flag)

b Same as flag 7 (carry/borrow flag)

# 9.12.2.6 Addressing Modes

as understands the following addressing modes for the D30V. Rn in the following refers to any of the numbered registers, but *not* the control registers.

Rn	Register direct	
@Rn	Register indirect	
@Rn+	Register indirect with post-increment	
@Rn-	Register indirect with post-decrement	
@-SP	Register indirect with pre-decrement	
@(disp, Rn)		
-	Register indirect with displacement	
addr	PC relative address (for branch or rep).	
#imm	Immediate data (the ' <b>#</b> ' is optional and ignored)	

# 9.12.3 Floating Point

The D30V has no hardware floating point, but the .float and .double directives generates IEEE floating-point numbers for compatibility with other development tools.

# 9.12.4 Opcodes

For detailed information on the D30V machine instruction set, see D30V Architecture: A VLIW Microprocessor for Multimedia Applications (Mitsubishi Electric Corp.). as implements all the standard D30V opcodes. The only changes are those described in the section on size modifiers

# 9.13 Epiphany Dependent Features

# 9.13.1 Options

as has two additional command-line options for the Epiphany architecture.

-mepiphany

Specifies that the both 32 and 16 bit instructions are allowed. This is the default behavior.

-mepiphany16

Restricts the permitted instructions to just the 16 bit set.

# 9.13.2 Epiphany Syntax

# 9.13.2.1 Special Characters

The presence of a ';' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ''' character can be used to separate statements on the same line.

# 9.14 H8/300 Dependent Features

# 9.14.1 Options

The Renesas H8/300 version of as has one machine-dependent option:

### -h-tick-hex

Support H'00 style hex constants in addition to 0x00 style.

### -mach=name

Sets the H8300 machine variant. The following machine names are recognised: h8300h, h8300hn, h8300s, h8300sn, h8300sx and h8300sxn.

## 9.14.2 Syntax

# 9.14.2.1 Special Characters

;;' is the line comment character.

'\$' can be used instead of a newline to separate statements. Therefore you may not use '\$' in symbol names on the H8/300.

# 9.14.2.2 Register Names

You can use predefined symbols of the form 'rnh' and 'rnl' to refer to the H8/300 registers as sixteen 8-bit general-purpose registers. *n* is a digit from '0' to '7'); for instance, both 'r0h' and 'r7l' are valid register names.

You can also use the eight predefined symbols 'rn' to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols 'ern' ('er0' ... 'er7') to refer to the 32-bit general purpose registers.

The two control registers are called pc (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and ccr (condition code register; an 8-bit register). r7 is used as the stack pointer, and can also be called sp.

## 9.14.2.3 Addressing Modes

as understands the following addressing modes for the H8/300:

rn	Register direct
@rn	Register indirect
@(d, rn) @(d:16, rn @(d:24, rn	
@r <i>n</i> +	Register indirect with post-increment
0-rn	Register indirect with pre-decrement

@aa	
@aa:8	
@aa:16	
@aa:24	Absolute address aa. (The address size ':24' only makes sense on the H8/300H.)
#xx	
#xx:8	
#xx:16	
#xx:32	Immediate data xx. You may specify the ':8', ':16', or ':32' for clarity, if you wish; but as neither requires this nor uses it—the data size required is taken from context.
00aa	
@@aa:8	Memory indirect. You may specify the ':8' for clarity, if you wish; but as neither requires this nor uses it.

# 9.14.3 Floating Point

The H8/300 family has no hardware floating point, but the .float directive generates IEEE floating-point numbers for compatibility with other development tools.

# 9.14.4 H8/300 Machine Directives

as has the following machine-dependent directives for the H8/300:

.h8300h	Recognize and emit additional instructions for the $H8/300H$ variant, and also make .int emit 32-bit numbers rather than the usual (16-bit) for the $H8/300$ family.
.h8300s	Recognize and emit additional instructions for the H8S variant, and also make .int emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.
.h8300hn	Recognize and emit additional instructions for the $H8/300H$ variant in normal mode, and also make .int emit 32-bit numbers rather than the usual (16-bit) for the $H8/300$ family.
.h8300sn	Recognize and emit additional instructions for the H8S variant in normal mode,

.h8300sn Recognize and emit additional instructions for the H8S variant in normal model and also make .int emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

On the H8/300 family (including the H8/300H) '.word' directives generate 16-bit numbers.

## 9.14.5 Opcodes

For detailed information on the H8/300 machine instruction set, see H8/300 Series Programming Manual. For information specific to the H8/300H, see H8/300H Series Programming Manual (Renesas).

as implements all the standard H8/300 opcodes. No additional pseudo-instructions are needed on this family.

Four H8/300 instructions (add, cmp, mov, sub) are defined with variants using the suffixes '.b', '.w', and '.1' to specify the size of a memory operand. as supports these suffixes, but does not require them; since one of the operands is always a register, as can deduce the correct size.

For example, since r0 refers to a 16-bit register,

mov r0,@foo is equivalent to mov.w r0,@foo

If you use the size suffixes, **as** issues a warning when the suffix and the register size do not match.

# 9.15 HPPA Dependent Features

# 9.15.1 Notes

As a back end for GNU CC **as** has been thoroughly tested and should work extremely well. We have tested it only minimally on hand written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original **as** port (version 1.3X) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA **as** port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

# 9.15.2 Options

as has no machine-dependent command-line options for the HPPA.

## 9.15.3 Syntax

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences.

First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

Some obscure expression parsing problems may affect hand written code which uses the **spop** instructions, or code which makes significant use of the ! line separator.

**as** is much less forgiving about missing arguments and other similar oversights than the HP assembler. **as** notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug.

Finally, as allows you to use an external symbol without explicitly importing the symbol. *Warning:* in the future this will be an error for HPPA targets.

Special characters for HPPA targets include:

';' is the line comment character.

'!' can be used instead of a newline to separate statements.

Since '\$' has no special meaning, you may use it in symbol names.

# 9.15.4 Floating Point

The HPPA family uses IEEE floating-point numbers.

# 9.15.5 HPPA Assembler Directives

as for the HPPA supports many additional directives for compatibility with the native assembler. This section describes them only briefly. For detailed information on HPPA-specific assembler directives, see *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001).

as does not support the following assembler directives described in the HP manual:

.endm .liston

.enter	.locct
.leave	.macro
.listoff	

Beyond those implemented for compatibility, as supports one additional assembler directive for the HPPA: .param. It conveys register argument locations for static functions. Its syntax closely follows the .export directive.

These are the additional directives in **as** for the HPPA:

### .block n

.blockz n Reserve n bytes of storage, and initialize them to zero.

- .call Mark the beginning of a procedure call. Only the special case with *no arguments* is allowed.
- .callinfo [ param=value, ... ] [ flag, ... ]

Specify a number of parameters and flags that define the environment for a procedure.

param may be any of 'frame' (frame size), 'entry\_gr' (end of general register range), 'entry\_fr' (end of float register range), 'entry\_sr' (end of space register range).

The values for *flag* are 'calls' or 'caller' (proc has subroutines), 'no\_calls' (proc does not call subroutines), 'save\_rp' (preserve return pointer), 'save\_sp' (proc preserves stack pointer), 'no\_unwind' (do not unwind this proc), 'hpux\_int' (proc is interrupt routine).

.code Assemble into the standard section called '\$TEXT\$', subsection '\$CODE\$'.

.copyright "*string*"

In the SOM object format, insert *string* into the object code, marked as a copyright string.

.copyright "string"

In the ELF object format, insert *string* into the object code, marked as a version string.

- .enter Not yet supported; the assembler rejects programs containing this directive.
- .entry Mark the beginning of a procedure.

.exit Mark the end of a procedure.

.export name [ ,typ ] [ ,param=r ]

Make a procedure name available to callers. *typ*, if present, must be one of 'absolute', 'code' (ELF only, not SOM), 'data', 'entry', 'data', 'entry', 'millicode', 'plabel', 'pri\_prog', or 'sec\_prog'.

param, if present, provides either relocation information for the procedure arguments and result, or a privilege level. param may be 'argwn' (where n ranges from 0 to 3, and indicates one of four one-word arguments); 'rtnval' (the procedure's result); or 'priv\_lev' (privilege level). For arguments or the result, r specifies how to relocate, and must be one of 'no' (not relocatable), 'gr' (argument is in general register), 'fr' (in floating point register), or 'fu' (upper half of float register). For 'priv\_lev', r is an integer.

.half n Define a two-byte integer constant n; synonym for the portable as directive .short.

### .import name [, typ]

Converse of .export; make a procedure available to call. The arguments use the same conventions as the first two arguments for .export.

#### .label name

Define name as a label for the current assembly location.

.leave Not yet supported; the assembler rejects programs containing this directive.

### .origin 1c

Advance location counter to *lc*. Synonym for the as portable directive .org.

### .param name [,typ] [,param=r]

Similar to .export, but used for static procedures.

- .proc Use preceding the first statement of a procedure.
- .procend Use following the last statement of a procedure.
- label .reg expr

Synonym for .equ; define *label* with the absolute expression *expr* as its value.

### .space secname [, params]

Switch to section *secname*, creating a new section by that name if necessary. You may only use *params* when creating a new section, not when switching to an existing one. *secname* may identify a section by number rather than by name.

If specified, the list *params* declares attributes of the section, identified by keywords. The keywords recognized are 'spnum=exp' (identify this section by the number exp, an absolute expression), 'sort=exp' (order sections according to this sort key when linking; exp is an absolute expression), 'unloadable' (section contains no loadable data), 'notdefined' (this section defined elsewhere), and 'private' (data in this section not available to other programs).

#### .spnum secnam

Allocate four bytes of storage, and initialize them with the section number of the section named *secnam*. (You can define the section number with the HPPA .space directive.)

## .string "str"

Copy the characters in the string str to the object file. See Section 3.6.1.1 [Strings], page 33, for information on escape sequences you can use in **as** strings.

*Warning!* The HPPA version of .string differs from the usual as definition: it does *not* write a zero byte after copying *str*.

#### .stringz "str"

Like .string, but appends a zero byte after copying str to object file.

Similar to .space, but selects a subsection *name* within the current section. You may only specify *params* when you create a subsection (in the first instance of .subspa for this *name*).

If specified, the list *params* declares attributes of the subsection, identified by keywords. The keywords recognized are 'quad=expr' ("quadrant" for this subsection), 'align=expr' (alignment for beginning of this subsection; a power of two), 'access=expr' (value for "access rights" field), 'sort=expr' (sorting order for this subspace in link), 'code\_only' (subsection contains only code), 'unloadable' (subsection cannot be loaded into memory), 'comdat' (subsection is comdat), 'common' (subsection is common block), 'dup\_comm' (subsection may have duplicate names), or 'zero' (subsection is all zeros, do not write in object file).

.nsubspa always creates a new subspace with the given name, even if one with the same name already exists.

'comdat', 'common' and 'dup\_comm' can be used to implement various flavors of one-only support when using the SOM linker. The SOM linker only supports specific combinations of these flags. The details are not documented. A brief description is provided here.

'comdat' provides a form of linkonce support. It is useful for both code and data subspaces. A 'comdat' subspace has a key symbol marked by the 'is\_comdat' flag or 'ST\_COMDAT'. Only the first subspace for any given key is selected. The key symbol becomes universal in shared links. This is similar to the behavior of 'secondary\_def' symbols.

'common' provides Fortran named common support. It is only useful for data subspaces. Symbols with the flag 'is\_common' retain this flag in shared links. Referencing a 'is\_common' symbol in a shared library from outside the library doesn't work. Thus, 'is\_common' symbols must be output whenever they are needed.

'common' and 'dup\_comm' together provide Cobol common support. The subspaces in this case must all be the same length. Otherwise, this support is similar to the Fortran common support.

'dup\_comm' by itself provides a type of one-only support for code. Only the first 'dup\_comm' subspace is selected. There is a rather complex algorithm to compare subspaces. Code symbols marked with the 'dup\_common' flag are hidden. This support was intended for "C++ duplicate inlines".

A simplified technique is used to mark the flags of symbols based on the flags of their subspace. A symbol with the scope SS\_UNIVERSAL and type ST\_ENTRY, ST\_CODE or ST\_DATA is marked with the corresponding settings of 'comdat', 'common' and 'dup\_comm' from the subspace, respectively. This avoids having to introduce additional directives to mark these symbols. The HP assembler sets 'is\_common' from 'common'. However, it doesn't set the 'dup\_common' from 'dup\_comm'. It doesn't have 'comdat' support.

```
.version "str"
```

Write str as version identifier in object code.

# 9.15.6 Opcodes

For detailed information on the HPPA machine instruction set, see *PA-RISC* Architecture and Instruction Set Reference Manual (HP 09740-90039).

# 9.16 80386 Dependent Features

The i386 version **as** supports both the original Intel 386 architecture in both 16 and 32-bit mode as well as AMD x86-64 architecture extending the Intel architecture to 64-bits.

# **9.16.1** Options

The i386 version of **as** has a few machine dependent options:

--32 | --x32 | --64

Select the word size, either 32 bits or 64 bits. '--32' implies Intel i386 architecture, while '--x32' and '--64' imply AMD x86-64 architecture with 32-bit or 64-bit word-size respectively.

These options are only available with the ELF object file format, and require that the necessary BFD support has been included (on a 32-bit platform you have to add –enable-64-bit-bfd to configure enable 64-bit usage and use x86-64 as target platform).

- -n By default, x86 GAS replaces multiple nop instructions used for alignment within code sections with multi-byte nop instructions such as leal 0(%esi,1),%esi. This switch disables the optimization if a single byte nop (0x90) is explicitly specified as the fill byte for alignment.
- --divide On SVR4-derived platforms, the character '/' is treated as a comment character, which means that it cannot be used in expressions. The '--divide' option turns '/' into a normal character. This does not disable '/' at the beginning of a line starting a comment, or affect using '#' for starting a comment.

## -march=CPU[+EXTENSION...]

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: i8086, i186, i286, i386, i486, i586, i686, pentium, pentiumpro, pentiumii, pentiumiii, pentium4, prescott, nocona, core, core2, corei7, iamcu, k6, k6\_2, athlon, opteron, k8, amdfam10, bdver1, bdver2, bdver3, bdver4, znver1, znver2, znver3, znver4, btver1, btver2, generic32 and generic64.

In addition to the basic instruction set, the assembler can be told to accept various extension mnemonics. For example, -march=i686+sse4+vmx extends i686 with sse4 and vmx. The following extensions are currently supported: 8087, 287, 387, 687, cmov, fxsr, mmx, sse, sse2, sse3, sse4a, ssse3, sse4.1, sse4.2, sse4, avx, avx2, lahf\_sahf, monitor, adx, rdseed, prfchw, smap, mpx, sha, rdpid, ptwrite, cet, gfni, vaes, vpclmulqdq, prefetchwt1, clflushopt, se1, clwb, movdiri, movdir64b, enqcmd, serialize, tsxldtrk, kl, widekl, hreset, avx512f, avx512cd, avx512er, avx512pf, avx512vl, avx512bw, avx512dq, avx512ifma, avx512vbmi, avx512\_4fmaps. avx512\_4vnniw, avx512\_vpopcntdq, avx512\_vbmi2. avx512\_vnni, avx512\_bitalg, avx512\_vp2intersect, tdx, avx512\_bf16, avx\_vnni, avx512\_fp16, prefetchi, avx\_ifma, avx\_vnni\_int8, cmpccxadd, wrmsrns, msrlist, avx\_ne\_convert, rao\_int, fred, lkgs, amx\_int8, amx\_bf16, amx\_fp16, amx\_complex, amx\_tile, vmx, vmfunc, smx, xsave, xsaveopt, xsavec, xsaves, aes, pclmul, fsgsbase, rdrnd, f16c, bmi2, fma, movbe, ept, lzcnt, popcnt, hle, rtm, tsx, invpcid, clflush, mwaitx, clzero, wbnoinvd, pconfig, waitpkg, uintr, cldemote, rdpru, mcommit, sev\_es, lwp, fma4, xop, cx16, syscall, rdtscp, 3dnow, 3dnowa, sse4a, sse5, snp, invlpgb, tlbsync, svme and padlock. Note that these extension mnemonics can be prefixed with no to revoke the respective (and any dependent) functionality.

When the .arch directive is used with -march, the .arch directive will take precedent.

#### -mtune=CPU

This option specifies a processor to optimize for. When used in conjunction with the **-march** option, only instructions of the processor specified by the **-march** option will be generated.

Valid CPU values are identical to the processor list of -march=CPU.

-msse2avx

This option specifies that the assembler should encode SSE instructions with VEX prefix.

## -muse-unaligned-vector-move

This option specifies that the assembler should encode aligned vector move as unaligned vector move.

#### -msse-check=none

-msse-check=warning

#### -msse-check=error

These options control if the assembler should check SSE instructions. -mssecheck=none will make the assembler not to check SSE instructions, which is the default. -msse-check=warning will make the assembler issue a warning for any SSE instruction. -msse-check=error will make the assembler issue an error for any SSE instruction.

#### -mavxscalar=128

#### -mavxscalar=256

These options control how the assembler should encode scalar AVX instructions. -mavxscalar=128 will encode scalar AVX instructions with 128bit vector length, which is the default. -mavxscalar=256 will encode scalar AVX instructions with 256bit vector length.

WARNING: Don't use this for production code - due to CPU errata the resulting code may not work on certain models.

### -mvexwig=0

#### -mvexwig=1

These options control how the assembler should encode VEX.W-ignored (WIG) VEX instructions. -mvexwig=0 will encode WIG VEX instructions with vex.w = 0, which is the default. -mvexwig=1 will encode WIG EVEX instructions with vex.w = 1.

WARNING: Don't use this for production code - due to CPU errata the resulting code may not work on certain models.

# -mevexlig=128 -mevexlig=256

## -mevexlig=512

These options control how the assembler should encode length-ignored (LIG) EVEX instructions. -mevexlig=128 will encode LIG EVEX instructions with 128bit vector length, which is the default. -mevexlig=256 and -mevexlig=512 will encode LIG EVEX instructions with 256bit and 512bit vector length, respectively.

## -mevexwig=0

## -mevexwig=1

These options control how the assembler should encode w-ignored (WIG) EVEX instructions. -mevexwig=0 will encode WIG EVEX instructions with evex.w = 0, which is the default. -mevexwig=1 will encode WIG EVEX instructions with evex.w = 1.

## -mmnemonic=att

#### -mmnemonic=intel

This option specifies instruction mnemonic for matching instructions. The .att\_mnemonic and .intel\_mnemonic directives will take precedent.

#### -msyntax=att

## -msyntax=intel

This option specifies instruction syntax when processing instructions. The .att\_syntax and .intel\_syntax directives will take precedent.

#### -mnaked-reg

This option specifies that registers don't require a '%' prefix. The .att\_syntax and .intel\_syntax directives will take precedent.

#### -madd-bnd-prefix

This option forces the assembler to add BND prefix to all branches, even if such prefix was not explicitly specified in the source code.

### -mno-shared

On ELF target, the assembler normally optimizes out non-PLT relocations against defined non-weak global branch targets with default visibility. The '-mshared' option tells the assembler to generate code which may go into a shared library where all non-weak global branch targets with default visibility can be preempted. The resulting code is slightly bigger. This option only affects the handling of branch instructions.

#### -mbig-obj

On PE/COFF target this option forces the use of big object file format, which allows more than 32768 sections.

## -momit-lock-prefix=no

## -momit-lock-prefix=yes

These options control how the assembler should encode lock prefix. This option is intended as a workaround for processors, that fail on lock prefix. This option can only be safely used with single-core, single-thread computers -momit-lock-

prefix=yes will omit all lock prefixes. -momit-lock-prefix=no will encode lock prefix as usual, which is the default.

### -mfence-as-lock-add=no

## -mfence-as-lock-add=yes

These options control how the assembler should encode lfence, mfence and sfence. -mfence-as-lock-add=yes will encode lfence, mfence and sfence as 'lock addl \$0x0, (%rsp)' in 64-bit mode and 'lock addl \$0x0, (%esp)' in 32-bit mode. -mfence-as-lock-add=no will encode lfence, mfence and sfence as usual, which is the default.

## -mrelax-relocations=no

## -mrelax-relocations=yes

These options control whether the assembler should generate relax relocations, R\_386\_GOT32X, in 32-bit mode, or R\_X86\_64\_GOTPCRELX and R\_X86\_64\_REX\_GOTPCRELX, in 64-bit mode. -mrelax-relocations=yes will generate relax relocations. -mrelax-relocations=no will not generate relax relocations. The default can be controlled by a configure option --enable-x86-relax-relocations.

### -malign-branch-boundary=NUM

This option controls how the assembler should align branches with segment prefixes or NOP. *NUM* must be a power of 2. It should be 0 or no less than 16. Branches will be aligned within *NUM* byte boundary. -malign-branch-boundary=0, which is the default, doesn't align branches.

## -malign-branch=TYPE[+TYPE...]

This option specifies types of branches to align. *TYPE* is combination of 'jcc', which aligns conditional jumps, 'fused', which aligns fused conditional jumps, 'jmp', which aligns unconditional jumps, 'call' which aligns calls, 'ret', which aligns rets, 'indirect', which aligns indirect jumps and calls. The default is -malign-branch=jcc+fused+jmp.

## -malign-branch-prefix-size=NUM

This option specifies the maximum number of prefixes on an instruction to align branches. *NUM* should be between 0 and 5. The default *NUM* is 5.

## -mbranches-within-32B-boundaries

This option aligns conditional jumps, fused conditional jumps and unconditional jumps within 32 byte boundary with up to 5 segment prefixes on an instruction. It is equivalent to -malign-branch-boundary=32 -malignbranch=jcc+fused+jmp-malign-branch-prefix-size=5. The default doesn't align branches.

## -mlfence-after-load=no

## -mlfence-after-load=yes

These options control whether the assembler should generate lfence after load instructions. -mlfence-after-load=yes will generate lfence. -mlfence-afterload=no will not generate lfence, which is the default.

```
-mlfence-before-indirect-branch=none
-mlfence-before-indirect-branch=all
-mlfence-before-indirect-branch=register
-mlfence-before-indirect-branch=memory
```

These options control whether the assembler should generate lfence before indirect near branch instructions. -mlfence-before-indirect-branch=all will generate lfence before indirect near branch via register and issue a warning before indirect near branch via memory. It also implicitly sets -mlfencebefore-ret=shl when there's no explicit -mlfence-before-ret=. -mlfencebefore-indirect-branch=register will generate lfence before indirect near branch via register. -mlfence-before-indirect-branch=memory will issue a warning before indirect near branch via memory. -mlfence-before-indirectbranch=none will not generate lfence nor issue warning, which is the default. Note that lfence won't be generated before indirect near branch via register with -mlfence-after-load=yes since lfence will be generated after loading branch target register.

```
-mlfence-before-ret=none
```

```
-mlfence-before-ret=shl
```

```
-mlfence-before-ret=or
```

```
-mlfence-before-ret=yes
```

```
-mlfence-before-ret=not
```

These options control whether the assembler should generate lfence before ret. -mlfence-before-ret=or will generate generate or instruction with lfence. -mlfence-before-ret=shl/yes will generate shl instruction with lfence. -mlfence-before-ret=not will generate not instruction with lfence. -mlfence-before-ret=none will not generate lfence, which is the default.

## -mx86-used-note=no

```
-mx86-used-note=yes
```

These options control whether the assembler should generate GNU\_PROPERTY\_X86\_ISA\_1\_USED and GNU\_PROPERTY\_X86\_FEATURE\_2\_USED GNU property notes. The default can be controlled by the --enable-x86-used-note configure option.

## -mevexrcig=rne

## -mevexrcig=rd

```
-mevexrcig=ru
```

```
-mevexrcig=rz
```

These options control how the assembler should encode SAE-only EVEX instructions. -mevexrcig=rne will encode RC bits of EVEX instruction with 00, which is the default. -mevexrcig=rd, -mevexrcig=ru and -mevexrcig=rz will encode SAE-only EVEX instructions with 01, 10 and 11 RC bits, respectively.

-mamd64

### -mintel64

This option specifies that the assembler should accept only AMD64 or Intel64 ISA in 64-bit mode. The default is to accept common, Intel64 only and AMD64 ISAs.

## -00 | -0 | -01 | -02 | -0s

Optimize instruction encoding with smaller instruction size. '-0' and '-01' encode 64-bit register load instructions with 64-bit immediate as 32-bit register load instructions with 31-bit or 32-bits immediates, encode 64-bit register clearing instructions with 32-bit register clearing instructions, encode 256-bit/512-bit VEX/EVEX vector register clearing instructions with 128-bit VEX vector register clearing instructions with 128-bit VEX vector register clearing instructions, and encode 128-bit/256-bit EVEX packed integer logical instructions with 128-bit/256-bit VEX packed integer logical.

'-O2' includes '-O1' optimization plus encodes 256-bit/512-bit EVEX vector register clearing instructions with 128-bit EVEX vector register clearing instructions. In 64-bit mode VEX encoded instructions with commutative source operands will also have their source operands swapped if this allows using the 2-byte VEX prefix form instead of the 3-byte one. Certain forms of AND as well as OR with the same (register) operand specified twice will also be changed to TEST.

'-Os' includes '-O2' optimization plus encodes 16-bit, 32-bit and 64-bit register tests with immediate as 8-bit register test with immediate. '-O0' turns off this optimization.

## 9.16.2 x86 specific Directives

## .lcomm symbol , length[, alignment]

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. Since *symbol* is not declared global, it is normally not visible to 1d. The optional third parameter, *alignment*, specifies the desired alignment of the symbol in the bss section.

This directive is only available for COFF based x86 targets.

### .largecomm symbol , length[, alignment]

This directive behaves in the same way as the comm directive except that the data is placed into the *.lbss* section instead of the *.bss* section Section 7.13 [Comm], page 58.

The directive is intended to be used for data which requires a large amount of space, and it is only available for ELF based x86\_64 targets.

## .value expression [, expression]

This directive behaves in the same way as the .short directive, taking a series of comma separated expressions and storing them as two-byte wide values into the current section.

.insn [prefix[,...]] [encoding] major-opcode[+r|/extension] [,operand[,...]] This directive allows composing instructions which as may not know about yet, or which it has no way of expressing (which can be the case for certain alternative encodings). It assumes certain basic structure in how operands are encoded, and it also only recognizes - with a few extensions as per below - operands otherwise valid for instructions. Therefore there is no guarantee that everything can be expressed (e.g. the original Intel Xeon Phi's MVEX encodings cannot be expressed).

- prefix expresses one or more opcode prefixes in the usual way. Legacy encoding prefixes altering meaning (0x66, 0xF2, 0xF3) may be specified as high byte of <major-opcode> (perhaps already including an encoding space prefix). Note that there can only be one such prefix. Segment overrides are better specified in the respective memory operand, as long as there is one.
- *encoding* is used to specify VEX, XOP, or EVEX encodings. The syntax tries to resemble that used in documentation:
  - VEX[.len][.prefix][.space][.w]
  - EVEX[.len][.prefix][.space][.w]
  - XOPspace[.len][.prefix][.w]

## Here

- len can be LIG, 128, 256, or (EVEX only) 512 as well as L0 / L1 for VEX / XOP and L0...L3 for EVEX
- prefix can be NP, 66, F3, or F2
- *space* can be
  - 0f, 0f38, 0f3a, or M0...M31 for VEX
  - 08...1f for XOP
  - 0f, 0f38, 0f3a, or M0...M15 for EVEX
- w can be WIG, WO, or W1

## Defaults:

- Omitted *len* means "infer from operand size" if there is at least one sized vector operand, or LIG otherwise. (Obviously *len* has to be omitted when there's EVEX rounding control specified later in the operands.)
- Omitted *prefix* means NP.
- Omitted space (VEX/EVEX only) implies encoding space is taken from major-opcode.
- Omitted w means "infer from GPR operand size" in 64-bit code if there is at least one GPR(-like) operand, or WIG otherwise.
- major-opcode is an absolute expression specifying the instruction opcode. Legacy encoding prefixes altering encoding space (0x0f, 0x0f38, 0x0f3a) have to be specified as high byte(s) here. "Degenerate" ModR/M bytes, as present in e.g. certain FPU opcodes or sub-spaces like that of major opcode 0x0f01, generally want encoding as immediate operand (such opcodes wouldn't normally have non-immediate operands); in some cases it may be possible to also encode these as low byte of the major opcode, but there are potential ambiguities. Also note that after stripping encoding prefixes, the residual has to fit in two bytes (16 bits). +r can be suffixed to the major opcode expression to specify register-only encoding forms not using

a ModR/M byte. <code>/extension</code> can alternatively be suffixed to the major opcode expression to specify an extension opcode, encoded in bits 3-5 of the ModR/M byte.

• operand is an instruction operand expressed the usual way. Register operands are primarily used to express register numbers as encoded in ModR/M byte and REX/VEX/XOP/EVEX prefixes. In certain cases the register type (really: size) is also used to derive other encoding attributes, if these aren't specified explicitly. Note that there is no consistency checking among operands, so entirely bogus mixes of operands are possible. Note further that only operands actually encoded in the instruction should be specified. Operands like '%cl' in shift/rotate instructions have to be omitted, or else they'll be encoded as an ordinary (register) operand. Operand order may also not match that of the actual instruction (see below).

Encoding of operands: While for a memory operand (of which there can be only one) it is clear how to encode it in the resulting ModR/M byte, register operands are encoded strictly in this order (operand counts do not include immediate ones in the enumeration below, and if there was an extension opcode specified it counts as a register operand; VEX.vvvv is meant to cover XOP and EVEX as well):

- VEX.vvvv for 1-register-operand VEX/XOP/EVEX insns,
- ModR/M.rm, ModR/M.reg for 2-operand insns,
- ModR/M.rm, VEX.vvvv, ModR/M.reg for 3-operand insns, and
- Imm{4,5}, ModR/M.rm, VEX.vvvv, ModR/M.reg for 4-operand insns,

obviously with the ModR/M.rm slot skipped when there is a memory operand, and obviously with the ModR/M.reg slot skipped when there is an extension opcode. For Intel syntax of course the opposite order applies. With +r (and hence no ModR/M) there can only be a single register operand for legacy encodings. VEX and alike can have two register operands, where the second (first in Intel syntax) would go into VEX.vvvv.

Immediate operands (including immediate-like displacements, i.e. when not part of ModR/M addressing) are emitted in the order specified, regardless of AT&T or Intel syntax. Since it may not be possible to infer the size of such immediates, they can be suffixed by  $\{:sn\}$  or  $\{:un\}$ , representing signed / unsigned immediates of the given number of bits respectively. When emitting such operands, the number of bits will be rounded up to the smallest suitable of 8, 16, 32, or 64. Immediates wider than 32 bits are permitted in 64-bit code only.

For EVEX encoding memory operands with a displacement need to know Disp8 scaling size in order to use an 8-bit displacement. For many instructions this can be inferred from the types of other operands specified. In Intel syntax 'DWORD PTR' and alike can be used to specify the respective size. In AT&T syntax the memory operands can be suffixed by {:dn} to specify the size (in bytes). This can be combined with an embedded broadcast specifier: '8(%eax){1to8:d8}'.

## 9.16.3 i386 Syntactical Considerations

## 9.16.3.1 AT&T Syntax versus Intel Syntax

as now supports assembly using Intel assembler syntax. .intel\_syntax selects Intel mode, and .att\_syntax switches back to the usual AT&T mode for compatibility with the output of gcc. Either of these directives may have an optional argument, prefix, or noprefix specifying whether registers require a '%' prefix. AT&T System V/386 assembler syntax is quite different from Intel syntax. We mention these differences because almost all 80386 documents use Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by '\$'; Intel immediate operands are undelimited (Intel 'push 4' is AT&T 'pushl \$4'). AT&T register operands are preceded by '%'; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by '\*'; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel 'add eax, 4' is 'addl \$4, %eax'. The 'source, dest' convention is maintained for compatibility with previous Unix assemblers. Note that 'bound', 'invlpga', and instructions with 2 immediate operands, such as the 'enter' instruction, do *not* have reversed order. Section 9.16.17 [i386-Bugs], page 198.
- In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of 'b', 'w', 'l' and 'q' specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Mnemonic suffixes of 'x', 'y' and 'z' specify xmm (128-bit vector), ymm (256-bit vector) and zmm (512-bit vector) memory references, only when there's no other way to disambiguate an instruction. Intel syntax accomplishes this by prefixing memory operands (*not* the instruction mnemonics) with 'byte ptr', 'word ptr', 'dword ptr', 'qword ptr', 'xmmword ptr', 'ymmword ptr' and 'zmmword ptr'. Thus, Intel syntax 'mov al, byte ptr foo' is 'movb foo, %al' in AT&T syntax. In Intel syntax, 'fword ptr', 'tbyte ptr' and 'oword ptr' specify 48-bit, 80-bit and 128-bit memory references.

In 64-bit code, 'movabs' can be used to encode the 'mov' instruction with the 64-bit displacement or immediate operand.

- Immediate form long jumps and calls are 'lcall/ljmp \$section, \$offset' in AT&T syntax; the Intel syntax is 'call/jmp far section:offset'. Also, the far return instruction is 'lret \$stack-adjust' in AT&T syntax; Intel syntax is 'ret far stack-adjust'.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

## 9.16.3.2 Special Characters

The presence of a '#' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

If the --divide command-line option has not been specified then the '/' character appearing anywhere on a line also introduces a line comment.

The ';' character can be used to separate statements on the same line.

# 9.16.4 i386-Mnemonics

# 9.16.4.1 Instruction Naming

Instruction mnemonics are suffixed with one character modifiers which specify the size of operands. The letters 'b', 'w', 'l' and 'q' specify byte, word, long and quadruple word operands. If no suffix is specified by an instruction then as tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, 'mov %ax, %bx' is equivalent to 'movw %ax, %bx'; also, 'mov \$1, %bx' is equivalent to 'movw \$1, bx'. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing mnemonic suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the mnemonic suffix.)

When there is no sizing suffix and no (suitable) register operands to deduce the size of memory operands, with a few exceptions and where long operand size is possible in the first place, operand size will default to long in 32- and 64-bit modes. Similarly it will default to short in 16-bit mode. Noteworthy exceptions are

- Instructions with an implicit on-stack operand as well as branches, which default to quad in 64-bit mode.
- Sign- and zero-extending moves, which default to byte size source operands.
- Floating point insns with integer operands, which default to short (for perhaps historical reasons).
- CRC32 with a 64-bit destination, which defaults to a quad source operand.

Different encoding options can be specified via pseudo prefixes:

- '{disp8}' prefer 8-bit displacement.
- '{disp32}' prefer 32-bit displacement.
- '{disp16}' prefer 16-bit displacement.
- '{load}' prefer load-form instruction.
- '{store}' prefer store-form instruction.
- '{vex}' encode with VEX prefix.
- '{vex3}' encode with 3-byte VEX prefix.
- '{evex}' encode with EVEX prefix.
- '{rex}' prefer REX prefix for integer and legacy vector instructions (x86-64 only). Note that this differs from the 'rex' prefix which generates REX prefix unconditionally.
- '{nooptimize}' disable instruction size optimization.

Mnemonics of Intel VNNI/IFMA instructions are encoded with the EVEX prefix by default. The pseudo '{vex}' prefix can be used to encode mnemonics of Intel VNNI/IFMA instructions with the VEX prefix.

The Intel-syntax conversion instructions

- 'cbw' sign-extend byte in '%al' to word in '%ax',
- 'cwde' sign-extend word in '%ax' to long in '%eax',
- 'cwd' sign-extend word in '%ax' to long in '%dx:%ax',
- 'cdq' sign-extend dword in '%eax' to quad in '%edx:%eax',

- 'cdqe' sign-extend dword in '%eax' to quad in '%rax' (x86-64 only),
- 'cqo' sign-extend quad in '%rax' to octuple in '%rdx:%rax' (x86-64 only),

are called 'cbtw', 'cwtl', 'cwtd', 'cltd', 'cltq', and 'cqto' in AT&T naming. as accepts either naming for these instructions.

The Intel-syntax extension instructions

- 'movsx' sign-extend 'reg8/mem8' to 'reg16'.
- 'movsx' sign-extend 'reg8/mem8' to 'reg32'.
- 'movsx' sign-extend 'reg8/mem8' to 'reg64' (x86-64 only).
- 'movsx' sign-extend 'reg16/mem16' to 'reg32'
- 'movsx' sign-extend 'reg16/mem16' to 'reg64' (x86-64 only).
- 'movsxd' sign-extend 'reg32/mem32' to 'reg64' (x86-64 only).
- 'movzx' zero-extend 'reg8/mem8' to 'reg16'.
- 'movzx' zero-extend 'reg8/mem8' to 'reg32'.
- 'movzx' zero-extend 'reg8/mem8' to 'reg64' (x86-64 only).
- 'movzx' zero-extend 'reg16/mem16' to 'reg32'
- 'movzx' zero-extend 'reg16/mem16' to 'reg64' (x86-64 only).

are called 'movsbw/movsxb/movsx', 'movsbl/movsxb/movsx', 'movsbq/movsxb/movsx', 'movsbq/movsxb/movsx', 'movsbl/movsxw', 'movsbq/movsxw', 'movsbl/movsxb/movzxb/movzx', 'movzbl/movzxb/movzxw' and 'movzwq/movzxw' in AT&T syntax.

Far call/jump instructions are 'lcall' and 'ljmp' in AT&T syntax, but are 'call far' and 'jump far' in Intel convention.

## 9.16.4.2 AT&T Mnemonic versus Intel Mnemonic

as supports assembly using Intel mnemonic. .intel\_mnemonic selects Intel mnemonic with Intel syntax, and .att\_mnemonic switches back to the usual AT&T mnemonic with AT&T syntax for compatibility with the output of gcc. Several x87 instructions, 'fadd', 'fdiv', 'fdivp', 'fdivr', 'fdivrp', 'fmul', 'fsub', 'fsubp', 'fsubr' and 'fsubrp', are implemented in AT&T System V/386 assembler with different mnemonics from those in Intel IA32 specification. gcc generates those instructions with AT&T mnemonic.

• 'movslq' with AT&T mnemonic only accepts 64-bit destination register. 'movsxd' should be used to encode 16-bit or 32-bit destination register with both AT&T and Intel mnemonics.

# 9.16.5 Register Naming

Register operands are always prefixed with '%'. The 80386 registers consist of

- the 8 32-bit registers '%eax' (the accumulator), '%ebx', '%ecx', '%edx', '%edi', '%esi', '%ebp' (the frame pointer), and '%esp' (the stack pointer).
- the 8 16-bit low-ends of these: '%ax', '%bx', '%cx', '%dx', '%di', '%si', '%bp', and '%sp'.
- the 8 8-bit registers: '%ah', '%al', '%bh', '%bl', '%ch', '%cl', '%dh', and '%dl' (These are the high-bytes and low-bytes of '%ax', '%bx', '%cx', and '%dx')

- the 6 section registers '%cs' (code section), '%ds' (data section), '%ss' (stack section), '%es', '%fs', and '%gs'.
- the 5 processor control registers '%cr0', '%cr2', '%cr3', '%cr4', and '%cr8'.
- the 6 debug registers '%db0', '%db1', '%db2', '%db3', '%db6', and '%db7'.
- the 2 test registers '%tr6' and '%tr7'.
- the 8 floating point register stack '%st' or equivalently '%st(0)', '%st(1)', '%st(2)', '%st(3)', '%st(4)', '%st(5)', '%st(6)', and '%st(7)'. These registers are overloaded by 8 MMX registers '%mm0', '%mm1', '%mm2', '%mm3', '%mm4', '%mm5', '%mm6' and '%mm7'.
- the 8 128-bit SSE registers registers '%xmm0', '%xmm1', '%xmm2', '%xmm3', '%xmm4', '%xmm5', '%xmm6' and '%xmm7'.

The AMD x86-64 architecture extends the register set by:

- enhancing the 8 32-bit registers to 64-bit: '%rax' (the accumulator), '%rbx', '%rcx', '%rdx', '%rdi', '%rsi', '%rbp' (the frame pointer), '%rsp' (the stack pointer)
- the 8 extended registers '%r8'-'%r15'.
- the 8 32-bit low ends of the extended registers: '%r8d'-'%r15d'.
- the 8 16-bit low ends of the extended registers: '%r8w'-'%r15w'.
- the 8 8-bit low ends of the extended registers: '%r8b'-'%r15b'.
- the 4 8-bit registers: '%sil', '%dil', '%bpl', '%spl'.
- the 8 debug registers: '%db8'-'%db15'.
- the 8 128-bit SSE registers: '%xmm8'-'%xmm15'.

With the AVX extensions more registers were made available:

• the 16 256-bit SSE '%ymm0'-'%ymm15' (only the first 8 available in 32-bit mode). The bottom 128 bits are overlaid with the 'xmm0'-'xmm15' registers.

The AVX512 extensions added the following registers:

- the 32 512-bit registers '%zmm0'-'%zmm31' (only the first 8 available in 32-bit mode). The bottom 128 bits are overlaid with the '%xmm0'-'%xmm31' registers and the first 256 bits are overlaid with the '%ymm0'-'%ymm31' registers.
- the 8 mask registers '%k0'-'%k7'.

## 9.16.6 Instruction Prefixes

Instruction prefixes are used to modify the following instruction. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to change operand and address sizes. (Most instructions that normally operate on 32-bit operands will use 16-bit operands if the instruction has an "operand size" prefix.) Instruction prefixes are best written on the same line as the instruction they act upon. For example, the 'scas' (scan string) instruction is repeated with:

repne scas %es:(%edi),%al

You may also place prefixes on the lines immediately preceding the instruction, but this circumvents checks that **as** does with prefixes, and will not work with all prefixes.

Here is a list of instruction prefixes:

• Section override prefixes 'cs', 'ds', 'ss', 'es', 'fs', 'gs'. These are automatically added by specifying using the section:memory-operand form for memory references.

• Operand/Address size prefixes 'data16' and 'addr16' change 32-bit operands/addresses into 16-bit operands/addresses, while 'data32' and 'addr32' change 16-bit ones (in a .code16 section) into 32-bit operands/addresses. These prefixes *must* appear on the same line of code as the instruction they modify. For example, in a 16-bit .code16 section, you might write:

addr32 jmpl \*(%ebx)

- The bus lock prefix 'lock' inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).
- The wait for coprocessor prefix 'wait' waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.
- The 'rep', 'repe', and 'repne' prefixes are added to string instructions to make them repeat '%ecx' times ('%cx' times if the current address size is 16-bits).
- The 'rex' family of prefixes is used by x86-64 to encode extensions to i386 instruction set. The 'rex' prefix has four bits an operand size overwrite (64) used to change operand size from 32-bit to 64-bit and X, Y and Z extensions bits used to extend the register set.

You may write the 'rex' prefixes directly. The 'rex64xyz' instruction emits 'rex' prefix with all the bits set. By omitting the 64, x, y or z you may write other prefixes as well. Normally, there is no need to write the prefixes explicitly, since gas will automatically generate them based on the instruction operands.

## 9.16.7 Memory References

An Intel syntax indirect memory reference of the form

```
section:[base + index*scale + disp]
```

is translated into the AT&T syntax

```
section:disp(base, index, scale)
```

where base and index are the optional 32-bit base and index registers, disp is the optional displacement, and scale, taking the values 1, 2, 4, and 8, multiplies index to calculate the address of the operand. If no scale is specified, scale is taken to be 1. section specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax must be preceded by a '%'. If you specify a section override which coincides with the default section register, **as** does not output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: '-4(%ebp)', Intel: '[ebp - 4]'

base is '%ebp'; disp is '-4'. section is missing, and the default section is used ('%ss' for addressing with '%ebp' as the base register). index, scale are both missing.

# AT&T: 'foo(,%eax,4)', Intel: '[foo + eax\*4]'

index is '%eax' (scaled by a scale 4); disp is 'foo'. All other fields are missing. The section register here defaults to '%ds'. AT&T: 'foo(,1)'; Intel '[foo]'

This uses the value pointed to by 'foo' as a memory operand. Note that base and index are both missing, but there is only one ','. This is a syntactic exception.

AT&T: '%gs:foo'; Intel 'gs:foo'

This selects the contents of the variable 'foo' with section register section being '%gs'.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with '\*'. If no '\*' is specified, as always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand, but no register operand, *must* specify its size (byte, word, long, or quadruple) with an instruction mnemonic suffix ('b', 'w', 'l' or 'q', respectively).

The x86-64 architecture adds an RIP (instruction pointer relative) addressing. This addressing mode is specified by using 'rip' as a base register. Only constant offsets are valid. For example:

AT&T: '1234(%rip)', Intel: '[rip + 1234]'

Points to the address 1234 bytes past the end of the current instruction.

AT&T: 'symbol(%rip)', Intel: '[rip + symbol]'

Points to the symbol in RIP relative way, this is shorter than the default absolute addressing.

Other addressing modes remain unchanged in x86-64 architecture, except registers used are 64-bit instead of 32-bit.

## 9.16.8 Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long displacement is used. We do not support word (16-bit) displacement jumps in 32-bit mode (i.e. prefixing the jump instruction with the 'data16' instruction prefix), since the 80386 insists upon masking '%eip' to 16 bits after the word displacement is added. (See also see Section 9.16.15 [i386-Arch], page 196)

Note that the 'jcxz', 'jecxz', 'loop', 'loopz', 'loope', 'loopnz' and 'loopne' instructions only come in byte displacements, so that if you use these instructions (gcc does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding 'jcxz foo' to

```
jcxz cx_zero
jmp cx_nonzero
cx_zero: jmp foo
cx_nonzero:
```

## 9.16.9 Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an instruction mnemonic suffix and a constructor associated with it. Instruction

mnemonic suffixes specify the operand's data type. Constructors build these data types into memory.

- Floating point constructors are '.float' or '.single', '.double', '.tfloat', '.hfloat', and '.bfloat16' for 32-, 64-, 80-, and 16-bit (two flavors) formats respectively. The former three correspond to instruction mnemonic suffixes 's', '1', and 't'. 't' stands for 80-bit (ten byte) real. The 80387 only supports this format via the 'fldt' (load 80-bit real to stack top) and 'fstpt' (store 80-bit real and pop stack) instructions.
- Integer constructors are '.word', '.long' or '.int', and '.quad' for the 16-, 32-, and 64-bit integer formats. The corresponding instruction mnemonic suffixes are 's' (short), 'l' (long), and 'q' (quad). As with the 80-bit real format, the 64-bit 'q' format is only present in the 'fildq' (load quad integer to stack top) and 'fistpq' (store quad integer and pop stack) instructions.

Register to register operations should not use instruction mnemonic suffixes. 'fstl %st, %st(1)' will give a warning, and be assembled as if you wrote 'fst %st, %st(1)', since all register to register operations use 80-bit floating point operands. (Contrast this with 'fstl %st, mem', which converts '%st' from 80-bit to 64-bit floating point format, then stores the result in the 4 byte location 'mem')

# 9.16.10 Intel's MMX and AMD's 3DNow! SIMD Operations

as supports Intel's MMX instruction set (SIMD instructions for integer data), available on Intel's Pentium MMX processors and Pentium II processors, AMD's K6 and K6-2 processors, Cyrix' M2 processor, and probably others. It also supports AMD's 3DNow! instruction set (SIMD instructions for 32-bit floating point data) available on AMD's K6-2 processor and possibly others in the future.

Currently, as does not support Intel's floating point SIMD, Katmai (KNI).

The eight 64-bit MMX operands, also used by 3DNow!, are called '%mm0', '%mm1', ... '%mm7'. They contain eight 8-bit integers, four 16-bit integers, two 32-bit integers, one 64-bit integer, or two 32-bit floating point values. The MMX registers cannot be used at the same time as the floating point stack.

See Intel and AMD documentation, keeping in mind that the operand order in instructions is reversed from the Intel syntax.

# 9.16.11 AMD's Lightweight Profiling Instructions

as supports AMD's Lightweight Profiling (LWP) instruction set, available on AMD's Family 15h (Orochi) processors.

LWP enables applications to collect and manage performance data, and react to performance events. The collection of performance data requires no context switches. LWP runs in the context of a thread and so several counters can be used independently across multiple threads. LWP can be used in both 64-bit and legacy 32-bit modes.

For detailed information on the LWP instruction set, see the AMD Lightweight Profiling Specification available at Lightweight Profiling Specification (http://developer.amd.com/cpu/LWP).

## 9.16.12 Bit Manipulation Instructions

as supports the Bit Manipulation (BMI) instruction set.

BMI instructions provide several instructions implementing individual bit manipulation operations such as isolation, masking, setting, or resetting.

## 9.16.13 AMD's Trailing Bit Manipulation Instructions

**as** supports AMD's Trailing Bit Manipulation (TBM) instruction set, available on AMD's BDVER2 processors (Trinity and Viperfish).

TBM instructions provide instructions implementing individual bit manipulation operations such as isolating, masking, setting, resetting, complementing, and operations on trailing zeros and ones.

## 9.16.14 Writing 16-bit Code

While as normally writes only "pure" 32-bit i386 code or 64-bit x86-64 code depending on the default configuration, it also supports writing code to run in real mode or in 16-bit protected mode code segments. To do this, put a '.code16' or '.code16gcc' directive before the assembly language instructions to be run in 16-bit mode. You can switch as to writing 32-bit code with the '.code32' directive or 64-bit code with the '.code64' directive.

'.code16gcc' provides experimental support for generating 16-bit code from gcc, and differs from '.code16' in that 'call', 'ret', 'enter', 'leave', 'push', 'pop', 'pusha', 'popa', 'pushf', and 'popf' instructions default to 32-bit size. This is so that the stack pointer is manipulated in the same way over function calls, allowing access to function parameters at the same stack offsets as in 32-bit mode. '.code16gcc' also automatically adds address size prefixes where necessary to use the 32-bit addressing modes that gcc generates.

The code which **as** generates in 16-bit mode will not necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you must refrain from using *any* 32-bit constructs which require **as** to output address or operand size prefixes.

Note that writing 16-bit code instructions by explicitly specifying a prefix or an instruction mnemonic suffix within a 32-bit code section generates different machine instructions than those generated for a 16-bit code segment. In a 32-bit code section, the following code generates the machine opcode bytes '66 6a 04', which pushes the value '4' onto the stack, decrementing '%esp' by 2.

## pushw \$4

The same code in a 16-bit code section would generate the machine opcode bytes '6a 04' (i.e., without the operand size prefix), which is correct since the processor default operand size is assumed to be 16 bits in a 16-bit code section.

# 9.16.15 Specifying CPU Architecture

as may be told to assemble for a particular CPU (sub-)architecture with the .arch cpu\_type directive. This directive enables a warning when gas detects an instruction that is not supported on the CPU specified. The choices for cpu\_type are:

'default'	'push'	'pop'	
ʻi8086'	ʻi186'	ʻi286'	ʻi386'
ʻi486'	ʻi586'	ʻi686'	'pentium'
'pentiumpro'	'pentiumii'	'pentiumiii'	'pentium4'
'prescott'	'nocona'	'core'	'core2'
'corei7'	'iamcu'		

'amdfam10'       'bdver1'       'bdver2'       'bdver3'         'bdver4'       'znver1'       'znver2'       'znver3'         'znver4'       'btver1'       'btver2'       'generic32'         'generic64'       '.cmov'       'fxsr'       '.mmx'         'ssse'       '.sse2'       'sse3'       'sse4'         'ssse'       'sse4'       'sse4'       'sse4'         'avx'       '.vmx'       'ssmx'       'ept'         'clflush'       'movbe'       'xsave'       'xsaveopt'         'aes'       'pclmul'       'fma'       'ffsgsbase'         'rdrnd'       'ffc'       'ssave'       'xsaveopt'         'aes'       'pclmul'       'fma'       'sfgsbase'         'rdrnd'       'ffc'       'avx2'       'bmi2'         'lzcnt'       'popcnt'       'invpcid'       'vmfunc'         'smap'       'adx'       'rdreed'       'prfchw'         'smap'       'avx512cd'       'avx512er'       'sav512pf'         'avx512/'       'avx512_vbmi'       'avx512_vp2intersect'         'avx512_popontdd'       'avx512_vp2intersect'       'idt'         'tdx'       'avx_ifma'       'avx_vnni_int8'         'savx512_popo	'k6'	'k6_2'	'athlon'	'k8'		
'znver4'       'btver1'       'btver2'       'generic32'         'generic64'       '.cmov'       '.fxsr'       '.mmx'         '.sse'       '.sse2'       '.sse3'       '.sse4a'         '.sses3'       '.sse4.1'       '.sse4.2'       '.sse4'         '.avx'       '.vmx'       '.smx'       '.ept'         '.clflush'       '.movbe'       '.xsave'       '.xsave'         '.aes'       '.pclmul'       '.fma'       '.fggsbase'         '.rdrnd'       '.fl6c'       '.avx2'       '.bmi2'         '.lzcnt'       '.popcnt'       '.invpcid'       '.vmfunc'         '.laf_sahf'       '.adx'       '.rdseed'       '.prfchw'         '.smap'       '.mpx'       '.sha'       '.prefetcht1'         '.avx512f'       '.avx512cd'       '.avx512etr'       '.avx512pf'         '.avx512vbmi'       '.avx512_tfmaps'<.avx512_truniw'	'amdfam10'	'bdver1'	'bdver2'	'bdver3'		
'generic64'       .cmov'       .fxsr'       .mmx'         '.sse'       .sse2'       .sse3'       .sse4a'         '.sse'       .sse4.1'       .sse4.2'       .sse4'         '.avx'       .vmx'       .smx'       .ept'         '.clflush'       .movbe'       .xsave'       .xsaveopt'         '.aes'       .pclmul'       .fma'       .fsgsbase'         '.rdrnd'       .fl6C'       .avx2'       .bmi2'         '.lzcnt'       .popcnt'       .invpcid'       .vmfunc'         '.monitor'       .hle'       .rtm'       .tsx'         '.lahf_sahf'       .adx'       .rdseed'       .prfchw'         '.smap'       .mpx'       .sha'       .prefetcht1'         '.avx512f'       .avx512cd'       .avx512dq'       .avx512pf'         '.avx512vbmi'       .avx512_vbmi2'       .avx512_unni'       .avx512_vbmi4'         '.avx512_vbogontdd'.avx512_vbmi2'       .avx512_unni'       .avx512_vbi4'       .avx_untia'         '.avx512_vbia1g'       .avx512_vbmi2'       .avx512_upi4'       .avx512_upi4'         '.avx512_vbi4'       .avx_untia'       .avx512_upi4'       .avx512_upi4'         '.dwo'       .rdpid'       .ptwrite'       .ibt'	'bdver4'	'znver1'	'znver2'	'znver3'		
'generic64'       '.cmov'       '.fxsr'       '.mmx'         '.sse'       '.sse2'       '.sse3'       '.sse4a'         '.sse3'       '.sse4.1'       '.sse4.2'       '.sse4'         '.avx'       '.vmx'       '.smx'       '.sse4'         '.avx'       '.vmx'       '.ssm'       '.ept'         '.lflush'       '.movbe'       '.xsave'       '.xsaveopt'         '.aes'       '.pclmul'       '.fma'       '.fsgsbase'         '.rdrnd'       '.fl6c'       '.avx2'       '.bmi2'         '.lzcnt'       '.popcnt'       '.invpcid'       '.vmfunc'         '.monitor'       '.hle'       '.rtm'       '.tsx'         '.lahf_sahf'       '.adx'       '.rdseed'       '.prefetchwt1'         '.avx512f'       '.avx512cd'       '.avx512er'       '.avx512pf'         '.avx512vbmi'       '.avx512_dmaps'       '.avx512_druni*       '.avx512_fma'         '.avx512_vpopcntdd'.avx512_vbmi2'       '.avx512_upuit       '.avx512_tmi*       '.avx512_tmi*         '.avx512_vpopcntdd'.avx512_vbmi2'       '.avx512_upuit       '.avx512_tmi*       '.avx512_tmi*         '.avx512_vbmi3'       '.avx_ifma'       '.avx_upuit*       '.ibt'         '.prefetchi'       '.avx_ifma'       '	'znver4'	'btver1'	'btver2'	'generic32'		
<pre>'.ssse3' '.sse4.1' '.sse4.2' '.sse4' '.avx' '.vmx' '.smx' '.ept' '.clflush' '.movbe' '.xsave' '.xsaveopt' '.aes' '.pclmul' '.fma' '.fsgsbase' '.rdrnd' '.f16c' '.avx2' '.bmi2' '.lzcnt' '.popcnt' '.invpcid' '.vmfunc' '.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512rl' '.avx512_d' '.avx512er' '.avx512pf' '.avx512vl' '.avx512_vbmi2' '.avx512_dvnniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_vvnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2int=rsect' '.tdx' '.avx_ifma' '.avx_11eff6' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_11st' '.avx_ne_convert' '.rao_int' '.amx_int8' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcnd' '.tsxldtrk' '.amx_int8' '.amx_tile' '.amx_complex' '.amx_tile' '.atx' '.uintr' '.hreset' '.dtd' '.widekl' '.uintr' '.hreset' '.stycall' '.avx_fiff6' '.avx_f166' '.amx_int8' '.amx_tile' '.amx_int8' '.adma' '.seve' '.syscall' '.rdtscp' '.seve' '.upr' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'generic64'	'.cmov'	'.fxsr'			
'.avx'       '.vmx'       '.smx'       '.ept'         '.clflush'       '.movbe'       '.xsave'       '.xsaveopt'         '.aes'       '.pclmul'       '.fma'       '.fsgsbase'         '.rdrnd'       '.fl6c'       '.avx2'       '.bmi2'         '.lzcnt'       '.popcnt'       '.invpcid'       '.vmfunc'         '.monitor'       '.hle'       '.rtm'       '.tsx'         '.lahf_sahf'       '.adx'       '.rdseed'       '.prfchw'         '.smap'       '.mpx'       '.sha'       '.prefetchvt1'         '.avx512tf'       '.avx512cd'       '.avx512er'       '.avx512pf'         '.avx512vbmi'       '.avx512_vbmi2'       '.avx512_vtnni'       '.avx512_vpont'         '.avx512_vponctdd'       '.avx512_vbmi2'       '.avx512_vponit'       .avx512_vponit'         '.avx512_vponctdd'       '.avx512_vponit'       '.avx512_vponit'       .avx512_vponit'         '.avx512_vponctdd'       '.avx_ifma'       '.avx512_vponit'       '.avx512_vponit'         '.avx512_vponctdd'       '.avx_ifma'       '.avx_up_tinte'       '.ibt'         '.avx_ifma'       '.avx_up_tinte'       '.ibt'         '.clwb'       '.rdpid'       '.pvrite'       '.ibt'         '.fred'       '.lkgs' <t< td=""><td>'.sse'</td><td>'.sse2'</td><td>'.sse3'</td><td>'.sse4a'</td></t<>	'.sse'	'.sse2'	'.sse3'	'.sse4a'		
<pre>'.clflush' '.movbe' '.xsave' '.xsaveopt' '.aes' '.pclmul' '.fma' '.fsgsbase' '.rdrnd' '.f16c' '.avx2' '.bmi2' '.lzcnt' '.popcnt' '.invpcid' '.vmfunc' '.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512rl' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512_dfmaps''.avx512_dvnniw' '.avx512vlmi' '.avx512_dfmaps''.avx512_tfma' '.avx512vlmi' '.avx512_tbmi2' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_tp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.whoninvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.amx_tfle' '.amx_int8' '.amx_tfle' '.amx_int8' '.amx_tfle' '.kl' '.widekl' '.uintr' '.hreset' '.shsch' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.ssse3'	'.sse4.1'	'.sse4.2'	'.sse4'		
<pre>'.clflush' '.movbe' '.xsave' '.xsaveopt' '.aes' '.pclmul' '.fma' '.fsgsbase' '.rdrnd' '.f16c' '.avx2' '.bmi2' '.lzcnt' '.popcnt' '.invpcid' '.vmfunc' '.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512rl' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512_dfmaps''.avx512_dvnniw' '.avx512vlmi' '.avx512_dfmaps''.avx512_tfma' '.avx512vlmi' '.avx512_tbmi2' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_tp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.whoninvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.amx_tfle' '.amx_int8' '.amx_tfle' '.amx_int8' '.amx_tfle' '.kl' '.widekl' '.uintr' '.hreset' '.shsch' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.avx'	'.vmx'	'.smx'	'.ept'		
<pre>'.aes' '.pclmul' '.fma' '.fsgsbase' '.rdrnd' '.f16c' '.avx2' '.bmi2' '.lzcnt' '.popcnt' '.invpcid' '.vmfunc' '.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.sel' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512_dfmaps' .avx512_druniw' '.avx512vbmi' '.avx512_dfmaps' .avx512_druniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_truniw' '.avx512_vpopcntdd'.avx512_bf16' '.avx512_vpni' '.avx512_bitalg' '.avx512_bf16' '.avx512_pf16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.whoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.amx_int8' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.danx_int8' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.adanw' '.gtna' '.sves' '.sef16' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.clflush'	'.movbe'	'.xsave'			
<pre>'.rdrnd' '.f16c' '.avx2' '.bmi2' '.lzcnt' '.popcnt' '.invpcid' '.vmfunc' '.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.sel' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512bw' '.avx512dq' '.avx512ifma' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vnni' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_unni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_unni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.awx_ift8' '.amx_tf16' '.amx_int8' '.amx_tf16' '.amx_fp16' '.amx_complex' '.adtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.aes'	'.pclmul'	'.fma'			
<pre>'.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.sel' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512bw' '.avx512dq' '.avx512ifma' '.avx512vbmi' '.avx512_4fmaps'.avx512_4vnniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_pf16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.adm_tile' '.kl' '.widekl' '.uintr' '.hreset' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.rdrnd'	'.f16c'	'.avx2'			
<pre>'.monitor' '.hle' '.rtm' '.tsx' '.lahf_sahf' '.adx' '.rdseed' '.prfchw' '.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512bw' '.avx512dq' '.avx512ifma' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.adm_tile' '.kl' '.widekl' '.uintr' '.hreset' '.shsta' '.adm_tile' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.lzcnt'	'.popcnt'	<pre>`.invpcid'</pre>	'.vmfunc'		
<pre>'.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.show' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.stwae' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.monitor'			'.tsx'		
<pre>'.smap' '.mpx' '.sha' '.prefetchwt1' '.clflushopt' '.xsavec' '.xsaves' '.se1' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vhmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_vpni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2int=rsect' '.tdx' '.avx_unni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_unni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.shstk' '.gfni' '.vaes' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.slodnow' '.gtnscp' '.svme' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.lahf_sahf'	'.adx'	'.rdseed'	'.prfchw'		
<pre>'.clflushopt' '.xsavec' '.xsaves' '.sel' '.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_vvnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.ftma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.smap'	'.mpx'	'.sha'			
<pre>'.avx512f' '.avx512cd' '.avx512er' '.avx512pf' '.avx512vl' '.avx512bw' '.avx512dq' '.avx512ifma' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vp2intersect' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>			'.xsaves'	'.se1'		
<pre>'.avx512vl' '.avx512bw' '.avx512dq' '.avx512ifma' '.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vplintersect' '.avx512_bitalg' '.avx512_bf16' '.avx512_vplintersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>		'.avx512cd'	'.avx512er'	'.avx512pf'		
<pre>'.avx512vbmi' '.avx512_4fmaps''.avx512_4vnniw' '.avx512_vpopcntdd'.avx512_vbmi2' '.avx512_vnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.avx512vl'	'.avx512bw'	'.avx512dq'			
<pre>'.avx512_vpopcntdq'.avx512_vbmi2' '.avx512_vnni' '.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>						
<pre>'.avx512_bitalg' '.avx512_bf16' '.avx512_vp2intersect' '.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>						
<pre>'.tdx' '.avx_vnni' '.avx512_fp16' '.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>						
<pre>'.clwb' '.rdpid' '.ptwrite' '.ibt' '.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.mwaitx' '.rdpru'</pre>	'.tdx'	`.avx_vnni'	'.avx512_fp16'			
<pre>'.prefetchi' '.avx_ifma' '.avx_vnni_int8' '.cmpccxadd' '.wrmsrns' '.msrlist' '.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.clwb'	'.rdpid'		'.ibt'		
<pre>'.avx_ne_convert' '.rao_int' '.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.prefetchi'		<pre>`.avx_vnni_int8</pre>	3'		
<pre>'.fred' '.lkgs' '.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	`.cmpccxadd'	'.wrmsrns'	'.msrlist'			
<pre>'.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.avx_ne_convert'	'.rao_int'				
<pre>'.wbnoinvd' '.pconfig' '.waitpkg' '.cldemote' '.shstk' '.gfni' '.vaes' '.vpclmulqdq' '.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.fred'	'.lkgs'				
<pre>'.movdiri' '.movdir64b' '.enqcmd' '.tsxldtrk' '.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.wbnoinvd'	`.pconfig'	'.waitpkg'	`.cldemote'		
<pre>'.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.shstk'	'.gfni'	'.vaes'	`.vpclmulqdq'		
<pre>'.amx_int8' '.amx_bf16' '.amx_fp16' '.amx_complex' '.amx_tile' '.kl' '.widekl' '.uintr' '.hreset' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.movdiri'	'.movdir64b'	'.enqcmd'	'.tsxldtrk'		
<pre>'.kl' '.widekl' '.uintr' '.hreset' '.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	'.amx_int8'	`.amx_bf16'	'.amx_fp16'			
<pre>'.3dnow' '.3dnowa' '.sse4a' '.sse5' '.syscall' '.rdtscp' '.svme' '.lwp' '.fma4' '.xop' '.cx16' '.padlock' '.clzero' '.mwaitx' '.rdpru'</pre>	`.amx_complex'	`.amx_tile'				
'.syscall'       '.rdtscp'       '.svme'         '.lwp'       '.fma4'       '.xop'       '.cx16'         '.padlock'       '.clzero'       '.mwaitx'       '.rdpru'	'.kl'	'.widekl'	'.uintr'			
'.lwp''.fma4''.xop''.cx16''.padlock''.clzero''.mwaitx''.rdpru'	'.3dnow'	'.3dnowa'	'.sse4a'	'.sse5'		
'.padlock' '.clzero' '.mwaitx' '.rdpru'	'.syscall'	`.rdtscp'	'.svme'			
<pre>'.padlock' '.clzero' '.mwaitx' '.rdpru' '.mcommit' '.sev es' '.snp' '.invlpgb'</pre>	'.lwp'	'.fma4'	'.xop'			
'.mcommit' '.sev es' '.snp' '.invlpgb'	'.padlock'	`.clzero'	`.mwaitx'	'.rdpru'		
	'.mcommit'					
'.tlbsync'		'.sev_es'	'.snp'	'.invlpgb'		

Apart from the warning, there are only two other effects on as operation; Firstly, if you specify a CPU other than 'i486', then shift by one instructions such as 'sarl \$1, %eax' will automatically use a two byte opcode sequence. The larger three byte opcode sequence is used on the 486 (and when no architecture is specified) because it executes faster on the 486. Note that you can explicitly request the two byte opcode by writing 'sarl %eax'. Secondly, if you specify 'i8086', 'i186', or 'i286', and '.code16' or '.code16gcc' then byte offset

conditional jumps will be promoted when necessary to a two instruction sequence consisting of a conditional jump of the opposite sense around an unconditional jump to the target.

Note that the sub-architecture specifiers (starting with a dot) can be prefixed with **no** to revoke the respective (and any dependent) functionality.

Following the CPU architecture (but not a sub-architecture, which are those starting with a dot), you may specify 'jumps' or 'nojumps' to control automatic promotion of conditional jumps. 'jumps' is the default, and enables jump promotion; All external jumps will be of the long variety, and file-local jumps will be promoted as necessary. (see Section 9.16.8 [i386-Jumps], page 194) 'nojumps' leaves external conditional jumps as byte offset jumps, and warns about file-local conditional jumps that as promotes. Unconditional jumps are treated as for 'jumps'.

For example .arch i8086,nojumps

# 9.16.16 AMD64 ISA vs. Intel64 ISA

There are some discrepancies between AMD64 and Intel64 ISAs.

- For 'movsxd' with 16-bit destination register, AMD64 supports 32-bit source operand and Intel64 supports 16-bit source operand.
- For far branches (with explicit memory operand), both ISAs support 32- and 16-bit operand size. Intel64 additionally supports 64-bit operand size, encoded as 'ljmpq' and 'lcallq' in AT&T syntax and with an explicit 'tbyte ptr' operand size specifier in Intel syntax.
- 'lfs', 'lgs', and 'lss' similarly allow for 16- and 32-bit operand size (32- and 48-bit memory operand) in both ISAs, while Intel64 additionally supports 64-bit operand sise (80-bit memory operands).

# 9.16.17 AT&T Syntax bugs

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we're stuck with it.

For example

fsub %st,%st(3)

results in '%st(3)' being updated to '%st - %st(3)' rather than the expected '%st(3) - %st'. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is '%st' and the destination register is '%st(i)'.

## 9.16.18 Notes

There is some trickery concerning the 'mul' and 'imul' instructions that deserves mention. The 16-, 32-, 64- and 128-bit expanding multiplies (base opcode 'Oxf6'; extension 4 for 'mul' and 5 for 'imul') can be output only in the one operand form. Thus, 'imul %ebx, %eax' does *not* select the expanding multiply; the expanding multiply would clobber the '%edx' register, and this would confuse gcc output. Use 'imul %ebx' to get the 64-bit product in '%edx:%eax'.

We have added a two operand form of 'imul' when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying '%eax' by 69, for example, can be done with 'imul \$69, %eax' rather than 'imul \$69, %eax'.

# 9.17 IA-64 Dependent Features

## 9.17.1 Options

#### -mconstant-gp

This option instructs the assembler to mark the resulting object file as using the "constant GP" model. With this model, it is assumed that the entire program uses a single global pointer (GP) value. Note that this option does not in any fashion affect the machine code emitted by the assembler. All it does is turn on the EF\_IA\_64\_CONS\_GP flag in the ELF file header.

#### -mauto-pic

This option instructs the assembler to mark the resulting object file as using the "constant GP without function descriptor" data model. This model is like the "constant GP" model, except that it additionally does away with function descriptors. What this means is that the address of a function refers directly to the function's code entry-point. Normally, such an address would refer to a function descriptor, which contains both the code entry-point and the GPvalue needed by the function. Note that this option does not in any fashion affect the machine code emitted by the assembler. All it does is turn on the EF\_IA\_64\_NOFUNCDESC\_CONS\_GP flag in the ELF file header.

```
-milp32
```

-milp64

-mlp64

-mp64 These options select the data model. The assembler defaults to -mlp64 (LP64 data model).

-mle

-mbe These options select the byte order. The -mle option selects little-endian byte order (default) and -mbe selects big-endian byte order. Note that IA-64 machine code always uses little-endian byte order.

## -mtune=itanium1

-mtune=itanium2

Tune for a particular IA-64 CPU, *itanium1* or *itanium2*. The default is *itanium2*.

#### -munwind-check=warning

#### -munwind-check=error

These options control what the assembler will do when performing consistency checks on unwind directives. -munwind-check=warning will make the assembler issue a warning when an unwind directive check fails. This is the default. -munwind-check=error will make the assembler issue an error when an unwind directive check fails.

#### -mhint.b=ok

#### -mhint.b=warning

#### -mhint.b=error

These options control what the assembler will do when the 'hint.b' instruction is used. -mhint.b=ok will make the assembler accept 'hint.b'. -mint.b=warning

will make the assembler issue a warning when 'hint.b' is used. -mhint.b=error will make the assembler treat 'hint.b' as an error, which is the default.

-x

-xexplici	t
	These options turn on dependency violation checking.
-xauto	This option instructs the assembler to automatically insert stop bits where necessary to remove dependency violations. This is the default mode.
-xnone	This option turns off dependency violation checking.
-xdebug	This turns on debug output intended to help tracking down bugs in the depen- dency violation checker.
-xdebugn	This is a shortcut for -xnone -xdebug.
-xdebugx	This is a shortcut for -xexplicit -xdebug.

## 9.17.2 Syntax

The assembler syntax closely follows the IA-64 Assembly Language Reference Guide.

## 9.17.2.1 Special Characters

'//' is the line comment token.

';' can be used instead of a newline to separate statements.

## 9.17.2.2 Register Names

The 128 integer registers are referred to as 'rn'. The 128 floating-point registers are referred to as 'fn'. The 128 application registers are referred to as 'arn'. The 128 control registers are referred to as 'crn'. The 64 one-bit predicate registers are referred to as 'pn'. The 8 branch registers are referred to as 'bn'. In addition, the assembler defines a number of aliases: 'gp' ('r1'), 'sp' ('r12'), 'rp' ('b0'), 'ret0' ('r8'), 'ret1' ('r9'), 'ret2' ('r10'), 'ret3' ('r9'), 'fargn' ('f8+n'), and 'fretn' ('f8+n').

For convenience, the assembler also defines aliases for all named application and control registers. For example, 'ar.bsp' refers to the register backing store pointer ('ar17'). Similarly, 'cr.eoi' refers to the end-of-interrupt register ('cr67').

## 9.17.2.3 IA-64 Processor-Status-Register (PSR) Bit Names

The assembler defines bit masks for each of the bits in the IA-64 processor status register. For example, 'psr.ic' corresponds to a value of 0x2000. These masks are primarily intended for use with the 'ssm'/'sum' and 'rsm'/'rum' instructions, but they can be used anywhere else where an integer constant is expected.

## 9.17.2.4 Relocations

In addition to the standard IA-64 relocations, the following relocations are implemented by as:

#### @slotcount(V)

Convert the address offset V into a slot count. This pseudo function is available only on VMS. The expression V must be known at assembly time: it can't reference undefined symbols or symbols in different sections.

# 9.17.3 Opcodes

For detailed information on the IA-64 machine instruction set, see the IA-64 Assembly Language Reference Guide available at

http://developer.intel.com/design/itanium/arch\_spec.htm

# 9.18 IP2K Dependent Features

## 9.18.1 IP2K Options

The Ubicom IP2K version of as has a few machine dependent options:

-mip2022ext

as can assemble the extended IP2022 instructions, but it will only do so if this is specifically allowed via this command line option.

-mip2022 This option restores the assembler's default behaviour of not permitting the extended IP2022 instructions to be assembled.

## 9.18.2 IP2K Syntax

## 9.18.2.1 Special Characters

The presence of a ';' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The IP2K assembler does not currently support a line separator character.

# 9.19 LM32 Dependent Features

## 9.19.1 Options

```
-mmultiply-enabled
Enable multiply instructions.
```

-mdivide-enabled Enable divide instructions.

-mbarrel-shift-enabled Enable barrel-shift instructions.

-msign-extend-enabled Enable sign extend instructions.

-muser-enabled

Enable user defined instructions.

-micache-enabled

Enable instruction cache related CSRs.

-mdcache-enabled Enable data cache related CSRs.

-mbreak-enabled

- Enable break instructions.
- -mall-enabled Enable all instructions and CSRs.

## 9.19.2 Syntax

## 9.19.2.1 Register Names

LM32 has 32 x 32-bit general purpose registers 'r0', 'r1', ... 'r31'.

The following aliases are defined: 'gp' - 'r26', 'fp' - 'r27', 'sp' - 'r28', 'ra' - 'r29', 'ea' - 'r30', 'ba' - 'r31'.

LM32 has the following Control and Status Registers (CSRs).

IE	Interrupt enable.
IM	Interrupt mask.
IP	Interrupt pending.
ICC	Instruction cache control.
DCC	Data cache control.
CC	Cycle counter.
CFG	Configuration.
EBA	Exception base address.
DC	Debug control.

DEBA	Debug exception base address.
JTX	JTAG transmit.
JRX	JTAG receive.
BPO	Breakpoint 0.
BP1	Breakpoint 1.
BP2	Breakpoint 2.
BP3	Breakpoint 3.
WPO	Watchpoint 0.
WP1	Watchpoint 1.
WP2	Watchpoint 2.
WP3	Watchpoint 3.

# 9.19.2.2 Relocatable Expression Modifiers

The assembler supports several modifiers when using relocatable addresses in LM32 instruction operands. The general syntax is the following:

modifier(relocatable-expression)

lo	
	This modifier allows you to use bits 0 through 15 of an address expression as 16 bit relocatable expression.
hi	
	This modifier allows you to use bits 16 through 23 of an address expression as 16 bit relocatable expression.
	For example
	ori r4, r4, lo(sym+10) orhi r4, r4, hi(sym+10)
gp	
	This modified creates a 16-bit relocatable expression that is the offset of the symbol from the global pointer. mva r4, gp(sym)
got	
	This modifier places a symbol in the GOT and creates a 16-bit relocatable expression that is the offset into the GOT of this symbol. lw r4, (gp+got(sym))
gotofflo16	3
	This modifier allows you to use the bits 0 through 15 of an address which is an offset from the GOT.
gotoffhi10	3
	This modifier allows you to use the bits 16 through 31 of an address which is an offset from the GOT. orhi r4, r4, gotoffhi16(lsym) addi r4, r4, gotofflo16(lsym)

# 9.19.2.3 Special Characters

The presence of a '#' on a line indicates the start of a comment that extends to the end of the current line. Note that if a line starts with a '#' character then it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

A semicolon (';') can be used to separate multiple statements on the same line.

# 9.19.3 Opcodes

For detailed information on the LM32 machine instruction set, see http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/.

as implements all the standard LM32 opcodes.

## 9.20 M32C Dependent Features

as can assemble code for several different members of the Renesas M32C family. Normally the default is to assemble code for the M16C microprocessor. The -m32c option may be used to change the default to the M32C microprocessor.

## 9.20.1 M32C Options

The Renesas M32C version of as has these machine-dependent options:

-m32c Assemble M32C instructions.

-m16c Assemble M16C instructions (default).

**-relax** Enable support for link-time relaxations.

-h-tick-hex

Support H'00 style hex constants in addition to 0x00 style.

## 9.20.2 M32C Syntax

## 9.20.2.1 Symbolic Operand Modifiers

The assembler supports several modifiers when using symbol addresses in M32C instruction operands. The general syntax is the following:

%modifier(symbol)

## %dsp8

%dsp16

These modifiers override the assembler's assumptions about how big a symbol's address is. Normally, when it sees an operand like 'sym[a0]' it assumes 'sym' may require the widest displacement field (16 bits for '-m16c', 24 bits for '-m32c'). These modifiers tell it to assume the address will fit in an 8 or 16 bit (respectively) unsigned displacement. Note that, of course, if it doesn't actually fit you will get linker errors. Example:

mov.w %dsp8(sym)[a0],r1 mov.b #0,%dsp8(sym)[a0]

## %hi8

This modifier allows you to load bits 16 through 23 of a 24 bit address into an 8 bit register. This is useful with, for example, the M16C 'smovf' instruction, which expects a 20 bit address in 'r1h' and 'a0'. Example:

mov.b #%hi8(sym),r1h
mov.w #%lo16(sym),a0
smovf.b

#### %lo16

Likewise, this modifier allows you to load bits 0 through 15 of a 24 bit address into a 16 bit register.

#### %hi16

This modifier allows you to load bits 16 through 31 of a 32 bit address into a 16 bit register. While the M32C family only has 24 bits of address space,

it does support addresses in pairs of 16 bit registers (like 'a1a0' for the 'lde' instruction). This modifier is for loading the upper half in such cases. Example:

```
mov.w #%hi16(sym),a1
mov.w #%lo16(sym),a0
...
lde.w [a1a0],r1
```

## 9.20.2.2 Special Characters

The presence of a ';' character on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The '|' character can be used to separate statements on the same line.

# 9.21 M32R Dependent Features

## 9.21.1 M32R Options

The Renesas M32R version of as has a few machine dependent options:

- -m32rx as can assemble code for several different members of the Renesas M32R family. Normally the default is to assemble code for the M32R microprocessor. This option may be used to change the default to the M32RX microprocessor, which adds some more instructions to the basic M32R instruction set, and some additional parameters to some of the original instructions.
- -m32r2 This option changes the target processor to the M32R2 microprocessor.
- -m32r This option can be used to restore the assembler's default behaviour of assembling for the M32R microprocessor. This can be useful if the default has been changed by a previous command-line option.
- -little This option tells the assembler to produce little-endian code and data. The default is dependent upon how the toolchain was configured.
- -EL This is a synonym for *-little*.
- -big This option tells the assembler to produce big-endian code and data.
- -EB This is a synonym for *-big*.
- -KPIC This option specifies that the output of the assembler should be marked as position-independent code (PIC).
- -parallel

This option tells the assembler to attempts to combine two sequential instructions into a single, parallel instruction, where it is legal to do so.

-no-parallel

This option disables a previously enabled *-parallel* option.

-no-bitinst

This option disables the support for the extended bit-field instructions provided by the M32R2. If this support needs to be re-enabled the *-bitinst* switch can be used to restore it.

-0 This option tells the assembler to attempt to optimize the instructions that it produces. This includes filling delay slots and converting sequential instructions into parallel ones. This option implies *-parallel*.

#### -warn-explicit-parallel-conflicts

Instructs as to produce warning messages when questionable parallel instructions are encountered. This option is enabled by default, but gcc disables it when it invokes as directly. Questionable instructions are those whose behaviour would be different if they were executed sequentially. For example the code fragment 'mv r1, r2 || mv r3, r1' produces a different result from 'mv r1, r2 \n mv r3, r1' since the former moves r1 into r3 and then r2 into r1, whereas the later moves r2 into r1 and r3.

-Wp This is a shorter synonym for the *-warn-explicit-parallel-conflicts* option.

#### -no-warn-explicit-parallel-conflicts

Instructs **as** not to produce warning messages when questionable parallel instructions are encountered.

-Wnp This is a shorter synonym for the *-no-warn-explicit-parallel-conflicts* option.

#### -ignore-parallel-conflicts

This option tells the assembler's to stop checking parallel instructions for constraint violations. This ability is provided for hardware vendors testing chip designs and should not be used under normal circumstances.

#### -no-ignore-parallel-conflicts

This option restores the assembler's default behaviour of checking parallel instructions to detect constraint violations.

-Ip This is a shorter synonym for the *-ignore-parallel-conflicts* option.

-nIp This is a shorter synonym for the *-no-ignore-parallel-conflicts* option.

#### -warn-unmatched-high

This option tells the assembler to produce a warning message if a .high pseudo op is encountered without a matching .low pseudo op. The presence of such an unmatched pseudo op usually indicates a programming error.

#### -no-warn-unmatched-high

Disables a previously enabled *-warn-unmatched-high* option.

-Wuh This is a shorter synonym for the *-warn-unmatched-high* option.

-Wnuh This is a shorter synonym for the *-no-warn-unmatched-high* option.

# 9.21.2 M32R Directives

The Renesas M32R version of **as** has a few architecture specific directives:

#### low expression

The low directive computes the value of its expression and places the lower 16-bits of the result into the immediate-field of the instruction. For example:

```
or3 r0, r0, \#low(0x12345678); compute r0 = r0 | 0x5678
add3, r0, r0, \#low(fred); compute r0 = r0 + low 16-bits of address of fred
```

#### high expression

The high directive computes the value of its expression and places the upper 16-bits of the result into the immediate-field of the instruction. For example:

```
seth r0, #high(0x12345678) ; compute r0 = 0x12340000
```

seth, r0, #high(fred) ; compute r0 = upper 16-bits of address of fred

#### shigh expression

The shigh directive is very similar to the high directive. It also computes the value of its expression and places the upper 16-bits of the result into the immediate-field of the instruction. The difference is that shigh also checks to see if the lower 16-bits could be interpreted as a signed number, and if so it assumes that a borrow will occur from the upper-16 bits. To compensate for this the shigh directive pre-biases the upper 16 bit value by adding one to it. For example: For example:

```
seth r0, #shigh(0x12345678) ; compute r0 = 0x12340000
seth r0, #shigh(0x00008000) ; compute r0 = 0x00010000
```

In the second example the lower 16-bits are  $0 \times 8000$ . If these are treated as a signed value and sign extended to 32-bits then the value becomes  $0 \times 10000$ . If this value is then added to  $0 \times 00010000$  then the result is  $0 \times 00008000$ .

This behaviour is to allow for the different semantics of the or3 and add3 instructions. The or3 instruction treats its 16-bit immediate argument as unsigned whereas the add3 treats its 16-bit immediate as a signed value. So for example:

```
seth r0, #shigh(0x00008000)
add3 r0, r0, #low(0x00008000)
```

Produces the correct result in r0, whereas:

seth r0, #shigh(0x00008000)
or3 r0, r0, #low(0x00008000)

Stores 0xffff8000 into r0.

Note - the **shigh** directive does not know where in the assembly source code the lower 16-bits of the value are going set, so it cannot check to make sure that an **or3** instruction is being used rather than an **add3** instruction. It is up to the programmer to make sure that correct directives are used.

- .m32r The directive performs a similar thing as the -m32r command line option. It tells the assembler to only accept M32R instructions from now on. An instructions from later M32R architectures are refused.
- .m32rx The directive performs a similar thing as the -m32rx command line option. It tells the assembler to start accepting the extra instructions in the M32RX ISA as well as the ordinary M32R ISA.
- .m32r2 The directive performs a similar thing as the -m32r2 command line option. It tells the assembler to start accepting the extra instructions in the M32R2 ISA as well as the ordinary M32R ISA.
- .little The directive performs a similar thing as the *-little* command line option. It tells the assembler to start producing little-endian code and data. This option should be used with care as producing mixed-endian binary files is fraught with danger.
- .big The directive performs a similar thing as the *-big* command line option. It tells the assembler to start producing big-endian code and data. This option should be used with care as producing mixed-endian binary files is fraught with danger.

## 9.21.3 M32R Warnings

There are several warning and error messages that can be produced by **as** which are specific to the M32R:

# output of 1st instruction is the same as an input to 2nd instruction - is this intentional ?

This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the left hand instruction is used as an input register in the right hand instruction. For example in this code fragment 'mv r1, r2 || neg r3, r1' register r1 is the destination of the move instruction and the input to the neg instruction.

# output of 2nd instruction is the same as an input to 1st instruction - is this intentional ?

This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the right hand instruction is used as an input register in the left hand instruction. For example in this code fragment 'mv r1,  $r2 \mid \mid neg r2$ , r3' register r2 is the destination of the neg instruction and the input to the move instruction.

#### instruction '...' is for the M32RX only

This message is produced when the assembler encounters an instruction which is only supported by the M32Rx processor, and the '-m32rx' command-line flag has not been specified to allow assembly of such instructions.

#### unknown instruction '...'

This message is produced when the assembler encounters an instruction which it does not recognize.

#### only the NOP instruction can be issued in parallel on the m32r

This message is produced when the assembler encounters a parallel instruction which does not involve a NOP instruction and the '-m32rx' command-line flag has not been specified. Only the M32Rx processor is able to execute two instructions in parallel.

#### instruction '...' cannot be executed in parallel.

This message is produced when the assembler encounters a parallel instruction which is made up of one or two instructions which cannot be executed in parallel.

#### Instructions share the same execution pipeline

This message is produced when the assembler encounters a parallel instruction whose components both use the same execution pipeline.

#### Instructions write to the same destination register.

This message is produced when the assembler encounters a parallel instruction where both components attempt to modify the same register. For example these code fragments will produce this message: 'mv r1, r2 || neg r1, r3' 'jl r0 || mv r14, r1' 'st r2, @-r1 || mv r1, r3' 'mv r1, r2 || ld r0, @r1+' 'cmp r1, r2 || addx r3, r4' (Both write to the condition bit)

# 9.22 M680x0 Dependent Features

## 9.22.1 M680x0 Options

The Motorola 680x0 version of as has a few machine dependent options:

#### '-march=architecture'

This option specifies a target architecture. The following architectures are recognized: 68000, 68010, 68020, 68030, 68040, 68060, cpu32, isaa, isaaplus, isab, isac and cfv4e.

'-mcpu=cpu'

This option specifies a target cpu. When used in conjunction with the -march option, the cpu must be within the specified architecture. Also, the generic features of the architecture are used for instruction generation, rather than those of the specific chip.

- '-m[no-]68851'
- '-m[no-]68881'
- '-m[no-]div'
- '-m[no-]usp'
- '-m[no-]float'
- '-m[no-]mac'
- '-m[no-]emac'

Enable or disable various architecture specific features. If a chip or architecture by default supports an option (for instance -march=isaaplus includes the -mdiv option), explicitly disabling the option will override the default.

'-1' You can use the '-1' option to shorten the size of references to undefined symbols. If you do not use the '-1' option, references to undefined symbols are wide enough for a full long (32 bits). (Since as cannot know where these symbols end up, as can only allocate space for the linker to fill in later. Since as does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

#### '--register-prefix-optional'

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a '%' before any use of a register name. This is intended to let the assembler distinguish between C variables and functions named 'a0' through 'a7', and so on. The '%' is always accepted, but is not required for certain configurations, notably 'sun3'. The '--register-prefix-optional' option may be used to permit omitting the '%' even for configurations for which it is normally required. If this is done, it will generally be impossible to refer to C variables and functions with the same names as register names.

'--bitwise-or'

Normally the character '|' is treated as a comment character, which means that it can not be used in expressions. The '--bitwise-or' option turns '|' into a

normal character. In this mode, you must either use C style comments, or start comments with a '#' character at the beginning of a line.

#### '--base-size-default-16 --base-size-default-32'

If you use an addressing mode with a base register without specifying the size, as will normally use the full 32 bit value. For example, the addressing mode 'a00(%d0)' is equivalent to 'a00(%d0:1)'. You may use the '--base-size-default-16' option to tell as to default to using the 16 bit value. In this case, 'a00(%d0)' is equivalent to 'a00(%d0:w)'. You may use the '--base-size-default-32' option to restore the default behaviour.

```
'--disp-size-default-16 --disp-size-default-32'
```

If you use an addressing mode with a displacement, and the value of the displacement is not known, as will normally assume that the value is 32 bits. For example, if the symbol 'disp' has not been defined, as will assemble the addressing mode '%a0@(disp,%d0)' as though 'disp' is a 32 bit value. You may use the '--disp-size-default-16' option to tell as to instead assume that the displacement is 16 bits. In this case, as will assemble '%a0@(disp,%d0)' as though 'disp' is a 16 bit value. You may use the '--disp-size-default-32' option to restore the default behaviour.

- '--pcrel' Always keep branches PC-relative. In the M680x0 architecture all branches are defined as PC-relative. However, on some processors they are limited to word displacements maximum. When as needs a long branch that is not available, it normally emits an absolute jump instead. This option disables this substitution. When this option is given and no long branches are available, only word branches will be emitted. An error message will be generated if a word branch cannot reach its target. This option has no effect on 68020 and other processors that have long branches. see Section 9.22.6.1 [Branch Improvement], page 219.
- '-m68000' as can assemble code for several different members of the Motorola 680x0 family. The default depends upon how as was configured when it was built; normally, the default is to assemble code for the 68020 microprocessor. The following options may be used to change the default. These options control which instructions and addressing modes are permitted. The members of the 680x0 family are very similar. For detailed information about the differences, see the Motorola manuals.
  - '-m68000'
    '-m68ec000'
    '-m68hc000'
    '-m68hc001'
    '-m68008'
    '-m68302'
    '-m68306'
    '-m68307'
    '-m68322'
  - '-m68356' Assemble for the 68000. '-m68008', '-m68302', and so on are synonyms for '-m68000', since the chips are the same from the point of view of the assembler.

Assemble for the 68010. '-m68010' '-m68020' '-m68ec020' Assemble for the 68020. This is normally the default. '-m68030' '-m68ec030' Assemble for the 68030. '-m68040' '-m68ec040' Assemble for the 68040. '-m68060' '-m68ec060' Assemble for the 68060. '-mcpu32' '-m68330' '-m68331' '-m68332' '-m68333' '-m68334' '-m68336' '-m68340' '-m68341' '-m68349' '-m68360' Assemble for the CPU32 family of chips. '-m5200' '-m5202' '-m5204' '-m5206' '-m5206e' '-m521x' '-m5249' '-m528x' '-m5307' '-m5407' '-m547x' '-m548x' '-mcfv4' Assemble for the ColdFire family of chips. '-mcfv4e' '-m68881' '-m68882' Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always

support floating point instructions.

```
'-mno-68881'
```

Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

'-m68851' Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; '-m68851' and '-m68040' should not be used together.

```
'-mno-68851'
```

Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

## 9.22.2 Syntax

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of as uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, 'movl' is equivalent to 'mov.l'.

In the following table *apc* stands for any of the address registers ('%a0' through '%a7'), the program counter ('%pc'), the zero-address relative to the program counter ('%zpc'), a suppressed address register ('%za0' through '%za7'), or it may be omitted entirely. The use of *size* means one of 'w' or 'l', and it may be omitted, along with the leading colon, unless a scale is also specified. The use of *scale* means one of '1', '2', '4', or '8', and it may always be omitted along with the leading colon.

The following addressing modes are understood:

Immediate

```
'#number'
```

Data Register

'%d0' through '%d7'

Address Register

'%a0' through '%a7'

'%a7' is also known as '%sp', i.e., the Stack Pointer. %a6 is also known as '%fp', the Frame Pointer.

```
Address Register Indirect
'%a0@' through '%a7@'
```

```
Address Register Postincrement
'%a0@+' through '%a7@+'
```

- Address Register Predecrement '%a0@-' through '%a7@-'
- Indirect Plus Offset

'apc@(number)'

```
Index 'apc@(number,register:size:scale)'
The number may be omitted.
```

Postindex	'apc@(number)@(onumber,register:size:scale)'
	The <i>onumber</i> or the <i>register</i> , but not both, may be omitted.
Preindex	'apc@(number,register:size:scale)@(onumber)'
	The <i>number</i> may be omitted. Omitting the <i>register</i> produces the Postindex addressing mode.

Absolute 'symbol', or 'digits', optionally followed by ':b', ':w', or ':l'.

#### 9.22.3 Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (see Section 9.22.2 [Syntax], page 216). as can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible.

In the following table *apc* stands for any of the address registers ('%a0' through '%a7'), the program counter ('%pc'), the zero-address relative to the program counter ('%zpc'), or a suppressed address register ('%za0' through '%za7'). The use of *size* means one of 'w' or '1', and it may always be omitted along with the leading dot. The use of *scale* means one of '1', '2', '4', or '8', and it may always be omitted along with the leading asterisk.

The following additional addressing modes are understood:

```
Address Register Indirect
```

'(%a0)' through '(%a7)' '%a7' is also known as '%sp', i.e., the Stack Pointer. %a6 is also known as '%fp', the Frame Pointer.

```
Address Register Postincrement
'(%a0)+' through '(%a7)+'
```

```
Address Register Predecrement
'-(%a0)' through '-(%a7)'
```

#### Indirect Plus Offset

'number(%a0)' through 'number(%a7)', or 'number(%pc)'.

The number may also appear within the parentheses, as in '(number, %a0)'. When used with the pc, the number may be omitted (with an address register, omitting the number produces Address Register Indirect mode).

```
Index 'number(apc,register.size*scale)'
```

The number may be omitted, or it may appear within the parentheses. The *apc* may be omitted. The *register* and the *apc* may appear in either order. If both *apc* and *register* are address registers, and the *size* and *scale* are omitted, then the first register is taken as the base register, and the second as the index register.

Postindex '([number,apc],register.size\*scale,onumber)'

The onumber, or the register, or both, may be omitted. Either the number or the apc may be omitted, but not both.

Preindex '([number, apc, register.size\*scale], onumber)' The number, or the apc, or the register, or any two of them, may be omitted. The onumber may be omitted. The register and the apc may appear in either order. If both apc and register are address registers, and the size and scale are omitted, then the first register is taken as the base register, and the second as the index register.

## 9.22.4 Floating Point

Packed decimal (P) format floating literals are not supported. Feel free to add the code!

The floating point formats generated by directives are these.

.float Single precision floating point constants.

.double Double precision floating point constants.

.extend

.ldouble Extended precision (long double) floating point constants.

## 9.22.5 680x0 Machine Directives

In order to be compatible with the Sun assembler the  $680 \times 0$  assembler understands the following directives.

- .data1 This directive is identical to a .data 1 directive.
- .data2 This directive is identical to a .data 2 directive.
- .even This directive is a special case of the .align directive; it aligns the output to an even byte boundary.
- .skip This directive is identical to a .space directive.
- .arch name

Select the target architecture and extension features. Valid values for name are the same as for the -march command-line option. This directive cannot be specified after any instructions have been assembled. If it is given multiple times, or in conjunction with the -march option, all uses must be for the same architecture and extension set.

.cpu name Select the target cpu. Valid values for name are the same as for the -mcpu command-line option. This directive cannot be specified after any instructions have been assembled. If it is given multiple times, or in conjunction with the -mopt option, all uses must be for the same cpu.

## 9.22.6 Opcodes

#### 9.22.6.1 Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by substituting 'j' for 'b' at the start of a Motorola mnemonic.

The following table summarizes the pseudo-operations. A \* flags cases that are more fully described after the table:

	Displace	ement			
Pseudo-Op	  BYTE +	WORD	68020 LONG	68000/10, not ABSOLUTE LONG	PC-relative OK JUMP **
jra * jXX * dbXX	bsrs  bras  bXXs   N/A   N/A	bsrw braw bXXw dbXXw fbXXw	bsrl bral bXXl dbXX;bras;bral fbXXl	jsr jmp bNXs;jmp dbXX;bras;jmp N/A	
XX: condi NX: negat		ondition	XX		

\*—see full description below
\*\*—this expansion mode is disallowed by '--pcrel'

#### jbsr

jra These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target. This instruction will be a byte or word branch is that is sufficient. Otherwise, a long branch will be emitted if available. If no long branches are available and the '--pcrel' option is not given, an absolute long jump will be emitted instead. If no long branches are available, the '--pcrel' option is given, and a word branch cannot reach the target, an error message is generated.

In addition to standard branch operands, **as** allows these pseudo-operations to have all operands that are allowed for jsr and jmp, substituting these instructions if the operand given is not valid for a branch instruction.

jXX Here, 'jXX' stands for an entire family of pseudo-operations, where XX is a conditional branch or condition-code test. The full list of pseudo-ops in this family is:

jhi jls jcc jcs jne jeq jvc jvs jpl jmi jge jlt jgt jle

Usually, each of these pseudo-operations expands to a single branch instruction. However, if a word branch is not sufficient, no long branches are available, and the '--pcrel' option is not given, as issues a longer code fragment in terms of NX, the opposite condition to XX. For example, under these conditions:

jXX foo

gives

b*NX*s oof jmp foo oof:

dbXX

The fu	ıll fami	ly of ps	seudo-c	peratic	ons cove	ered he	re is
	dbhi	dbls	dbcc	dbcs	dbne	dbeq	dbvc
		-		dbge	dblt	dbgt	dble
	dbf	dbra	dbt				

Motorola 'dbXX' instructions allow word displacements only. When a word displacement is sufficient, each of these pseudo-operations expands to the corresponding Motorola instruction. When a word displacement is not sufficient and long branches are available, when the source reads 'dbXX foo', as emits

```
dbXX oo1
bras oo2
oo1:bral foo
oo2:
```

If, however, long branches are not available and the '--pcrel' option is not given, as emits

dbXX oo1 bras oo2 oo1:jmp foo oo2:

fjXX This family includes

fjne fjeq fjge fjlt fjgt fjle fjf fjt fjgl fjgle fjnge fjngl fjngle fjngt fjnle fjnlt fjoge fjogl fjogt fjole fjolt fjor fjseq fjsf fjsne fjst fjueq fjuge fjugt fjule fjult fjun

Each of these pseudo-operations always expands to a single Motorola coprocessor branch instruction, word or long. All Motorola coprocessor branch instructions allow both word and long displacements.

## 9.22.6.2 Special Characters

Line comments are introduced by the '|' character appearing anywhere on a line, unless the --bitwise-or command-line option has been specified.

An asterisk ('\*') as the first character on a line marks the start of a line comment as well.

A hash character ('#') as the first character on a line also marks the start of a line comment, but in this case it could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31). If the hash character appears elsewhere on a line it is used to introduce an immediate value. (This is for compatibility with Sun's assembler).

Multiple statements on the same line can appear if they are separated by the ';' character.

# 9.23 M68HC11 and M68HC12 Dependent Features

## 9.23.1 M68HC11 and M68HC12 Options

The Motorola 68HC11 and 68HC12 version of as have a few machine dependent options.

- -m68hc11 This option switches the assembler into the M68HC11 mode. In this mode, the assembler only accepts 68HC11 operands and mnemonics. It produces code for the 68HC11.
- -m68hc12 This option switches the assembler into the M68HC12 mode. In this mode, the assembler also accepts 68HC12 operands and mnemonics. It produces code for the 68HC12. A few 68HC11 instructions are replaced by some 68HC12 instructions as recommended by Motorola specifications.
- -m68hcs12

This option switches the assembler into the M68HCS12 mode. This mode is similar to '-m68hc12' but specifies to assemble for the 68HCS12 series. The only difference is on the assembling of the 'movb' and 'movw' instruction when a PC-relative operand is used.

- -mm9s12x This option switches the assembler into the M9S12X mode. This mode is similar to '-m68hc12' but specifies to assemble for the S12X series which is a superset of the HCS12.
- -mm9s12xg

This option switches the assembler into the XGATE mode for the RISC coprocessor featured on some S12X-family chips.

--xgate-ramoffset

This option instructs the linker to offset RAM addresses from S12X address space into XGATE address space.

- -mshort This option controls the ABI and indicates to use a 16-bit integer ABI. It has no effect on the assembled instructions. This is the default.
- -mlong This option controls the ABI and indicates to use a 32-bit integer ABI.
- -mshort-double

This option controls the ABI and indicates to use a 32-bit float ABI. This is the default.

-mlong-double

This option controls the ABI and indicates to use a 64-bit float ABI.

--strict-direct-mode

You can use the '--strict-direct-mode' option to disable the automatic translation of direct page mode addressing into extended mode when the instruction does not support direct mode. For example, the 'clr' instruction does not support direct page mode addressing. When it is used with the direct page mode, as will ignore it and generate an absolute addressing. This option prevents as from doing this, and the wrong usage of the direct page mode will raise an error.

#### --short-branches

The '--short-branches' option turns off the translation of relative branches into absolute branches when the branch offset is out of range. By default as transforms the relative branch ('bsr', 'bgt', 'bge', 'beq', 'ble', 'ble', 'blt', 'bhi', 'bcc', 'bls', 'bcs', 'bmi', 'bvs', 'bvs', 'bra') into an absolute branch when the offset is out of the -128 .. 127 range. In that case, the 'bsr' instruction is translated into a 'jsr', the 'bra' instruction is translated into a 'jmp' and the conditional branches instructions are inverted and followed by a 'jmp'. This option disables these translations and as will generate an error if a relative branch is out of range. This option does not affect the optimization associated to the 'jbra', 'jbsr' and 'jbXX' pseudo opcodes.

#### --force-long-branches

The '--force-long-branches' option forces the translation of relative branches into absolute branches. This option does not affect the optimization associated to the 'jbra', 'jbsr' and 'jbXX' pseudo opcodes.

#### --print-insn-syntax

You can use the '--print-insn-syntax' option to obtain the syntax description of the instruction when an error is detected.

--print-opcodes

The '--print-opcodes' option prints the list of all the instructions with their syntax. The first item of each line represents the instruction name and the rest of the line indicates the possible operands for that instruction. The list is printed in alphabetical order. Once the list is printed **as** exits.

#### --generate-example

The '--generate-example' option is similar to '--print-opcodes' but it generates an example for each instruction instead.

## 9.23.2 Syntax

In the M68HC11 syntax, the instruction name comes first and it may be followed by one or several operands (up to three). Operands are separated by comma (','). In the normal mode, **as** will complain if too many operands are specified for a given instruction. In the MRI mode (turned on with '-M' option), it will treat them as comments. Example:

```
inx
lda #23
bset 2,x #4
brclr *bot #8 foo
```

The presence of a ';' character or a '!' character anywhere on a line indicates the start of a comment that extends to the end of that line.

A '\*' or a '#' character at the start of a line also introduces a line comment, but these characters do not work elsewhere on the line. If the first character of the line is a '#' then as well as starting a comment, the line could also be logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The M68HC11 assembler does not currently support a line separator character.

The following addressing modes are understood for 68HC11 and 68HC12:

#### Immediate

'#number'

Address Register

'number,X', 'number,Y'

The number may be omitted in which case 0 is assumed.

Direct Addressing mode '\*symbol', or '\*digits'

Absolute 'symbol', or 'digits'

The M68HC12 has other more complex addressing modes. All of them are supported and they are represented below:

#### Constant Offset Indexed Addressing Mode

#### 'number, reg'

The number may be omitted in which case 0 is assumed. The register can be either 'X', 'Y', 'SP' or 'PC'. The assembler will use the smaller post-byte definition according to the constant value (5-bit constant offset, 9-bit constant offset or 16-bit constant offset). If the constant is not known by the assembler it will use the 16-bit constant offset post-byte and the value will be resolved at link time.

#### Offset Indexed Indirect

#### '[number,reg]'

The register can be either 'X', 'Y', 'SP' or 'PC'.

Auto Pre-Increment/Pre-Decrement/Post-Increment/Post-Decrement

'number,-reg' 'number,+reg' 'number,reg-' 'number,reg+'

The number must be in the range '-8'..'+8' and must not be 0. The register can be either 'X', 'Y', 'SP' or 'PC'.

#### Accumulator Offset

## 'acc,reg'

The accumulator register can be either 'A', 'B' or 'D'. The register can be either 'X', 'Y', 'SP' or 'PC'.

#### Accumulator D offset indexed-indirect

'[D,reg]'

The register can be either 'X', 'Y', 'SP' or 'PC'.

#### For example:

ldab 1024,sp ldd [10,x] orab 3,+x stab -2,yldx a,pc sty [d,sp]

# 9.23.3 Symbolic Operand Modifiers

The assembler supports several modifiers when using symbol addresses in 68HC11 and 68HC12 instruction operands. The general syntax is the following:

%modifier(symbol)

- %addr This modifier indicates to the assembler and linker to use the 16-bit physical address corresponding to the symbol. This is intended to be used on memory window systems to map a symbol in the memory bank window. If the symbol is in a memory expansion part, the physical address corresponds to the symbol address within the memory bank window. If the symbol is not in a memory expansion part, this is the symbol address (using or not using the %addr modifier has no effect in that case).
- %page This modifier indicates to use the memory page number corresponding to the symbol. If the symbol is in a memory expansion part, its page number is computed by the linker as a number used to map the page containing the symbol in the memory bank window. If the symbol is not in a memory expansion part, the page number is 0.
- %hi This modifier indicates to use the 8-bit high part of the physical address of the symbol.
- %10 This modifier indicates to use the 8-bit low part of the physical address of the symbol.

For example a 68HC12 call to a function 'foo\_example' stored in memory expansion part could be written as follows:

call %addr(foo\_example),%page(foo\_example)

and this is equivalent to

```
call foo_example
```

And for 68HC11 it could be written as follows:

```
ldab #%page(foo_example)
stab _page_switch
jsr %addr(foo_example)
```

## 9.23.4 Assembler Directives

The 68HC11 and 68HC12 version of as have the following specific assembler directives:

**.relax** The relax directive is used by the 'GNU Compiler' to emit a specific relocation to mark a group of instructions for linker relaxation. The sequence of instructions within the group must be known to the linker so that relaxation can be performed.

```
.mode [mshort|mlong|mshort-double|mlong-double]
```

This directive specifies the ABI. It overrides the '-mshort', '-mlong', '-mshort-double' and '-mlong-double' options.

```
.far symbol
```

This directive marks the symbol as a 'far' symbol meaning that it uses a 'call/rtc' calling convention as opposed to 'jsr/rts'. During a final link, the linker will identify references to the 'far' symbol and will verify the proper calling convention.

.interrupt symbol

This directive marks the symbol as an interrupt entry point. This information is then used by the debugger to correctly unwind the frame across interrupts.

.xrefb symbol

This directive is defined for compatibility with the 'Specification for Motorola 8 and 16-Bit Assembly Language Input Standard' and is ignored.

#### 9.23.5 Floating Point

Packed decimal (P) format floating literals are not supported. Feel free to add the code! The floating point formats generated by directives are these.

.float Single precision floating point constants.

.double Double precision floating point constants.

.extend

.ldouble Extended precision (long double) floating point constants.

## 9.23.6 Opcodes

## 9.23.6.1 Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by prepending 'j' to the start of Motorola mnemonic. These pseudo opcodes are not affected by the '--short-branches' or '--force-long-branches' options.

The following table summarizes the pseudo-operations.

	+		Displacer	nent Width	
	   -	short-br	Option anches		ong-branches
Op	+  BYTE			+   BYTE	WORD
bra jbsr jbra bXX	bsr   bra   bsr   bsr   bra	<pc-rel> <pc-rel> <pc-rel> <pc-rel> <pc-rel> <pc-rel></pc-rel></pc-rel></pc-rel></pc-rel></pc-rel></pc-rel>	jmp <abs> <error></error></abs>	bsr <pc-rel>   bra <pc-rel>     bXX <pc-rel></pc-rel></pc-rel></pc-rel>	jmp <abs> bNX +3; jmp <abs></abs></abs>
XX: c	+	 ion		+	

NX: negative of condition XX

jbsr

jbra These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

jbXX Here, 'jbXX' stands for an entire family of pseudo-operations, where XX is a conditional branch or condition-code test. The full list of pseudo-ops in this family is:

jbcc jbeq jbge jbgt jbhi jbvs jbpl jblo

## jbcs jbne jblt jble jbls jbvc jbmi

For the cases of non-PC relative displacements and long displacements, **as** issues a longer code fragment in terms of NX, the opposite condition to XX. For example, for the non-PC relative case:

jbXX foo

gives

```
bNXs oof
jmp foo
oof:
```

## 9.24 S12Z Dependent Features

The Freescale S12Z version of as has a few machine dependent features.

## 9.24.1 S12Z Options

The S12Z version of as recognizes the following options:

#### '-mreg-prefix=prefix'

You can use the '-mreg-prefix=pfx' option to indicate that the assembler should expect all register names to be prefixed with the string pfx.

For an explanation of what this means and why it might be needed, see Section 9.24.2.3 [S12Z Register Notation], page 229.

#### '-mdollar-hex'

The '-mdollar-hex' option affects the way that literal hexadecimal constants are represented. When this option is specified, the assembler will consider the '\$' character as the start of a hexadecimal integer constant. Without this option, the standard value of '0x' is expected.

If you use this option, then you cannot have symbol names starting with '\$'. '-mdollar-hex' is implied if the '--traditional-format' (see Section 2.15 [traditional-format], page 29) is used.

## 9.24.2 Syntax

#### 9.24.2.1 Overview

In the S12Z syntax, the instruction name comes first and it may be followed by one, or by several operands. In most cases the maximum number of operands is three. Operands are separated by a comma (','). A comma however does not act as a separator if it appears within parentheses ('()') or within square brackets ('[]'). as will complain if too many, too few or inappropriate operands are specified for a given instruction.

Some instructions accept and (in certain situations require) a suffix indicating the size of the operand. The suffix is separated from the instruction name by a period ('.') and may be one of 'b', 'w', 'p' or 'l' indicating 'byte' (a single byte), 'word' (2 bytes), 'pointer' (3 bytes) or 'long' (4 bytes) respectively.

Example:

```
bset.b 0xA98, #5
mov.b
        #6, 0x2409
ld
        d0, #4
mov.l
        (d0, x), 0x2409
        d0
inc
cmp
        d0, #12
blt
        *-4
        x, 0x2409
lea
st
        y, (1, x)
```

The presence of a ';' character anywhere on a line indicates the start of a comment that extends to the end of that line.

A '\*' or a '#' character at the start of a line also introduces a line comment, but these characters do not work elsewhere on the line. If the first character of the line is a '#' then as well as starting a comment, the line could also be logical line number directive (see Section 3.3

[Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The S12Z assembler does not currently support a line separator character.

## 9.24.2.2 Addressing Modes

The following addressing modes are understood for the S12Z.

Immediate

'#number'

Immediate Bit Field

'#width:offset'

Bit field instructions in the immediate mode require the width and offset to be specified. The width parameter specifies the number of bits in the field. It should be a number in the range [1,32]. Offset determines the position within the field where the operation should start. It should be a number in the range [0,31].

Relative '\*symbol', or '\*[+-]digits'

Program counter relative addresses have a width of 15 bits. Thus, they must be within the range [-32768, 32767].

Register 'reg'

Some instructions accept a register as an operand. In general, reg may be a data register ('D0', 'D1'  $\dots$  'D7'), the 'X' register or the 'Y' register.

A few instructions accept as an argument the stack pointer register ('S'), and/or the program counter ('P').

Some very special instructions accept arguments which refer to the condition code register. For these arguments the syntax is 'CCR', 'CCH' or 'CCL' which refer to the complete condition code register, the condition code register high byte and the condition code register low byte respectively.

#### Absolute Direct

'symbol', or 'digits'

Absolute Indirect

'[symbol', or 'digits]'

Constant Offset Indexed

'(number,reg)'

Reg may be either 'X', 'Y', 'S' or 'P' or one of the data registers 'D0', 'D1' ... 'D7'. If any of the registers 'D2' ... 'D5' are specified, then the register value is treated as a signed value. Otherwise it is treated as unsigned. Number may be any integer in the range [-8388608,8388607].

#### Offset Indexed Indirect

'[number,reg]'

Reg may be either 'X', 'Y', 'S' or 'P'. Number may be any integer in the range [-8388608,8388607].

#### Auto Pre-Increment/Pre-Decrement/Post-Increment/Post-Decrement '-reg', '+reg', 'reg-' or 'reg+'

This addressing mode is typically used to access a value at an address, and simultaneously to increment/decrement the register pointing to that address. Thus reg may be any of the 24 bit registers 'X', 'Y', or 'S'. Pre-increment and post-decrement are not available for register 'S' (only post-increment and pre-decrement are available).

#### Register Offset Direct

'(data-reg,reg)'

Reg can be either 'X', 'Y', or 'S'. Data-reg must be one of the data registers 'D0', 'D1' ... 'D7'. If any of the registers 'D2' ... 'D5' are specified, then the register value is treated as a signed value. Otherwise it is treated as unsigned.

#### Register Offset Indirect

'[data-reg,reg]'

Reg can be either 'X' or 'Y'. Data-reg must be one of the data registers 'D0', 'D1' ... 'D7'. If any of the registers 'D2' ... 'D5' are specified, then the register value is treated as a signed value. Otherwise it is treated as unsigned.

#### For example:

trap	#197	;; Immediate mode
bra	*+49	;; Relative mode
bra	.LO	;; ditto
jmp	0xFE0034	;; Absolute direct mode
jmp	[0xFD0012]	;; Absolute indirect mode
inc.b	(4,x)	;; Constant offset indexed mode
jsr	(45, d0)	;; ditto
dec.w	[4,y]	;; Constant offset indexed indirect mode
clr.p	(-s)	;; Pre-decrement mode
neg.l	(d0, s)	;; Register offset direct mode
com.b	[d1, x]	;; Register offset indirect mode
psh		

#### 9.24.2.3 Register Notation

Without a register prefix (see Section 9.24.1 [S12Z Options], page 227), S12Z assembler code is expected in the traditional format like this:

```
lea s, (-2,s)
st d2, (0,s)
ld x, symbol
tfr d2, d6
cmp d6, #1532
```

However, if **as** is started with (for example) '-mreg-prefix=%' then all register names must be prefixed with '%' as follows:

lea %s, (-2,%s)
st %d2, (0,%s)
ld %x, symbol
tfr %d2, %d6
cmp %d6, #1532

The register prefix feature is intended to be used by compilers to avoid ambiguity between symbols and register names. Consider the following assembler instruction:

st d0, d1

The destination operand of this instruction could either refer to the register 'D1', or it could refer to the symbol named "d1". If the latter is intended then as must be invoked with '-mreg-prefix=pfx' and the code written as

#### st pfxd0, d1

where pfx is the chosen register prefix. For this reason, compiler back-ends should choose a register prefix which cannot be confused with a symbol name.

# 9.25 Meta Dependent Features

## 9.25.1 Options

The Imagination Technologies Meta architecture is implemented in a number of versions, with each new version adding new features such as instructions and registers. For precise details of what instructions each core supports, please see the chip's technical reference manual.

The following table lists all available Meta options.

```
-mcpu=metac11
```

Generate code for Meta 1.1.

-mcpu=metac12

Generate code for Meta 1.2.

```
-mcpu=metac21
```

Generate code for Meta 2.1.

```
-mfpu=metac21
```

Allow code to use FPU hardware of Meta 2.1.

## 9.25.2 Syntax

## 9.25.2.1 Special Characters

'!' is the line comment character.

You can use ';' instead of a newline to separate statements.

Since '\$' has no special meaning, you may use it in symbol names.

## 9.25.2.2 Register Names

Registers can be specified either using their mnemonic names, such as 'DOReO', or using the unit plus register number separated by a '.', such as 'DO.O'.

# 9.26 MicroBlaze Dependent Features

The Xilinx MicroBlaze processor family includes several variants, all using the same core instruction set. This chapter covers features of the GNU assembler that are specific to the MicroBlaze architecture. For details about the MicroBlaze instruction set, please see the MicroBlaze Processor Reference Guide (UG081) available at www.xilinx.com.

## 9.26.1 Directives

A number of assembler directives are available for MicroBlaze.

```
.data8 expression, ...
```

This directive is an alias for .byte. Each expression is assembled into an eight-bit value.

.data16 expression, ...

This directive is an alias for .hword. Each expression is assembled into an 16-bit value.

#### .data32 expression,...

This directive is an alias for .word. Each expression is assembled into an 32-bit value.

.ent name[,label]

This directive is an alias for .func denoting the start of function name at (optional) label.

.end name[,label]

This directive is an alias for .endfunc denoting the end of function name.

.gpword label,...

This directive is an alias for .rva. The resolved address of *label* is stored in the data section.

.weakext label

Declare that label is a weak external symbol.

- .rodata Switch to .rodata section. Equivalent to .section .rodata
- .sdata2 Switch to .sdata2 section. Equivalent to .section .sdata2
- .sdata Switch to .sdata section. Equivalent to .section .sdata
- .bss Switch to .bss section. Equivalent to .section .bss
- .sbss Switch to .sbss section. Equivalent to .section .sbss

## 9.26.2 Syntax for the MicroBlaze

## 9.26.2.1 Special Characters

The presence of a '#' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

## 9.27 MIPS Dependent Features

GNU **as** for MIPS architectures supports several different MIPS processors, and MIPS ISA levels I through V, MIPS32, and MIPS64. For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall). For an overview of MIPS assembly conventions, see "Appendix D: Assembly Language Programming" in the same work.

## 9.27.1 Assembler options

The MIPS configurations of GNU as support these special options:

-G num Set the "small data" limit to n bytes. The default limit is 8 bytes. See Section 9.27.4 [Controlling the use of small data accesses], page 242.

-EB

- -EL Any MIPS configuration of **as** can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use '-EB' to select big-endian output, and '-EL' for little-endian.
- -KPIC Generate SVR4-style PIC. This option tells the assembler to generate SVR4-style position-independent macro expansions. It also tells the assembler to mark the output file as PIC.

-mvxworks-pic

Generate VxWorks PIC. This option tells the assembler to generate VxWorksstyle position-independent macro expansions.

-mips1 -mips2 -mips3 -mips4 -mips5 -mips32r2 -mips32r3 -mips32r5 -mips32r6 -mips64 -mips64r2 -mips64r5 -mips64r6

Generate code for a particular MIPS Instruction Set Architecture level. '-mips1' corresponds to the R2000 and R3000 processors, '-mips2' to the R6000 processor, '-mips3' to the R4000 processor, and '-mips4' to the R8000 and R10000 processors. '-mips5', '-mips32', '-mips32r2', '-mips32r3', '-mips32r5', '-mips32r6', '-mips64', '-mips64r2', '-mips64r3', '-mips64r5', and '-mips64r6' correspond to generic MIPS V, MIPS32, MIPS32 Release 2, MIPS32 Release 3, MIPS32 Release 5, MIPS32 Release 6, MIPS64, and MIPS64 Release 2, MIPS64 Release 3, MIPS64 Release 5, and MIPS64 Release 6 ISA processors, respectively. You

can also switch instruction sets during the assembly; see Section 9.27.5 [MIPS ISA], page 243.

-mgp32

-mfp32 Some macros have different expansions for 32-bit and 64-bit registers. The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 32 bits wide at all times. '-mgp32' controls the size of general-purpose registers and '-mfp32' controls the size of floating-point registers.

The .set gp=32 and .set fp=32 directives allow the size of registers to be changed for parts of an object. The default value is restored by .set gp=default and .set fp=default.

On some MIPS variants there is a 32-bit mode flag; when this flag is set, 64-bit instructions generate a trap. Also, some 32-bit OSes only save the 32-bit registers on a context switch, so it is essential never to use the 64-bit registers.

-mgp64

-mfp64 Assume that 64-bit registers are available. This is provided in the interests of symmetry with '-mgp32' and '-mfp32'.

The .set gp=64 and .set fp=64 directives allow the size of registers to be changed for parts of an object. The default value is restored by .set gp=default and .set fp=default.

-mfpxx Make no assumptions about whether 32-bit or 64-bit floating-point registers are available. This is provided to support having modules compatible with either '-mfp32' or '-mfp64'. This option can only be used with MIPS II and above. The .set fp=xx directive allows a part of an object to be marked as not making assumptions about 32-bit or 64-bit FP registers. The default value is restored by .set fp=default.

#### -modd-spreg

#### -mno-odd-spreg

Enable use of floating-point operations on odd-numbered single-precision registers when supported by the ISA. '-mfpxx' implies '-mno-odd-spreg', otherwise the default is '-modd-spreg'

### -mips16

#### -no-mips16

Generate code for the MIPS 16 processor. This is equivalent to putting .module mips16 at the start of the assembly file. '-no-mips16' turns off this option.

#### -mmips16e2

#### -mno-mips16e2

Enable the use of MIPS16e2 instructions in MIPS16 mode. This is equivalent to putting .module mips16e2 at the start of the assembly file. '-mno-mips16e2' turns off this option.

#### -mmicromips

#### -mno-micromips

Generate code for the microMIPS processor. This is equivalent to putting .module micromips at the start of the assembly file. '-mno-micromips' turns

off this option. This is equivalent to putting .module nomicromips at the start of the assembly file.

-msmartmi	•
-mno-smar	tmips Enables the SmartMIPS extensions to the MIPS32 instruction set, which pro-
	vides a number of new instructions which target smartcard and cryptographic
	applications. This is equivalent to putting .module smartmips at the start of
	the assembly file. '-mno-smartmips' turns off this option.
-mips3d	
-no-mips3	
	Generate code for the MIPS-3D Application Specific Extension. This tells the assembler to accept MIPS-3D instructions. '-no-mips3d' turns off this option.
-mdmx	
-no-mdmx	Generate code for the MDMX Application Specific Extension. This tells the assembler to accept MDMX instructions. '-no-mdmx' turns off this option.
-mdsp	
-mno-dsp	Generate code for the DSP Release 1 Application Specific Extension. This tells
	the assembler to accept DSP Release 1 instructions. '-mno-dsp' turns off this option.
-mdspr2	
-mno-dspr	
	Generate code for the DSP Release 2 Application Specific Extension. This option implies '-mdsp'. This tells the assembler to accept DSP Release 2 instructions. '-mno-dspr2' turns off this option.
-mdspr3	
-mno-dspr	
	Generate code for the DSP Release 3 Application Specific Extension. This option implies '-mdsp' and '-mdspr2'. This tells the assembler to accept DSP Release 3 instructions. '-mno-dspr3' turns off this option.
-mmt	
-mno-mt	Generate code for the MT Application Specific Extension. This tells the assembler to accept MT instructions. '-mno-mt' turns off this option.
-mmcu	
-mno-mcu	Generate code for the MCU Application Specific Extension. This tells the assembler to accept MCU instructions. '-mno-mcu' turns off this option.
-mmsa	
-mno-msa	Generate code for the MIPS SIMD Architecture Extension. This tells the assembler to accept MSA instructions. '-mno-msa' turns off this option.
-mxpa	
-mno-xpa	Generate code for the MIPS eXtended Physical Address (XPA) Extension. This tells the assembler to accept XPA instructions. '-mno-xpa' turns off this option.

# -mvirt

# -mno-virt

Generate code for the Virtualization Application Specific Extension. This tells the assembler to accept Virtualization instructions. '-mno-virt' turns off this option.

### -mcrc

-mno-crc Generate code for the cyclic redundancy check (CRC) Application Specific Extension. This tells the assembler to accept CRC instructions. '-mno-crc' turns off this option.

# -mginv

# -mno-ginv

Generate code for the Global INValidate (GINV) Application Specific Extension. This tells the assembler to accept GINV instructions. '-mno-ginv' turns off this option.

### -mloongson-mmi

-mno-loongson-mmi

Generate code for the Loongson MultiMedia extensions Instructions (MMI) Application Specific Extension. This tells the assembler to accept MMI instructions. '-mno-loongson-mmi' turns off this option.

#### -mloongson-cam

# -mno-loongson-cam

Generate code for the Loongson Content Address Memory (CAM) Application Specific Extension. This tells the assembler to accept CAM instructions. '-mno-loongson-cam' turns off this option.

### -mloongson-ext

-mno-loongson-ext

Generate code for the Loongson EXTensions (EXT) instructions Application Specific Extension. This tells the assembler to accept EXT instructions. '-mno-loongson-ext' turns off this option.

### -mloongson-ext2

### -mno-loongson-ext2

Generate code for the Loongson EXTensions R2 (EXT2) instructions Application Specific Extension. This tells the assembler to accept EXT2 instructions. '-mno-loongson-ext2' turns off this option.

### -minsn32

### -mno-insn32

Only use 32-bit instruction encodings when generating code for the microMIPS processor. This option inhibits the use of any 16-bit instructions. This is equivalent to putting .set insn32 at the start of the assembly file. '-mno-insn32' turns off this option. This is equivalent to putting .set noinsn32 at the start of the assembly file. By default '-mno-insn32' is selected, allowing all instructions to be used.

# -mfix7000

# -mno-fix7000

Cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

### -mfix-rm7000

#### -mno-fix-rm7000

Cause nops to be inserted if a dmult or dmultu instruction is followed by a load instruction.

### -mfix-loongson2f-jump

# -mno-fix-loongson2f-jump

Eliminate instruction fetch from outside 256M region to work around the Loongson2F 'jump' instructions. Without it, under extreme cases, the kernel may crash. The issue has been solved in latest processor batches, but this fix has no side effect to them.

### -mfix-loongson2f-nop

### -mno-fix-loongson2f-nop

Replace nops by or at, at, zero to work around the Loongson2F 'nop' errata. Without it, under extreme cases, the CPU might deadlock. The issue has been solved in later Loongson2F batches, but this fix has no side effect to them.

### -mfix-loongson3-llsc

# -mno-fix-loongson3-llsc

Insert 'sync' before 'll' and 'lld' to work around Loongson3 LLSC errata. Without it, under extrame cases, the CPU might deadlock. The default can be controlled by the --enable-mips-fix-loongson3-llsc=[yes|no] configure option.

### -mfix-vr4120

### -mno-fix-vr4120

Insert nops to work around certain VR4120 errata. This option is intended to be used on GCC-generated code: it is not designed to catch all problems in hand-written assembler code.

### -mfix-vr4130

### -mno-fix-vr4130

Insert nops to work around the VR4130 'mflo'/'mfhi' errata.

### -mfix-24k

### -mno-fix-24k

Insert nops to work around the 24K 'eret'/'deret' errata.

### -mfix-cn63xxp1

### -mno-fix-cn63xxp1

Replace pref hints 0 - 4 and 6 - 24 with hint 28 to work around certain CN63XXP1 errata.

### -mfix-r5900

### -mno-fix-r5900

Do not attempt to schedule the preceding instruction into the delay slot of a branch instruction placed at the end of a short loop of six instructions or fewer and always schedule a **nop** instruction there instead. The short loop bug under certain conditions causes loops to execute only once or twice, due to a hardware bug in the R5900 chip.

-m4010

-no-m4010

Generate code for the LSI R4010 chip. This tells the assembler to accept the R4010-specific instructions ('addciu', 'ffc', etc.), and to not schedule 'nop' instructions around accesses to the 'HI' and 'LO' registers. '-no-m4010' turns off this option.

-m4650

-no-m4650

Generate code for the MIPS R4650 chip. This tells the assembler to accept the 'mad' and 'madu' instruction, and to not schedule 'nop' instructions around accesses to the 'HI' and 'LO' registers. '-no-m4650' turns off this option.

-m3900

-no-m3900

-m4100

-no-m4100

For each option '-mnnnn', generate code for the MIPS Rnnnn chip. This tells the assembler to accept instructions specific to that chip, and to schedule for that chip's hazards.

-march=cpu

Generate code for a particular MIPS CPU. It is exactly equivalent to '-mcpu', except that there are more value of cpu understood. Valid cpu value are:

2000, 3000, 3900, 4000, 4010, 4100, 4111, vr4120, vr4130, vr4181, 4300, 4400, 4600, 4650, 5000, rm5200, rm5230, rm5231, rm5261, rm5721, vr5400, vr5500, 6000, rm7000, 8000, rm9000, 10000, 12000, 14000, 16000, 4kc, 4km, 4kp, 4ksc, 4kec, 4kem, 4kep, 4ksd, m4k, m4kp, m14k, m14kc, m14ke, m14kec, 24kc, 24kcf2\_1, 24kf, 24kf1\_1, 24kec, 24kef2\_1, 24kef, 24kef1\_1, 34kc, 34kf2\_1, 34kf, 34kf1\_1, 34kn, 74kc, 74kf2\_1, 74kf, 74kf1\_1, 74kf3\_2, 1004kc, 1004kf2\_1, 1004kf, 1004kf1\_1, interaptiv, interaptiv-mr2, m5100, m5101, p5600, 5kc, 5kf, 20kc, 25kf, sb1, sb1a, i6400, i6500, p6600, loongson2e, loong-son2f, gs464, gs464e, gs264e, octeon, octeon+, octeon2, octeon3, xlr, xlp

For compatibility reasons, 'nx' and 'bfx' are accepted as synonyms for 'nf1\_1'. These values are deprecated.

-mtune=cpu

Schedule and tune for a particular MIPS CPU. Valid *cpu* values are identical to '-march=*cpu*'.

-mabi=abi

Record which ABI the source code uses. The recognized arguments are: '32', 'n32', 'o64', '64' and 'eabi'.

# -msym32

# -mno-sym32

Equivalent to adding .set sym32 or .set nosym32 to the beginning of the assembler input. See Section 9.27.3 [MIPS Symbol Sizes], page 242.

-nocpp This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GNU as, there is no need for '-nocpp', because the GNU assembler itself never runs the C preprocessor.

### -msoft-float

-mhard-float

Disable or enable floating-point instructions. Note that by default floating-point instructions are always allowed even with CPU targets that don't have support for these instructions.

### -msingle-float

### -mdouble-float

Disable or enable double-precision floating-point operations. Note that by default double-precision floating-point operations are always allowed even with CPU targets that don't have support for these operations.

# --construct-floats

#### --no-construct-floats

The --no-construct-floats option disables the construction of double width floating point constants by loading the two halves of the value into the two single width floating point registers that make up the double width register. This feature is useful if the processor support the FR bit in its status register, and this bit is known (by the programmer) to be set. This bit prevents the aliasing of the double width register by the single width registers.

By default --construct-floats is selected, allowing construction of these floating point constants.

#### --relax-branch

#### --no-relax-branch

The '--relax-branch' option enables the relaxation of out-of-range branches. Any branches whose target cannot be reached directly are converted to a small instruction sequence including an inverse-condition branch to the physically next instruction, and a jump to the original target is inserted between the two instructions. In PIC code the jump will involve further instructions for address calculation.

The BC1ANY2F, BC1ANY2T, BC1ANY4F, BC1ANY4T, BPOSGE32 and BPOSGE64 instructions are excluded from relaxation, because they have no complementing counterparts. They could be relaxed with the use of a longer sequence involving another branch, however this has not been implemented and if their target turns out of reach, they produce an error even if branch relaxation is enabled.

Also no MIPS16 branches are ever relaxed.

By default '--no-relax-branch' is selected, causing any out-of-range branches to produce an error.

```
-mignore-branch-isa
-mno-ignore-branch-isa
```

Ignore branch checks for invalid transitions between ISA modes.

The semantics of branches does not provide for an ISA mode switch, so in most cases the ISA mode a branch has been encoded for has to be the same as the ISA mode of the branch's target label. If the ISA modes do not match, then such a branch, if taken, will cause the ISA mode to remain unchanged and instructions that follow will be executed in the wrong ISA mode causing the program to misbehave or crash.

In the case of the BAL instruction it may be possible to relax it to an equivalent JALX instruction so that the ISA mode is switched at the run time as required. For other branches no relaxation is possible and therefore GAS has checks implemented that verify in branch assembly that the two ISA modes match, and report an error otherwise so that the problem with code can be diagnosed at the assembly time rather than at the run time.

However some assembly code, including generated code produced by some versions of GCC, may incorrectly include branches to data labels, which appear to require a mode switch but are either dead or immediately followed by valid instructions encoded for the same ISA the branch has been encoded for. While not strictly correct at the source level such code will execute as intended, so to help with these cases '-mignore-branch-isa' is supported which disables ISA mode checks for branches.

By default '-mno-ignore-branch-isa' is selected, causing any invalid branch requiring a transition between ISA modes to produce an error.

#### -mnan=encoding

This option indicates whether the source code uses the IEEE 2008 NaN encoding (-mnan=2008) or the original MIPS encoding (-mnan=legacy). It is equivalent to adding a .nan directive to the beginning of the source file. See Section 9.27.10 [MIPS NaN Encodings], page 246.

-mnan=legacy is the default if no -mnan option or .nan directive is used.

### --trap

### --no-break

**as** automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes **as** to generate code to take a trap exception rather than a break exception when an error is detected. The trap instructions are only supported at Instruction Set Architecture level 2 and higher.

#### --break

--no-trap

Generate code to take a break exception rather than a trap exception when an error is detected. This is the default.

### -mpdr

-mno-pdr Control generation of .pdr sections. Off by default on IRIX, on elsewhere.

-mshared -mno-shared

When generating code using the Unix calling conventions (selected by '-KPIC' or '-mcall\_shared'), gas will normally generate code which can go into a shared library. The '-mno-shared' option tells gas to generate code which uses the calling convention, but can not go into a shared library. The resulting code is slightly more efficient. This option only affects the handling of the '.cpload' and '.cpsetup' pseudo-ops.

# 9.27.2 High-level assembly macros

MIPS assemblers have traditionally provided a wider range of instructions than the MIPS architecture itself. These extra instructions are usually referred to as "macro" instructions<sup>1</sup>.

Some MIPS macro instructions extend an underlying architectural instruction while others are entirely new. An example of the former type is and, which allows the third operand to be either a register or an arbitrary immediate value. Examples of the latter type include bgt, which branches to the third operand when the first operand is greater than the second operand, and ulh, which implements an unaligned 2-byte load.

One of the most common extensions provided by macros is to expand memory offsets to the full address range (32 or 64 bits) and to allow symbolic offsets such as 'my\_data + 4' to be used in place of integer constants. For example, the architectural instruction lbu allows only a signed 16-bit offset, whereas the macro lbu allows code such as 'lbu \$4,array+32769(\$5)'. The implementation of these symbolic offsets depends on several factors, such as whether the assembler is generating SVR4-style PIC (selected by -KPIC, see Section 9.27.1 [Assembler options], page 233), the size of symbols (see Section 9.27.3 [Directives to override the size of symbols], page 242), and the small data limit (see Section 9.27.4 [Controlling the use of small data accesses], page 242).

Sometimes it is undesirable to have one assembly instruction expand to several machine instructions. The directive .set nomacro tells the assembler to warn when this happens. .set macro restores the default behavior.

Some macro instructions need a temporary register to store intermediate results. This register is usually \$1, also known as \$at, but it can be changed to any core register reg using .set at=reg. Note that \$at always refers to \$1 regardless of which register is being used as the temporary register.

Implicit uses of the temporary register in macros could interfere with explicit uses in the assembly code. The assembler therefore warns whenever it sees an explicit use of the temporary register. The directive .set noat silences this warning while .set at restores the default behavior. It is safe to use .set noat while .set nomacro is in effect since single-instruction macros never need a temporary register.

Note that while the GNU assembler provides these macros for compatibility, it does not make any attempt to optimize them with the surrounding code.

<sup>&</sup>lt;sup>1</sup> The term "macro" is somewhat overloaded here, since these macros have no relation to those defined by .macro, see Section 7.63 [.macro], page 69.

# 9.27.3 Directives to override the size of symbols

The n64 ABI allows symbols to have any 64-bit value. Although this provides a great deal of flexibility, it means that some macros have much longer expansions than their 32-bit counterparts. For example, the non-PIC expansion of 'dla \$4,sym' is usually:

```
lui $4,%highest(sym)
lui $1,%hi(sym)
daddiu $4,$4,%higher(sym)
daddiu $1,$1,%lo(sym)
dsll32 $4,$4,0
daddu $4,$4,$1
```

whereas the 32-bit expansion is simply:

lui \$4,%hi(sym) daddiu \$4,\$4,%lo(sym)

n64 code is sometimes constructed in such a way that all symbolic constants are known to have 32-bit values, and in such cases, it's preferable to use the 32-bit expansion instead of the 64-bit expansion.

You can use the .set sym32 directive to tell the assembler that, from this point on, all expressions of the form 'symbol' or 'symbol + offset' have 32-bit values. For example:

```
.set sym32
dla $4,sym
lw $4,sym+16
sw $4,sym+0x8000($4)
```

will cause the assembler to treat 'sym', sym+16 and sym+0x8000 as 32-bit values. The handling of non-symbolic addresses is not affected.

The directive .set nosym32 ends a .set sym32 block and reverts to the normal behavior. It is also possible to change the symbol size using the command-line options -msym32 and -mno-sym32.

These options and directives are always accepted, but at present, they have no effect for anything other than n64.

# 9.27.4 Controlling the use of small data accesses

It often takes several instructions to load the address of a symbol. For example, when 'addr' is a 32-bit symbol, the non-PIC expansion of 'dla \$4,addr' is usually:

```
lui $4,%hi(addr)
daddiu $4,$4,%lo(addr)
```

The sequence is much longer when 'addr' is a 64-bit symbol. See Section 9.27.3 [Directives to override the size of symbols], page 242.

In order to cut down on this overhead, most embedded MIPS systems set aside a 64kilobyte "small data" area and guarantee that all data of size n and smaller will be placed in that area. The limit n is passed to both the assembler and the linker using the command-line option -G n, see Section 9.27.1 [Assembler options], page 233. Note that the same value of nmust be used when linking and when assembling all input files to the link; any inconsistency could cause a relocation overflow error.

The size of an object in the .bss section is set by the .comm or .lcomm directive that defines it. The size of an external object may be set with the .extern directive. For example, '.extern sym,4' declares that the object at sym is 4 bytes in length, while leaving sym otherwise undefined.

When no -G option is given, the default limit is 8 bytes. The option -G 0 prevents any data from being automatically classified as small.

It is also possible to mark specific objects as small by putting them in the special sections .sdata and .sbss, which are "small" counterparts of .data and .bss respectively. The toolchain will treat such data as small regardless of the -G setting.

On startup, systems that support a small data area are expected to initialize register \$28, also known as \$gp, in such a way that small data can be accessed using a 16-bit offset from that register. For example, when 'addr' is small data, the 'dla \$4,addr' instruction above is equivalent to:

daddiu \$4,\$28,%gp\_rel(addr)

Small data is not supported for SVR4-style PIC.

# 9.27.5 Directives to override the ISA level

GNU as supports an additional directive to change the MIPS Instruction Set Architecture level on the fly: .set mipsn. n should be a number from 0 to 5, or 32, 32r2, 32r3, 32r5, 32r6, 64, 64r2, 64r3, 64r5 or 64r6. The values other than 0 make the assembler accept instructions for the corresponding ISA level, from that point on in the assembly. .set mipsn affects not only which instructions are permitted, but also how certain macros are expanded. .set mips0 restores the ISA level to its original level: either the level you selected with command-line options, or the default for your configuration. You can use this feature to permit specific MIPS III instructions while assembling in 32 bit mode. Use this directive with care!

The .set arch=cpu directive provides even finer control. It changes the effective CPU target and allows the assembler to use instructions specific to a particular CPU. All CPUs supported by the '-march' command-line option are also selectable by this directive. The original value is restored by .set arch=default.

The directive .set mips16 puts the assembler into MIPS 16 mode, in which it will assemble instructions for the MIPS 16 processor. Use .set nomips16 to return to normal 32 bit mode.

Traditional MIPS assemblers do not support this directive.

The directive .set micromips puts the assembler into microMIPS mode, in which it will assemble instructions for the microMIPS processor. Use .set nomicromips to return to normal 32 bit mode.

Traditional MIPS assemblers do not support this directive.

# 9.27.6 Directives to control code generation

The .module directive allows command-line options to be set directly from assembly. The format of the directive matches the .set directive but only those options which are relevant to a whole module are supported. The effect of a .module directive is the same as the corresponding command-line option. Where .set directives support returning to a default then the .module directives do not as they define the defaults.

These module-level directives must appear first in assembly.

Traditional MIPS assemblers do not support this directive.

The directive .set insn32 makes the assembler only use 32-bit instruction encodings when generating code for the microMIPS processor. This directive inhibits the use of any 16-bit instructions from that point on in the assembly. The .set noinsn32 directive allows 16-bit instructions to be accepted.

Traditional MIPS assemblers do not support this directive.

# 9.27.7 Directives for extending MIPS 16 bit instructions

By default, MIPS 16 instructions are automatically extended to 32 bits when necessary. The directive .set noautoextend will turn this off. When .set noautoextend is in effect, any 32 bit instruction must be explicitly extended with the .e modifier (e.g., li.e \$4,1000). The directive .set autoextend may be used to once again automatically extend instructions when necessary.

This directive is only meaningful when in MIPS 16 mode. Traditional MIPS assemblers do not support this directive.

# 9.27.8 Directive to mark data as an instruction

The .insn directive tells as that the following data is actually instructions. This makes a difference in MIPS 16 and microMIPS modes: when loading the address of a label which precedes instructions, as automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

The .global and .globl directives supported by as will by default mark the symbol as pointing to a region of data not code. This means that, for example, any instructions following such a symbol will not be disassembled by objdump as it will regard them as data. To change this behavior an optional section name can be placed after the symbol name in the .global directive. If this section exists and is known to be a code section, then the symbol will be marked as pointing at code not data. Ie the syntax for the directive is:

.global symbol[ section] [, symbol[ section]] ...,

Here is a short example:

.global foo .text, bar, baz .data foo: nop bar: .word 0x0 baz: .word 0x1

# 9.27.9 Directives to control the FP ABI

## 9.27.9.1 History of FP ABIs

The MIPS ABIs support a variety of different floating-point extensions where callingconvention and register sizes vary for floating-point data. The extensions exist to support a wide variety of optional architecture features. The resulting ABI variants are generally incompatible with each other and must be tracked carefully.

Traditionally the use of an explicit .gnu\_attribute 4, n directive is used to indicate which ABI is in use by a specific module. It was then left to the user to ensure that command-line options and the selected ABI were compatible with some potential for inconsistencies.

# 9.27.9.2 Supported FP ABIs

The supported floating-point ABI variants are:

### 0 - No floating-point

This variant is used to indicate that floating-point is not used within the module at all and therefore has no impact on the ABI. This is the default.

### 1 - Double-precision

This variant indicates that double-precision support is used. For 64-bit ABIs this means that 64-bit wide floating-point registers are required. For 32-bit ABIs this means that 32-bit wide floating-point registers are required and double-precision operations use pairs of registers.

# 2 - Single-precision

This variant indicates that single-precision support is used. Double precision operations will be supported via soft-float routines.

### 3 - Soft-float

This variant indicates that although floating-point support is used all operations are emulated in software. This means the ABI is modified to pass all floatingpoint data in general-purpose registers.

### 4 - Deprecated

This variant existed as an initial attempt at supporting 64-bit wide floating-point registers for O32 ABI on a MIPS32r2 CPU. This has been superseded by 5, 6 and 7.

### 5 - Double-precision 32-bit CPU, 32-bit or 64-bit FPU

This variant is used by 32-bit ABIs to indicate that the floating-point code in the module has been designed to operate correctly with either 32-bit wide or 64-bit wide floating-point registers. Double-precision support is used. Only O32 currently supports this variant and requires a minimum architecture of MIPS II.

### 6 - Double-precision 32-bit FPU, 64-bit FPU

This variant is used by 32-bit ABIs to indicate that the floating-point code in the module requires 64-bit wide floating-point registers. Double-precision support is used. Only O32 currently supports this variant and requires a minimum architecture of MIPS32r2.

### 7 - Double-precision compat 32-bit FPU, 64-bit FPU

This variant is used by 32-bit ABIs to indicate that the floating-point code in the module requires 64-bit wide floating-point registers. Double-precision support is used. This differs from the previous ABI as it restricts use of odd-numbered single-precision registers. Only O32 currently supports this variant and requires a minimum architecture of MIPS32r2.

# 9.27.9.3 Automatic selection of FP ABI

In order to simplify and add safety to the process of selecting the correct floating-point ABI, the assembler will automatically infer the correct .gnu\_attribute 4, n directive based on command-line options and .module overrides. Where an explicit .gnu\_attribute 4, n directive has been seen then a warning will be raised if it does not match an inferred setting.

The floating-point ABI is inferred as follows. If '-msoft-float' has been used the module will be marked as soft-float. If '-msingle-float' has been used then the module will be marked as single-precision. The remaining ABIs are then selected based on the FP register width. Double-precision is selected if the width of GP and FP registers match and the special double-precision variants for 32-bit ABIs are then selected depending on '-mfpxx', '-mfp64' and '-mno-odd-spreg'.

# 9.27.9.4 Linking different FP ABI variants

Modules using the default FP ABI (no floating-point) can be linked with any other (singular) FP ABI variant.

Special compatibility support exists for O32 with the four double-precision FP ABI variants. The '-mfpxx' FP ABI is specifically designed to be compatible with the standard double-precision ABI and the '-mfp64' FP ABIs. This makes it desirable for O32 modules to be built as '-mfpxx' to ensure the maximum compatibility with other modules produced for more specific needs. The only FP ABIs which cannot be linked together are the standard double-precision ABI and the full '-mfp64' ABI with '-modd-spreg'.

# 9.27.10 Directives to record which NaN encoding is being used

The IEEE 754 floating-point standard defines two types of not-a-number (NaN) data: "signalling" NaNs and "quiet" NaNs. The original version of the standard did not specify how these two types should be distinguished. Most implementations followed the i387 model, in which the first bit of the significand is set for quiet NaNs and clear for signalling NaNs. However, the original MIPS implementation assigned the opposite meaning to the bit, so that it was set for signalling NaNs and clear for quiet NaNs.

The 2008 revision of the standard formally suggested the i387 choice and as from Sep 2012 the current release of the MIPS architecture therefore optionally supports that form. Code that uses one NaN encoding would usually be incompatible with code that uses the other NaN encoding, so MIPS ELF objects have a flag (EF\_MIPS\_NAN2008) to record which encoding is being used.

Assembly files can use the .nan directive to select between the two encodings. '.nan 2008' says that the assembly file uses the IEEE 754-2008 encoding while '.nan legacy' says that the file uses the original MIPS encoding. If several .nan directives are given, the final setting is the one that is used.

The command-line options -mnan=legacy and -mnan=2008 can be used instead of '.nan legacy' and '.nan 2008' respectively. However, any .nan directive overrides the command-line setting.

'.nan legacy' is the default if no .nan directive or -mnan option is given.

Note that GNU as does not produce NaNs itself and therefore these directives do not affect code generation. They simply control the setting of the EF\_MIPS\_NAN2008 flag.

Traditional MIPS assemblers do not support these directives.

# 9.27.11 Directives to save and restore options

The directives .set push and .set pop may be used to save and restore the current settings for all the options which are controlled by .set. The .set push directive saves the current settings on a stack. The .set pop directive pops the stack and restores the settings. These directives can be useful inside an macro which must change an option such as the ISA level or instruction reordering but does not want to change the state of the code which invoked the macro.

Traditional MIPS assemblers do not support these directives.

# 9.27.12 Directives to control generation of MIPS ASE instructions

The directive .set mips3d makes the assembler accept instructions from the MIPS-3D Application Specific Extension from that point on in the assembly. The .set nomips3d directive prevents MIPS-3D instructions from being accepted.

The directive .set smartmips makes the assembler accept instructions from the Smart-MIPS Application Specific Extension to the MIPS32 ISA from that point on in the assembly. The .set nosmartmips directive prevents SmartMIPS instructions from being accepted.

The directive .set mdmx makes the assembler accept instructions from the MDMX Application Specific Extension from that point on in the assembly. The .set nomdmx directive prevents MDMX instructions from being accepted.

The directive .set dsp makes the assembler accept instructions from the DSP Release 1 Application Specific Extension from that point on in the assembly. The .set nodsp directive prevents DSP Release 1 instructions from being accepted.

The directive .set dspr2 makes the assembler accept instructions from the DSP Release 2 Application Specific Extension from that point on in the assembly. This directive implies .set dsp. The .set nodspr2 directive prevents DSP Release 2 instructions from being accepted.

The directive .set dspr3 makes the assembler accept instructions from the DSP Release 3 Application Specific Extension from that point on in the assembly. This directive implies .set dsp and .set dspr2. The .set nodspr3 directive prevents DSP Release 3 instructions from being accepted.

The directive .set mt makes the assembler accept instructions from the MT Application Specific Extension from that point on in the assembly. The .set nomt directive prevents MT instructions from being accepted.

The directive .set mcu makes the assembler accept instructions from the MCU Application Specific Extension from that point on in the assembly. The .set nomcu directive prevents MCU instructions from being accepted.

The directive .set msa makes the assembler accept instructions from the MIPS SIMD Architecture Extension from that point on in the assembly. The .set nomsa directive prevents MSA instructions from being accepted.

The directive .set virt makes the assembler accept instructions from the Virtualization Application Specific Extension from that point on in the assembly. The .set novirt directive prevents Virtualization instructions from being accepted.

The directive .set xpa makes the assembler accept instructions from the XPA Extension from that point on in the assembly. The .set noxpa directive prevents XPA instructions from being accepted.

The directive .set mips16e2 makes the assembler accept instructions from the MIPS16e2 Application Specific Extension from that point on in the assembly, whenever in MIPS16 mode. The .set nomips16e2 directive prevents MIPS16e2 instructions from being accepted,

in MIPS16 mode. Neither directive affects the state of MIPS16 mode being active itself which has separate controls.

The directive .set crc makes the assembler accept instructions from the CRC Extension from that point on in the assembly. The .set nocrc directive prevents CRC instructions from being accepted.

The directive .set ginv makes the assembler accept instructions from the GINV Extension from that point on in the assembly. The .set noginv directive prevents GINV instructions from being accepted.

The directive .set loongson-mmi makes the assembler accept instructions from the MMI Extension from that point on in the assembly. The .set noloongson-mmi directive prevents MMI instructions from being accepted.

The directive .set loongson-cam makes the assembler accept instructions from the Loongson CAM from that point on in the assembly. The .set noloongson-cam directive prevents Loongson CAM instructions from being accepted.

The directive .set loongson-ext makes the assembler accept instructions from the Loongson EXT from that point on in the assembly. The .set noloongson-ext directive prevents Loongson EXT instructions from being accepted.

The directive .set loongson-ext2 makes the assembler accept instructions from the Loongson EXT2 from that point on in the assembly. This directive implies .set loognson-ext. The .set noloongson-ext2 directive prevents Loongson EXT2 instructions from being accepted.

Traditional MIPS assemblers do not support these directives.

# 9.27.13 Directives to override floating-point options

The directives .set softfloat and .set hardfloat provide finer control of disabling and enabling float-point instructions. These directives always override the default (that hard-float instructions are accepted) or the command-line options ('-msoft-float' and '-mhard-float').

The directives .set singlefloat and .set doublefloat provide finer control of disabling and enabling double-precision float-point operations. These directives always override the default (that double-precision operations are accepted) or the command-line options ('-msingle-float' and '-mdouble-float').

Traditional MIPS assemblers do not support these directives.

### 9.27.14 Syntactical considerations for the MIPS assembler

# 9.27.14.1 Special Characters

The presence of a '#' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

# 9.28 MMIX Dependent Features

## 9.28.1 Command-line Options

The MMIX version of **as** has some machine-dependent options.

When '--fixed-special-register-names' is specified, only the register names specified in Section 9.28.3.3 [MMIX-Regs], page 251, are recognized in the instructions PUT and GET.

You can use the '--globalize-symbols' to make all symbols global. This option is useful when splitting up a mmixal program into several files.

The '--gnu-syntax' turns off most syntax compatibility with mmixal. Its usability is currently doubtful.

The '--relax' option is not fully supported, but will eventually make the object file prepared for linker relaxation.

If you want to avoid inadvertently calling a predefined symbol and would rather get an error, for example when using as with a compiler or other machine-generated code, specify '--no-predefined-syms'. This turns off built-in predefined definitions of all such symbols, including rounding-mode symbols, segment symbols, 'BIT' symbols, and TRAP symbols used in mmix "system calls". It also turns off predefined special-register names, except when used in PUT and GET instructions.

By default, some instructions are expanded to fit the size of the operand or an external symbol (see Section 9.28.2 [MMIX-Expand], page 250). By passing '--no-expand', no such expansion will be done, instead causing errors at link time if the operand does not fit.

The mmixal documentation (see [mmixsite], page 250) specifies that global registers allocated with the 'GREG' directive (see [MMIX-greg], page 252) and initialized to the same non-zero value, will refer to the same global register. This isn't strictly enforceable in as since the final addresses aren't known until link-time, but it will do an effort unless the '--no-merge-gregs' option is specified. (Register merging isn't yet implemented in 1d.)

as will warn every time it expands an instruction to fit an operand unless the option '-x' is specified. It is believed that this behaviour is more useful than just mimicking mmixal's behaviour, in which instructions are only expanded if the '-x' option is specified, and assembly fails otherwise, when an instruction needs to be expanded. It needs to be kept in mind that mmixal is both an assembler and linker, while as will expand instructions that at link stage can be contracted. (Though linker relaxation isn't yet implemented in ld.) The option '-x' also implies '--linker-allocated-gregs'.

If instruction expansion is enabled, **as** can expand a 'PUSHJ' instruction into a series of instructions. The shortest expansion is to not expand it, but just mark the call as redirectable to a stub, which 1d creates at link-time, but only if the original 'PUSHJ' instruction is found not to reach the target. The stub consists of the necessary instructions to form a jump to the target. This happens if **as** can assert that the 'PUSHJ' instruction can reach such a stub. The option '--no-pushj-stubs' disables this shorter expansion, and the longer series of instructions is then created at assembly-time. The option '--no-stubs' is a synonym, intended for compatibility with future releases, where generation of stubs for other instructions may be implemented.

Usually a two-operand-expression (see [GREG-base], page 253) without a matching 'GREG' directive is treated as an error by as. When the option '--linker-allocated-gregs' is

in effect, they are instead passed through to the linker, which will allocate as many global registers as is needed.

# 9.28.2 Instruction expansion

When **as** encounters an instruction with an operand that is either not known or does not fit the operand size of the instruction, **as** (and **ld**) will expand the instruction into a sequence of instructions semantically equivalent to the operand fitting the instruction. Expansion will take place for the following instructions:

'GETA' Expands to a sequence of four instructions: SETL, INCML, INCMH and INCH. The operand must be a multiple of four.

Conditional branches

A branch instruction is turned into a branch with the complemented condition and prediction bit over five instructions; four instructions setting \$255 to the operand value, which like with GETA must be a multiple of four, and a final GO \$255,\$255,0.

- 'PUSHJ' Similar to expansion for conditional branches; four instructions set \$255 to the operand value, followed by a PUSHGO \$255,\$255,0.
- 'JMP' Similar to conditional branches and PUSHJ. The final instruction is GO \$255,\$255,0.

The linker ld is expected to shrink these expansions for code assembled with '--relax' (though not currently implemented).

# 9.28.3 Syntax

The assembly syntax is supposed to be upward compatible with that described in Sections 1.3 and 1.4 of 'The Art of Computer Programming, Volume 1'. Draft versions of those chapters as well as other MMIX information is located at http://www-cs-faculty.stanford.edu/ ~knuth/mmix-news.html. Most code examples from the mmixal package located there should work unmodified when assembled and linked as single files, with a few noteworthy exceptions (see Section 9.28.4 [MMIX-mmixal], page 254).

Before an instruction is emitted, the current location is aligned to the next four-byte boundary. If a label is defined at the beginning of the line, its value will be the aligned value.

In addition to the traditional hex-prefix '0x', a hexadecimal number can also be specified by the prefix character '#'.

After all operands to an MMIX instruction or directive have been specified, the rest of the line is ignored, treated as a comment.

# 9.28.3.1 Special Characters

The characters '\*' and '#' are line comment characters; each start a comment at the beginning of a line, but only at the beginning of a line. A '#' prefixes a hexadecimal number if found elsewhere on a line. If a '#' appears at the start of a line the whole line is treated as a comment, but the line can also act as a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Two other characters, '%' and '!', each start a comment anywhere on the line. Thus you can't use the 'modulus' and 'not' operators in expressions normally associated with these two characters.

A ';' is a line separator, treated as a new-line, so separate instructions can be specified on a single line.

### 9.28.3.2 Symbols

The character ':' is permitted in identifiers. There are two exceptions to it being treated as any other symbol character: if a symbol begins with ':', it means that the symbol is in the global namespace and that the current prefix should not be prepended to that symbol (see [MMIX-prefix], page 254). The ':' is then not considered part of the symbol. For a symbol in the label position (first on a line), a ':' at the end of a symbol is silently stripped off. A label is permitted, but not required, to be followed by a ':', as with many other assembly formats.

The character '@' in an expression, is a synonym for '.', the current location.

In addition to the common forward and backward local symbol formats (see Section 5.3 [Symbol Names], page 43), they can be specified with upper-case 'B' and 'F', as in '8B' and '9F'. A local label defined for the current position is written with a 'H' appended to the number:

3H LDB \$0,\$1,2

This and traditional local-label formats cannot be mixed: a label must be defined and referred to using the same format.

There's a minor caveat: just as for the ordinary local symbols, the local symbols are translated into ordinary symbols using control characters are to hide the ordinal number of the symbol. Unfortunately, these symbols are not translated back in error messages. Thus you may see confusing error messages when local symbols are used. Control characters '\003' (control-C) and '\004' (control-D) are used for the MMIX-specific local-symbol syntax.

The symbol 'Main' is handled specially; it is always global.

By defining the symbols '\_\_.MMIX.start..text' and '\_\_.MMIX.start..data', the address of respectively the '.text' and '.data' segments of the final program can be defined, though when linking more than one object file, the code or data in the object file containing the symbol is not guaranteed to be start at that position; just the final executable. See [MMIX-loc], page 252.

# 9.28.3.3 Register names

Local and global registers are specified as '\$0' to '\$255'. The recognized special register names are 'rJ', 'rA', 'rB', 'rC', 'rD', 'rE', 'rF', 'rG', 'rH', 'rI', 'rK', 'rL', 'rM', 'rN', 'rO', 'rP', 'rQ', 'rR', 'rS', 'rT', 'rU', 'rV', 'rX', 'rY', 'rZ', 'rBB', 'rTT', 'rWW', 'rXX', 'rYY' and 'rZZ'. A leading ':' is optional for special register names.

Local and global symbols can be equated to register names and used in place of ordinary registers.

Similarly for special registers, local and global symbols can be used. Also, symbols equated from numbers and constant expressions are allowed in place of a special register, except when either of the options --no-predefined-syms and --fixed-special-register-names are

specified. Then only the special register names above are allowed for the instructions having a special register operand; GET and PUT.

### 9.28.3.4 Assembler Directives

LOC

The LOC directive sets the current location to the value of the operand field, which may include changing sections. If the operand is a constant, the section is set to either .data if the value is 0x200000000000000 or larger, else it is set to .text. Within a section, the current location may only be changed to monotonically higher addresses. A LOC expression must be a previously defined symbol or a "pure" constant.

An example, which sets the label *prev* to the current location, and updates the current location to eight bytes forward:

prev LOC @+8

When a LOC has a constant as its operand, a symbol \_\_.MMIX.start..text or \_\_.MMIX.start..data is defined depending on the address as mentioned above. Each such symbol is interpreted as special by the linker, locating the section at that address. Note that if multiple files are linked, the first object file with that section will be mapped to that address (not necessarily the file with the LOC definition).

LOCAL

Example:

LOCAL external\_symbol LOCAL 42 .local asymbol

This directive-operation generates a link-time assertion that the operand does not correspond to a global register. The operand is an expression that at linktime resolves to a register symbol or a number. A number is treated as the register having that number. There is one restriction on the use of this directive: the pseudo-directive must be placed in a section with contents, code or data.

IS

The IS directive:

asymbol IS an\_expression

sets the symbol 'asymbol' to 'an\_expression'. A symbol may not be set more than once using this directive. Local labels may be set using this directive, for example:

5H IS @+4

### GREG

This directive reserves a global register, gives it an initial value and optionally gives it a symbolic name. Some examples:

```
areg GREG
breg GREG data_value
GREG data_buffer
.greg creg, another_data_value
```

The symbolic register name can be used in place of a (non-special) register. If a value isn't provided, it defaults to zero. Unless the option '--no-merge-gregs' is specified, non-zero registers allocated with this directive may be eliminated by as; another register with the same value used in its place. Any of the instructions 'CSWAP', 'GO', 'LDA', 'LDBU', 'LDB', 'LDHT', 'LDOU', 'LDO', 'LDSF', 'LDTU', 'LDT', 'LDUNC', 'LDVTS', 'LDWU', 'LDW', 'PREGO', 'PRELD', 'PREST', 'PUSHGO', 'STBU', 'STB', 'STCO', 'STHT', 'STOU', 'STSF', 'STTU', 'STT', 'STUNC', 'SYNCID', can have a value nearby an initial value in place of its second and third operands. Here, "nearby" is defined as within the range 0...255 from the initial value of such an allocated register.

```
buffer1 BYTE 0,0,0,0,0
buffer2 BYTE 0,0,0,0,0
...
GREG buffer1
LDOU $42,buffer2
```

In the example above, the 'Y' field of the LDOUI instruction (LDOU with a constant Z) will be replaced with the global register allocated for 'buffer1', and the 'Z' field will have the value 5, the offset from 'buffer1' to 'buffer2'. The result is equivalent to this code:

```
buffer1 BYTE 0,0,0,0,0
buffer2 BYTE 0,0,0,0,0
...
tmpreg GREG buffer1
LDOU $42,tmpreg,(buffer2-buffer1)
```

Global registers allocated with this directive are allocated in order higher-tolower within a file. Other than that, the exact order of register allocation and elimination is undefined. For example, the order is undefined when more than one file with such directives are linked together. With the options '-x' and '--linker-allocated-gregs', 'GREG' directives for two-operand cases like the one mentioned above can be omitted. Sufficient global registers will then be allocated by the linker.

BYTE

The 'BYTE' directive takes a series of operands separated by a comma. If an operand is a string (see Section 3.6.1.1 [Strings], page 33), each character of that string is emitted as a byte. Other operands must be constant expressions without forward references, in the range 0...255. If you need operands having expressions with forward references, use '.byte' (see Section 7.11 [Byte], page 54). An operand can be omitted, defaulting to a zero value.

WYDE TETRA OCTA

> The directives 'WYDE', 'TETRA' and 'OCTA' emit constants of two, four and eight bytes size respectively. Before anything else happens for the directive, the current location is aligned to the respective constant-size boundary. If a label is defined at the beginning of the line, its value will be that after the alignment. A single operand can be omitted, defaulting to a zero value emitted for the directive. Operands can be expressed as strings (see Section 3.6.1.1 [Strings],

page 33), in which case each character in the string is emitted as a separate constant of the size indicated by the directive.

#### PREFIX

The 'PREFIX' directive sets a symbol name prefix to be prepended to all symbols (except local symbols, see Section 9.28.3.2 [MMIX-Symbols], page 251), that are not prefixed with ':', until the next 'PREFIX' directive. Such prefixes accumulate. For example,

```
PREFIX a
PREFIX b
c IS 0
```

defines a symbol 'abc' with the value 0.

BSPEC ESPEC

A pair of 'BSPEC' and 'ESPEC' directives delimit a section of special contents (without specified semantics). Example:

```
BSPEC 42
TETRA 1,2,3
ESPEC
```

The single operand to 'BSPEC' must be number in the range 0...255. The 'BSPEC' number 80 is used by the GNU binutils implementation.

# 9.28.4 Differences to mmixal

The binutils **as** and **ld** combination has a few differences in function compared to **mmixal** (see [mmixsite], page 250).

The replacement of a symbol with a GREG-allocated register (see [GREG-base], page 253) is not handled the exactly same way in **as** as in **mmixal**. This is apparent in the **mmixal** example file **inout.mms**, where different registers with different offsets, eventually yielding the same address, are used in the first instruction. This type of difference should however not affect the function of any program unless it has specific assumptions about the allocated register number.

Line numbers (in the 'mmo' object format) are currently not supported.

Expression operator precedence is not that of mmixal: operator precedence is that of the C programming language. It's recommended to use parentheses to explicitly specify wanted operator precedence whenever more than one type of operators are used.

The serialize unary operator &, the fractional division operator '//', the logical not operator ! and the modulus operator '%' are not available.

Symbols are not global by default, unless the option '--globalize-symbols' is passed. Use the '.global' directive to globalize symbols (see Section 7.41 [Global], page 63).

Operand syntax is a bit stricter with as than mmixal. For example, you can't say addu 1,2,3, instead you must write addu \$1,\$2,3.

You can't LOC to a lower address than those already visited (i.e., "backwards").

A LOC directive must come before any emitted code.

Predefined symbols are visible as file-local symbols after use. (In the ELF file, that is—the linked mmo file has no notion of a file-local symbol.)

Some mapping of constant expressions to sections in LOC expressions is attempted, but that functionality is easily confused and should be avoided unless compatibility with mmixal is required. A LOC expression to '0x200000000000000' or higher, maps to the '.data' section and lower addresses map to the '.text' section (see [MMIX-loc], page 252).

The code and data areas are each contiguous. Sparse programs with far-away LOC directives will take up the same amount of space as a contiguous program with zeros filled in the gaps between the LOC directives. If you need sparse programs, you might try and get the wanted effect with a linker script and splitting up the code parts into sections (see Section 7.85 [Section], page 77). Assembly code for this, to be compatible with mmixal, would look something like:

```
.if 0
LOC away_expression
.else
.section away,"ax"
.fi
```

as will not execute the LOC directive and mmixal ignores the lines with .. This construct can be used generally to help compatibility.

Symbols can't be defined twice-not even to the same value.

Instruction mnemonics are recognized case-insensitive, though the 'IS' and 'GREG' pseudooperations must be specified in upper-case characters.

There's no unicode support.

The following is a list of programs in 'mmix.tar.gz', available at http:// www-cs-faculty.stanford.edu/~knuth/mmix-news.html, last checked with the version dated 2001-08-25 (md5sum c393470cfc86fac040487d22d2bf0172) that assemble with mmixal but do not assemble with as:

silly.mms

LOC to a previous address.

sim.mms Redefines symbol 'Done'.

test.mms Uses the serial operator '&'.

# 9.29 MSP 430 Dependent Features

# 9.29.1 Options

- -mmcu selects the mcu architecture. If the architecture is 430Xv2 then this also enables NOP generation unless the -mN is also specified.
- -mcpu selects the cpu architecture. If the architecture is 430Xv2 then this also enables NOP generation unless the -mN is also specified.

# -msilicon-errata=name[,name...]

Implements a fixup for named silicon errata. Multiple silicon errata can be specified by multiple uses of the -msilicon-errata option and/or by including the errata names, separated by commas, on an individual -msilicon-errata option. Errata names currently recognised by the assembler are:

- cpu4 PUSH #4 and PUSH #8 need longer encodings on the MSP430. This option is enabled by default, and cannot be disabled.
- cpu8 Do not set the SP to an odd value.
- cpull Do not update the SR and the PC in the same instruction.
- cpu12 Do not use the PC in a CMP or BIT instruction.
- cpu13 Do not use an arithmetic instruction to modify the SR.
- cpu19 Insert NOP after CPUOFF.

### -msilicon-errata-warn=name[,name...]

Like the -msilicon-errata option except that instead of fixing the specified errata, a warning message is issued instead. This option can be used alongside -msilicon-errata to generate messages whenever a problem is fixed, or on its own in order to inspect code for potential problems.

- -mP enables polymorph instructions handler.
- -mQ enables relaxation at assembly time. DANGEROUS!
- -ml indicates that the input uses the large code model.
- -mn enables the generation of a NOP instruction following any instruction that might change the interrupts enabled/disabled state. The pipelined nature of the MSP430 core means that any instruction that changes the interrupt state (EINT, DINT, BIC #8, SR, BIS #8, SR or MOV.W <>, SR) must be followed by a NOP instruction in order to ensure the correct processing of interrupts. By default it is up to the programmer to supply these NOP instructions, but this command-line option enables the automatic insertion by the assembler, if they are missing.
- -mN disables the generation of a NOP instruction following any instruction that might change the interrupts enabled/disabled state. This is the default behaviour.
- -my tells the assembler to generate a warning message if a NOP does not immediately follow an instruction that enables or disables interrupts. This is the default.

Note that this option can be stacked with the **-mn** option so that the assembler will both warn about missing NOP instructions and then insert them automatically.

- -mY disables warnings about missing NOP instructions.
- -md mark the object file as one that requires data to copied from ROM to RAM at execution startup. Disabled by default.

### -mdata-region=region

Select the region data will be placed in. Region placement is performed by the compiler and linker. The only effect this option will have on the assembler is that if *upper* or *either* is selected, then the symbols to initialise high data and bss will be defined. Valid *region* values are:

none

lower

upper

either

# 9.29.2 Syntax

# 9.29.2.1 Macros

The macro syntax used on the MSP 430 is like that described in the MSP 430 Family Assembler Specification. Normal **as** macros should still work.

Additional built-in macros are:

- **llo(exp)** Extracts least significant word from 32-bit expression 'exp'.
- lhi(exp) Extracts most significant word from 32-bit expression 'exp'.
- hlo(exp) Extracts 3rd word from 64-bit expression 'exp'.
- hhi(exp) Extracts 4rd word from 64-bit expression 'exp'.

They normally being used as an immediate source operand.

mov #llo(1), r10 ; == mov #1, r10
mov #lhi(1), r10 ; == mov #0, r10

# 9.29.2.2 Special Characters

A semicolon (';') appearing anywhere on a line starts a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Multiple statements can appear on the same line provided that they are separated by the '{' character.

The character '\$' in jump instructions indicates current location and implemented only for TI syntax compatibility.

# 9.29.2.3 Register Names

General-purpose registers are represented by predefined symbols of the form 'rN' (for global registers), where N represents a number between 0 and 15. The leading letters may be in either upper or lower case; for example, 'r13' and 'R7' are both valid register names.

Register names 'PC', 'SP' and 'SR' cannot be used as register names and will be treated as variables. Use 'r0', 'r1', and 'r2' instead.

# 9.29.2.4 Assembler Extensions

**@rN** As destination operand being treated as '0(**rn**)'

0(rN) As source operand being treated as '@rn'

jCOND +N Skips next N bytes followed by jump instruction and equivalent to 'jCOND \$+N+2'

Also, there are some instructions, which cannot be found in other assemblers. These are branch instructions, which has different opcodes upon jump distance. They all got PC relative addressing mode.

bne label A polymorph instruction which is 'jne label' or 'jeq +4; br label'

blt label A polymorph instruction which is 'jl label' or 'jge +4; br label'

bltn label

A polymorph instruction which is 'jn label' or 'jn +2; jmp +4; br label'

bltu label

A polymorph instruction which is 'jlo label' or 'jhs +2; br label'

bge label A polymorph instruction which is 'jge label' or 'jl +4; br label'

bgeu label

A polymorph instruction which is 'jhs label' or 'jlo +4; br label'

bgt label A polymorph instruction which is 'jeq +2; jge label' or 'jeq +6; jl +4; br label'

bgtu label

A polymorph instruction which is 'jeq +2; jhs label' or 'jeq +6; jlo +4; br label'

bleu label

A polymorph instruction which is 'jeq label; jlo label' or 'jeq +2; jhs +4; br label'

jump label

A polymorph instruction which is 'jmp label' or 'br label'

# 9.29.3 Floating Point

The MSP 430 family uses IEEE 32-bit floating-point numbers.

# 9.29.4 MSP 430 Machine Directives

.file This directive is ignored; it is accepted for compatibility with other MSP 430 assemblers.

*Warning:* in other versions of the GNU assembler, .file is used for the directive called .app-file in the MSP 430 support.

- .line This directive is ignored; it is accepted for compatibility with other MSP 430 assemblers.
- .arch Sets the target microcontroller in the same way as the -mmcu command-line option.
- .cpu Sets the target architecture in the same way as the -mcpu command-line option.
- .profiler

This directive instructs assembler to add new profile entry to the object file.

.refsym This directive instructs assembler to add an undefined reference to the symbol following the directive. The maximum symbol name length is 1023 characters. No relocation is created for this symbol; it will exist purely for pulling in object files from archives. Note that this reloc is not sufficient to prevent garbage collection; use a KEEP() directive in the linker file to preserve such objects.

# .mspabi\_attribute

This directive tells the assembler what the MSPABI build attributes for this file are. This is used for validating the command line options passed to the assembler against the options the original source file was compiled with. The expected format is: '.mspabi\_attribute tag\_name, tag\_value' For example, to set the tag OFBA\_MSPABI\_Tag\_ISA to MSP430X: '.mspabi\_attribute 4, 2' See the MSP430 EABI, document slaa534 for the details on tag names and values.

# 9.29.5 Opcodes

**as** implements all the standard MSP 430 opcodes. No additional pseudo-instructions are needed on this family.

For information on the 430 machine instruction set, see MSP430 User's Manual, document slau049d, Texas Instrument, Inc.

# 9.29.6 Profiling Capability

It is a performance hit to use gcc's profiling approach for this tiny target. Even more – jtag hardware facility does not perform any profiling functions. However we've got gdb's built-in simulator where we can do anything.

We define new section '.profiler' which holds all profiling information. We define new pseudo operation '.profiler' which will instruct assembler to add new profile entry to the object file. Profile should take place at the present address.

Pseudo operation format:

'.profiler flags,function\_to\_profile [, cycle\_corrector, extra]' where:

'flags' is a combination	of the following characters:
--------------------------	------------------------------

- s function entry
- x function exit
- i function is in init section
- f function is in fini section
- 1 library call
- c libc standard call
- d stack value demand
- I interrupt service routine
- P prologue start
- p prologue end
- E epilogue start
- e epilogue end
- j long jump / sjlj unwind
- a an arbitrary code fragment
- t extra parameter saved (a constant value like frame size)

### function\_to\_profile

a function address

### cycle\_corrector

a value which should be added to the cycle counter, zero if omitted.

### extra any extra parameter, zero if omitted.

```
For example:
```

```
.global fxx
.type fxx,@function
fxx:
.LFrameOffset_fxx=0x08
.profiler "scdP", fxx
                          ; function entry.
  ; we also demand stack value to be saved
 push r11
 push r10
 push r9
 push r8
.profiler "cdpt",fxx,0, .LFrameOffset_fxx ; check stack value at this point
  ; (this is a prologue end)
  ; note, that spare var filled with
  ; the farme size
 mov r15,r8
. . .
```

.profiler cdE,fxx ; check stack pop r8 pop r9 pop r10 pop r11 .profiler xcde,fxx,3 ; exit adds 3 to the cycle counter ret ; cause 'ret' insn takes 3 cycles

# 9.30 NDS32 Dependent Features

The NDS32 processors family includes high-performance and low-power 32-bit processors for high-end to low-end. GNU as for NDS32 architectures supports NDS32 ISA version 3. For detail about NDS32 instruction set, please see the AndeStar ISA User Manual which is available at http://www.andestech.com/en/index/index.htm

# 9.30.1 NDS32 Options

The NDS32 configurations of GNU **as** support these special options:

- -01 Optimize for performance.
- -Os Optimize for space.
- -EL Produce little endian data output.
- -EB Produce little endian data output.
- -mpic Generate PIC.
- -mno-fp-as-gp-relax

Suppress fp-as-gp relaxation for this file.

# -mb2bb-relax

Back-to-back branch optimization.

-mno-all-relax

Suppress all relaxation for this file.

-march=<arch name>

Assemble for architecture <arch name> which could be v3, v3j, v3m, v3f, v3s, v2, v2j, v2f, v2s.

# -mbaseline=<baseline>

Assemble for baseline <br/> baseline > which could be v2, v3, v3m.

# -mfpu-freg=FREG

Specify a FPU configuration.

0 8 SP / 4 DP registers 1 16 SP / 8 DP registers 2 32 SP / 16 DP registers 3 32 SP / 32 DP registers

### -mabi=abi

Specify a abi version <abi> could be v1, v2, v2fp, v2fpp.

# -m[no-]mac

Enable/Disable Multiply instructions support.

# -m[no-]div

Enable/Disable Divide instructions support.

# -m[no-]16bit-ext

Enable/Disable 16-bit extension

# -m[no-]dx-regs

Enable/Disable d0/d1 registers

-m[no-]pe:	rf-ext Enable/Disable Performance extension
-m[no-]pe:	rf2-ext Enable/Disable Performance extension 2
-m[no-]st:	ring-ext Enable/Disable String extension
-m[no-]re	duced-regs Enable/Disable Reduced Register configuration (GPR16) option
-m[no-]au	dio-isa-ext Enable/Disable AUDIO ISA extension
-m[no-]fp	u-sp-ext Enable/Disable FPU SP extension
-m[no-]fp	u-dp-ext Enable/Disable FPU DP extension
-m[no-]fp	u-fma Enable/Disable FPU fused-multiply-add instructions
-mall-ext	

Turn on all extensions and instructions support

# 9.30.2 Syntax

# 9.30.2.1 Special Characters

Use '#' at column 1 and '!' anywhere in the line except inside quotes.

Multiple instructions in a line are allowed though not recommended and should be separated by ';'.

Assembler is not case-sensitive in general except user defined label. For example, 'jral f1' is different from 'jral f1' while it is the same as 'JRAL F1'.

# 9.30.2.2 Register Names

```
General purpose registers (GPR)
```

There are 32 32-bit general purpose registers \$r0 to \$r31.

```
Accumulators d0 and d1
```

64-bit accumulators: \$d0.hi, \$d0.lo, \$d1.hi, and \$d1.lo.

### Assembler reserved register \$ta

Register ta (r15) is reserved for assembler using.

# Operating system reserved registers \$p0 and \$p1

Registers p0 (\$r26) and \$p1 (\$r27) are used by operating system as scratch registers.

### Frame pointer \$fp

Register \$r28 is regarded as the frame pointer.

#### Global pointer

Register \$r29 is regarded as the global pointer.

#### Link pointer

Register \$r30 is regarded as the link pointer.

#### Stack pointer

Register \$r31 is regarded as the stack pointer.

## 9.30.2.3 Pseudo Instructions

#### li rt5,imm32

load 32-bit integer into register rt5. 'sethi rt5, hi20(imm32)' and then 'ori rt5, reg, lo12(imm32)'.

#### la rt5,var

Load 32-bit address of var into register rt5. 'sethi rt5,hi20(var)' and then 'ori reg,rt5,lo12(var)'

### 1.[bhw] rt5,var

Load value of var into register rt5. 'sethi \$ta,hi20(var)' and then 'l[bhw]i rt5,[\$ta+lo12(var)]'

### 1.[bh]s rt5,var

Load value of var into register rt5. 'sethi \$ta,hi20(var)' and then 'l[bh]si rt5,[\$ta+lo12(var)]'

## l.[bhw]p rt5,var,inc

Load value of var into register rt5 and increment \$ta by amount inc. 'la \$ta,var' and then 'l[bhw]i.bi rt5, [\$ta],inc'

#### l.[bhw]pc rt5,inc

Continue loading value of var into register rt5 and increment \$ta by amount inc. 'l[bhw]i.bi rt5,[\$ta],inc.'

# l.[bh]sp rt5,var,inc

Load value of var into register rt5 and increment \$ta by amount inc. 'la \$ta,var' and then 'l[bh]si.bi rt5, [\$ta],inc'

### 1.[bh]spc rt5,inc

Continue loading value of var into register rt5 and increment \$ta by amount inc. '1[bh]si.bi rt5,[\$ta],inc.'

#### s.[bhw] rt5,var

Store register rt5 to var. 'sethi \$ta,hi20(var)' and then 's[bhw]i rt5,[\$ta+lo12(var)]'

# s.[bhw]p rt5,var,inc

Store register rt5 to var and increment \$ta by amount inc. 'la \$ta,var' and then 's[bhw]i.bi rt5,[\$ta],inc'

### s.[bhw]pc rt5,inc

Continue storing register rt5 to var and increment \$ta by amount inc. 's[bhw]i.birt5,[\$ta],inc.'

not rt5,ra						
	Alias of 'nor rt5,ra5,ra5'.					
neg rt5,ra	Alias of 'subri rt5,ra5,0'.					
br rb5	Depending on how it is assembled, it is translated into 'r5 rb5' or 'jr rb5'.					
b label	Branch to label depending on how it is assembled, it is translated into 'j8 label', 'j label', or "'la \$ta,label' 'br \$ta'".					
bral rb5	Alias of jral br5 depending on how it is assembled, it is translated into 'jral5 rb5' or 'jral rb5'.					
bal fname	Alias of jal fname depending on how it is assembled, it is translated into 'jal fname' or "'la \$ta,fname' 'bral \$ta'".					
call fname	Call function fname same as 'jal fname'.					
move rt5,r	a5 For 16-bit, this is 'mov55 rt5,ra5'. For no 16-bit, this is 'ori rt5,ra5,0'.					
move rt5,v	This is the same as 'l.w rt5,var'.					
move rt5,i	mm32 This is the same as 'li rt5,imm32'.					
pushm ra5,	rb5 Push contents of registers from ra5 to rb5 into stack.					
push ra5	Push content of register ra5 into stack. (same 'pushm ra5,ra5').					
push.d var						
	Push value of double-word variable var into stack.					
push.w var	Push value of word variable var into stack.					
push.h var	Push value of half-word variable var into stack.					
push.b var						
	Push value of byte variable var into stack.					
pusha var	r Push 32-bit address of variable var into stack.					
pushi imm3	22 Push 32-bit immediate value into stack.					
popm ra5,r	b5 Pop top of stack values into registers ra5 to rb5.					
pop rt5	Pop top of stack value into register. (same as 'popm rt5,rt5'.)					
pop.d var,	ra5 Pop value of double-word variable var from stack using register ra5 as 2nd scratch register. (1st is \$ta)					

	Pop value of word variable var from stack using register ra5.
pop.h var,	, <b>ra5</b> Pop value of half-word variable var from stack using register ra5.
pop.b var	,ra5 Pop value of byte variable var from stack using register ra5.

pop.w var,ra5

# 9.31 Nios II Dependent Features

# **9.31.1** Options

### -relax-section

Replace identified out-of-range branches with PC-relative jmp sequences when possible. The generated code sequences are suitable for use in position-independent code, but there is a practical limit on the extended branch range because of the length of the sequences. This option is the default.

### -relax-all

Replace branch instructions not determinable to be in range and all call instructions with jmp and callr sequences (respectively). This option generates absolute relocations against the target symbols and is not appropriate for position-independent code.

### -no-relax

Do not replace any branches or calls.

- -EB Generate big-endian output.
- -EL Generate little-endian output. This is the default.

### -march=architecture

This option specifies the target architecture. The assembler issues an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: r1, r2. The default is r1.

# 9.31.2 Syntax

# 9.31.2.1 Special Characters

'#' is the line comment character. ';' is the line separator character.

# 9.31.3 Nios II Machine Relocations

### %hiadj(expression)

Extract the upper 16 bits of expression and add one if the 15th bit is set.

The value of %hiadj(expression) is:

((expression >> 16) & 0xffff) + ((expression >> 15) & 0x01)

The %hiadj relocation is intended to be used with the addi, 1d or st instructions along with a %10, in order to load a 32-bit constant.

movhi r2, %hiadj(symbol)
addi r2, r2, %lo(symbol)

# %hi(expression)

Extract the upper 16 bits of expression.

### %lo(expression)

Extract the lower 16 bits of expression.

# %gprel(expression)

Subtract the value of the symbol \_gp from expression.

The intention of the **%gprel** relocation is to have a fast small area of memory which only takes a 16-bit immediate to access.

.section .sdata fastint: .int 123 .section .text ldw r4, %gprel(fastint)(gp)

```
%call(expression)
```

%call\_lo(expression)

```
%call_hiadj(expression)
%got(expression)
%got_lo(expression)
%got_hiadj(expression)
%gotoff(expression)
%gotoff_lo(expression)
%gotoff_hiadj(expression)
%tls_gd(expression)
```

%tls\_ie(expression)

```
%tls_le(expression)
```

%tls\_ldm(expression)

# %tls\_ldo(expression)

These relocations support the ABI for Linux Systems documented in the Nios II Processor Reference Handbook.

# 9.31.4 Nios II Machine Directives

```
.align expression [, expression]
           This is the generic .align directive, however this aligns to a power of two.
.half expression
           Create an aligned constant 2 bytes in size.
```

.word expression

Create an aligned constant 4 bytes in size.

.dword expression

Create an aligned constant 8 bytes in size.

.2byte expression

Create an unaligned constant 2 bytes in size.

.4byte expression

Create an unaligned constant 4 bytes in size.

.8byte expression

Create an unaligned constant 8 bytes in size.

### .16byte expression

Create an unaligned constant 16 bytes in size.

.set noat	Allows assembly	code to us	e at	register	without	warning.	Macro	or	relaxation
	expansions gener	ate warnin	gs.						

.set at Assembly code using at register generates warnings, and macro expansion and relaxation are enabled.

.set nobreak

Allows assembly code to use ba and bt registers without warning.

.set break

Turns warnings back on for using ba and bt registers.

.set norelax

Do not replace any branches or calls.

.set relaxsection Replace identified out-of-range branches with jmp sequences (default).

.set relaxsection Replace all branch and call instructions with jmp and callr sequences.

.set ... All other .set are the normal use.

# 9.31.5 Opcodes

as implements all the standard Nios II opcodes documented in the Nios II Processor Reference Handbook, including the assembler pseudo-instructions.

# 9.32 NS32K Dependent Features

# 9.32.1 Syntax

# 9.32.1.1 Special Characters

The presence of a '#' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

If Sequent compatibility has been configured into the assembler then the '|' character appearing as the first character on a line will also indicate the start of a line comment.

The ';' character can be used to separate statements on the same line.

# 9.33 OPENRISC Dependent Features

# 9.33.1 OpenRISC Syntax

The assembler syntax follows the OpenRISC 1000 Architecture Manual.

### 9.33.1.1 Special Characters

A '#' character appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

';' can be used instead of a newline to separate statements.

### 9.33.1.2 Register Names

The OpenRISC register file contains 32 general pupose registers.

- The 32 general purpose registers are referred to as 'rn'.
- The stack pointer register 'r1' can be referenced using the alias 'sp'.
- The frame pointer register 'r2' can be referenced using the alias 'fp'.
- The link register 'r9' can be referenced using the alias 'lr'.

Floating point operations use the same general purpose registers. The instructions lf.itof.s (single precision) and lf.itof.d (double precision) can be used to convert integer values to floating point. Likewise, instructions lf.ftoi.s (single precision) and lf.ftoi.d (double precision) can be used to convert floating point to integer.

OpenRISC also contains privileged special purpose registers (SPRs). The SPRs are accessed using the l.mfspr and l.mtspr instructions.

### 9.33.1.3 Relocations

ELF relocations are available as defined in the OpenRISC architecture specification.

R\_OR1K\_HI\_16\_IN\_INSN is obtained using 'hi' and R\_OR1K\_LO\_16\_IN\_INSN and R\_OR1K\_ SLO16 are obtained using 'lo'. For signed offsets R\_OR1K\_AHI16 is obtained from 'ha'. For example:

```
1.movhi r5, hi(symbol)
l.ori r5, r5, lo(symbol)
l.movhi r5, ha(symbol)
l.addi r5, r5, lo(symbol)
```

These "high" mnemonics extract bits 31:16 of their operand, and the "low" mnemonics extract bits 15:0 of their operand.

The PC relative relocation R\_OR1K\_GOTPC\_HI16 can be obtained by enclosing an operand inside of 'gotpchi'. Likewise, the R\_OR1K\_GOTPC\_LO16 relocation can be obtained using 'gotpclo'. These are mostly used when assembling PIC code. For example, the standard PIC sequence on OpenRISC to get the base of the global offset table, PC relative, into a register, can be performed as:

```
l.jal 0x8
l.movhi r17, gotpchi(_GLOBAL_OFFSET_TABLE_-4)
l.ori r17, r17, gotpclo(_GLOBAL_OFFSET_TABLE_+0)
```

l.add r17, r17, r9

Several relocations exist to allow the link editor to perform GOT data references. The  $R_OR1K_GOT16$  relocation can obtained by enclosing an operand inside of 'got'. For example, assuming the GOT base is in register r17.

1.lwz r19, got(a)(r17)
1.lwz r21, 0(r19)

Also, several relocations exist for local GOT references. The R\_OR1K\_GOTOFF\_AHI16 relocation can obtained by enclosing an operand inside of 'gotoffha'. Likewise, R\_OR1K\_GOTOFF\_LO16 and R\_OR1K\_GOTOFF\_SLO16 can be obtained by enclosing an operand inside of 'gotofflo'. For example, assuming the GOT base is in register r17:

```
l.movhi r19, gotoffha(symbol)
l.add r19, r19, r17
l.lwz r19, gotofflo(symbol)(r19)
```

The above PC relative relocations use a 1.jal (jump) instruction and reading of the link register to load the PC. OpenRISC also supports page offset PC relative locations without a jump instruction using the 1.adrp instruction. By default the 1.adrp instruction will create an R\_OR1K\_PCREL\_PG21 relocation. Likewise, BFD\_RELOC\_OR1K\_LO13 and BFD\_RELOC\_OR1K\_SLO13 can be obtained by enclosing an operand inside of 'po'. For example:

```
1.adrp r3, symbol
1.ori r4, r3, po(symbol)
1.lbz r5, po(symbol)(r3)
1.sb po(symbol)(r3), r6
```

Likewise the page offset relocations can be used with GOT references. The relocation R\_OR1K\_GOT\_PG21 can be obtained by enclosing an l.adrp immediate operand inside of 'got'. Likewise, R\_OR1K\_GOT\_LO13 can be obtained by enclosing an operand inside of 'gotpo'. For example to load the value of a GOT symbol into register 'r5' we can do:

```
l.adrp r17, got(_GLOBAL_OFFSET_TABLE_)
l.lwz r5, gotpo(symbol)(r17)
```

There are many relocations that can be requested for access to thread local storage variables. All of the OpenRISC TLS mnemonics are supported:

- R\_OR1K\_TLS\_GD\_HI16 is requested using 'tlsgdhi'.
- R\_OR1K\_TLS\_GD\_L016 is requested using 'tlsgdlo'.
- R\_OR1K\_TLS\_GD\_PG21 is requested using 'tldgd'.
- R\_OR1K\_TLS\_GD\_L013 is requested using 'tlsgdpo'.
- R\_OR1K\_TLS\_LDM\_HI16 is requested using 'tlsldmhi'.
- R\_OR1K\_TLS\_LDM\_LO16 is requested using 'tlsldmlo'.
- R\_OR1K\_TLS\_LDM\_PG21 is requested using 'tldldm'.
- R\_OR1K\_TLS\_LDM\_LO13 is requested using 'tlsldmpo'.
- R\_OR1K\_TLS\_LDO\_HI16 is requested using 'dtpoffhi'.
- R\_OR1K\_TLS\_LDO\_L016 is requested using 'dtpofflo'.
- R\_OR1K\_TLS\_IE\_HI16 is requested using 'gottpoffhi'.
- R\_OR1K\_TLS\_IE\_AHI16 is requested using 'gottpoffha'.

- R\_OR1K\_TLS\_IE\_LO16 is requested using 'gottpofflo'.
- R\_OR1K\_TLS\_IE\_PG21 is requested using 'gottp'.
- R\_OR1K\_TLS\_IE\_LO13 is requested using 'gottppo'.
- R\_OR1K\_TLS\_LE\_HI16 is requested using 'tpoffhi'.
- R\_OR1K\_TLS\_LE\_AHI16 is requested using 'tpoffha'.
- R\_OR1K\_TLS\_LE\_LO16 is requested using 'tpofflo'.
- R\_OR1K\_TLS\_LE\_SL016 also is requested using 'tpofflo' depending on the instruction format.

Here are some example TLS model sequences.

First, General Dynamic:

```
l.movhi r17, tlsgdhi(symbol)
l.ori r17, r17, tlsgdlo(symbol)
l.add r17, r17, r16
l.or r3, r17, r17
l.jal plt(__tls_get_addr)
l.nop
```

Initial Exec:

```
1.movhi r17, gottpoffhi(symbol)
1.add r17, r17, r16
1.lwz r17, gottpofflo(symbol)(r17)
1.add r17, r17, r10
1.lbs r17, 0(r17)
```

And finally, Local Exec:

```
l.movhi r17, tpoffha(symbol)
l.add r17, r17, r10
l.addi r17, r17, tpofflo(symbol)
l.lbs r17, 0(r17)
```

# 9.33.2 Floating Point

OpenRISC uses IEEE floating-point numbers.

## 9.33.3 OpenRISC Machine Directives

The OpenRISC version of as supports the following additional machine directives:

- .align This must be followed by the desired alignment in bytes.
- .word On the OpenRISC, the .word directive produces a 32 bit value.
- .nodelay On the OpenRISC, the .nodelay directive sets a flag in elf binaries indicating that the binary is generated catering for no delay slots.
- .proc This directive is ignored. Any text following it on the same line is also ignored.
- .endproc This directive is ignored. Any text following it on the same line is also ignored.

# 9.33.4 Opcodes

For detailed information on the OpenRISC machine instruction set, see http://www.openrisc.io/architecture/.

as implements all the standard OpenRISC opcodes.

# 9.34 PDP-11 Dependent Features

### **9.34.1** Options

The PDP-11 version of **as** has a rich set of machine dependent options.

### 9.34.1.1 Code Generation Options

```
-mpic | -mno-pic
```

Generate position-independent (or position-dependent) code.

The default is to generate position-independent code.

### 9.34.1.2 Instruction Set Extension Options

These options enables or disables the use of extensions over the base line instruction set as introduced by the first PDP-11 CPU: the KA11. Most options come in two variants: a -mextension that enables extension, and a -mno-extension that disables extension.

The default is to enable all extensions.

```
-mall | -mall-extensions
```

Enable all instruction set extensions.

-mno-extensions

Disable all instruction set extensions.

-mcis | -mno-cis

Enable (or disable) the use of the commercial instruction set, which consists of these instructions: ADDNI, ADDN, ADDPI, ADDP, ASHNI, ASHN, ASHPI, ASHP, CMPCI, CMPC, CMPNI, CMPN, CMPPI, CMPP, CVTLNI, CVTLN, CVTLPI, CVTLP, CVTNLI, CVTNL, CVTNPI, CVTNPI, CVTPLI, CVTPL, CVTPNI, CVTPN, DIVPI, DIVP, L2DR, L3DR, L0CCI, L0CC, MATCI, MATC, MOVCI, MOVC, MOVRCI, MOVRC, MOVTCI, MOVTC, MULPI, MULP, SCANCI, SCANC, SKPCI, SKPC, SPANCI, SPANC, SUBNI, SUBN, SUBPI, and SUBP.

-mcsm | -mno-csm

Enable (or disable) the use of the CSM instruction.

-meis | -mno-eis

Enable (or disable) the use of the extended instruction set, which consists of these instructions: ASHC, ASH, DIV, MARK, MUL, RTT, SOB SXT, and XOR.

```
-mfis | -mkev11
```

-mno-fis | -mno-kev11

Enable (or disable) the use of the KEV11 floating-point instructions: FADD, FDIV, FMUL, and FSUB.

-mfpp | -mfpu | -mfp-11

```
-mno-fpp | -mno-fpu | -mno-fp-11
```

Enable (or disable) the use of FP-11 floating-point instructions: ABSF, ADDF, CFCC, CLRF, CMPF, DIVF, LDCFF, LDCIF, LDEXP, LDF, LDFPS, MODF, MULF, NEGF, SETD, SETF, SETI, SETL, STCFF, STCFI, STEXP, STF, STFPS, STST, SUBF, and TSTF.

```
-mlimited-eis | -mno-limited-eis
```

Enable (or disable) the use of the limited extended instruction set: MARK, RTT, SOB, SXT, and XOR.

The -mno-limited-eis options also implies -mno-eis.

#### -mmfpt | -mno-mfpt

Enable (or disable) the use of the MFPT instruction.

### -mmultiproc | -mno-multiproc

Enable (or disable) the use of multiprocessor instructions: TSTSET and WRTLCK.

-mmxps | -mno-mxps

Enable (or disable) the use of the MFPS and MTPS instructions.

-mspl | -mno-spl

Enable (or disable) the use of the SPL instruction.

Enable (or disable) the use of the microcode instructions: LDUB, MED, and XFC.

## 9.34.1.3 CPU Model Options

These options enable the instruction set extensions supported by a particular CPU, and disables all other extensions.

-mka11	KA11 CPU. Base line instruction set only.
-mkb11	KB11 CPU. Enable extended instruction set and SPL.
-mkd11a	KD11-A CPU. Enable limited extended instruction set.
-mkd11b	KD11-B CPU. Base line instruction set only.
-mkd11d	KD11-D CPU. Base line instruction set only.
-mkd11e	KD11-E CPU. Enable extended instruction set, MFPS, and MTPS.
-mkd11f   ·	-mkd11h   -mkd11q KD11-F, KD11-H, or KD11-Q CPU. Enable limited extended instruction set, MFPS, and MTPS.
-mkd11k	KD11-K CPU. Enable extended instruction set, LDUB, MED, MFPS, MFPT, MTPS, and XFC.
-mkd11z	KD11-Z CPU. Enable extended instruction set, $\texttt{CSM}, \texttt{MFPS}, \texttt{MFPT}, \texttt{MTPS}, \texttt{and} \texttt{SPL}.$
-mf11	F11 CPU. Enable extended instruction set, MFPS, MFPT, and MTPS.
-mj11	J11 CPU. Enable extended instruction set, CSM, MFPS, MFPT, MTPS, SPL, TSTSET, and WRTLCK.
-mt11	T11 CPU. Enable limited extended instruction set, MFPS, and MTPS.

# 9.34.1.4 Machine Model Options

These options enable the instruction set extensions supported by a particular machine model, and disables all other extensions.

-m11/03 Same as -mkd11f.

-m11/04 Same as -mkd11d.	
-m11/05   -m11/10 Same as -mkd11b.	
-m11/15   -m11/20 Same as -mka11.	
-m11/21 Same as -mt11.	
-m11/23   -m11/24 Same as -mf11.	
-m11/34 Same as -mkd11e.	
-m11/34a Ame as -mkd11e -mfpp.	
-m11/35   -m11/40 Same as -mkd11a.	
-m11/44 Same as -mkd11z.	
-m11/45   -m11/50   -m11/55   -m11/70 Same as -mkb11.	
-m11/53   -m11/73   -m11/83   -m11/84   -m11/93   -m11/94 Same as -mj11.	F
-m11/60 Same as -mkd11k.	

### 9.34.2 Assembler Directives

The PDP-11 version of as has a few machine dependent assembler directives.

.bss Switch to the bss section.

.even Align the location counter to an even number.

# 9.34.3 PDP-11 Assembly Language Syntax

as supports both DEC syntax and BSD syntax. The only difference is that in DEC syntax, a **#** character is used to denote an immediate constants, while in BSD syntax the character for this purpose is **\$**.

general-purpose registers are named r0 through r7. Mnemonic alternatives for r6 and r7 are sp and pc, respectively.

Floating-point registers are named ac0 through ac3, or alternatively fr0 through fr3.

Comments are started with a **#** or a / character, and extend to the end of the line. (FIXME: clash with immediates?)

Multiple statements on the same line can be separated by the ';' character.

### 9.34.4 Instruction Naming

Some instructions have alternative names.

BCC	BHIS
BCS	BLO

L2DR	L2D
L3DR	L3D
SYS	TRAP

# 9.34.5 Synthetic Instructions

The JBR and JCC synthetic instructions are not supported yet.

# 9.35 picoJava Dependent Features

### 9.35.1 Options

as has two additional command-line options for the picoJava architecture.

-ml This option selects little endian data output.

-mb This option selects big endian data output.

# 9.35.2 PJ Syntax

### 9.35.2.1 Special Characters

The presence of a '!' or '/' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

# 9.36 PowerPC Dependent Features

# 9.36.1 Options

The PowerPC chip family includes several successive levels, using the same core instruction set, but including a few additional instructions at each level. There are exceptions to this however. For details on what instructions each variant supports, please see the chip's architecture reference manual.

The following table lists all available PowerPC options.

-a32	Generate ELF32 or XCOFF32.		
-a64	Generate ELF64 or XCOFF64.		
-K PIC	Set EF_PPC_RELOCATABLE_LIB in ELF flags.		
-mpwrx   -	mpwr2 Generate code for POWER/2 (RIOS2).		
-mpwr	Generate code for POWER (RIOS1)		
-m601	Generate code for PowerPC 601.		
-mppc, -mp	Generate code for PowerPC 603/604.		
-m403, -m4	105		
	Generate code for PowerPC $403/405$ .		
-m440	Generate code for PowerPC 440. BookE and some 405 instructions.		
-m464	Generate code for PowerPC 464.		
-m476	Generate code for PowerPC 476.		
-m7400, -m	17410, -m7450, -m7455 Generate code for PowerPC 7400/7410/7450/7455.		
-m750cl, -	-mgekko, -mbroadway Generate code for PowerPC 750CL/Gekko/Broadway.		
-m821, -m8	350, -m860		
	Generate code for PowerPC 821/850/860.		
-mppc64,-	-m620 Generate code for PowerPC 620/625/630.		
-me200z2,	-me200z4 Generate code for e200 variants, e200z2 with LSP, e200z4 with SPE.		
-me300	Generate code for PowerPC e300 family.		
-me500, -m	ne500x2 Generate code for Motorola e500 core complex.		
-me500mc	Generate code for Freescale e500mc core complex.		
-me500mc64			
	Generate code for Freescale e500mc64 core complex.		

-me5500	Generate code for Freescale e5500 core complex.	
-me6500	Generate code for Freescale e6500 core complex.	
-mlsp	Enable LSP instructions. (Disables SPE and SPE2.)	
-mspe	Generate code for Motorola SPE instructions. (Disables LSP.)	
-mspe2	Generate code for Freescale SPE2 instructions. (Disables LSP.)	
-mtitan	Generate code for AppliedMicro Titan core complex.	
-mppc64br	idge	
	Generate code for PowerPC 64, including bridge insns.	
-mbooke	Generate code for 32-bit BookE.	
-ma2	Generate code for A2 architecture.	
-maltivec		
	Generate code for processors with AltiVec instructions.	
-mvle	Generate code for Freescale PowerPC VLE instructions.	
-mvsx	Generate code for processors with Vector-Scalar (VSX) instructions.	
-mhtm	Generate code for processors with Hardware Transactional Memory instructions.	
-mpower4,	-mpwr4 Generate code for Power4 architecture.	
-mpower5,	-mpwr5, -mpwr5x Generate code for Power5 architecture.	
-mpower6,	-mpwr6 Generate code for Power6 architecture.	
-mpower7,	-mpwr7 Generate code for Power7 architecture.	
-mpower8,	-mpwr8 Generate code for Power8 architecture.	
-mpower9,	-mpwr9 Generate code for Power9 architecture.	
-mpower10	, -mpwr10 Generate code for Power10 architecture.	
-mfuture	Generate code for 'future' architecture.	
-mcell		
-mcell	Generate code for Cell Broadband Engine architecture.	
-mcom	Generate code Power/PowerPC common instructions.	
-many	Generate code for any architecture (PWR/PWRX/PPC).	
-mregnames		
mr ogname	Allow symbolic names for registers.	

-mno-regna	ames
	Do not allow symbolic names for registers.
-mrelocata	
	Support for GCC's -mrelocatable option.
-mrelocata	able-lib Support for GCC's -mrelocatable-lib option.
-memb	Set PPC_EMB bit in ELF flags.
-mlittle,	-mlittle-endian, -le Generate code for a little endian machine.
-mbig, -mb	<b>ig-endian, -be</b> Generate code for a big endian machine.
-msolaris	
	Generate code for Solaris.
-mno-solar	ris
	Do not generate code for Solaris.
-nops=cour	lt

If an alignment directive inserts more than *count* nops, put a branch at the beginning to skip execution of the nops.

# 9.36.2 PowerPC Assembler Directives

A number of assembler directives are available for PowerPC. The following table is far from complete.

```
.machine "string"
```

This directive allows you to change the machine for which code is generated. "string" may be any of the -m cpu selection options (without the -m) enclosed in double quotes, "push", or "pop". .machine "push" saves the currently selected cpu, which may be restored with .machine "pop".

### 9.36.3 PowerPC Syntax

#### 9.36.3.1 Special Characters

The presence of a '#' on a line indicates the start of a comment that extends to the end of the current line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

If the assembler has been configured for the ppc-\*-solaris\* target then the '!' character also acts as a line comment character. This can be disabled via the -mno-solaris command-line option.

The ';' character can be used to separate statements on the same line.

# 9.37 PRU Dependent Features

# 9.37.1 Options

#### -mlink-relax

Assume that LD would optimize LDI32 instructions by checking the upper 16 bits of the *expression*. If they are all zeros, then LD would shorten the LDI32 instruction to a single LDI. In such case **as** will output DIFF relocations for diff expressions.

#### -mno-link-relax

Assume that LD would not optimize LDI32 instructions. As a consequence, DIFF relocations will not be emitted.

#### -mno-warn-regname-label

Do not warn if a label name matches a register name. Usually assembler programmers will want this warning to be emitted. C compilers may want to turn this off.

## 9.37.2 Syntax

### 9.37.2.1 Special Characters

'#' and ';' are the line comment characters.

## 9.37.3 PRU Machine Relocations

#### %pmem(expression)

Convert *expression* from byte-address to a word-address. In other words, shift right by two.

#### %label(expression)

Mark the given operand as a label. This is useful if you need to jump to a label that matches a register name.

r1:

jmp r1 ; Will jump to register R1 jmp %label(r1) ; Will jump to label r1

# 9.37.4 PRU Machine Directives

.align expression [, expression]

This is the generic .align directive, however this aligns to a power of two.

```
.word expression
```

Create an aligned constant 4 bytes in size.

#### .dword expression

Create an aligned constant 8 bytes in size.

#### .2byte expression

Create an unaligned constant 2 bytes in size.

#### .4byte expression

Create an unaligned constant 4 bytes in size.

```
.8byte expression
```

Create an unaligned constant 8 bytes in size.

```
.16byte expression
```

Create an unaligned constant 16 bytes in size.

```
.set no_warn_regname_label
```

Do not output warnings when a label name matches a register name. Equivalent to passing the **-mno-warn-regname-label** command-line option.

# 9.37.5 Opcodes

**as** implements all the standard PRU core V3 opcodes in the original pasm assembler. Older cores are not supported by **as**.

GAS also implements the LDI32 pseudo instruction for loading a 32-bit immediate value into a register.

ldi32 sp, \_\_stack\_top ldi32 r14, 0x12345678

# 9.38 RISC-V Dependent Features

# 9.38.1 RISC-V Options

The following table lists all available RISC-V specific options.

-fpic

-fPIC Generate position-inc	dependent code
-----------------------------	----------------

-fno-pic Don't generate position-independent code (default)

#### -march=ISA

Select the base isa, as specified by ISA. For example -march=rv32ima. If this option and the architecture attributes aren't set, then assembler will check the default configure setting -with-arch=ISA.

#### -misa-spec=ISAspec

Select the default is spec version. If the version of ISA isn't set by -march, then assembler helps to set the version according to the default chosen spec. If this option isn't set, then assembler will check the default configure setting –with-isa-spec=ISAspec.

#### -mpriv-spec=PRIVspec

Select the privileged spec version. We can decide whether the CSR is valid or not according to the chosen spec. If this option and the privilege attributes aren't set, then assembler will check the default configure setting –with-privspec=PRIVspec.

#### -mabi=ABI

Selects the ABI, which is either "ilp32" or "lp64", optionally followed by "f", "d", or "q" to indicate single-precision, double-precision, or quad-precision floating-point calling convention, or none to indicate the soft-float calling convention. Also, "ilp32" can optionally be followed by "e" to indicate the RVE ABI, which is always soft-float.

-mrelax Take advantage of linker relaxations to reduce the number of instructions required to materialize symbol addresses. (default)

#### -mno-relax

Don't do linker relaxations.

#### -march-attr

Generate the default contents for the riscv elf attribute section if the .attribute directives are not set. This section is used to record the information that a linker or runtime loader needs to check compatibility. This information includes ISA string, stack alignment requirement, unaligned memory accesses, and the major, minor and revision version of privileged specification.

#### -mno-arch-attr

Don't generate the default riscv elf attribute section if the .attribute directives are not set.

#### -mcsr-check

Enable the CSR checking for the ISA-dependent CRS and the read-only CSR. The ISA-dependent CSR are only valid when the specific ISA is set. The read-only CSR can not be written by the CSR instructions.

#### -mno-csr-check

Don't do CSR checking.

### -mlittle-endian

Generate code for a little endian machine.

#### -mbig-endian

Generate code for a big endian machine.

# 9.38.2 RISC-V Directives

The following table lists all available RISC-V specific directives.

```
.align size-log-2
```

Align to the given boundary, with the size given as log2 the number of bytes to align to.

#### .half value

- .word value
- .dword value

Emits a half-word, word, or double-word value at the current position.

.dtprelword value

#### .dtpreldword value

Emits a DTP-relative word (or double-word) at the current position. This is meant to be used by the compiler in shared libraries for DWARF debug info for thread local variables.

.bss Sets the current section to the BSS section.

- .uleb128 value
- .sleb128 value

Emits a signed or unsigned LEB128 value at the current position. This only accepts constant expressions, because symbol addresses can change with relaxation, and we don't support relocations to modify LEB128 values at link time.

#### .option argument

Modifies RISC-V specific assembler options inline with the assembly code. This is used when particular instruction sequences must be assembled with a specific set of options. For example, since we relax addressing sequences to shorter GP-relative sequences when possible the initial load of GP must not be relaxed and should be emitted as something like

.option push .option norelax la gp, \_\_global\_pointer\$ .option pop

in order to produce after linker relaxation the expected auipc gp, %pcrel\_hi(\_\_global\_pointer\$) addi gp, gp, %pcrel\_lo(\_\_global\_pointer\$)

instead of just

addi gp, gp, 0

It's not expected that options are changed in this manner during regular use, but there are a handful of esoteric cases like the one above where users need to disable particular features of the assembler for particular code sequences. The complete list of option arguments is shown below:

push	
рор	Pushes or pops the current option stack. These should be used whenever changing an option in line with assembly code in order to ensure the user's command-line options are respected for the bulk of the file being assembled.
rvc	
norvc	Enables or disables the generation of compressed instructions. In- structions are opportunistically compressed by the RISC-V assem- bler when possible, but sometimes this behavior is not desirable, especially when handling alignments.
pic	
nopic	Enables or disables position-independent code generation. Unless you really know what you're doing, this should only be at the top of a file.
relax	
norelax	Enables or disables relaxation. The RISC-V assembler and linker opportunistically relax some code sequences, but sometimes this behavior is not desirable.
csr-check	
no-csr-che	eck
	Enables or disables the CSR checking.
	cension[version] [,,+extension_n[version_n]] cension [,,-extension_n]
	Enables or disables the extensions for specific code region. For example, '.option arch, +m2p0' means add m extension with ver- sion 2.0, and '.option arch, -f, -d' means remove extensions, f and d, from the architecture string. Note that, '.option arch, +c, -c' have the same behavior as '.option rvc, norvc'. However, they are also undesirable sometimes. Besides, '.option arch, -i' is illegal, since we cannot remove the base i extension anytime. If you want to reset the whole ISA string, you can also use '.option arch, =rv32imac' to overwrite the previous settings.

```
.insn type, operand [,...,operand_n]
.insn insn_length, value
.insn value
```

This directive permits the numeric representation of an instructions and makes the assembler insert the operands according to one of the instruction formats for '.insn' (Section 9.38.5 [RISC-V-Formats], page 290). For example, the instruction 'add a0, a1, a2' could be written as '.insn r 0x33, 0, 0, a0, a1, a2'. But in fact, the instruction formats are difficult to use for some users, so most of them are using '.word' to encode the instruction directly, rather than using '.insn'. It is fine for now, but will be wrong when the mapping symbols are supported, since '.word' will not be shown as an instruction, it should be shown as data. Therefore, we also support two more formats of the '.insn', the instruction 'add a0, a1, a2' could also be written as '.insn 0x4, 0xc58533' or '.insn 0xc58533'. When the *insn\_length* is set, then assembler will check if the *value* is a valid *insn\_length* bytes instruction.

#### .attribute tag, value

Set the object attribute tag to value.

The tag is either an attribute number, or one of the following: Tag\_RISCV\_ arch, Tag\_RISCV\_stack\_align, Tag\_RISCV\_unaligned\_access, Tag\_RISCV\_ priv\_spec, Tag\_RISCV\_priv\_spec\_minor, Tag\_RISCV\_priv\_spec\_revision.

### 9.38.3 RISC-V Assembler Modifiers

The RISC-V assembler supports following modifiers for relocatable addresses used in RISC-V instruction operands. However, we also support some pseudo instructions that are easier to use than these modifiers.

%lo(symbol)

The low 12 bits of absolute address for symbol.

```
%hi(symbol)
```

The high 20 bits of absolute address for *symbol*. This is usually used with the %lo modifier to represent a 32-bit absolute address.

lui a0, %hi(symbol) // R\_RISCV\_HI20
addi a0, a0, %lo(symbol) // R\_RISCV\_L012\_I
lui a0, %hi(symbol) // R\_RISCV\_HI20
load/store a0, %lo(symbol)(a0) // R\_RISCV\_L012\_I/S

#### %pcrel\_lo(label)

The low 12 bits of relative address between pc and symbol. The symbol is related to the high part instruction which is marked by *label*.

#### %pcrel\_hi(symbol)

The high 20 bits of relative address between pc and symbol. This is usually used with the %pcrel\_lo modifier to represent a +/-2GB pc-relative range.

label: auipc a0, %pcrel\_hi(symbol) // R\_RISCV\_PCREL\_HI20 addi a0, a0, %pcrel\_lo(label) // R\_RISCV\_PCREL\_L012\_I auipc a0, %pcrel\_hi(symbol) // R\_RISCV\_PCREL\_HI20 load/store a0, %pcrel\_lo(label)(a0) // R\_RISCV\_PCREL\_L012\_I/S Or you can use the pseudo lla/lw/sw/... instruction to do this. lla a0, symbol

#### %got\_pcrel\_hi(symbol)

The high 20 bits of relative address between pc and the GOT entry of symbol. This is usually used with the %pcrel\_lo modifier to access the GOT entry.

```
label:
auipc a0, %got_pcrel_hi(symbol) // R_RISCV_GOT_HI20
addi a0, a0, %pcrel_lo(label) // R_RISCV_PCREL_L012_I
label:
auipc a0, %got_pcrel_hi(symbol) // R_RISCV_GOT_HI20
load/store a0, %pcrel_lo(label)(a0) // R_RISCV_PCREL_L012_I/S
```

Also, the pseudo la instruction with PIC has similar behavior.

#### %tprel\_add(symbol)

This is used purely to associate the R\_RISCV\_TPREL\_ADD relocation for TLS relaxation. This one is only valid as the fourth operand to the normally 3 operand add instruction.

#### %tprel\_lo(symbol)

The low 12 bits of relative address between tp and symbol.

#### %tprel\_hi(symbol)

The high 20 bits of relative address between tp and symbol. This is usually used with the %tprel\_lo and %tprel\_add modifiers to access the thread local variable symbol in TLS Local Exec.

```
luia5, %tprel_hi(symbol)// R_RISCV_TPREL_HI20adda5, a5, tp, %tprel_add(symbol)// R_RISCV_TPREL_ADDload/store t0, %tprel_lo(symbol)(a5)// R_RISCV_TPREL_L012_I/S
```

#### %tls\_ie\_pcrel\_hi(symbol)

The high 20 bits of relative address between pc and GOT entry. It is usually used with the %pcrel\_lo modifier to access the thread local variable *symbol* in TLS Initial Exec.

```
la.tls.ie a5, symbol
add a5, a5, tp
load/store t0, 0(a5)
```

The pseudo la.tls.ie instruction can be expended to

```
label:
auipc a5, %tls_ie_pcrel_hi(symbol) // R_RISCV_TLS_GOT_HI20
load a5, %pcrel_lo(label)(a5) // R_RISCV_PCREL_L012_I
```

#### %tls\_gd\_pcrel\_hi(symbol)

The high 20 bits of relative address between pc and GOT entry. It is usually used with the %pcrel\_lo modifier to access the thread local variable *symbol* in TLS Global Dynamic.

```
la.tls.gd a0, symbol
call __tls_get_addr@plt
mv a5, a0
load/store t0, 0(a5)
```

The pseudo la.tls.gd instruction can be expended to

```
label:
auipc a0, %tls_gd_pcrel_hi(symbol) // R_RISCV_TLS_GD_HI20
addi a0, a0, %pcrel_lo(label) // R_RISCV_PCREL_L012_I
```

# 9.38.4 RISC-V Floating Point

The RISC-V architecture uses IEEE floating-point numbers.

The RISC-V Zfa extension includes a load-immediate instruction for floating-point registers, which allows specifying the immediate (from a pool of 32 predefined values defined in the specification) as operand. E.g. to load the value 0.0625 as single-precision FP value into the FP register ft1 one of the following instructions can be used:

fli.s ft1,  $0.0625 \ \# \ dec \ floating-point \ literal fli.s \ ft1, \ 0x1p-4 \ \# \ hex \ floating-point \ literal fli.s \ ft1, \ 0x0.8p-3 \ fli.s \ ft1, \ 0x1.0p-4 \ fli.s \ ft1, \ 0x2p-5 \ fli.s \ ft1, \ 0x4p-6 \ ...$ 

As can be seen, many valid ways exist to express a floating-point value. This is realized by parsing the value operand using strtof() and comparing the parsed value against built-in float-constants that are written as hex floating-point literals.

This approach works on all machines that use IEEE 754. However, there is a chance that this fails on other machines with the following error message:

Error: improper fli value operand Error: illegal operands 'fli.s ft1,0.0625

The error indicates that parsing '0x1p-4' and '0.0625' to single-precision floating point numbers will not result in two equal values on that machine.

If you encounter this problem, then please report it.

## 9.38.5 RISC-V Instruction Formats

The RISC-V Instruction Set Manual Volume I: User-Level ISA lists 15 instruction formats where some of the formats have multiple variants. For the '.insn' pseudo directive the assembler recognizes some of the formats. Typically, the most general variant of the instruction format is used by the '.insn' directive.

The following table lists the abbreviations used in the table of instruction formats:

opcode	Unsigned immediate or op- code name for 7-bits opcode.
opcode2	Unsigned immediate or op- code name for 2-bits opcode.
func7	Unsigned immediate for 7- bits function code.
func6	Unsigned immediate for 6- bits function code.
func4	Unsigned immediate for 4- bits function code.

func3	Unsigned immediate for 3- bits function code.
func2	Unsigned immediate for 2- bits function code.
rd	Destination register number for operand x, can be GPR or FPR.
rd'	Destination register number for operand x, only accept s0-s1, a0-a5, fs0- fs1 and fa0-fa5.
rs1	First source register num- ber for operand x, can be GPR or FPR.
rs1'	First source register num- ber for operand x, only accept s0-s1, a0-a5, fs0- fs1 and fa0-fa5.
rs2	Second source register num- ber for operand x, can be GPR or FPR.
rs2'	Second source register num- ber for operand x, only accept s0-s1, a0-a5, fs0- fs1 and fa0-fa5.
simm12	Sign-extended 12-bit immedi- ate for operand x.
simm20	Sign-extended 20-bit immedi- ate for operand x.
simm6	Sign-extended 6-bit immedi- ate for operand x.
uimm5	Unsigned 5-bit immediate for operand x.
uimm6	Unsigned 6-bit immediate for operand x.
uimm8	Unsigned 8-bit immediate for operand x.
symbol	Symbol or lable reference for operand x.

The following table lists all available opcode name:

	$\sim$	1	٦
1		ì	J

C1	
C2	Opcode space for compressed instructions.
LOAD	Opcode space for load instructions.
LOAD_FP	Opcode space for floating-point load instructions.
STORE	Opcode space for store instructions.
STORE_FP	Opcode space for floating-point store instructions.
AUIPC	Opcode space for aujpc instruction.
LUI	Opcode space for lui instruction.
BRANCH	Opcode space for branch instructions.
JAL	Opcode space for jal instruction.
JALR	Opcode space for jalr instruction.
OP	Opcode space for ALU instructions.
OP_32	Opcode space for 32-bits ALU instructions.
OP_IMM	Opcode space for ALU with immediate instructions.
OP_IMM_32	
	Opcode space for 32-bits ALU with immediate instructions.
OP_FP	Opcode space for floating-point operation instructions.
MADD	Opcode space for madd instruction.
MSUB	Opcode space for msub instruction.
NMADD	Opcode space for nmadd instruction.
NMSUB	Opcode space for msub instruction.
AMO	Opcode space for atomic memory operation instructions.
MISC_MEM	Opcode space for misc instructions.
SYSTEM	Opcode space for system instructions.
CUSTOM_0	
CUSTOM_1	
CUSTOM_2	

CUSTOM\_3 Opcode space for customize instructions.

An instruction is two or four bytes in length and must be aligned on a 2 byte boundary. The first two bits of the instruction specify the length of the instruction, 00, 01 and 10 indicates a two byte instruction, 11 indicates a four byte instruction.

The following table lists the RISC-V instruction formats that are available with the '.insn' pseudo directive:

R type: .insn r opcode6, func3, func7, rd, rs1, rs2 +----+ | func7 | rs2 | rs1 | func3 | rd | opcode6 | +----+ 12 7 25 20 15 0 31 R type with 4 register operands: .insn r opcode6, func3, func2, rd, rs1, rs2, rs3 R4 type: .insn r4 opcode6, func3, func2, rd, rs1, rs2, rs3 +----+ | rs3 | func2 | rs2 | rs1 | func3 | rd | opcode6 | +----+ 25 20 15 12 7 31 27 0 I type: .insn i opcode6, func3, rd, rs1, simm12 I type: .insn i opcode6, func3, rd, simm12(rs1) +----+ | simm12[11:0] | rs1 | func3 | rd | opcode6 | +----+ 20 15 12 7 0 31 S type: .insn s opcode6, func3, rs2, simm12(rs1) +----+ | simm12[11:5] | rs2 | rs1 | func3 | simm12[4:0] | opcode6 | +----+ 7 20 15 12 0 31 25 B type: .insn s opcode6, func3, rs1, rs2, symbol SB type: .insn sb opcode6, func3, rs1, rs2, symbol +----+ | simm12[12|10:5] | rs2 | rs1 | func3 | simm12[4:1|11] | opcode6 | +----+ 15 12 7 31 25 20 0 U type: .insn u opcode6, rd, simm20 +----+ | simm20[20|10:1|11|19:12] | rd | opcode6 | +----+ 31 12 7 0 J type: .insn j opcode6, rd, symbol UJ type: .insn uj opcode6, rd, symbol | simm20[20] | simm20[10:1] | simm20[11] | simm20[19:12] | rd | opcode6 | 20 31 30 21 12 7 0 CR type: .insn cr opcode2, func4, rd, rs2 +----+

| func4 | rd/rs1 | rs2 | opcode2 | +----+ 15 12 7 2 0 CI type: .insn ci opcode2, func3, rd, simm6 | func3 | simm6[5] | rd/rs1 | simm6[4:0] | opcode2 | +----+ 2 13 12 7 0 15 CIW type: .insn ciw opcode2, func3, rd', uimm8 +----+ | func3 | uimm8[7:0] | rd' | opcode2 | +----+ 15 5 2 0 13 CSS type: .insn css opcode2, func3, rd, uimm6 +----+ | func3 | uimm6[5:0] | rd | opcode2 | +----+ 7 2 15 13 0 CL type: .insn cl opcode2, func3, rd', uimm5(rs1') | func3 | uimm5[4:2] | rs1' | uimm5[1:0] | rd' | opcode2 | +----+ 10 7 5 2 0 15 13 CS type: .insn cs opcode2, func3, rs2', uimm5(rs1') +----+ | func3 | uimm5[4:2] | rs1' | uimm5[1:0] | rs2' | opcode2 | +----+ 10 7 2 15 13 5 0 CA type: .insn ca opcode2, func6, func2, rd', rs2' | func6 | rd'/rs1' | func2 | rs2' | opcode2 | +----+ 7 5 2 15 10 0 CB type: .insn cb opcode2, func3, rs1', symbol +----+ | func3 | simm8[8|4:3] | rs1' | simm8[7:6|2:1|5] | opcode2 | 15 13 10 7 2 0 CJ type: .insn cj opcode2, symbol +----+ | func3 | simm11[11|4|9:8|10|6|7|3:1|5] | opcode2 | +----+ 2 0 15 13

For the complete list of all instruction format variants see The RISC-V Instruction Set Manual Volume I: User-Level ISA.

### 9.38.6 RISC-V Object Attribute

RISC-V attributes have a string value if the tag number is odd and an integer value if the tag number is even.

 $Tag_RISCV_stack_align$  (4)

Tag\_RISCV\_strict\_align records the N-byte stack alignment for this object. The default value is 16 for RV32I or RV64I, and 4 for RV32E.

The smallest value will be used if object files with different Tag\_RISCV\_stack\_align values are merged.

Tag\_RISCV\_arch (5)

Tag\_RISCV\_arch contains a string for the target architecture taken from the option -march. Different architectures will be integrated into a superset when object files are merged.

Note that the version information of the target architecture must be presented explicitly in the attribute and abbreviations must be expanded. The version information, if not given by -march, must be in accordance with the default specified by the tool. For example, the architecture RV32I has to be recorded in the attribute as RV32I2PO in which 2PO stands for the default version of its base ISA. On the other hand, the architecture RV32G has to be presented as RV32I2PO\_M2PO\_A2PO\_F2PO\_D2PO in which the abbreviation G is expanded to the IMAFD combination with default versions of the standard extensions.

#### $Tag_RISCV_unaligned_access$ (6)

Tag\_RISCV\_unaligned\_access is 0 for files that do not allow any unaligned memory accesses, and 1 for files that do allow unaligned memory accesses.

 $Tag_RISCV_priv_spec$  (8)

 $Tag_RISCV_priv_spec_minor$  (10)

 $Tag_RISCV_priv_spec_revision$  (12)

Tag\_RISCV\_priv\_spec contains the major/minor/revision version information of the privileged specification. It will report errors if object files of different privileged specification versions are merged.

### 9.38.7 RISC-V Custom (Vendor-Defined) Extensions

The following table lists the custom (vendor-defined) RISC-V extensions supported and provides the location of their publicly-released documentation:

#### XTheadBa

The XTheadBa extension provides instructions for address calculations.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### **XTheadBb**

The XTheadBb extension provides instructions for basic bit-manipulation

```
It is documented in https://github.com/T-head-Semi/
thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2.
0.0.pdf.
```

### **XTheadBs**

The XTheadBs extension provides single-bit instructions.

```
It is documented in https://github.com/T-head-Semi/
thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2.
0.0.pdf.
```

#### **XTheadCmo**

The XTheadCmo extension provides instructions for cache management.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### **XTheadCondMov**

The XTheadCondMov extension provides instructions for conditional moves.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### XTheadFMemIdx

The XTheadFMemIdx extension provides floating-point memory operations.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### **XTheadFmv**

The XTheadFmv extension provides access to the upper 32 bits of a doubleprecision floating point register.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.1.0/xthead-2022-11-07-2. 1.0.pdf.

#### **XTheadInt**

The XTheadInt extension provides access to ISR stack management instructions.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.1.0/xthead-2022-11-07-2. 1.0.pdf.

#### **XTheadMac**

The XTheadMac extension provides multiply-accumulate instructions.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### XTheadMemIdx

The XTheadMemIdx extension provides GPR memory operations.

```
It is documented in https://github.com/T-head-Semi/
thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2.
0.0.pdf.
```

#### **XTheadMemPair**

The XTheadMemPair extension provides two-GP-register memory operations.

It is documented in https://github.com/T-head-Semi/ thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2. 0.0.pdf.

#### **XTheadSync**

The XTheadSync extension provides instructions for multi-processor synchronization.

```
It is documented in https://github.com/T-head-Semi/
thead-extension-spec/releases/download/2.0.0/xthead-2022-09-05-2.
0.0.pdf.
```

#### XVentanaCondOps

XVentanaCondOps extension provides instructions for branchless sequences that perform conditional arithmetic, conditional bitwise-logic, and conditional select operations.

```
It is documented at https://github.com/ventanamicro/
ventana-custom-extensions/releases/download/v1.0.0/
ventana-custom-extensions-v1.0.0.pdf.
```

# 9.39 RL78 Dependent Features

### 9.39.1 RL78 Options

relax	Enable support	for link-time	relaxation.
-------	----------------	---------------	-------------

- **norelax** Disable support for link-time relaxation (default).
- mg10 Mark the generated binary as targeting the G10 variant of the RL78 architecture.
- mg13 Mark the generated binary as targeting the G13 variant of the RL78 architecture.

mg14

mr178 Mark the generated binary as targeting the G14 variant of the RL78 architecture. This is the default.

#### m32bit-doubles

Mark the generated binary as one that uses 32-bits to hold the **double** floating point type. This is the default.

#### m64bit-doubles

Mark the generated binary as one that uses 64-bits to hold the **double** floating point type.

### 9.39.2 Symbolic Operand Modifiers

The RL78 has three modifiers that adjust the relocations used by the linker:

#### %lo16()

When loading a 20-bit (or wider) address into registers, this modifier selects the 16 least significant bits.

movw ax,#%lo16(\_sym)

#### %hi16()

When loading a 20-bit (or wider) address into registers, this modifier selects the 16 most significant bits. movw ax,#%hi16(\_sym)

%hi8()

When loading a 20-bit (or wider) address into registers, this modifier selects the 8 bits that would go into CS or ES (i.e. bits 23..16). mov es, #%hi8(\_sym)

### 9.39.3 Assembler Directives

In addition to the common directives, the RL78 adds these:

- .double Output a constant in "double" format, which is either a 32-bit or a 64-bit floating point value, depending upon the setting of the -m32bit-doubles|-m64bit-doubles command-line option.
- .bss Select the BSS section.
- .3byte Output a constant value in a three byte format.
- .int
- .word Output a constant value in a four byte format.

# 9.39.4 Syntax for the RL78

# 9.39.4.1 Special Characters

The presence of a ';' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The '|' character can be used to separate statements on the same line.

# 9.40 RX Dependent Features

# 9.40.1 RX Options

The Renesas RX port of as has a few target specific command-line options:

#### -m32bit-doubles

This option controls the ABI and indicates to use a 32-bit float ABI. It has no effect on the assembled instructions, but it does influence the behaviour of the '.double' pseudo-op. This is the default.

#### -m64bit-doubles

This option controls the ABI and indicates to use a 64-bit float ABI. It has no effect on the assembled instructions, but it does influence the behaviour of the '.double' pseudo-op.

#### -mbig-endian

This option controls the ABI and indicates to use a big-endian data ABI. It has no effect on the assembled instructions, but it does influence the behaviour of the '.short', '.hword', '.int', '.word', '.long', '.quad' and '.octa' pseudo-ops.

#### -mlittle-endian

This option controls the ABI and indicates to use a little-endian data ABI. It has no effect on the assembled instructions, but it does influence the behaviour of the '.short', '.hword', '.int', '.word', '.long', '.quad' and '.octa' pseudo-ops. This is the default.

#### -muse-conventional-section-names

This option controls the default names given to the code (.text), initialised data (.data) and uninitialised data sections (.bss).

#### -muse-renesas-section-names

This option controls the default names given to the code (P), initialised data  $(D_1)$  and uninitialised data sections  $(B_1)$ . This is the default.

#### -msmall-data-limit

This option tells the assembler that the small data limit feature of the RX port of GCC is being used. This results in the assembler generating an undefined reference to a symbol called \_\_gp for use by the relocations that are needed to support the small data limit feature. This option is not enabled by default as it would otherwise pollute the symbol table.

-mpid This option tells the assembler that the position independent data of the RX port of GCC is being used. This results in the assembler generating an undefined reference to a symbol called \_\_pid\_base, and also setting the RX\_PID flag bit in the e\_flags field of the ELF header of the object file.

#### -mint-register=num

This option tells the assembler how many registers have been reserved for use by interrupt handlers. This is needed in order to compute the correct values for the **%gpreg** and **%pidreg** meta registers.

#### -mgcc-abi

This option tells the assembler that the old GCC ABI is being used by the assembled code. With this version of the ABI function arguments that are passed on the stack are aligned to a 32-bit boundary.

-mrx-abi This option tells the assembler that the official RX ABI is being used by the assembled code. With this version of the ABI function arguments that are passed on the stack are aligned to their natural alignments. This option is the default.

#### -mcpu=name

This option tells the assembler the target CPU type. Currently the rx100, rx200, rx600, rx610, rxv2, rxv3 and rxv3-dfpu are recognised as valid cpu names. Attempting to assemble an instructionnot supported by the indicated cpu type will result in an error message being generated.

#### -mno-allow-string-insns

This option tells the assembler to mark the object file that it is building as one that does not use the string instructions SMOVF, SCMPU, SMOVB, SMOVU, SUNTIL SWHILE or the RMPA instruction. In addition the mark tells the linker to complain if an attempt is made to link the binary with another one that does use any of these instructions.

Note - the inverse of this option, -mallow-string-insns, is not needed. The assembler automatically detects the use of the the instructions in the source code and labels the resulting object file appropriately. If no string instructions are detected then the object file is labelled as being one that can be linked with either string-using or string-banned object files.

### 9.40.2 Symbolic Operand Modifiers

The assembler supports one modifier when using symbol addresses in RX instruction operands. The general syntax is the following:

#### %gp(symbol)

The modifier returns the offset from the  $\_gp$  symbol to the specified symbol as a 16-bit value. The intent is that this offset should be used in a register+offset move instruction when generating references to small data. Ie, like this:

```
mov.W %gp(_foo)[%gpreg], r1
```

The assembler also supports two meta register names which can be used to refer to registers whose values may not be known to the programmer. These meta register names are:

%gpreg The small data address register.

%pidreg The PID base address register.

Both registers normally have the value r13, but this can change if some registers have been reserved for use by interrupt handlers or if both the small data limit and position independent data features are being used at the same time.

# 9.40.3 Assembler Directives

The RX version of **as** has the following specific assembler directives:

- .3byte Inserts a 3-byte value into the output file at the current location.
- .fetchalign

If the next opcode following this directive spans a fetch line boundary (8 byte boundary), the opcode is aligned to that boundary. If the next opcode does not span a fetch line, this directive has no effect. Note that one or more labels may be between this directive and the opcode; those labels are aligned as well. Any inserted bytes due to alignment will form a NOP opcode.

## 9.40.4 Floating Point

The floating point formats generated by directives are these.

- .float Single precision (32-bit) floating point constants.
- .double If the -m64bit-doubles command-line option has been specified then then double directive generates double precision (64-bit) floating point constants, otherwise it generates single precision (32-bit) floating point constants. To force the generation of 64-bit floating point constants used the dc.d directive instead.

# 9.40.5 Syntax for the RX

## 9.40.5.1 Special Characters

The presence of a ';' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The '!' character can be used to separate statements on the same line.

# 9.41 IBM S/390 Dependent Features

The s390 version of **as** supports two architectures modes and eleven chip levels. The architecture modes are the Enterprise System Architecture (ESA) and the newer z/Architecture mode. The chip levels are g5 (or arch3), g6, z900 (or arch5), z990 (or arch6), z9-109, z9-ec (or arch7), z10 (or arch8), z196 (or arch9), zEC12 (or arch10), z13 (or arch11), z14 (or arch12), z15 (or arch13), or z16 (or arch14).

### 9.41.1 Options

The following table lists all available s390 specific options:

-m31 | -m64

Select 31- or 64-bit ABI implying a word size of 32- or 64-bit.

These options are only available with the ELF object file format, and require that the necessary BFD support has been included (on a 31-bit platform you must add –enable-64-bit-bfd on the call to the configure script to enable 64-bit usage and use s390x as target platform).

-mesa | -mzarch

Select the architecture mode, either the Enterprise System Architecture (esa) mode or the z/Architecture mode (zarch).

The 64-bit instructions are only available with the z/Architecture mode. The combination of '-m64' and '-mesa' results in a warning message.

-march=CPU

This option specifies the target processor. The following processor names are recognized: g5 (or arch3), g6, z900 (or arch5), z990 (or arch6), z9-109, z9-ec (or arch7), z10 (or arch8), z196 (or arch9), zEC12 (or arch10), z13 (or arch11), z14 (or arch12), z15 (or arch13), and z16 (or arch14).

Assembling an instruction that is not supported on the target processor results in an error message.

The processor names starting with **arch** refer to the edition number in the Principle of Operations manual. They can be used as alternate processor names and have been added for compatibility with the IBM XL compiler.

arch3, g5 and g6 cannot be used with the '-mzarch' option since the z/Architecture mode is not supported on these processor levels.

There is no arch4 option supported. arch4 matches -march=arch5 -mesa.

#### -mregnames

Allow symbolic names for registers.

#### -mno-regnames

Do not allow symbolic names for registers.

#### -mwarn-areg-zero

Warn whenever the operand for a base or index register has been specified but evaluates to zero. This can indicate the misuse of general purpose register 0 as an address register.

# 9.41.2 Special Characters

'#' is the line comment character.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used instead of a newline to separate statements.

### 9.41.3 Instruction syntax

The assembler syntax closely follows the syntax outlined in Enterprise Systems Architecture/390 Principles of Operation (SA22-7201) and the z/Architecture Principles of Operation (SA22-7832).

Each instruction has two major parts, the instruction mnemonic and the instruction operands. The instruction format varies.

# 9.41.3.1 Register naming

The **as** recognizes a number of predefined symbols for the various processor registers. A register specification in one of the instruction formats is an unsigned integer between 0 and 15. The specific instruction and the position of the register in the instruction format denotes the type of the register. The register symbols are prefixed with ' $\chi$ ':

%rN	the 16 general purpose registers, 0 <= N <= 15
% fN	the 16 floating point registers, 0 <= N <= 15
%aN	the 16 access registers, 0 <= N <= 15
%cN	the 16 control registers, 0 <= N <= 15
%lit	an alias for the general purpose register $\%r13$
$\%\mathrm{sp}$	an alias for the general purpose register $\%r15$

# 9.41.3.2 Instruction Mnemonics

All instructions documented in the Principles of Operation are supported with the mnemonic and order of operands as described. The instruction mnemonic identifies the instruction format (Section 9.41.3.4 [s390 Formats], page 307) and the specific operation code for the instruction. For example, the 'lr' mnemonic denotes the instruction format 'RR' with the operation code '0x18'.

The definition of the various mnemonics follows a scheme, where the first character usually hint at the type of the instruction:

- a add instruction, for example 'al' for add logical 32-bit
- b branch instruction, for example 'bc' for branch on condition

	С	compare or convert instruction, for example 'cr' for compare register 32-bit
	d	divide instruction, for example 'dlr' devide logical register 64-bit to 32-bit
	i	insert instruction, for example 'ic' insert character
	1	load instruction, for example 'ltr' load and test register
	mv	move instruction, for example ' $mvc$ ' move character
	m	multiply instruction, for example 'mh' multiply halfword
	n	and instruction, for example 'ni' and immediate
	0	or instruction, for example 'oc' or character
	sla, sll	shift left single instruction
	$\operatorname{sra},\operatorname{srl}$	shift right single instruction
	$\operatorname{st}$	store instruction, for example 'stm' store multiple
	S	subtract instruction, for example 'slr' subtract logical 32-bit
	t	test or translate instruction, of example 'tm' test under mask
	х	exclusive or instruction, for example ' $\mathbf{xc}$ ' exclusive or character
Certain characters at the end of the mnemonic may describe a property of the instruction:		
c the instruction uses a 8-bit character operand		
	f the instruction extends a 32-bit operand to 64 bit	
	g the	operands are treated as 64-bit values
	h the	operand uses a 16-bit halfword operand

- i the instruction uses an immediate operand
- l the instruction uses unsigned, logical operands

- m the instruction uses a mask or operates on multiple values
- r if r is the last character, the instruction operates on registers
- y the instruction uses 20-bit displacements

There are many exceptions to the scheme outlined in the above lists, in particular for the privileged instructions. For non-privileged instruction it works quite well, for example the instruction 'clgfr' c: compare instruction, l: unsigned operands, g: 64-bit operands, f: 32- to 64-bit extension, r: register operands. The instruction compares an 64-bit value in a register with the zero extended 32-bit value from a second register. For a complete list of all mnemonics see appendix B in the Principles of Operation.

# 9.41.3.3 Instruction Operands

Instruction operands can be grouped into three classes, operands located in registers, immediate operands, and operands in storage.

A register operand can be located in general, floating-point, access, or control register. The register is identified by a four-bit field. The field containing the register operand is called the R field.

Immediate operands are contained within the instruction and can have 8, 16 or 32 bits. The field containing the immediate operand is called the I field. Dependent on the instruction the I field is either signed or unsigned.

A storage operand consists of an address and a length. The address of a storage operands can be specified in any of these ways:

- The content of a single general R
- The sum of the content of a general register called the base register B plus the content of a displacement field D
- The sum of the contents of two general registers called the index register X and the base register B plus the content of a displacement field
- The sum of the current instruction address and a 32-bit signed immediate field multiplied by two.

The length of a storage operand can be:

- Implied by the instruction
- Specified by a bitmask
- Specified by a four-bit or eight-bit length field L
- Specified by the content of a general register

The notation for storage operand addresses formed from multiple fields is as follows:

Dn(Bn) the address for operand number n is formed from the content of general register Bn called the base register and the displacement field Dn.

Dn(Xn,Bn)

the address for operand number n is formed from the content of general register Xn called the index register, general register Bn called the base register and the displacement field Dn.

#### Dn(Ln,Bn)

the address for operand number n is formed from the content of general register Bn called the base register and the displacement field Dn. The length of the operand n is specified by the field Ln.

The base registers Bn and the index registers Xn of a storage operand can be skipped. If Bn and Xn are skipped, a zero will be stored to the operand field. The notation changes as follows:

full notation	short notation
$\mathrm{Dn}(0,\mathrm{Bn})$	Dn(Bn)
Dn(0,0)	Dn
$\mathrm{Dn}(0)$	Dn
Dn(Ln,0)	Dn(Ln)

## 9.41.3.4 Instruction Formats

The Principles of Operation manuals lists 35 instruction formats where some of the formats have multiple variants. For the '.insn' pseudo directive the assembler recognizes some of the formats. Typically, the most general variant of the instruction format is used by the '.insn' directive.

The following table lists the abbreviations used in the table of instruction formats:

OpCode / OpCd Part of the op code.

Bx	Base register number for operand x.
Dx	Displacement for operand x.
DLx	Displacement lower 12 bits for operand x.
DHx	Displacement higher 8-bits for operand x.
Rx	Register number for operand x.
Xx	Index register number for operand x.
Ix	Signed immediate for operand x.
Ux	Unsigned immediate for operand x.

An instruction is two, four, or six bytes in length and must be aligned on a 2 byte boundary. The first two bits of the instruction specify the length of the instruction, 00 indicates a two byte instruction, 01 and 10 indicates a four byte instruction, and 11 indicates a six byte instruction.

The following table lists the s390 instruction formats that are available with the '.insn' pseudo directive:

E format

+----+ | OpCode | +----+ 0 15 RI format: <insn> R1,I2 +----+ | OpCode | R1 | OpCd | I2 | +----+ 8 12 16 0 31 RIE format: <insn> R1,R3,I2 +----+ | OpCode | R1 | R3 | I2 |//////| OpCode | 0 32 40 8 12 16 47 RIL format: <insn> R1,I2 +----+ | OpCode | R1 |OpCd| I2 +----+ 0 8 12 16 47 RILU format: <insn> R1,U2 +----+ U2 | OpCode | R1 |OpCd| +----+ 0 8 12 16 47 RIS format: <insn> R1,I2,M3,D4(B4) +----+ | OpCode | R1 | M3 | B4 | D4 | I2 | Opcode | 0 12 16 20 32 8 36 47 RR format: <insn> R1,R2 +----+ | OpCode | R1 | R2 | +----+ 0 8 12 15 RRE format: <insn> R1,R2 +----+ OpCode |/////| R1 | R2 | 1 +----+

0 16 24 28 31 RRF format: <insn> R1,R2,R3,M4 +----+ | OpCode | R3 | M4 | R1 | R2 | +----+ 0 16 20 24 28 31 RRS format: <insn> R1,R2,M3,D4(B4) +----+ | OpCode | R1 | R3 | B4 | D4 | M3 |////| OpCode | 8 12 16 20 32 36 40 47 0 RS format: <insn> R1,R3,D2(B2) +----+ | OpCode | R1 | R3 | B2 | D2 | +----+ 12 16 20 0 8 31 RSE format: <insn> R1,R3,D2(B2) +----+ | OpCode | R1 | R3 | B2 | D2 |//////| OpCode | +----+ 0 8 12 16 20 32 40 47 RSI format: <insn> R1,R3,I2 +----+ | OpCode | R1 | R3 | I2 +----+ 8 12 16 0 47 RSY format: <insn> R1,R3,D2(B2) +----+ | OpCode | R1 | R3 | B2 | DL2 | DH2 | OpCode | 32 40 47 8 12 16 20 0 RX format: <insn> R1,D2(X2,B2) +----+ | OpCode | R1 | X2 | B2 | D2 \_\_\_\_\_ +----+ 0 8 12 16 20 31 RXE format: <insn> R1,D2(X2,B2) +----+ | OpCode | R1 | X2 | B2 | D2 |////// OpCode | 12 16 20 32 0 8 40 47 RXF format: <insn> R1,R3,D2(X2,B2) +----+

	OpCode								-	
(					20				40 40	
RXY format:	<insn> R1</insn>	-			_1				4	
	OpCode	R1	X2	B2	I	DL2	Ι	DH2	OpCode	e
	) 8								40	
S format: <:	insn> D2(E			-+	_+		+			
	0p0	Code		B2	Ì	D2	I			
(				16			31			
SI format:	<insn> D1( +</insn>			-+	_+		+			
	DpCode		[2	B1	I	D1	Ι			
	) {						31			
SIY format:	<insn> D1</insn>						4		4	
I	OpCode	]	[2	B1	I	DL1	D	H1	OpCode	e
	) 8								40	
SIL format:	<insn> D1</insn>			-+	-+		+			+
	Op(	Code		B1	I	D1	I	I	2	Ι
- (										47
SS format:	<insn> D1(</insn>							1		
	OpCode	R1	R3	B1	Ι	D1	B2	1	D2	Ι
(		3	12	16	20		32			47
SSE format:	<insn> D1</insn>				_1			_4		
	0p0	Code		B1	I	D1	B2		D2	Ι
(	+ ) {	3	12							
SSF format:	<insn> D1</insn>							<b>_+</b>		+
	OpCode	R3	OpCo	1  B1	I	D1	B2	1	D2	Ι
	+ ) {	+ 3	12		20		32			+ 47

VRV format:											
	OpCode	V1	V2	B2	I	D2		M3	Ι	Opcode	
	)										+ 47
VRI format:							-+	_+	_+		+
	OpCode	V1	V2	I	13		M5	M4	Ι	Opcode	Ι
	)										47
VRX format:					-+			-+	-+		+
	OpCode	V1	R2	B2	I	D2		M3	Ι	Opcode	Ι
	)										47
VRS format:					-+			-+	-+		+
	OpCode	R1	V3	B2	I	D2		M4	Ι	Opcode	Ι
	)							32			47
VRR format: <insn> V1,V2,V3,M4,M5,M6 +++++++++</insn>											
	OpCode	V1	V2	V3	///	M6	M5	M4	Ι	Opcode	Ι
		8			2						47
VSI format: <insn> V1,D2(B2),I3</insn>											
										Opcode	
	)			16							

For the complete list of all instruction format variants see the Principles of Operation manuals.

### 9.41.3.5 Instruction Aliases

A specific bit pattern can have multiple mnemonics, for example the bit pattern '0xa7000000' has the mnemonics 'tmh' and 'tmlh'. In addition, there are a number of mnemonics recognized by as that are not present in the Principles of Operation. These are the short forms of the branch instructions, where the condition code mask operand is encoded in the mnemonic. This is relevant for the branch instructions, the compare and branch instructions, and the compare and trap instructions.

For the branch instructions there are 20 condition code strings that can be used as part of the mnemonic in place of a mask operand in the instruction format:

instruction	short form
bcr M1,R2	b <m>r R2</m>
bc M1,D2(X2,B2)	b <m> D2(X2,B2)</m>
brc M1,I2	j <m> I2</m>
brcl M1,I2	jg <m> I2</m>

In the mnemonic for a branch instruction the condition code string  $<\!m\!>$  can be any of the following:

0	jump on overflow / if ones
h	jump on A high
р	jump on plus
nle	jump on not low or equal
1	jump on A low
m	jump on minus
nhe	jump on not high or equal
lh	jump on low or high
ne	jump on A not equal B
nz	jump on not zero / if not zeros
e	jump on A equal B
$\mathbf{Z}$	jump on zero / if zeroes
nlh	jump on not low or high
he	jump on high or equal
nl	jump on A not low
nm	$\operatorname{jump}$ on not minus / if not mixed

le	jump on low or equal
nh	jump on A not high
np	jump on not plus
no	jump on not overflow / if not ones

For the compare and branch, and compare and trap instructions there are 12 condition code strings that can be used as part of the mnemonic in place of a mask operand in the instruction format:

instruction	short form
crb R1,R2,M3,D4(B4)	crb <m> R1,R2,D4(B4)</m>
cgrb R1,R2,M3,D4(B4)	cgrb <m> R1,R2,D4(B4)</m>
crj R1,R2,M3,I4	crj <m> R1,R2,I4</m>
cgrj R1,R2,M3,I4	cgrj <m> R1,R2,I4</m>
cib R1,I2,M3,D4(B4)	cib <m> R1,I2,D4(B4)</m>
cgib R1,I2,M3,D4(B4)	cgib <m> R1,I2,D4(B4)</m>
cij R1,I2,M3,I4	cij <m> R1,I2,I4</m>
cgij R1,I2,M3,I4	cgij <m> R1,I2,I4</m>
$\operatorname{crt}$ R1,R2,M3	crt <m> R1,R2</m>
cgrt R1,R2,M3	cgrt <m> R1,R2</m>
cit R1,I2,M3	cit <m> R1,I2</m>
cgit R1,I2,M3	cgit <m> R1,I2</m>
clrb R1,R2,M3,D4(B4)	clrb <m> R1,R2,D4(B4)</m>
clgrb R1,R2,M3,D4(B4)	clgrb <m> R1,R2,D4(B4)</m>
clrj R1,R2,M3,I4	clrj <m> R1,R2,I4</m>
clgrj R1, R2, M3, I4	clgrj <m> R1,R2,I4</m>

clib R1,I2,M3,D4(B4)	clib <m> R1,I2,D4(B4)</m>
clgib R1,I2,M3,D4(B4)	clgib <m> R1,I2,D4(B4)</m>
clij R1,I2,M3,I4	clij <m> R1,I2,I4</m>
clgij R1,I2,M3,I4	clgij <m> R1,I2,I4</m>
clrt R1,R2,M3	clrt <m> R1,R2</m>
clgrt R1,R2,M3	clgrt <m> <math>R1,R2</math></m>
clfit R1,I2,M3	clfit <m> R1,I2</m>
clgit R1,I2,M3	clgit <m> R1,I2</m>

In the mnemonic for a compare and branch and compare and trap instruction the condition code string <m> can be any of the following:

- h jump on A high
- nle jump on not low or equal
- l jump on A low
- nhe jump on not high or equal
- ne jump on A not equal B
- lh jump on low or high
- e jump on A equal B
- nlh jump on not low or high
- nl jump on A not low
- he jump on high or equal
- nh jump on A not high
- le jump on low or equal

## 9.41.3.6 Instruction Operand Modifier

If a symbol modifier is attached to a symbol in an expression for an instruction operand field, the symbol term is replaced with a reference to an object in the global offset table (GOT) or the procedure linkage table (PLT). The following expressions are allowed: 'symbol@modifier + constant', 'symbol@modifier + label + constant', and 'symbol@modifier - label + constant'. The term 'symbol' is the symbol that will be entered into the GOT or PLT, 'label' is a local label, and 'constant' is an arbitrary expression that the assembler can evaluate to a constant value.

The term '(symbol + constant1)@modifier +/- label + constant2' is also accepted but a warning message is printed and the term is converted to 'symbol@modifier +/- label + constant1 + constant2'.

@got

- **@got12** The @got modifier can be used for displacement fields, 16-bit immediate fields and 32-bit pc-relative immediate fields. The @got12 modifier is synonym to @got. The symbol is added to the GOT. For displacement fields and 16-bit immediate fields the symbol term is replaced with the offset from the start of the GOT to the GOT slot for the symbol. For a 32-bit pc-relative field the pc-relative offset to the GOT slot from the current instruction address is used.
- **@gotent** The @gotent modifier can be used for 32-bit pc-relative immediate fields. The symbol is added to the GOT and the symbol term is replaced with the pc-relative offset from the current instruction to the GOT slot for the symbol.
- **@gotoff** The @gotoff modifier can be used for 16-bit immediate fields. The symbol term is replaced with the offset from the start of the GOT to the address of the symbol.
- **@gotplt** The @gotplt modifier can be used for displacement fields, 16-bit immediate fields, and 32-bit pc-relative immediate fields. A procedure linkage table entry is generated for the symbol and a jump slot for the symbol is added to the GOT. For displacement fields and 16-bit immediate fields the symbol term is replaced with the offset from the start of the GOT to the jump slot for the symbol. For a 32-bit pc-relative field the pc-relative offset to the jump slot from the current instruction address is used.
- **@plt**The @plt modifier can be used for 16-bit and 32-bit pc-relative immediate fields.A procedure linkage table entry is generated for the symbol. The symbol termis replaced with the relative offset from the current instruction to the PLT entryfor the symbol.
- **@pltoff** The @pltoff modifier can be used for 16-bit immediate fields. The symbol term is replaced with the offset from the start of the PLT to the address of the symbol.

#### @gotntpoff

The @gotntpoff modifier can be used for displacement fields. The symbol is added to the static TLS block and the negated offset to the symbol in the static TLS block is added to the GOT. The symbol term is replaced with the offset to the GOT slot from the start of the GOT.

#### @indntpoff

The @indntpoff modifier can be used for 32-bit pc-relative immediate fields. The symbol is added to the static TLS block and the negated offset to the symbol in the static TLS block is added to the GOT. The symbol term is replaced with the pc-relative offset to the GOT slot from the current instruction address.

For more information about the thread local storage modifiers 'gotntpoff' and 'indntpoff' see the ELF extension documentation 'ELF Handling For Thread-Local Storage'.

## 9.41.3.7 Instruction Marker

The thread local storage instruction markers are used by the linker to perform code optimization.

:tls\_load

The :tls\_load marker is used to flag the load instruction in the initial exec TLS model that retrieves the offset from the thread pointer to a thread local storage variable from the GOT.

```
:tls_gdcall
```

The :tls\_gdcall marker is used to flag the branch-and-save instruction to the \_\_tls\_get\_offset function in the global dynamic TLS model.

:tls\_ldcall

The :tls\_ldcall marker is used to flag the branch-and-save instruction to the \_\_tls\_get\_offset function in the local dynamic TLS model.

For more information about the thread local storage instruction marker and the linker optimizations see the ELF extension documentation 'ELF Handling For Thread-Local Storage'.

## 9.41.3.8 Literal Pool Entries

A literal pool is a collection of values. To access the values a pointer to the literal pool is loaded to a register, the literal pool register. Usually, register %r13 is used as the literal pool register (Section 9.41.3.1 [s390 Register], page 304). Literal pool entries are created by adding the suffix :lit1, :lit2, :lit4, or :lit8 to the end of an expression for an instruction operand. The expression is added to the literal pool and the operand is replaced with the offset to the literal in the literal pool.

- :lit1 The literal pool entry is created as an 8-bit value. An operand modifier must not be used for the original expression.
- :1it2 The literal pool entry is created as a 16 bit value. The operand modifier @got may be used in the original expression. The term 'x@got:lit2' will put the got offset for the global symbol x to the literal pool as 16 bit value.
- :lit4 The literal pool entry is created as a 32-bit value. The operand modifier @got and @plt may be used in the original expression. The term 'x@got:lit4' will put the got offset for the global symbol x to the literal pool as a 32-bit value. The term 'x@plt:lit4' will put the plt offset for the global symbol x to the literal pool as a 32-bit value.

:1it8 The literal pool entry is created as a 64-bit value. The operand modifier @got and @plt may be used in the original expression. The term 'x@got:lit8' will put the got offset for the global symbol x to the literal pool as a 64-bit value. The term 'x@plt:lit8' will put the plt offset for the global symbol x to the literal pool as a 64-bit value.

The assembler directive '.ltorg' is used to emit all literal pool entries to the current position.

### 9.41.4 Assembler Directives

as for s390 supports all of the standard ELF assembler directives as outlined in the main part of this document. Some directives have been extended and there are some additional directives, which are only available for the s390 as.

.insn This directive permits the numeric representation of an instructions and makes the assembler insert the operands according to one of the instructions formats for '.insn' (Section 9.41.3.4 [s390 Formats], page 307). For example, the instruction '1 %r1,24(%r15)' could be written as '.insn rx,0x58000000,%r1,24(%r15)'.

.short

.long

.quad This directive places one or more 16-bit (.short), 32-bit (.long), or 64-bit (.quad) values into the current section. If an ELF or TLS modifier is used only the following expressions are allowed: 'symbol@modifier + constant', 'symbol@modifier + label + constant', and 'symbol@modifier - label + constant'. The following modifiers are available:

@got

- **@got12** The @got modifier can be used for .short, .long and .quad. The @got12 modifier is synonym to @got. The symbol is added to the GOT. The symbol term is replaced with offset from the start of the GOT to the GOT slot for the symbol.
- **@gotoff** The @gotoff modifier can be used for .short, .long and .quad. The symbol term is replaced with the offset from the start of the GOT to the address of the symbol.
- **@gotplt** The @gotplt modifier can be used for .long and .quad. A procedure linkage table entry is generated for the symbol and a jump slot for the symbol is added to the GOT. The symbol term is replaced with the offset from the start of the GOT to the jump slot for the symbol.
- **@plt**The @plt modifier can be used for .long and .quad. A procedure<br/>linkage table entry us generated for the symbol. The symbol term<br/>is replaced with the address of the PLT entry for the symbol.
- **@pltoff** The @pltoff modifier can be used for .short, .long and .quad. The symbol term is replaced with the offset from the start of the PLT to the address of the symbol.

## @tlsgd

@tlsldmThe @tlsgd and @tlsldm modifier can be used for .long and .quad.A tls\_index structure for the symbol is added to the GOT. The

symbol term is replaced with the offset from the start of the GOT to the tls\_index structure.

# @gotntpoff @indntpoff

The @gotntpoff and @indntpoff modifier can be used for .long and .quad. The symbol is added to the static TLS block and the negated offset to the symbol in the static TLS block is added to the GOT. For @gotntpoff the symbol term is replaced with the offset from the start of the GOT to the GOT slot, for @indntpoff the symbol term is replaced with the address of the GOT slot.

- **@dtpoff** The @dtpoff modifier can be used for .long and .quad. The symbol term is replaced with the offset of the symbol relative to the start of the TLS block it is contained in.
- **Ontpoff** The @ntpoff modifier can be used for .long and .quad. The symbol term is replaced with the offset of the symbol relative to the TCB pointer.

For more information about the thread local storage modifiers see the ELF extension documentation 'ELF Handling For Thread-Local Storage'.

.ltorg This directive causes the current contents of the literal pool to be dumped to the current location (Section 9.41.3.8 [s390 Literal Pool Entries], page 316).

#### .machine STRING[+EXTENSION] ...

This directive allows changing the machine for which code is generated. string may be any of the -march= selection options, or push, or pop. .machine push saves the currently selected cpu, which may be restored with .machine pop. Be aware that the cpu string has to be put into double quotes in case it contains characters not appropriate for identifiers. So you have to write "z9-109" instead of just z9-109. Extensions can be specified after the cpu name, separated by plus characters. Valid extensions are: htm, nohtm, vx, novx. They extend the basic instruction set with features from a higher cpu level, or remove support for a feature from the given cpu level.

Example: z13+nohtm allows all instructions of the z13 cpu except instructions from the HTM facility.

#### .machinemode string

This directive allows one to change the architecture mode for which code is being generated. string may be esa, zarch, zarch\_nohighgprs, push, or pop. .machinemode zarch\_nohighgprs can be used to prevent the highgprs flag from being set in the ELF header of the output file. This is useful in situations where the code is gated with a runtime check which makes sure that the code is only executed on kernels providing the highgprs feature. .machinemode push saves the currently selected mode, which may be restored with .machinemode pop.

## 9.41.5 Floating Point

The assembler recognizes both the IEEE floating-point instruction and the hexadecimal floating-point instructions. The floating-point constructors '.float', '.single', and '.double' always emit the IEEE format. To assemble hexadecimal floating-point constants the '.long' and '.quad' directives must be used.

## 9.42 SCORE Dependent Features

## 9.42.1 Options

The following table lists all available SCORE options.

- -G num This option sets the largest size of an object that can be referenced implicitly with the gp register. The default value is 8.
- -EB Assemble code for a big-endian cpu
- -EL Assemble code for a little-endian cpu
- -FIXDD Assemble code for fix data dependency
- -NWARN Assemble code for no warning message for fix data dependency
- -SCORE5 Assemble code for target is SCORE5
- -SCORE5U Assemble code for target is SCORE5U
- -SCORE7 Assemble code for target is SCORE7, this is default setting
- -SCORE3 Assemble code for target is SCORE3
- -march=score7

Assemble code for target is SCORE7, this is default setting

-march=score3

Assemble code for target is SCORE3

- -USE\_R1 Assemble code for no warning message when using temp register r1
- -KPIC Generate code for PIC. This option tells the assembler to generate score positionindependent macro expansions. It also tells the assembler to mark the output file as PIC.
- -00 Assembler will not perform any optimizations

-V Sunplus release version

## 9.42.2 SCORE Assembler Directives

A number of assembler directives are available for SCORE. The following table is far from complete.

.set nwarn

Let the assembler not to generate warnings if the source machine language instructions happen data dependency.

.set fixdd

Let the assembler to insert bubbles (32 bit nop instruction / 16 bit nop! Instruction) if the source machine language instructions happen data dependency.

.set nofixdd

Let the assembler to generate warnings if the source machine language instructions happen data dependency. (Default)

.set r1 Let the assembler not to generate warnings if the source program uses r1. allow user to use r1

- set nor1 Let the assembler to generate warnings if the source program uses r1. (Default)
- .sdata Tell the assembler to add subsequent data into the sdata section
- .rdata Tell the assembler to add subsequent data into the rdata section

#### .frame "frame-register", "offset", "return-pc-register"

Describe a stack frame. "frame-register" is the frame register, "offset" is the distance from the frame register to the virtual frame pointer, "return-pc-register" is the return program register. You must use ".ent" before ".frame" and only one ".frame" can be used per ".ent".

#### .mask "bitmask", "frameoffset"

Indicate which of the integer registers are saved in the current function's stack frame, this is for the debugger to explain the frame chain.

.ent "proc-name"

Set the beginning of the procedure "proc\_name". Use this directive when you want to generate information for the debugger.

#### .end proc-name

Set the end of a procedure. Use this directive to generate information for the debugger.

.bss Switch the destination of following statements into the bss section, which is used for data that is uninitialized anywhere.

### 9.42.3 SCORE Syntax

#### 9.42.3.1 Special Characters

The presence of a '#' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

## 9.43 Renesas / SuperH SH Dependent Features

## 9.43.1 Options

 ${\tt as}$  has following command-line options for the Renesas (formerly Hitachi) / SuperH SH family.

little	Generate little endian code.
big	Generate big endian code.
relax	Alter jump instructions for long displacements.
small	Align sections to 4 byte boundaries, not 16.
dsp	Enable sh-dsp insns, and disable sh3e / sh4 insns.
renesas	Disable optimization with section symbol for compatibility with Renesas assembler.
allow-re	eg-prefix Allow '\$' as a register name prefix.
fdpic	Generate an FDPIC object file.
isa=sh4	<b>  sh4a</b> Specify the sh4 or sh4a instruction set.
isa=dsp	Enable sh-dsp insns, and disable sh3e / sh4 insns.
isa=fp	Enable sh2e, sh3e, sh4, and sh4a insn sets.
isa=all	Enable sh1, sh2, sh2e, sh3, sh3e, sh4, sh4a, and sh-dsp insn sets.
-h-tick-he	Support H'00 style hex constants in addition to $0x00$ style.

## 9.43.2 Syntax

## 9.43.2.1 Special Characters

'!' is the line comment character.

You can use ';' instead of a newline to separate statements.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Since '\$' has no special meaning, you may use it in symbol names.

## 9.43.2.2 Register Names

You can use the predefined symbols 'r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10', 'r11', 'r12', 'r13', 'r14', and 'r15' to refer to the SH registers.

The SH also has these control registers:

pr	procedure register (holds return address)
pc	program counter
mach macl	high and low multiply accumulator registers
sr	status register
gbr	global base register
vbr	vector base register (for interrupt vectors)

### 9.43.2.3 Addressing Modes

as understands the following addressing modes for the SH. Rn in the following refers to any of the numbered registers, but *not* the control registers.

lent

Rn	Register direct
@Rn	Register indirect
@-Rn	Register indirect with pre-decrem

**@Rn+** Register indirect with post-increment

#### @(disp, Rn)

Register indirect with displacement

- @(RO, Rn) Register indexed
- @(disp, GBR)

GBR offset

@(RO, GBR)

GBR indexed

addr

@(disp, PC)

PC relative address (for branch or for addressing memory). The **as** implementation allows you to use the simpler form *addr* anywhere a PC relative address is called for; the alternate form is supported for compatibility with other assemblers.

*#imm* Immediate data

## 9.43.3 Floating Point

SH2E, SH3E and SH4 groups have on-chip floating-point unit (FPU). Other SH groups can use .float directive to generate IEEE floating-point numbers.

SH2E and SH3E support single-precision floating point calculations as well as entirely PCAPI compatible emulation of double-precision floating point calculations. SH2E and

SH3E instructions are a subset of the floating point calculations conforming to the IEEE754 standard.

In addition to single-precision and double-precision floating-point operation capability, the on-chip FPU of SH4 has a 128-bit graphic engine that enables 32-bit floating-point data to be processed 128 bits at a time. It also supports 4 \* 4 array operations and inner product operations. Also, a superscalar architecture is employed that enables simultaneous execution of two instructions (including FPU instructions), providing performance of up to twice that of conventional architectures at the same frequency.

### 9.43.4 SH Machine Directives

uaword

ualong

uaquad

as will issue a warning when a misaligned .word, .long, or .quad directive is used. You may use .uaword, .ualong, or .uaquad to indicate that the value is intentionally misaligned.

#### 9.43.5 Opcodes

For detailed information on the SH machine instruction set, see SH-Microcomputer User's Manual (Renesas) or SH-4 32-bit CPU Core Architecture (SuperH) and SuperH (SH) 64-Bit RISC Series (SuperH).

**as** implements all the standard SH opcodes. No additional pseudo-instructions are needed on this family. Note, however, that because **as** supports a simpler form of PC-relative addressing, you may simply write (for example)

mov.l bar,r0

where other assemblers might require an explicit displacement to **bar** from the program counter:

mov.l @(disp, PC)

## 9.44 SPARC Dependent Features

### 9.44.1 Options

The SPARC chip family includes several successive versions, using the same core instruction set, but including a few additional instructions at each version. There are exceptions to this however. For details on what instructions each variant supports, please see the chip's architecture reference manual.

By default, **as** assumes the core instruction set (SPARC v6), but "bumps" the architecture level as needed: it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

If not configured for SPARC v9 (sparc64-\*-\*) GAS will not bump past sparclite by default, an option must be passed to enable the v9 instructions.

GAS treats sparclite as being compatible with v8, unless an architecture is explicitly requested. SPARC v9 is always incompatible with sparclite.

```
-Av6 | -Av7 | -Av8 | -Aleon | -Asparclet | -Asparclite

-Av8plus | -Av8plusa | -Av8plusb | -Av8plusc | -Av8plusd |

-Av8plusv | -Av8plusm | -Av8plusm8

-Av9 | -Av9a | -Av9b | -Av9c | -Av9d | -Av9e | -Av9v | -Av9m | -Av9m8

-Asparc | -Asparcvis | -Asparcvis2 | -Asparcfmaf | -Asparcima

-Asparcvis3 | -Asparcvis3r | -Asparc5 | -Asparc6
```

Use one of the '-A' options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, **as** reports a fatal error if it encounters an instruction or feature requiring an incompatible or higher level.

```
'-Av8plus', '-Av8plusa', '-Av8plusb', '-Av8plusc', '-Av8plusd', and '-Av8plusv' select a 32 bit environment.
```

'-Av9', '-Av9a', '-Av9b', '-Av9c', '-Av9d', '-Av9e', '-Av9v' and '-Av9m' select a 64 bit environment and are not available unless GAS is explicitly configured with 64 bit environment support.

'-Av8plusa' and '-Av9a' enable the SPARC V9 instruction set with UltraSPARC VIS 1.0 extensions.

'-Av8plusb' and '-Av9b' enable the UltraSPARC VIS 2.0 instructions, as well as the instructions enabled by '-Av8plusa' and '-Av9a'.

'-Av8plusc' and '-Av9c' enable the UltraSPARC Niagara instructions, as well as the instructions enabled by '-Av8plusb' and '-Av9b'.

'-Av8plusd' and '-Av9d' enable the floating point fused multiply-add, VIS 3.0, and HPC extension instructions, as well as the instructions enabled by '-Av8plusc' and '-Av9c'.

'-Av8pluse' and '-Av9e' enable the cryptographic instructions, as well as the instructions enabled by '-Av8plusd' and '-Av9d'.

'-Av8plusv' and '-Av9v' enable floating point unfused multiply-add, and integer multiply-add, as well as the instructions enabled by '-Av8pluse' and '-Av9e'.

'-Av8plusm' and '-Av9m' enable the VIS 4.0, subtract extended, xmpmul, xmontmul and xmontsqr instructions, as well as the instructions enabled by '-Av8plusv' and '-Av9v'. '-Av8plusm8' and '-Av9m8' enable the instructions introduced in the Oracle SPARC Architecture 2017 and the M8 processor, as well as the instructions enabled by '-Av8plusm' and '-Av9m'.

'-Asparc' specifies a v9 environment. It is equivalent to '-Av9' if the word size is 64-bit, and '-Av8plus' otherwise.

'-Asparcvis' specifies a v9a environment. It is equivalent to '-Av9a' if the word size is 64-bit, and '-Av8plusa' otherwise.

'-Asparcvis2' specifies a v9b environment. It is equivalent to '-Av9b' if the word size is 64-bit, and '-Av8plusb' otherwise.

'-Asparcfmaf' specifies a v9b environment with the floating point fused multiplyadd instructions enabled.

'-Asparcima' specifies a v9b environment with the integer multiply-add instructions enabled.

'-Asparcvis3' specifies a v9b environment with the VIS 3.0, HPC , and floating point fused multiply-add instructions enabled.

'-Asparcvis3r' specifies a v9b environment with the VIS 3.0, HPC, and floating point unfused multiply-add instructions enabled.

'-Asparc5' is equivalent to '-Av9m'.

'-Asparc6' is equivalent to '-Av9m8'.

```
-xarch=v8plus | -xarch=v8plusa | -xarch=v8plusb | -xarch=v8plusc
-xarch=v8plusd | -xarch=v8plusv | -xarch=v8plusm |
-xarch=v8plusm8 | -xarch=v9 | -xarch=v9a | -xarch=v9b
-xarch=v9c | -xarch=v9d | -xarch=v9e | -xarch=v9v
-xarch=v9m | -xarch=v9m8
-xarch=sparc | -xarch=sparcvis | -xarch=sparcvis2
-xarch=sparcfmaf | -xarch=sparcima | -xarch=sparcvis3
-xarch=sparcvis3r | -xarch=sparc5 | -xarch=sparc6
```

For compatibility with the SunOS v9 assembler. These options are equivalent to -Av8plus, -Av8plusa, -Av8plusb, -Av8plusc, -Av8plusd, -Av8plusv, -Av8plusm, -Av8plusm8, -Av9, -Av9a, -Av9b, -Av9c, -Av9d, -Av9e, -Av9v, -Av9m, -Av9m8, -Asparc, -Asparcvis, -Asparcvis2, -Asparcfmaf, -Asparcima, -Asparcvis3, -Asparcvis3r, -Asparc5 and -Asparc6 respectively.

- -bump Warn whenever it is necessary to switch to another level. If an architecture level is explicitly requested, GAS will not issue warnings until that level is reached, and will then bump the level as required (except between incompatible levels).
- -32 | -64 Select the word size, either 32 bits or 64 bits. These options are only available with the ELF object file format, and require that the necessary BFD support has been included.

#### --dcti-couples-detect

Warn if a DCTI (delayed control transfer instruction) couple is found when generating code for a variant of the SPARC architecture in which the execution of the couple is unpredictable, or very slow. This is disabled by default.

## 9.44.2 Enforcing aligned data

SPARC GAS normally permits data to be misaligned. For example, it permits the .long pseudo-op to be used on a byte boundary. However, the native SunOS assemblers issue an error when they see misaligned data.

You can use the --enforce-aligned-data option to make SPARC GAS also issue an error about misaligned data, just as the SunOS assemblers do.

The --enforce-aligned-data option is not the default because gcc issues misaligned data pseudo-ops when it initializes certain packed data structures (structures defined using the packed attribute). You may have to assemble with GAS in order to initialize packed data structures in your own code.

#### 9.44.3 Sparc Syntax

The assembler syntax closely follows The Sparc Architecture Manual, versions 8 and 9, as well as most extensions defined by Sun for their UltraSPARC and Niagara line of processors.

#### 9.44.3.1 Special Characters

A '!' character appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

';' can be used instead of a newline to separate statements.

### 9.44.3.2 Register Names

The Sparc integer register file is broken down into global, outgoing, local, and incoming.

- The 8 global registers are referred to as '%gn'.
- The 8 outgoing registers are referred to as '%on'.
- The 8 local registers are referred to as '%ln'.
- The 8 incoming registers are referred to as '%in'.
- The frame pointer register '%i6' can be referenced using the alias '%fp'.
- The stack pointer register '%o6' can be referenced using the alias '%sp'.

Floating point registers are simply referred to as '%fn'. When assembling for pre-V9, only 32 floating point registers are available. For V9 and later there are 64, but there are restrictions when referencing the upper 32 registers. They can only be accessed as double or quad, and thus only even or quad numbered accesses are allowed. For example, '%f34' is a legal floating point register, but '%f35' is not.

Floating point registers accessed as double can also be referred using the ' $\dn$ ' notation, where *n* is even. Similarly, floating point registers accessed as quad can be referred using the ' $\dn$ ' notation, where *n* is a multiple of 4. For example, ' $\fd$ ' can be denoted as both ' $\dd$ ' and ' $\dd$ '. On the other hand, ' $\fd$ ' can be denoted as ' $\dd$ ' but not as ' $\dd$ '.

Certain V9 instructions allow access to ancillary state registers. Most simply they can be referred to as 'asrn' where *n* can be from 16 to 31. However, there are some aliases defined to reference ASR registers defined for various UltraSPARC processors:

- The tick compare register is referred to as '%tick\_cmpr'.
- The system tick register is referred to as '%stick'. An alias, '%sys\_tick', exists but is deprecated and should not be used by new software.
- The system tick compare register is referred to as '%stick\_cmpr'. An alias, '%sys\_tick\_cmpr', exists but is deprecated and should not be used by new software.
- The software interrupt register is referred to as '%softint'.
- The set software interrupt register is referred to as '%set\_softint'. The mnemonic '%softint\_set' is provided as an alias.
- The clear software interrupt register is referred to as '%clear\_softint'. The mnemonic '%softint\_clear' is provided as an alias.
- The performance instrumentation counters register is referred to as '%pic'.
- The performance control register is referred to as '%pcr'.
- The graphics status register is referred to as '%gsr'.
- The V9 dispatch control register is referred to as '%dcr'.

Various V9 branch and conditional move instructions allow specification of which set of integer condition codes to test. These are referred to as '%xcc' and '%icc'.

Additionally, GAS supports the so-called "natural" condition codes; these are referred to as '%ncc' and reference to '%icc' if the word size is 32, '%xcc' if the word size is 64.

In V9, there are 4 sets of floating point condition codes which are referred to as '%fccn'. Several special privileged and non-privileged registers exist:

- The V9 address space identifier register is referred to as '%asi'.
- The V9 restorable windows register is referred to as '%canrestore'.
- The V9 savable windows register is referred to as '%cansave'.
- The V9 clean windows register is referred to as '%cleanwin'.
- The V9 current window pointer register is referred to as '%cwp'.
- The floating-point queue register is referred to as '%fq'.
- The V8 co-processor queue register is referred to as '%cq'.
- The floating point status register is referred to as '%fsr'.
- The other windows register is referred to as '%otherwin'.
- The V9 program counter register is referred to as '%pc'.
- The V9 next program counter register is referred to as '%npc'.
- The V9 processor interrupt level register is referred to as '%pil'.
- The V9 processor state register is referred to as '%pstate'.
- The trap base address register is referred to as '**%tba**'.
- The V9 tick register is referred to as '%tick'.
- The V9 trap level is referred to as '%t1'.
- The V9 trap program counter is referred to as '%tpc'.
- The V9 trap next program counter is referred to as '%tnpc'.
- The V9 trap state is referred to as '%tstate'.

- The V9 trap type is referred to as '%tt'.
- The V9 condition codes is referred to as '%ccr'.
- The V9 floating-point registers state is referred to as '%fprs'.
- The V9 version register is referred to as '%ver'.
- The V9 window state register is referred to as '%wstate'.
- The Y register is referred to as '%y'.
- The V8 window invalid mask register is referred to as '%wim'.
- The V8 processor state register is referred to as '%psr'.
- The V9 global register level register is referred to as '%gl'.

Several special register names exist for hypervisor mode code:

- The hyperprivileged processor state register is referred to as '%hpstate'.
- The hyperprivileged trap state register is referred to as '%htstate'.
- The hyperprivileged interrupt pending register is referred to as '%hintp'.
- The hyperprivileged trap base address register is referred to as '%htba'.
- The hyperprivileged implementation version register is referred to as '%hver'.
- The hyperprivileged system tick offset register is referred to as '%hstick\_offset'. Note that there is no '%hstick' register, the normal '%stick' is used.
- The hyperprivileged system tick enable register is referred to as '%hstick\_enable'.
- The hyperprivileged system tick compare register is referred to as '%hstick\_cmpr'.

### 9.44.3.3 Constants

Several Sparc instructions take an immediate operand field for which mnemonic names exist. Two such examples are 'membar' and 'prefetch'. Another example are the set of V9 memory access instruction that allow specification of an address space identifier.

The 'membar' instruction specifies a memory barrier that is the defined by the operand which is a bitmask. The supported mask mnemonics are:

- '#Sync' requests that all operations (including nonmemory reference operations) appearing prior to the membar must have been performed and the effects of any exceptions become visible before any instructions after the membar may be initiated. This corresponds to membar cmask field bit 2.
- '#MemIssue' requests that all memory reference operations appearing prior to the membar must have been performed before any memory operation after the membar may be initiated. This corresponds to membar cmask field bit 1.
- '#Lookaside' requests that a store appearing prior to the membar must complete before any load following the membar referencing the same address can be initiated. This corresponds to membar cmask field bit 0.
- '#StoreStore' defines that the effects of all stores appearing prior to the membar instruction must be visible to all processors before the effect of any stores following the membar. Equivalent to the deprecated stbar instruction. This corresponds to membar mmask field bit 3.

- '#LoadStore' defines all loads appearing prior to the membar instruction must have been performed before the effect of any stores following the membar is visible to any other processor. This corresponds to membar mmask field bit 2.
- '#StoreLoad' defines that the effects of all stores appearing prior to the membar instruction must be visible to all processors before loads following the membar may be performed. This corresponds to membar mmask field bit 1.
- '#LoadLoad' defines that all loads appearing prior to the membar instruction must have been performed before any loads following the membar may be performed. This corresponds to membar mmask field bit 0.

These values can be ored together, for example:

membar #Sync membar #StoreLoad | #LoadLoad membar #StoreLoad | #StoreStore

The prefetch and prefetcha instructions take a prefetch function code. The following prefetch function code constant mnemonics are available:

• '#n\_reads' requests a prefetch for several reads, and corresponds to a prefetch function code of 0.

'**#one\_read**' requests a prefetch for one read, and corresponds to a prefetch function code of 1.

'#n\_writes' requests a prefetch for several writes (and possibly reads), and corresponds to a prefetch function code of 2.

'**#one\_write**' requests a prefetch for one write, and corresponds to a prefetch function code of 3.

'**#page**' requests a prefetch page, and corresponds to a prefetch function code of 4.

'**#invalidate**' requests a prefetch invalidate, and corresponds to a prefetch function code of 16.

'**#unified**' requests a prefetch to the nearest unified cache, and corresponds to a prefetch function code of 17.

'#n\_reads\_strong' requests a strong prefetch for several reads, and corresponds to a prefetch function code of 20.

'#one\_read\_strong' requests a strong prefetch for one read, and corresponds to a prefetch function code of 21.

'#n\_writes\_strong' requests a strong prefetch for several writes, and corresponds to a prefetch function code of 22.

'#one\_write\_strong' requests a strong prefetch for one write, and corresponds to a prefetch function code of 23.

Onle one prefetch code may be specified. Here are some examples:

prefetch [%10 + %12], #one\_read prefetch [%g2 + 8], #n\_writes prefetcha [%g1] 0x8, #unified prefetcha [%o0 + 0x10] %asi, #n\_reads The actual behavior of a given prefetch function code is processor specific. If a processor does not implement a given prefetch function code, it will treat the prefetch instruction as a nop.

For instructions that accept an immediate address space identifier, as provides many mnemonics corresponding to V9 defined as well as UltraSPARC and Niagara extended values. For example, '#ASI\_P' and '#ASI\_BLK\_INIT\_QUAD\_LDD\_AIUS'. See the V9 and processor specific manuals for details.

### 9.44.3.4 Relocations

ELF relocations are available as defined in the 32-bit and 64-bit Sparc ELF specifications.

R\_SPARC\_HI22 is obtained using '%hi' and R\_SPARC\_L010 is obtained using '%lo'. Likewise R\_SPARC\_HIX22 is obtained from '%hix' and R\_SPARC\_L0X10 is obtained using '%lox'. For example:

```
sethi %hi(symbol), %g1
or %g1, %lo(symbol), %g1
sethi %hix(symbol), %g1
xor %g1, %lox(symbol), %g1
```

These "high" mnemonics extract bits 31:10 of their operand, and the "low" mnemonics extract bits 9:0 of their operand.

V9 code model relocations can be requested as follows:

- R\_SPARC\_HH22 is requested using '%hh'. It can also be generated using '%uhi'.
- R\_SPARC\_HM10 is requested using '%hm'. It can also be generated using '%ulo'.
- R\_SPARC\_LM22 is requested using '%lm'.
- R\_SPARC\_H44 is requested using '%h44'.
- R\_SPARC\_M44 is requested using '%m44'.
- R\_SPARC\_L44 is requested using '%144' or '%134'.
- R\_SPARC\_H34 is requested using '%h34'.

The '%134' generates a R\_SPARC\_L44 relocation because it calculates the necessary value, and therefore no explicit R\_SPARC\_L34 relocation needed to be created for this purpose.

The '%h34' and '%l34' relocations are used for the abs34 code model. Here is an example abs34 address generation sequence:

```
sethi %h34(symbol), %g1
sllx %g1, 2, %g1
or %g1, %l34(symbol), %g1
```

The PC relative relocation R\_SPARC\_PC22 can be obtained by enclosing an operand inside of '%pc22'. Likewise, the R\_SPARC\_PC10 relocation can be obtained using '%pc10'. These are mostly used when assembling PIC code. For example, the standard PIC sequence on Sparc to get the base of the global offset table, PC relative, into a register, can be performed as:

```
sethi %pc22(_GLOBAL_OFFSET_TABLE_-4), %17
add %17, %pc10(_GLOBAL_OFFSET_TABLE_+4), %17
```

Several relocations exist to allow the link editor to potentially optimize GOT data references. The R\_SPARC\_GOTDATA\_OP\_HIX22 relocation can obtained by enclosing an operand

inside of '%gdop\_hix22'. The R\_SPARC\_GOTDATA\_OP\_LOX10 relocation can obtained by enclosing an operand inside of '%gdop\_lox10'. Likewise, R\_SPARC\_GOTDATA\_OP can be obtained by enclosing an operand inside of '%gdop'. For example, assuming the GOT base is in register %17:

sethi %gdop\_hix22(symbol), %l1
xor %l1, %gdop\_lox10(symbol), %l1
ld [%l7 + %l1], %l2, %gdop(symbol)

There are many relocations that can be requested for access to thread local storage variables. All of the Sparc TLS mnemonics are supported:

- R\_SPARC\_TLS\_GD\_HI22 is requested using '%tgd\_hi22'.
- R\_SPARC\_TLS\_GD\_L010 is requested using '%tgd\_lo10'.
- R\_SPARC\_TLS\_GD\_ADD is requested using '%tgd\_add'.
- R\_SPARC\_TLS\_GD\_CALL is requested using '%tgd\_call'.
- R\_SPARC\_TLS\_LDM\_HI22 is requested using '%tldm\_hi22'.
- R\_SPARC\_TLS\_LDM\_LO10 is requested using '%tldm\_lo10'.
- R\_SPARC\_TLS\_LDM\_ADD is requested using '%tldm\_add'.
- R\_SPARC\_TLS\_LDM\_CALL is requested using '%tldm\_call'.
- R\_SPARC\_TLS\_LDO\_HIX22 is requested using '%tldo\_hix22'.
- R\_SPARC\_TLS\_LD0\_LOX10 is requested using '%tldo\_lox10'.
- R\_SPARC\_TLS\_LDO\_ADD is requested using '%tldo\_add'.
- R\_SPARC\_TLS\_IE\_HI22 is requested using '%tie\_hi22'.
- R\_SPARC\_TLS\_IE\_LO10 is requested using '%tie\_lo10'.
- R\_SPARC\_TLS\_IE\_LD is requested using '%tie\_ld'.
- R\_SPARC\_TLS\_IE\_LDX is requested using '%tie\_ldx'.
- R\_SPARC\_TLS\_IE\_ADD is requested using '%tie\_add'.
- R\_SPARC\_TLS\_LE\_HIX22 is requested using '%tle\_hix22'.
- R\_SPARC\_TLS\_LE\_LOX10 is requested using '%tle\_lox10'.

Here are some example TLS model sequences.

First, General Dynamic:

```
sethi %tgd_hi22(symbol), %l1
add %l1, %tgd_lo10(symbol), %l1
add %l7, %l1, %o0, %tgd_add(symbol)
call __tls_get_addr, %tgd_call(symbol)
nop
Local Dynamic:
```

```
sethi %tldm_hi22(symbol), %l1
add %l1, %tldm_lo10(symbol), %l1
add %l7, %l1, %o0, %tldm_add(symbol)
call __tls_get_addr, %tldm_call(symbol)
nop
```

```
%tldo_hix22(symbol), %11
  sethi
         %11, %tldo_lox10(symbol), %11
  xor
  add
         %00, %11, %11, %tldo_add(symbol)
Initial Exec:
  sethi
         %tie_hi22(symbol), %l1
  add
          %11, %tie_lo10(symbol), %11
  ld
          [%17 + %11], %00, %tie_ld(symbol)
  add
          %g7, %o0, %o0, %tie_add(symbol)
         %tie_hi22(symbol), %11
  sethi
  add
         %11, %tie_lo10(symbol), %11
          [%17 + %11], %00, %tie_ldx(symbol)
  ldx
  add
         %g7, %o0, %o0, %tie_add(symbol)
And finally, Local Exec:
```

sethi %tle\_hix22(symbol), %11
add %11, %tle\_lox10(symbol), %11
add %g7, %11, %11

When assembling for 64-bit, and a secondary constant addend is specified in an address expression that would normally generate an R\_SPARC\_LO10 relocation, the assembler will emit an R\_SPARC\_OLO10 instead.

#### 9.44.3.5 Size Translations

Often it is desirable to write code in an operand size agnostic manner. **as** provides support for this via operand size opcode translations. Translations are supported for loads, stores, shifts, compare-and-swap atomics, and the 'clr' synthetic instruction.

If generating 32-bit code, as will generate the 32-bit opcode. Whereas if 64-bit code is being generated, the 64-bit opcode will be emitted. For example ldn will be transformed into ld for 32-bit code and ldx for 64-bit code.

Here is an example meant to demonstrate all the supported opcode translations:

```
[%00], %01
  ldn
        [%00] %asi, %02
  ldna
        %01, [%00]
  stn
       %o2, [%o0] %asi
  stna
  slln
       %03, 3, %03
  srln %04, 8, %04
  sran %o5, 12, %o5
       [%00], %01, %02
  casn
  casna [%00] %asi, %01, %02
  clrn %g1
In 32-bit mode as will emit:
```

ld [%00], %01 lda [%00] %asi, %02 st %01, [%00] sta %02, [%00] %asi

```
sll %o3, 3, %o3
       %04, 8, %04
  srl
       %05, 12, %05
  sra
       [%00], %01, %02
  cas
  casa [%00] %asi, %01, %02
  clr %g1
And in 64-bit mode as will emit:
  ldx
        [%00], %01
  ldxa
        [%00] %asi, %02
        %01, [%00]
  stx
       %o2, [%o0] %asi
  stxa
  sllx %o3, 3, %o3
  srlx %04, 8, %04
  srax %o5, 12, %o5
  casx [%00], %01, %02
  casxa [%00] %asi, %01, %02
  clrx %g1
```

Finally, the '.nword' translating directive is supported as well. It is documented in the section on Sparc machine directives.

## 9.44.4 Floating Point

The Sparc uses IEEE floating-point numbers.

## 9.44.5 Sparc Machine Directives

The Sparc version of **as** supports the following additional machine directives:

.align	This must be followed	by the desired	alignment in bytes.
--------	-----------------------	----------------	---------------------

- .common This must be followed by a symbol name, a positive number, and "bss". This behaves somewhat like .comm, but the syntax is different.
- .half This is functionally identical to .short.
- .nword On the Sparc, the .nword directive produces native word sized value, ie. if assembling with -32 it is equivalent to .word, if assembling with -64 it is equivalent to .xword.
- .proc This directive is ignored. Any text following it on the same line is also ignored.

.register

This directive declares use of a global application or system register. It must be followed by a register name %g2, %g3, %g6 or %g7, comma and the symbol name for that register. If symbol name is **#scratch**, it is a scratch register, if it is **#ignore**, it just suppresses any errors about using undeclared global register, but does not emit any information about it into the object file. This can be useful e.g. if you save the register before use and restore it after.

.reserve This must be followed by a symbol name, a positive number, and "bss". This behaves somewhat like .lcomm, but the syntax is different.

- .seg This must be followed by "text", "data", or "data1". It behaves like .text, .data, or .data 1.
- .skip This is functionally identical to the .space directive.
- .word On the Sparc, the .word directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.
- .xword On the Sparc V9 processor, the .xword directive produces 64 bit values.

## 9.45 TIC54X Dependent Features

## **9.45.1 Options**

The TMS320C54X version of as has a few machine-dependent options.

You can use the '-mfar-mode' option to enable extended addressing mode. All addresses will be assumed to be > 16 bits, and the appropriate relocation types will be used. This option is equivalent to using the '.far\_mode' directive in the assembly code. If you do not use the '-mfar-mode' option, all references will be assumed to be 16 bits. This option may be abbreviated to '-mf'.

You can use the '-mcpu' option to specify a particular CPU. This option is equivalent to using the '.version' directive in the assembly code. For recognized CPU codes, see See Section 9.45.9 [.version], page 339. The default CPU version is '542'.

You can use the '-merrors-to-file' option to redirect error output to a file (this provided for those deficient environments which don't provide adequate output redirection). This option may be abbreviated to '-me'.

#### 9.45.2 Blocking

A blocked section or memory block is guaranteed not to cross the blocking boundary (usually a page, or 128 words) if it is smaller than the blocking size, or to start on a page boundary if it is larger than the blocking size.

#### 9.45.3 Environment Settings

'C54XDSP\_DIR' and 'A\_DIR' are semicolon-separated paths which are added to the list of directories normally searched for source and include files. 'C54XDSP\_DIR' will override 'A\_DIR'.

## 9.45.4 Constants Syntax

The TIC54X version of **as** allows the following additional constant formats, using a suffix to indicate the radix:

Binary	000000B, 011000b
Octal	10Q, 224q
Hexadecimal	45h, OFH

#### 9.45.5 String Substitution

A subset of allowable symbols (which we'll call subsyms) may be assigned arbitrary string values. This is roughly equivalent to C preprocessor #define macros. When **as** encounters one of these symbols, the symbol is replaced in the input stream by its string value. Subsym names **must** begin with a letter.

Subsyms may be defined using the .asg and .eval directives (See Section 9.45.9 [.asg], page 339, See Section 9.45.9 [.eval], page 339.

Expansion is recursive until a previously encountered symbol is seen, at which point substitution stops.

In this example, x is replaced with SYM2; SYM2 is replaced with SYM1, and SYM1 is replaced with x. At this point, x has already been encountered and the substitution stops.

.asg	"x",SYM1			
.asg	"SYM1",SYM2			
.asg	"SYM2",x			
add	x,a	; final code	e assembled is "add	x, a"

Macro parameters are converted to subsyms; a side effect of this is the normal **as** '\ARG' dereferencing syntax is unnecessary. Subsyms defined within a macro will have global scope, unless the .var directive is used to identify the subsym as a local macro variable see Section 9.45.9 [.var], page 339.

Substitution may be forced in situations where replacement might be ambiguous by placing colons on either side of the subsym. The following code:

```
.eval "10",x
LAB:X: add #x, a
When assembled becomes:
LAB10 add #10, a
```

Smaller parts of the string assigned to a subsym may be accessed with the following syntax:

```
:symbol(char_index):
```

Evaluates to a single-character string, the character at char\_index.

```
:symbol(start,length):
```

Evaluates to a substring of symbol beginning at start with length length.

#### 9.45.6 Local Labels

Local labels may be defined in two ways:

- \$N, where N is a decimal number between 0 and 9
- LABEL?, where LABEL is any legal symbol name.

Local labels thus defined may be redefined or automatically generated. The scope of a local label is based on when it may be undefined or reset. This happens when one of the following situations is encountered:

- .newblock directive see Section 9.45.9 [.newblock], page 339,
- The current section is changed (.sect, .text, or .data)
- Entering or leaving an included file
- The macro scope where the label was defined is exited

## 9.45.7 Math Builtins

The following built-in functions may be used to generate a floating-point value. All return a floating-point value except '\$cvi', '\$int', and '\$sgn', which return an integer value.

```
$acos(expr)
```

Returns the floating point arccosine of expr.

#### \$asin(expr)

Returns the floating point arcsine of expr.

#### \$atan(expr)

Returns the floating point arctangent of expr.

<pre>\$atan2(expr1,expr2) Returns the floating point arctangent of expr1 / expr2.</pre>
<pre>\$ceil(expr)</pre>
\$cosh(expr) Returns the floating point hyperbolic cosine of expr.
\$cos(expr) Returns the floating point cosine of expr.
<pre>\$cvf(expr) Returns the integer value expr converted to floating-point.</pre>
<pre>\$cvi(expr) Returns the floating point value expr converted to integer.</pre>
<pre>\$exp(expr) Returns the floating point value e ^ expr.</pre>
<pre>\$fabs(expr) Returns the floating point absolute value of expr.</pre>
<pre>\$floor(expr)</pre>
<pre>\$fmod(expr1,expr2) Returns the floating point remainder of expr1 / expr2.</pre>
<pre>\$int(expr) Returns 1 if expr evaluates to an integer, zero otherwise.</pre>
<pre>\$ldexp(expr1,expr2) Returns the floating point value expr1 * 2 ^ expr2.</pre>
<pre>\$log10(expr) Returns the base 10 logarithm of expr.</pre>
\$log(expr) Returns the natural logarithm of expr.
<pre>\$max(expr1,expr2) Returns the floating point maximum of expr1 and expr2.</pre>
<pre>\$min(expr1,expr2)</pre>
<pre>\$pow(expr1,expr2) Returns the floating point value expr1 ^ expr2.</pre>
<pre>\$round(expr) Returns the nearest integer to expr as a floating point number.</pre>
<pre>\$sgn(expr) Returns -1, 0, or 1 based on the sign of expr.</pre>

#### \$sin(expr)

Returns the floating point sine of expr.

#### \$sinh(expr)

Returns the floating point hyperbolic sine of expr.

#### \$sqrt(expr)

Returns the floating point square root of expr.

#### \$tan(expr)

Returns the floating point tangent of expr.

#### \$tanh(expr)

Returns the floating point hyperbolic tangent of expr.

#### \$trunc(expr)

Returns the integer value of expr truncated towards zero as floating point.

### 9.45.8 Extended Addressing

The LDX pseudo-op is provided for loading the extended addressing bits of a label or address. For example, if an address \_label resides in extended program memory, the value of \_label may be loaded as follows:

```
ldx #_label,16,a ; loads extended bits of _label
or #_label,a ; loads lower 16 bits of _label
bacc a ; full address is in accumulator A
```

### 9.45.9 Directives

.align [size]

- .even Align the section program counter on the next boundary, based on size. size may be any power of 2. .even is equivalent to .align with a size of 2.
  - 1 Align SPC to word boundary
  - 2 Align SPC to longword boundary (same as .even)
  - 128 Align SPC to page boundary
- .asg string, name

Assign name the string string. String replacement is performed on string before assignment.

#### .eval string, name

Evaluate the contents of string string and assign the result as a string to the subsym name. String replacement is performed on string before assignment.

#### .bss symbol, size [, [blocking\_flag] [,alignment\_flag]]

Reserve space for symbol in the .bss section. size is in words. If present, blocking\_flag indicates the allocated space should be aligned on a page boundary if it would otherwise cross a page boundary. If present, alignment\_flag causes the assembler to allocate size on a long word boundary.

```
.byte value [,...,value_n]
```

```
.ubyte value [,...,value_n]
```

```
.char value [,...,value_n]
```

```
.uchar value [,...,value_n]
```

Place one or more bytes into consecutive words of the current section. The upper 8 bits of each word is zero-filled. If a label is used, it points to the word allocated for the first byte encountered.

```
.clink ["section_name"]
```

Set STYP\_CLINK flag for this section, which indicates to the linker that if no symbols from this section are referenced, the section should not be included in the link. If *section\_name* is omitted, the current section is used.

```
.c_mode TBD.
```

```
.copy "filename" | filename
```

```
.include "filename" | filename
```

Read source statements from *filename*. The normal include search path is used. Normally .copy will cause statements from the included file to be printed in the assembly listing and .include will not, but this distinction is not currently implemented.

.data Begin assembling code into the .data section.

```
.double value [,...,value_n]
```

.ldouble value [,...,value\_n]

```
.float value [,...,value_n]
```

.xfloat value [,...,value\_n]

Place an IEEE single-precision floating-point representation of one or more floating-point values into the current section. All but .xfloat align the result on a longword boundary. Values are stored most-significant word first.

## .drlist

.drnolist

Control printing of directives to the listing file. Ignored.

```
.emsg string
```

```
.mmsg string
```

```
.wmsg string
```

Emit a user-defined error, message, or warning, respectively.

```
.far_mode
```

Use extended addressing when assembling statements. This should appear only once per file, and is equivalent to the -mfar-mode option see Section 9.45.1 [-mfar-mode], page 336.

```
.fclist
```

```
.fcnolist
```

Control printing of false conditional blocks to the listing file.

```
.field value [,size]
```

Initialize a bitfield of *size* bits in the current section. If *value* is relocatable, then *size* must be 16. *size* defaults to 16 bits. If *value* does not fit into *size* 

bits, the value will be truncated. Successive .field directives will pack starting at the current word, filling the most significant bits first, and aligning to the start of the next word if the field size does not fit into the space remaining in the current word. A .align directive with an operand of 1 will force the next .field directive to begin packing into a new word. If a label is used, it points to the word that contains the specified field.

```
.global symbol [,..., symbol_n]
```

```
.def symbol [,...,symbol_n]
```

```
.ref symbol [,...,symbol_n]
```

.def nominally identifies a symbol defined in the current file and available to other files. .ref identifies a symbol used in the current file but defined elsewhere. Both map to the standard .global directive.

```
.half value [,...,value_n]
```

```
.uhalf value [,...,value_n]
```

```
.short value [,...,value_n]
```

```
.ushort value [,...,value_n]
```

```
.int value [,...,value_n]
```

```
.uint value [,...,value_n]
```

```
.word value [,...,value_n]
```

```
.uword value [,...,value_n]
```

Place one or more values into consecutive words of the current section. If a label is used, it points to the word allocated for the first value encountered.

#### .label symbol

Define a special *symbol* to refer to the load time address of the current section program counter.

```
.length
```

.width Set the page length and width of the output listing file. Ignored.

```
.list
```

.nolist Control whether the source listing is printed. Ignored.

```
.long value [,...,value_n]
```

```
.ulong value [,...,value_n]
```

```
.xlong value [,...,value_n]
```

Place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. .long and .ulong align the result on a longword boundary; xlong does not.

## .loop [count]

.break [condition]

.endloop Repeatedly assemble a block of code. .loop begins the block, and .endloop marks its termination. *count* defaults to 1024, and indicates the number of times the block should be repeated. .break terminates the loop so that assembly begins after the .endloop directive. The optional *condition* will cause the loop to terminate only if it evaluates to zero.

macro_name .macro	[param1][,	param_n]
-------------------	------------	----------

[.mexit]

- .endm See the section on macros for more explanation (See Section 9.45.10 [TIC54X-Macros], page 344.
- .mlib "filename" | filename

Load the macro library filename. filename must be an archived library (BFD ar-compatible) of text files, expected to contain only macro definitions. The standard include search path is used.

#### .mlist

- .mnolist Control whether to include macro and loop block expansions in the listing output. Ignored.
- .mmregs Define global symbolic names for the 'c54x registers. Supposedly equivalent to executing .set directives for each register with its memory-mapped value, but in reality is provided only for compatibility and does nothing.

#### .newblock

This directive resets any TIC54X local labels currently defined. Normal **as** local labels are unaffected.

#### .option option\_list

Set listing options. Ignored.

```
.sblock "section_name" | section_name [,"name_n" | name_n]
```

Designate section\_name for blocking. Blocking guarantees that a section will start on a page boundary (128 words) if it would otherwise cross a page boundary. Only initialized sections may be designated with this directive. See also See Section 9.45.2 [TIC54X-Block], page 336.

```
.sect "section_name"
```

Define a named initialized section and make it the current section.

- symbol .set "value"
- symbol .equ "value"

Equate a constant value to a symbol, which is placed in the symbol table. symbol may not be previously defined.

#### .space size\_in\_bits

#### .bes size\_in\_bits

Reserve the given number of bits in the current section and zero-fill them. If a label is used with .space, it points to the **first** word reserved. With .bes, the label points to the **last** word reserved.

```
.sslist
```

```
.ssnolist
```

Controls the inclusion of subsym replacement in the listing output. Ignored.

```
.string "string" [,..., "string_n"]
```

```
.pstring "string" [,..., "string_n"]
```

Place 8-bit characters from *string* into the current section. .string zero-fills the upper 8 bits of each word, while .pstring puts two characters into each

is used, it points to the first word initialized.

word, filling the most-significant bits first. Unused space is zero-filled. If a label

```
[stag] .struct [offset]
[name_1] element [count_1]
[name_2] element [count_2]
[tname] .tag stagx [tcount]
...
[name_n] element [count_n]
[ssize] .endstruct
label .tag [stag]
```

Assign symbolic offsets to the elements of a structure. *stag* defines a symbol to use to reference the structure. *offset* indicates a starting value to use for the first element encountered; otherwise it defaults to zero. Each element can have a named offset, *name*, which is a symbol assigned the value of the element's offset into the structure. If *stag* is missing, these become global symbols. *count* adjusts the offset that many times, as if element were an array. element may be one of .byte, .word, .long, .float, or any equivalent of those, and the structure offset is adjusted accordingly. .field and .string are also allowed; the size of .field is one bit, and .string is considered to be one word in size. Only element descriptors, structure/union tags, .align and conditional assembly directives are allowed within .struct/.endstruct. .align aligns member offsets to word boundaries only. *ssize*, if provided, will always be assigned the size of the structure.

The .tag directive, in addition to being used to define a structure/union element within a structure, may be used to apply a structure to a symbol. Once applied to *label*, the individual structure elements may be applied to *label* to produce the desired offsets using *label* as the structure base.

.tab Set the tab size in the output listing. Ignored.

```
[utag] .union
[name_1] element [count_1]
[name_2] element [count_2]
[tname] .tag utagx[,tcount]
...
[name_n] element [count_n]
[usize] .endstruct
label .tag [utag]
Similar to .struct, but the offset after each element is reset to zero, and the
usize is set to the maximum of all defined elements. Starting offset for the union
is always zero.
[symbol] .usect "section_name", size, [,[blocking_flag] [,alignment_flag]]
```

[symbol] .usect "section\_name", size, [,[blocking\_flag] [,alignment\_flag]]
Reserve space for variables in a named, uninitialized section (similar to .bss).
.usect allows definitions sections independent of .bss. symbol points to the
first location reserved by this allocation. The symbol may be used as a variable
name. size is the allocated size in words. blocking\_flag indicates whether to
block this section on a page boundary (128 words) (see Section 9.45.2 [TIC54X-

Block], page 336). alignment flag indicates whether the section should be longword-aligned.

.var sym[,..., sym\_n]

Define a subsym to be a local variable within a macro. See See Section 9.45.10 [TIC54X-Macros], page 344.

#### .version version

Set which processor to build instructions for. Though the following values are accepted, the op is ignored.

## 9.45.10 Macros

Macros do not require explicit dereferencing of arguments (i.e., \ARG).

During macro expansion, the macro parameters are converted to subsyms. If the number of arguments passed the macro invocation exceeds the number of parameters defined, the last parameter is assigned the string equivalent of all remaining arguments. If fewer arguments are given than parameters, the missing parameters are assigned empty strings. To include a comma in an argument, you must enclose the argument in quotes.

The following built-in subsym functions allow examination of the string value of subsyms (or ordinary strings). The arguments are strings unless otherwise indicated (subsyms passed as args will be replaced by the strings they represent).

```
$symlen(str)
Returns the length of str.
```

```
$symcmp(str1,str2)
```

Returns 0 if str1 = str2, non-zero otherwise.

```
$firstch(str,ch)
```

Returns index of the first occurrence of character constant ch in str.

```
$lastch(str,ch)
```

Returns index of the last occurrence of character constant ch in str.

```
$isdefed(symbol)
```

Returns zero if the symbol symbol is not in the symbol table, non-zero otherwise.

### \$ismember(symbol,list)

Assign the first member of comma-separated string *list* to symbol; *list* is reassigned the remainder of the list. Returns zero if *list* is a null string. Both arguments must be subsyms.

#### \$iscons(expr)

Returns 1 if string expr is binary, 2 if octal, 3 if hexadecimal, 4 if a character, 5 if decimal, and zero if not an integer.

#### \$isname(name)

Returns 1 if *name* is a valid symbol name, zero otherwise.

#### \$isreg(reg)

Returns 1 if reg is a valid predefined register name (AR0-AR7 only).

#### \$structsz(stag)

Returns the size of the structure or union represented by stag.

### \$structacc(stag)

Returns the reference point of the structure or union represented by *stag*. Always returns zero.

## 9.45.11 Memory-mapped Registers

The following symbols are recognized as memory-mapped registers:

# 9.45.12 TIC54X Syntax

### 9.45.12.1 Special Characters

The presence of a ';' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The presence of an asterisk ('\*') at the start of a line also indicates a comment that extends to the end of that line.

The TIC54X assembler does not currently support a line separator character.

# 9.46 TIC6X Dependent Features

# 9.46.1 TIC6X Options

#### -march=arch

Enable (only) instructions from architecture *arch*. By default, all instructions are permitted.

The following values of arch are accepted: c62x, c64x, c64x+, c67x, c67x+, c67x+, c674x.

### -mdsbt

-mno-dsbt

The -mdsbt option causes the assembler to generate the Tag\_ABI\_DSBT attribute with a value of 1, indicating that the code is using DSBT addressing. The -mno-dsbt option, the default, causes the tag to have a value of 0, indicating that the code does not use DSBT addressing. The linker will emit a warning if objects of different type (DSBT and non-DSBT) are linked together.

-mpid=no

#### -mpid=near

-mpid=far

The -mpid= option causes the assembler to generate the Tag\_ABI\_PID attribute with a value indicating the form of data addressing used by the code. -mpid=no, the default, indicates position-dependent data addressing, -mpid=near indicates position-independent addressing with GOT accesses using near DP addressing, and -mpid=far indicates position-independent addressing with GOT accesses using far DP addressing. The linker will emit a warning if objects built with different settings of this option are linked together.

#### -mpic

-mno-pic The -mpic option causes the assembler to generate the Tag\_ABI\_PIC attribute with a value of 1, indicating that the code is using position-independent code addressing, The -mno-pic option, the default, causes the tag to have a value of 0, indicating position-dependent code addressing. The linker will emit a warning if objects of different type (position-dependent and position-independent) are linked together.

#### -mbig-endian

```
-mlittle-endian
```

Generate code for the specified endianness. The default is little-endian.

# 9.46.2 TIC6X Syntax

The presence of a ';' on a line indicates the start of a comment that extends to the end of the current line. If a '#' or '\*' appears as the first character of a line, the whole line is treated as a comment. Note that if a line starts with a '#' character then it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The '@' character can be used instead of a newline to separate statements.

Instruction, register and functional unit names are case-insensitive. as requires fullyspecified functional unit names, such as '.S1', '.L1X' or '.D1T2', on all instructions using a functional unit.

For some instructions, there may be syntactic ambiguity between register or functional unit names and the names of labels or other symbols. To avoid this, enclose the ambiguous symbol name in parentheses; register and functional unit names may not be enclosed in parentheses.

# 9.46.3 TIC6X Directives

Directives controlling the set of instructions accepted by the assembler have effect for instructions between the directive and any subsequent directive overriding it.

#### .arch arch

This has the same effect as -march=arch.

.cantunwind

Prevents unwinding through the current function. No personality routine or exception table data is required or permitted.

If this is not specified then frame unwinding information will be constructed from CFI directives. see Section 7.12 [CFI directives], page 54.

#### .c6xabi\_attribute tag, value

Set the C6000 EABI build attribute tag to value.

The tag is either an attribute number or one of Tag\_ISA, Tag\_ABI\_wchar\_t, Tag\_ABI\_stack\_align\_needed, Tag\_ABI\_stack\_align\_preserved, Tag\_ABI\_ DSBT, Tag\_ABI\_PID, Tag\_ABI\_PIC, TAG\_ABI\_array\_object\_alignment, TAG\_ ABI\_array\_object\_align\_expected, Tag\_ABI\_compatibility and Tag\_ABI\_ conformance. The value is either a number, "string", or number, "string" depending on the tag.

#### .ehtype symbol

Output an exception type table reference to symbol.

.endp Marks the end of and exception table or function. If preceded by a .handlerdata directive then this also switched back to the previous text section.

#### .handlerdata

Marks the end of the current function, and the start of the exception table entry for that function. Anything between this directive and the .endp directive will be added to the exception table entry.

Must be preceded by a CFI block containing a .cfi\_lsda directive.

.nocmp Disallow use of C64x+ compact instructions in the current text section.

#### .personalityindex index

Sets the personality routine for the current function to the ABI specified compact routine number *index* 

### .personality name

Sets the personality routine for the current function to name.

# .scomm symbol, size, align

Like .comm, creating a common symbol symbol with size size and alignment align, but unlike when using .comm, this symbol will be placed into the small BSS section by the linker.

# 9.47 TILE-Gx Dependent Features

## 9.47.1 Options

The following table lists all available TILE-Gx specific options:

-m32 | -m64

Select the word size, either 32 bits or 64 bits.

-EB | -EL Select the endianness, either big-endian (-EB) or little-endian (-EL).

# 9.47.2 Syntax

Block comments are delimited by '/\*' and '\*/'. End of line comments may be introduced by '#'.

Instructions consist of a leading opcode or macro name followed by whitespace and an optional comma-separated list of operands:

opcode [operand, ...]

Instructions must be separated by a newline or semicolon.

There are two ways to write code: either write naked instructions, which the assembler is free to combine into VLIW bundles, or specify the VLIW bundles explicitly.

Bundles are specified using curly braces:

{ add r3,r4,r5 ; add r7,r8,r9 ; lw r10,r11 }

A bundle can span multiple lines. If you want to put multiple instructions on a line, whether in a bundle or not, you need to separate them with semicolons as in this example.

A bundle may contain one or more instructions, up to the limit specified by the ISA (currently three). If fewer instructions are specified than the hardware supports in a bundle, the assembler inserts **fnop** instructions automatically.

The assembler will prefer to preserve the ordering of instructions within the bundle, putting the first instruction in a lower-numbered pipeline than the next one, etc. This fact, combined with the optional use of explicit fnop or nop instructions, allows precise control over which pipeline executes each instruction.

If the instructions cannot be bundled in the listed order, the assembler will automatically try to find a valid pipeline assignment. If there is no way to bundle the instructions together, the assembler reports an error.

The assembler does not yet auto-bundle (automatically combine multiple instructions into one bundle), but it reserves the right to do so in the future. If you want to force an instruction to run by itself, put it in a bundle explicitly with curly braces and use **nop** instructions (not **fnop**) to fill the remaining pipeline slots in that bundle.

## 9.47.2.1 Opcode Names

For a complete list of opcodes and descriptions of their semantics, see *TILE-Gx Instruction* Set Architecture, available upon request at www.tilera.com.

# 9.47.2.2 Register Names

General-purpose registers are represented by predefined symbols of the form 'rN', where N represents a number between 0 and 63. However, the following registers have canonical names that must be used instead:

r54	$\operatorname{sp}$	
r55	lr	
r56	$\operatorname{sn}$	
r57	idn0	
r58	idn1	
r59	udn0	
r60	udn1	
r61	udn2	
r62	udn3	
r63	zero	

The assembler will emit a warning if a numeric name is used instead of the non-numeric name. The .no\_require\_canonical\_reg\_names assembler pseudo-op turns off this warning. .require\_canonical\_reg\_names turns it back on.

# 9.47.2.3 Symbolic Operand Modifiers

The assembler supports several modifiers when using symbol addresses in TILE-Gx instruction operands. The general syntax is the following:

modifier(symbol)

The following modifiers are supported:

hw0	
	This modifier is used to load bits 0-15 of the symbol's address.
hw1	
	This modifier is used to load bits 16-31 of the symbol's address.
hw2	
	This modifier is used to load bits 32-47 of the symbol's address.
hw3	
	This modifier is used to load bits 48-63 of the symbol's address.
hw0_last	
	This modifier yields the same value as $hw0,$ but it also checks that the value does not overflow.
hw1_last	
	This modifier yields the same value as $hw1$ , but it also checks that the value does not overflow.

### hw2\_last

This modifier yields the same value as hw2, but it also checks that the value does not overflow.

A 48-bit symbolic value is constructed by using the following idiom:

moveli r0, hw2\_last(sym)
shl16insli r0, r0, hw1(sym)
shl16insli r0, r0, hw0(sym)

#### hw0\_got

This modifier is used to load bits 0-15 of the symbol's offset in the GOT entry corresponding to the symbol.

### hw0\_last\_got

This modifier yields the same value as hw0\_got, but it also checks that the value does not overflow.

#### hw1\_last\_got

This modifier is used to load bits 16-31 of the symbol's offset in the GOT entry corresponding to the symbol, and it also checks that the value does not overflow.

#### plt

This modifier is used for function symbols. It causes a *procedure linkage table*, an array of code stubs, to be created at the time the shared object is created or linked against, together with a global offset table entry. The value is a pc-relative offset to the corresponding stub code in the procedure linkage table. This arrangement causes the run-time symbol resolver to be called to look up and set the value of the symbol the first time the function is called (at latest; depending environment variables). It is only safe to leave the symbol unresolved this way if all references are function calls.

## hw0\_plt

This modifier is used to load bits 0-15 of the pc-relative address of a plt entry.

## hw1\_plt

This modifier is used to load bits 16-31 of the pc-relative address of a plt entry.

#### hw1\_last\_plt

This modifier yields the same value as hw1\_plt, but it also checks that the value does not overflow.

#### hw2\_last\_plt

This modifier is used to load bits 32-47 of the pc-relative address of a plt entry, and it also checks that the value does not overflow.

### hw0\_tls\_gd

This modifier is used to load bits 0-15 of the offset of the GOT entry of the symbol's TLS descriptor, to be used for general-dynamic TLS accesses.

#### hw0\_last\_tls\_gd

This modifier yields the same value as hw0\_tls\_gd, but it also checks that the value does not overflow.

### hw1\_last\_tls\_gd

This modifier is used to load bits 16-31 of the offset of the GOT entry of the symbol's TLS descriptor, to be used for general-dynamic TLS accesses. It also checks that the value does not overflow.

### hw0\_tls\_ie

This modifier is used to load bits 0-15 of the offset of the GOT entry containing the offset of the symbol's address from the TCB, to be used for initial-exec TLS accesses.

### hw0\_last\_tls\_ie

This modifier yields the same value as hw0\_tls\_ie, but it also checks that the value does not overflow.

### hw1\_last\_tls\_ie

This modifier is used to load bits 16-31 of the offset of the GOT entry containing the offset of the symbol's address from the TCB, to be used for initial-exec TLS accesses. It also checks that the value does not overflow.

### hw0\_tls\_le

This modifier is used to load bits 0-15 of the offset of the symbol's address from the TCB, to be used for local-exec TLS accesses.

### hw0\_last\_tls\_le

This modifier yields the same value as hw0\_tls\_le, but it also checks that the value does not overflow.

## hw1\_last\_tls\_le

This modifier is used to load bits 16-31 of the offset of the symbol's address from the TCB, to be used for local-exec TLS accesses. It also checks that the value does not overflow.

## tls\_gd\_call

This modifier is used to tag an instruction as the "call" part of a calling sequence for a TLS GD reference of its operand.

### tls\_gd\_add

This modifier is used to tag an instruction as the "add" part of a calling sequence for a TLS GD reference of its operand.

### tls\_ie\_load

This modifier is used to tag an instruction as the "load" part of a calling sequence for a TLS IE reference of its operand.

# 9.47.3 TILE-Gx Directives

# .align expression [, expression]

This is the generic .*align* directive. The first argument is the requested alignment in bytes.

### .allow\_suspicious\_bundles

Turns on error checking for combinations of instructions in a bundle that probably indicate a programming error. This is on by default.

### .no\_allow\_suspicious\_bundles

Turns off error checking for combinations of instructions in a bundle that probably indicate a programming error.

### .require\_canonical\_reg\_names

Require that canonical register names be used, and emit a warning if the numeric names are used. This is on by default.

## .no\_require\_canonical\_reg\_names

Permit the use of numeric names for registers that have canonical names.

# 9.48 TILEPro Dependent Features

# 9.48.1 Options

as has no machine-dependent command-line options for TILEPro.

# 9.48.2 Syntax

Block comments are delimited by '/\*' and '\*/'. End of line comments may be introduced by '#'.

Instructions consist of a leading opcode or macro name followed by whitespace and an optional comma-separated list of operands:

opcode [operand, ...]

Instructions must be separated by a newline or semicolon.

There are two ways to write code: either write naked instructions, which the assembler is free to combine into VLIW bundles, or specify the VLIW bundles explicitly.

Bundles are specified using curly braces:

{ add r3,r4,r5 ; add r7,r8,r9 ; lw r10,r11 }

A bundle can span multiple lines. If you want to put multiple instructions on a line, whether in a bundle or not, you need to separate them with semicolons as in this example.

A bundle may contain one or more instructions, up to the limit specified by the ISA (currently three). If fewer instructions are specified than the hardware supports in a bundle, the assembler inserts **fnop** instructions automatically.

The assembler will prefer to preserve the ordering of instructions within the bundle, putting the first instruction in a lower-numbered pipeline than the next one, etc. This fact, combined with the optional use of explicit fnop or nop instructions, allows precise control over which pipeline executes each instruction.

If the instructions cannot be bundled in the listed order, the assembler will automatically try to find a valid pipeline assignment. If there is no way to bundle the instructions together, the assembler reports an error.

The assembler does not yet auto-bundle (automatically combine multiple instructions into one bundle), but it reserves the right to do so in the future. If you want to force an instruction to run by itself, put it in a bundle explicitly with curly braces and use **nop** instructions (not **fnop**) to fill the remaining pipeline slots in that bundle.

# 9.48.2.1 Opcode Names

For a complete list of opcodes and descriptions of their semantics, see *TILE Processor User Architecture Manual*, available upon request at www.tilera.com.

# 9.48.2.2 Register Names

General-purpose registers are represented by predefined symbols of the form 'rN', where N represents a number between 0 and 63. However, the following registers have canonical names that must be used instead:

r54	$\operatorname{sp}$
r55	lr

r56	$\operatorname{sn}$
r57	idn0
r58	idn1
r59	udn0
r60	udn1
r61	udn2
r62	udn3
r63	zero

The assembler will emit a warning if a numeric name is used instead of the canonical name. The .no\_require\_canonical\_reg\_names assembler pseudo-op turns off this warning. .require\_canonical\_reg\_names turns it back on.

# 9.48.2.3 Symbolic Operand Modifiers

The assembler supports several modifiers when using symbol addresses in TILEPro instruction operands. The general syntax is the following:

modifier(symbol)

The following modifiers are supported:

### lo16

This modifier is used to load the low 16 bits of the symbol's address, signextended to a 32-bit value (sign-extension allows it to be range-checked against signed 16 bit immediate operands without complaint).

### hi16

This modifier is used to load the high 16 bits of the symbol's address, also sign-extended to a 32-bit value.

### ha16

ha16(N) is identical to hi16(N), except if lo16(N) is negative it adds one to the hi16(N) value. This way lo16 and ha16 can be added to create any 32-bit value using auli. For example, here is how you move an arbitrary 32-bit address into r3:

moveli r3, lo16(sym)
auli r3, r3, ha16(sym)

#### got

This modifier is used to load the offset of the GOT entry corresponding to the symbol.

### got\_lo16

This modifier is used to load the sign-extended low 16 bits of the offset of the GOT entry corresponding to the symbol.

#### got\_hi16

This modifier is used to load the sign-extended high 16 bits of the offset of the GOT entry corresponding to the symbol.

### got\_ha16

This modifier is like got\_hi16, but it adds one if got\_lo16 of the input value is negative.

## plt

This modifier is used for function symbols. It causes a *procedure linkage table*, an array of code stubs, to be created at the time the shared object is created or linked against, together with a global offset table entry. The value is a pc-relative offset to the corresponding stub code in the procedure linkage table. This arrangement causes the run-time symbol resolver to be called to look up and set the value of the symbol the first time the function is called (at latest; depending environment variables). It is only safe to leave the symbol unresolved this way if all references are function calls.

### tls\_gd

This modifier is used to load the offset of the GOT entry of the symbol's TLS descriptor, to be used for general-dynamic TLS accesses.

tls\_gd\_lo16

This modifier is used to load the sign-extended low 16 bits of the offset of the GOT entry of the symbol's TLS descriptor, to be used for general dynamic TLS accesses.

### tls\_gd\_hi16

This modifier is used to load the sign-extended high 16 bits of the offset of the GOT entry of the symbol's TLS descriptor, to be used for general dynamic TLS accesses.

#### tls\_gd\_ha16

This modifier is like tls\_gd\_hi16, but it adds one to the value if tls\_gd\_lo16 of the input value is negative.

#### tls\_ie

This modifier is used to load the offset of the GOT entry containing the offset of the symbol's address from the TCB, to be used for initial-exec TLS accesses.

### tls\_ie\_lo16

This modifier is used to load the low 16 bits of the offset of the GOT entry containing the offset of the symbol's address from the TCB, to be used for initial-exec TLS accesses.

### tls\_ie\_hi16

This modifier is used to load the high 16 bits of the offset of the GOT entry containing the offset of the symbol's address from the TCB, to be used for initial-exec TLS accesses.

#### tls\_ie\_ha16

This modifier is like tls\_ie\_hi16, but it adds one to the value if tls\_ie\_lo16 of the input value is negative.

#### tls\_le

This modifier is used to load the offset of the symbol's address from the TCB, to be used for local-exec TLS accesses.

#### tls\_le\_lo16

This modifier is used to load the low 16 bits of the offset of the symbol's address from the TCB, to be used for local-exec TLS accesses.

#### tls\_le\_hi16

This modifier is used to load the high 16 bits of the offset of the symbol's address from the TCB, to be used for local-exec TLS accesses.

#### tls\_le\_ha16

This modifier is like tls\_le\_hi16, but it adds one to the value if tls\_le\_lo16 of the input value is negative.

#### tls\_gd\_call

This modifier is used to tag an instruction as the "call" part of a calling sequence for a TLS GD reference of its operand.

### tls\_gd\_add

This modifier is used to tag an instruction as the "add" part of a calling sequence for a TLS GD reference of its operand.

#### tls\_ie\_load

This modifier is used to tag an instruction as the "load" part of a calling sequence for a TLS IE reference of its operand.

# 9.48.3 TILEPro Directives

#### .align expression [, expression]

This is the generic .*align* directive. The first argument is the requested alignment in bytes.

### .allow\_suspicious\_bundles

Turns on error checking for combinations of instructions in a bundle that probably indicate a programming error. This is on by default.

#### .no\_allow\_suspicious\_bundles

Turns off error checking for combinations of instructions in a bundle that probably indicate a programming error.

### .require\_canonical\_reg\_names

Require that canonical register names be used, and emit a warning if the numeric names are used. This is on by default.

#### .no\_require\_canonical\_reg\_names

Permit the use of numeric names for registers that have canonical names.

# 9.49 v850 Dependent Features

## 9.49.1 Options

as supports the following additional command-line options for the V850 processor family:

### -wsigned\_overflow

Causes warnings to be produced when signed immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

### -wunsigned\_overflow

Causes warnings to be produced when unsigned immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

- -mv850 Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- -mv850e Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- -mv850e1 Specifies that the assembled code should be marked as being targeted at the V850E1 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- -mv850any

Specifies that the assembled code should be marked as being targeted at the V850 processor but support instructions that are specific to the extended variants of the process. This allows the production of binaries that contain target specific code, but which are also intended to be used in a generic fashion. For example libgcc.a contains generic routines used by the code produced by GCC for all versions of the v850 architecture, together with support routines only used by the V850E architecture.

- -mv850e2 Specifies that the assembled code should be marked as being targeted at the V850E2 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- -mv850e2v3

Specifies that the assembled code should be marked as being targeted at the V850E2V3 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

#### -mv850e2v4

This is an alias for -mv850e3v5.

-mv850e3v5

Specifies that the assembled code should be marked as being targeted at the V850E3V5 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

-mrelax Enables relaxation. This allows the .longcall and .longjump pseudo ops to be used in the assembler source code. These ops label sections of code which are

either a long function call or a long branch. The assembler will then flag these sections of code and the linker will attempt to relax them.

#### -mgcc-abi

Marks the generated object file as supporting the old GCC ABI.

#### -mrh850-abi

Marks the generated object file as supporting the RH850 ABI. This is the default.

### -m8byte-align

Marks the generated object file as supporting a maximum 64-bits of alignment for variables defined in the source code.

#### -m4byte-align

Marks the generated object file as supporting a maximum 32-bits of alignment for variables defined in the source code. This is the default.

#### -msoft-float

Marks the generated object file as not using any floating point instructions and hence can be linked with other V850 binaries that do or do not use floating point. This is the default for binaries for architectures earlier than the e2v3.

#### -mhard-float

Marks the generated object file as one that uses floating point instructions - and hence can only be linked with other V850 binaries that use the same kind of floating point instructions, or with binaries that do not use floating point at all. This is the default for binaries the e2v3 and later architectures.

## 9.49.2 Syntax

# 9.49.2.1 Special Characters

'#' is the line comment character. If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Two dashes ('--') can also be used to start a line comment.

The ';' character can be used to separate statements on the same line.

# 9.49.2.2 Register Names

as supports the following names for registers:

```
general register 0
r0, zero
general register 1
r1
general register 2
r2, hp
```

general register 3 r3, spgeneral register 4 r4, gp general register 5 r5, tpgeneral register 6 r6 general register 7 r7general register 8 r8 general register 9 r9 general register 10 r10 general register 11 r11 general register 12 r12general register 13 r13 general register 14 r14general register 15 r15general register 16 r16general register 17 r17general register 18 r18general register 19 r19 general register 20 r20general register 21 r21

general register 22 r22general register 23 r23 general register 24 r24general register 25 r25general register 26 r26 general register 27 r27 general register 28 r28general register 29 r29 general register 30 r30, ep general register 31 r31, lp system register 0 eipc system register 1 eipsw system register 2 fepc system register 3 fepsw system register 4  $\operatorname{ecr}$ system register 5 psw system register 16 ctpc system register 17 ctpswsystem register 18 dbpc

```
system register 19
dbpsw
system register 20
```

ctbp

# 9.49.3 Floating Point

The V850 family uses IEEE floating-point numbers.

# 9.49.4 V850 Machine Directives

### .offset <expression>

Moves the offset into the current section to the specified amount.

### .section "name", <type>

This is an extension to the standard .section directive. It sets the current section to be <type> and creates an alias for this section called "name".

- .v850 Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- .v850e Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- .v850e1 Specifies that the assembled code should be marked as being targeted at the V850E1 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- .v850e2 Specifies that the assembled code should be marked as being targeted at the V850E2 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.
- .v850e2v3

Specifies that the assembled code should be marked as being targeted at the V850E2V3 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

.v850e2v4

Specifies that the assembled code should be marked as being targeted at the V850E3V5 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

.v850e3v5

Specifies that the assembled code should be marked as being targeted at the V850E3V5 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

# 9.49.5 Opcodes

as implements all the standard V850 opcodes.

as also implements the following pseudo ops:

```
hi0()
            Computes the higher 16 bits of the given expression and stores it into the
            immediate operand field of the given instruction. For example:
            'mulhi hi0(here - there), r5, r6'
            computes the difference between the address of labels 'here' and 'there', takes
            the upper 16 bits of this difference, shifts it down 16 bits and then multiplies it
            by the lower 16 bits in register 5, putting the result into register 6.
10()
            Computes the lower 16 bits of the given expression and stores it into the
            immediate operand field of the given instruction. For example:
            'addi lo(here - there), r5, r6'
            computes the difference between the address of labels 'here' and 'there', takes
            the lower 16 bits of this difference and adds it to register 5, putting the result
            into register 6.
hi()
            Computes the higher 16 bits of the given expression and then adds the value
            of the most significant bit of the lower 16 bits of the expression and stores the
            result into the immediate operand field of the given instruction. For example
            the following code can be used to compute the address of the label 'here' and
            store it into register 6:
            'movhi hi(here), r0, r6' 'movea lo(here), r6, r6'
            The reason for this special behaviour is that move performs a sign extension on
            its immediate operand. So for example if the address of 'here' was 0xFFFFFFF
            then without the special behaviour of the hi() pseudo-op the movhi instruction
```

hilo() Computes the 32 bit value of the given expression and stores it into the immediate operand field of the given instruction (which must be a mov instruction). For example:

'mov hilo(here), r6'

computes the absolute address of label 'here' and puts the result into register 6.

sdaoff() Computes the offset of the named variable from the start of the Small Data Area (whose address is held in register 4, the GP register) and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

'ld.w sdaoff(\_a\_variable)[gp],r6'

loads the contents of the location pointed to by the label '\_a\_variable' into register 6, provided that the label is located somewhere within +/- 32K of the address held in the GP register. [Note the linker assumes that the GP register contains a fixed address set to the address of the label called '\_gp'. This can

either be set up automatically by the linker, or specifically set by using the '--defsym \_\_gp=<value>' command-line option].

tdaoff() Computes the offset of the named variable from the start of the Tiny Data Area (whose address is held in register 30, the EP register) and stores the result as a 4,5, 7 or 8 bit unsigned value in the immediate operand field of the given instruction. For example:

'sld.w tdaoff(\_a\_variable)[ep],r6'

loads the contents of the location pointed to by the label '\_a\_variable' into register 6, provided that the label is located somewhere within +256 bytes of the address held in the EP register. [Note the linker assumes that the EP register contains a fixed address set to the address of the label called '\_\_ep'. This can either be set up automatically by the linker, or specifically set by using the '--defsym \_\_ep=<value>' command-line option].

zdaoff() Computes the offset of the named variable from address 0 and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

### 'movea zdaoff(\_a\_variable),zero,r6'

puts the address of the label '\_a\_variable' into register 6, assuming that the label is somewhere within the first 32K of memory. (Strictly speaking it also possible to access the last 32K of memory as well, as the offsets are signed).

ctoff() Computes the offset of the named variable from the start of the Call Table Area (whose address is held in system register 20, the CTBP register) and stores the result a 6 or 16 bit unsigned value in the immediate field of then given instruction or piece of data. For example:

'callt ctoff(table\_func1)'

will put the call the function whose address is held in the call table at the location labeled 'table\_func1'.

.longcall name

Indicates that the following sequence of instructions is a long call to function name. The linker will attempt to shorten this call sequence if name is within a 22bit offset of the call. Only valid if the -mrelax command-line switch has been enabled.

.longjump name

Indicates that the following sequence of instructions is a long jump to label name. The linker will attempt to shorten this code sequence if name is within a 22bit offset of the jump. Only valid if the -mrelax command-line switch has been enabled.

For information on the V850 instruction set, see V850 Family 32-/16-Bit single-Chip Microcontroller Architecture Manual from NEC. Ltd.

# 9.50 VAX Dependent Features

# 9.50.1 VAX Command-Line Options

The Vax version of **as** accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

#### -D (Debug)

- -S (Symbol Table)
- -T (Token Trace)

These are obsolete options used to debug old assemblers.

### -d (Displacement size for JUMPs)

This option expects a number following the '-d'. Like options that expect filenames, the number may immediately follow the '-d' (old standard) or constitute the whole of the command-line argument that follows '-d' (GNU standard).

### -V (Virtualize Interpass Temporary File)

Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. **as** always does this, so this option is redundant.

### -J (JUMPify Longer Branches)

Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

### -t (Temporary File Directory)

Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since **as** does not use a temporary disk file, this option makes no difference. '-t' needs exactly one filename.

The Vax version of the assembler accepts additional options when compiled for VMS:

'-h n' External symbol or section (used for global variables) names are not case sensitive on VAX/VMS and always mapped to upper case. This is contrary to the C language definition which explicitly distinguishes upper and lower case. To implement a standard conforming C compiler, names must be changed (mapped) to preserve the case information. The default mapping is to convert all lower case characters to uppercase and adding an underscore followed by a 6 digit hex value, representing a 24 digit binary value. The one digits in the binary value represent which characters are uppercase in the original symbol name.

The '-h n' option determines how we map names. This takes several values. No '-h' switch at all allows case hacking as described above. A value of zero ('-h0') implies names should be upper case, and inhibits the case hack. A value of 2 ('-h2') implies names should be all lower case, with no case hack. A value of 3 ('-h3') implies that case should be preserved. The value 1 is unused. The -H option directs **as** to display every mapped symbol during assembly.

Symbols whose names include a dollar sign '\$' are exceptions to the general name mapping. These symbols are normally only used to reference VMS library names. Such symbols are always mapped to upper case.

- '-+' The '-+' option causes **as** to truncate any symbol name larger than 31 characters. The '-+' option also prevents some code following the '\_main' symbol normally added to make the object file compatible with Vax-11 "C".
- '-1' This option is ignored for backward compatibility with as version 1.x.
- '-H' The '-H' option causes **as** to print every symbol which was changed by case mapping.

## 9.50.2 VAX Floating Point

Conversion of florums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit.

D, F, G and H floating point formats are understood.

Immediate floating literals (*e.g.* 'S' \*6.9') are rendered correctly. Again, rounding is towards zero in the boundary case.

The .float directive produces f format numbers. The .double directive produces d format numbers.

## 9.50.3 Vax Machine Directives

The Vax version of the assembler supports four directives for generating Vax floating point constants. They are described in the table below.

- .dfloat This expects zero or more flonums, separated by commas, and assembles Vax d format 64-bit floating point constants.
- .ffloat This expects zero or more florums, separated by commas, and assembles Vax f format 32-bit floating point constants.
- .gfloat This expects zero or more florums, separated by commas, and assembles Vax g format 64-bit floating point constants.
- .hfloat This expects zero or more florums, separated by commas, and assembles Vax h format 128-bit floating point constants.

## 9.50.4 VAX Opcodes

All DEC mnemonics are supported. Beware that case... instructions have exactly 3 operands. The dispatch table that follows the case... instruction should be made with .word statements. This is compatible with all unix assemblers we know of.

## 9.50.5 VAX Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches the target. Generally these mnemonics are made by substituting 'j' for 'b' at the start of a DEC mnemonic. This feature is included both for compatibility and to help compilers. If you do not need this feature, avoid these opcodes. Here are the mnemonics, and the code they can expand into.

jbsb	'Jsb' is already an instruction mnemonic, so we chose 'jbsb'.
	(byte displacement) bsbb
	(word displacement) bsbw
	(long displacement) jsb
jbr	
jr	Unconditional branch.
	(byte displacement) brb
	(word displacement) brw
	(long displacement) jmp
j <i>COND</i>	COND may be any one of the conditional branches neq, nequ, eql, eqlu, gtr, geq, lss, gtru, lequ, vc, vs, gequ, cc, lssu, cs. COND may also be one of the bit tests bs, bc, bss, bcs, bsc, bcc, bssi, bcci, lbs, lbc. NOTCOND is the opposite condition to COND.
	(byte displacement) bCOND
	<pre>(word displacement)     bNOTCOND foo ; brw ; foo:</pre>
	<pre>(long displacement)      bNOTCOND foo ; jmp ; foo:</pre>
jacbX	X may be one of b d f g h l w.
	(word displacement) OPCODE
	<pre>(long displacement)</pre>
jaobYYY	YYY may be one of lss leq.
jsob <i>ZZZ</i>	ZZZ may be one of geq gtr.
	(byte displacement) OPCODE
	<pre>(word displacement)</pre>

```
brb bar ;
                           foo: brw destination ;
                           bar:
          (long displacement)
                           OPCODE ..., foo ;
                           brb bar ;
                           foo: jmp destination ;
                           bar:
aobleg
aoblss
sobgeq
sobgtr
          (byte displacement)
                      OPCODE ...
          (word displacement)
                           OPCODE ..., foo ;
                           brb bar ;
                           foo: brw destination ;
                           bar:
          (long displacement)
                           OPCODE ..., foo ;
                           brb bar ;
                           foo: jmp destination ;
                           bar:
```

# 9.50.6 VAX Operands

The immediate character is '\$' for Unix compatibility, not '#' as DEC writes it.

The indirect character is '\*' for Unix compatibility, not '@' as DEC writes it.

The displacement sizing character is ''' (an accent grave) for Unix compatibility, not '~' as DEC writes it. The letter preceding ''' may have either case. 'G' is not understood, but all other letters (b i l s w) are understood.

Register names understood are r0 r1 r2 ... r15 ap fp sp pc. Upper and lower case letters are equivalent.

For instance

tstb \*w'\$4(r5)

Any expression is permitted in an operand. Operands are comma separated.

# 9.50.7 Not Supported on VAX

Vax bit fields can not be assembled with **as**. Someone can add the required code if they really need it.

# 9.50.8 VAX Syntax

# 9.50.8.1 Special Characters

The presence of a '#' appearing anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The ';' character can be used to separate statements on the same line.

# 9.51 Visium Dependent Features

# 9.51.1 Options

The Visium assembler implements one machine-specific option:

-mtune=arch

This option specifies the target architecture. If an attempt is made to assemble an instruction that will not execute on the target architecture, the assembler will issue an error message.

The following names are recognized: mcm24 mcm gr5 gr6

# 9.51.2 Syntax

# 9.51.2.1 Special Characters

Line comments are introduced either by the '!' character or by the ';' character appearing anywhere on a line.

A hash character ('#') as the first character on a line also marks the start of a line comment, but in this case it could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The Visium assembler does not currently support a line separator character.

# 9.51.2.2 Register Names

Registers can be specified either by using their canonical mnemonic names or by using their alias if they have one, for example 'sp'.

# 9.51.3 Opcodes

All the standard opcodes of the architecture are implemented, along with the following three pseudo-instructions: cmp, cmpc, move.

In addition, the following two illegal opcodes are implemented and used by the simulation:

stop 5-bit immediate, SourceA
trace 5-bit immediate, SourceA

# 9.52 WebAssembly Dependent Features

# 9.52.1 Notes

While WebAssembly provides its own module format for executables, this documentation describes how to use **as** to produce intermediate ELF object format files.

# 9.52.2 Syntax

The assembler syntax directly encodes sequences of opcodes as defined in the WebAssembly binary encoding specification at https://github.com/webassembly/spec/BinaryEncoding.md. Structured sexp-style expressions are not supported as input.

# 9.52.2.1 Special Characters

'#' and ';' are the line comment characters. Note that if '#' is the first character on a line then it can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

# 9.52.2.2 Relocations

Special relocations are available by using the '@plt', '@got', or '@got' suffixes after a constant expression, which correspond to the R\_ASMJS\_LEB128\_PLT, R\_ASMJS\_LEB128\_GOT, and R\_ASMJS\_LEB128\_GOT\_CODE relocations, respectively.

The '**@plt**' suffix is followed by a symbol name in braces; the symbol value is used to determine the function signature for which a PLT stub is generated. Currently, the symbol *name* is parsed from its last 'F' character to determine the argument count of the function, which is also necessary for generating a PLT stub.

# 9.52.2.3 Signatures

Function signatures are specified with the **signature** pseudo-opcode, followed by a simple function signature imitating a C++-mangled function type: F followed by an optional v, then a sequence of i, l, f, and d characters to mark i32, i64, f32, and f64 parameters, respectively; followed by a final E to mark the end of the function signature.

# 9.52.3 Floating Point

WebAssembly uses little-endian IEEE floating-point numbers.

# 9.52.4 Regular Opcodes

Ordinary instructions are encoded with the WebAssembly mnemonics as listed at: https://github.com/WebAssembly/design/blob/master/BinaryEncoding.md.

Opcodes are written directly in the order in which they are encoded, without going through an intermediate sexp-style expression as in the **was** format.

For "typed" opcodes (block, if, etc.), the type of the block is specified in square brackets following the opcode: if[i], if[].

# 9.52.5 WebAssembly Module Layout

as will only produce ELF output, not a valid WebAssembly module. It is possible to make as produce output in a single ELF section which becomes a valid WebAssembly module, but a linker script to do so may be preferable, as it doesn't require running the entire module through the assembler at once.

# 9.53 XGATE Dependent Features

## 9.53.1 XGATE Options

The Freescale XGATE version of **as** has a few machine dependent options.

- -mshort This option controls the ABI and indicates to use a 16-bit integer ABI. It has no effect on the assembled instructions. This is the default.
- -mlong This option controls the ABI and indicates to use a 32-bit integer ABI.

#### -mshort-double

This option controls the ABI and indicates to use a 32-bit float ABI. This is the default.

#### -mlong-double

This option controls the ABI and indicates to use a 64-bit float ABI.

#### --print-insn-syntax

You can use the '--print-insn-syntax' option to obtain the syntax description of the instruction when an error is detected.

#### --print-opcodes

The '--print-opcodes' option prints the list of all the instructions with their syntax. Once the list is printed as exits.

## 9.53.2 Syntax

In XGATE RISC syntax, the instruction name comes first and it may be followed by up to three operands. Operands are separated by commas (','). as will complain if too many operands are specified for a given instruction. The same will happen if you specified too few operands.

```
nop
ldl #23
CMP R1, R2
```

The presence of a ';' character or a '!' character anywhere on a line indicates the start of a comment that extends to the end of that line.

A '\*' or a '#' character at the start of a line also introduces a line comment, but these characters do not work elsewhere on the line. If the first character of the line is a '#' then as well as starting a comment, the line could also be logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The XGATE assembler does not currently support a line separator character.

The following addressing modes are understood for XGATE:

Inherent

Immediate 3 Bit Wide '#number' Immediate 4 Bit Wide

،،

'#number'

Immediate 8 Bit Wide '#number' Monadic Addressing 'reg' Dyadic Addressing 'reg, reg' Triadic Addressing 'reg, reg, reg' Relative Addressing 9 Bit Wide '\*symbol' Relative Addressing 10 Bit Wide '\*symbol' Index Register plus Immediate Offset 'reg, (reg, #number)' Index Register plus Register Offset 'reg, reg, reg' Index Register plus Register Offset with Post-increment 'reg, reg, reg+' Index Register plus Register Offset with Pre-decrement 'reg, reg, -reg' The register can be either 'R0', 'R1', 'R2', 'R3', 'R4', 'R5', 'R6' or 'R7'. Convene macro opcodes to deal with 16-bit values have been added. Immediate 16 Bit Wide '#number', or '\*symbol' For example:

ldw R1, #1024 ldw R3, timer ldw R1, (R1, #0) COM R1 stw R2, (R1, #0)

# 9.53.3 Assembler Directives

The XGATE version of **as** have the following specific assembler directives:

# 9.53.4 Floating Point

Packed decimal (P) format floating literals are not supported(yet).

The floating point formats generated by directives are these.

.float Single precision floating point constants.

.double Double precision floating point constants.

```
.extend
```

.ldouble Extended precision (long double) floating point constants.

# 9.53.5 Opcodes

# 9.54 XStormy16 Dependent Features

### 9.54.1 Syntax

# 9.54.1.1 Special Characters

'#' is the line comment character. If a '#' appears as the first character of a line, the whole line is treated as a comment, but in this case the line can also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

A semicolon (';') can be used to start a comment that extends from wherever the character appears on the line up to the end of the line.

The '|' character can be used to separate statements on the same line.

## 9.54.2 XStormy16 Machine Directives

#### .16bit\_pointers

Like the --16bit-pointers command-line option this directive indicates that the assembly code makes use of 16-bit pointers.

.32bit\_pointers

Like the --32bit-pointers command-line option this directive indicates that the assembly code makes use of 32-bit pointers.

#### .no\_pointers

Like the **--no-pointers** command-line option this directive indicates that the assembly code does not makes use pointers.

## 9.54.3 XStormy16 Pseudo-Opcodes

as implements all the standard XStormy16 opcodes.

as also implements the following pseudo ops:

**@lo()** Computes the lower 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:

```
'add r6, @lo(here - there)'
```

computes the difference between the address of labels 'here' and 'there', takes the lower 16 bits of this difference and adds it to register 6.

**Computes the higher 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:** 

'addc r7, @hi(here - there)'

computes the difference between the address of labels 'here' and 'there', takes the upper 16 bits of this difference, shifts it down 16 bits and then adds it, along with the carry bit, to the value in register 7.

# 9.55 Xtensa Dependent Features

This chapter covers features of the GNU assembler that are specific to the Xtensa architecture. For details about the Xtensa instruction set, please consult the *Xtensa Instruction Set Architecture (ISA) Reference Manual.* 

## 9.55.1 Command-line Options

--text-section-literals | --no-text-section-literals

Control the treatment of literal pools. The default is '--no-text-sectionliterals', which places literals in separate sections in the output file. This allows the literal pool to be placed in a data RAM/ROM. With '--textsection-literals', the literals are interspersed in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files, where the literals would otherwise be out of range of the L32R instructions in the text section. Literals are grouped into pools following .literal\_position directives or preceding ENTRY instructions. These options only affect literals referenced via PC-relative L32R instructions; literals for absolute mode L32R instructions are handled separately. See Section 9.55.5.4 [literal], page 384.

## --auto-litpools | --no-auto-litpools

Control the treatment of literal pools. The default is '--no-auto-litpools', which in the absence of '--text-section-literals' places literals in separate sections in the output file. This allows the literal pool to be placed in a data RAM/ROM. With '--auto-litpools', the literals are interspersed in the text section in order to keep them as close as possible to their references, explicit .literal\_position directives are not required. This may be necessary for very large functions, where single literal pool at the beginning of the function may not be reachable by L32R instructions at the end. These options only affect literals referenced via PC-relative L32R instructions; literals for absolute mode L32R instructions are handled separately. When used together with '--text-section-literals', '--auto-litpools' takes precedence. See Section 9.55.5.4 [literal], page 384.

### --absolute-literals | --no-absolute-literals

Indicate to the assembler whether L32R instructions use absolute or PC-relative addressing. If the processor includes the absolute addressing option, the default is to use absolute L32R relocations. Otherwise, only the PC-relative L32R relocations can be used.

--target-align | --no-target-align

Enable or disable automatic alignment to reduce branch penalties at some expense in code size. See Section 9.55.3.2 [Automatic Instruction Alignment], page 379. This optimization is enabled by default. Note that the assembler will always align instructions like LOOP that have fixed alignment requirements.

### --longcalls | --no-longcalls

Enable or disable transformation of call instructions to allow calls across a greater range of addresses. See Section 9.55.4.2 [Function Call Relaxation],

page 380. This option should be used when call targets can potentially be out of range. It may degrade both code size and performance, but the linker can generally optimize away the unnecessary overhead when a call ends up within range. The default is '--no-longcalls'.

### --transform | --no-transform

Enable or disable all assembler transformations of Xtensa instructions, including both relaxation and optimization. The default is '--transform'; '--no-transform' should only be used in the rare cases when the instructions must be exactly as specified in the assembly source. Using '--no-transform' causes out of range instruction operands to be errors.

### --rename-section oldname=newname

Rename the *oldname* section to *newname*. This option can be used multiple times to rename multiple sections.

### --trampolines | --no-trampolines

Enable or disable transformation of jump instructions to allow jumps across a greater range of addresses. See Section 9.55.4.3 [Jump Trampolines], page 381. This option should be used when jump targets can potentially be out of range. In the absence of such jumps this option does not affect code size or performance. The default is '--trampolines'.

### --abi-windowed | --abi-call0

Choose ABI tag written to the .xtensa.info section. ABI tag indicates ABI of the assembly code. A warning is issued by the linker on an attempt to link object files with inconsistent ABI tags. Default ABI is chosen by the Xtensa core configuration.

### 9.55.2 Assembler Syntax

Block comments are delimited by '/\*' and '\*/'. End of line comments may be introduced with either '#' or '//'.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

Instructions consist of a leading opcode or macro name followed by whitespace and an optional comma-separated list of operands:

```
opcode [operand, ...]
```

Instructions must be separated by a newline or semicolon (';').

FLIX instructions, which bundle multiple opcodes together in a single instruction, are specified by enclosing the bundled opcodes inside braces:

```
{
 [format]
 opcode0 [operands]
 opcode1 [operands]
 opcode2 [operands]
 ...
}
```

The opcodes in a FLIX instruction are listed in the same order as the corresponding instruction slots in the TIE format declaration. Directives and labels are not allowed inside the braces of a FLIX instruction. A particular TIE format name can optionally be specified immediately after the opening brace, but this is usually unnecessary. The assembler will automatically search for a format that can encode the specified opcodes, so the format name need only be specified in rare cases where there is more than one applicable format and where it matters which of those formats is used. A FLIX instruction can also be specified on a single line by separating the opcodes with semicolons:

{ [format;] opcode0 [operands]; opcode1 [operands]; opcode2 [operands]; ... }

If an opcode can only be encoded in a FLIX instruction but is not specified as part of a FLIX bundle, the assembler will choose the smallest format where the opcode can be encoded and will fill unused instruction slots with no-ops.

# 9.55.2.1 Opcode Names

See the Xtensa Instruction Set Architecture (ISA) Reference Manual for a complete list of opcodes and descriptions of their semantics.

If an opcode name is prefixed with an underscore character ('\_'), as will not transform that instruction in any way. The underscore prefix disables both optimization (see Section 9.55.3 [Xtensa Optimizations], page 379) and relaxation (see Section 9.55.4 [Xtensa Relaxation], page 380) for that particular instruction. Only use the underscore prefix when it is essential to select the exact opcode produced by the assembler. Using this feature unnecessarily makes the code less efficient by disabling assembler optimization and less flexible by disabling relaxation.

Note that this special handling of underscore prefixes only applies to Xtensa opcodes, not to either built-in macros or user-defined macros. When an underscore prefix is used with a macro (e.g., \_MOV), it refers to a different macro. The assembler generally provides built-in macros both with and without the underscore prefix, where the underscore versions behave as if the underscore carries through to the instructions in the macros. For example, \_MOV may expand to \_MOV.N.

The underscore prefix only applies to individual instructions, not to series of instructions. For example, if a series of instructions have underscore prefixes, the assembler will not transform the individual instructions, but it may insert other instructions between them (e.g., to align a LOOP instruction). To prevent the assembler from modifying a series of instructions as a whole, use the no-transform directive. See Section 9.55.5.3 [transform], page 384.

### 9.55.2.2 Register Names

The assembly syntax for a register file entry is the "short" name for a TIE register file followed by the index into that register file. For example, the general-purpose AR register file has a short name of a, so these registers are named a0...a15. As a special feature, sp is also supported as a synonym for a1. Additional registers may be added by processor configuration options and by designer-defined TIE extensions. An initial '\$' character is optional in all register names.

#### 9.55.3 Xtensa Optimizations

The optimizations currently supported by **as** are generation of density instructions where appropriate and automatic branch target alignment.

#### 9.55.3.1 Using Density Instructions

The Xtensa instruction set has a code density option that provides 16-bit versions of some of the most commonly used opcodes. Use of these opcodes can significantly reduce code size. When possible, the assembler automatically translates instructions from the core Xtensa instruction set into equivalent instructions from the Xtensa code density option. This translation can be disabled by using underscore prefixes (see Section 9.55.2.1 [Opcode Names], page 378), by using the '--no-transform' command-line option (see Section 9.55.1 [Command Line Options], page 376), or by using the no-transform directive (see Section 9.55.5.3 [transform], page 384).

It is a good idea *not* to use the density instructions directly. The assembler will automatically select dense instructions where possible. If you later need to use an Xtensa processor without the code density option, the same assembly code will then work without modification.

#### 9.55.3.2 Automatic Instruction Alignment

The Xtensa assembler will automatically align certain instructions, both to optimize performance and to satisfy architectural requirements.

As an optimization to improve performance, the assembler attempts to align branch targets so they do not cross instruction fetch boundaries. (Xtensa processors can be configured with either 32-bit or 64-bit instruction fetch widths.) An instruction immediately following a call is treated as a branch target in this context, because it will be the target of a return from the call. This alignment has the potential to reduce branch penalties at some expense in code size. This optimization is enabled by default. You can disable it with the '--no-target-align' command-line option (see Section 9.55.1 [Command-line Options], page 376).

The target alignment optimization is done without adding instructions that could increase the execution time of the program. If there are density instructions in the code preceding a target, the assembler can change the target alignment by widening some of those instructions to the equivalent 24-bit instructions. Extra bytes of padding can be inserted immediately following unconditional jump and return instructions. This approach is usually successful in aligning many, but not all, branch targets.

The LOOP family of instructions must be aligned such that the first instruction in the loop body does not cross an instruction fetch boundary (e.g., with a 32-bit fetch width, a LOOP instruction must be on either a 1 or 2 mod 4 byte boundary). The assembler knows about this restriction and inserts the minimal number of 2 or 3 byte no-op instructions to satisfy it. When no-op instructions are added, any label immediately preceding the original loop will be moved in order to refer to the loop instruction, not the newly generated no-op instruction. To preserve binary compatibility across processors with different fetch widths, the assembler conservatively assumes a 32-bit fetch width when aligning LOOP instructions (except if the first instruction in the loop is a 64-bit instruction).

Previous versions of the assembler automatically aligned ENTRY instructions to 4-byte boundaries, but that alignment is now the programmer's responsibility.

#### 9.55.4 Xtensa Relaxation

When an instruction operand is outside the range allowed for that particular instruction field, as can transform the code to use a functionally-equivalent instruction or sequence of instructions. This process is known as *relaxation*. This is typically done for branch instructions because the distance of the branch targets is not known until assembly-time. The Xtensa assembler offers branch relaxation and also extends this concept to function calls, MOVI instructions and other instructions with immediate fields.

#### 9.55.4.1 Conditional Branch Relaxation

When the target of a branch is too far away from the branch itself, i.e., when the offset from the branch to the target is too large to fit in the immediate field of the branch instruction, it may be necessary to replace the branch with a branch around a jump. For example,

```
beqz a2, L
may result in:
bnez.n a2, M
j L
M:
```

(The BNEZ.N instruction would be used in this example only if the density option is available. Otherwise, BNEZ would be used.)

This relaxation works well because the unconditional jump instruction has a much larger offset range than the various conditional branches. However, an error will occur if a branch target is beyond the range of a jump instruction. as cannot relax unconditional jumps. Similarly, an error will occur if the original input contains an unconditional jump to a target that is out of range.

Branch relaxation is enabled by default. It can be disabled by using underscore prefixes (see Section 9.55.2.1 [Opcode Names], page 378), the '--no-transform' command-line option (see Section 9.55.1 [Command-line Options], page 376), or the no-transform directive (see Section 9.55.5.3 [transform], page 384).

#### 9.55.4.2 Function Call Relaxation

Function calls may require relaxation because the Xtensa immediate call instructions (CALLO, CALL4, CALL8 and CALL12) provide a PC-relative offset of only 512 Kbytes in either direction. For larger programs, it may be necessary to use indirect calls (CALLX0, CALLX4, CALLX8 and CALLX12) where the target address is specified in a register. The Xtensa assembler can automatically relax immediate call instructions into indirect call instructions. This relaxation is done by loading the address of the called function into the callee's return address register and then using a CALLX instruction. So, for example:

```
call8 func
```

might be relaxed to:

.literal .L1, func 132r a8, .L1 callx8 a8 Because the addresses of targets of function calls are not generally known until link-time, the assembler must assume the worst and relax all the calls to functions in other source files, not just those that really will be out of range. The linker can recognize calls that were unnecessarily relaxed, and it will remove the overhead introduced by the assembler for those cases where direct calls are sufficient.

Call relaxation is disabled by default because it can have a negative effect on both code size and performance, although the linker can usually eliminate the unnecessary overhead. If a program is too large and some of the calls are out of range, function call relaxation can be enabled using the '--longcalls' command-line option or the longcalls directive (see Section 9.55.5.2 [longcalls], page 384).

#### 9.55.4.3 Jump Relaxation

Jump instruction may require relaxation because the Xtensa jump instruction (J) provide a PC-relative offset of only 128 Kbytes in either direction. One option is to use jump long (J.L) instruction, which depending on jump distance may be assembled as jump (J) or indirect jump (JX). However it needs a free register. When there's no spare register it is possible to plant intermediate jump sites (trampolines) between the jump instruction and its target. These sites may be located in areas unreachable by normal code execution flow, in that case they only contain intermediate jumps, or they may be inserted in the middle of code block, in which case there's an additional jump from the beginning of the trampoline to the instruction past its end. So, for example:

```
j 1f
...
retw
...
mov a10, a2
call8 func
...
1:
```

might be relaxed to:

```
j .L0_TR_1
...
retw
.L0_TR_1:
j 1f
...
mov a10, a2
call8 func
...
1:
```

```
j .L0_TR_1
...
retw
...
mov a10, a2
j .L0_TR_0
.L0_TR_1:
j 1f
.L0_TR_0:
call8 func
...
1:
...
```

The Xtensa assembler uses trampolines with jump around only when it cannot find suitable unreachable trampoline. There may be multiple trampolines between the jump instruction and its target.

This relaxation does not apply to jumps to undefined symbols, assuming they will reach their targets once resolved.

Jump relaxation is enabled by default because it does not affect code size or performance while the code itself is small. This relaxation may be disabled completely with '--no-trampolines' or '--no-transform' command-line options (see Section 9.55.1 [Command-line Options], page 376).

## 9.55.4.4 Other Immediate Field Relaxation

The assembler normally performs the following other relaxations. They can be disabled by using underscore prefixes (see Section 9.55.2.1 [Opcode Names], page 378), the '--no-transform' command-line option (see Section 9.55.1 [Command-line Options], page 376), or the no-transform directive (see Section 9.55.5.3 [transform], page 384).

The MOVI machine instruction can only materialize values in the range from -2048 to 2047. Values outside this range are best materialized with L32R instructions. Thus:

```
movi a0, 100000
```

is assembled into the following machine code:

```
.literal .L1, 100000
132r a0, .L1
```

The L8UI machine instruction can only be used with immediate offsets in the range from 0 to 255. The L16SI and L16UI machine instructions can only be used with offsets from 0 to 510. The L32I machine instruction can only be used with offsets from 0 to 1020. A load offset outside these ranges can be materialized with an L32R instruction if the destination register of the load is different than the source address register. For example:

```
132i a1, a0, 2040
```

is translated to:

```
.literal .L1, 2040
132r a1, .L1
add a1, a0, a1
132i a1, a1, 0
```

If the load destination and source address register are the same, an out-of-range offset causes an error. The Xtensa ADDI instruction only allows immediate operands in the range from -128 to 127. There are a number of alternate instruction sequences for the ADDI operation. First, if the immediate is 0, the ADDI will be turned into a MOV.N instruction (or the equivalent OR instruction if the code density option is not available). If the ADDI immediate is outside of the range -128 to 127, but inside the range -32896 to 32639, an ADDMI instruction or ADDMI/ADDI sequence will be used. Finally, if the immediate is outside of this range and a free register is available, an L32R/ADD sequence will be used with a literal allocated from the literal pool.

For example:

```
addi
               a5, a6, 0
       addi
               a5, a6, 512
       addi
               a5, a6, 513
       addi
               a5, a6, 50000
is assembled into the following:
       .literal .L1, 50000
       mov.n
              a5, a6
       addmi
               a5, a6, 0x200
               a5, a6, 0x200
       addmi
               a5, a5, 1
       addi
       132r
               a5, .L1
               a5, a6, a5
       add
```

## 9.55.5 Directives

. . .

The Xtensa assembler supports a region-based directive syntax:

.begin directive [options]

.end directive

All the Xtensa-specific directives that apply to a region of code use this syntax.

The directive applies to code between the .begin and the .end. The state of the option after the .end reverts to what it was before the .begin. A nested .begin/.end region can further change the state of the directive without having to be aware of its outer state. For example, consider:

```
.begin no-transform
L: add a0, a1, a2
.begin transform
M: add a0, a1, a2
.end transform
N: add a0, a1, a2
.end no-transform
```

The ADD opcodes at L and N in the outer no-transform region both result in ADD machine instructions, but the assembler selects an ADD.N instruction for the ADD at M in the inner transform region.

The advantage of this style is that it works well inside macros which can preserve the context of their callers.

The following directives are available:

#### 9.55.5.1 schedule

The schedule directive is recognized only for compatibility with Tensilica's assembler.

```
.begin [no-]schedule
.end [no-]schedule
```

This directive is ignored and has no effect on as.

## 9.55.5.2 longcalls

The longcalls directive enables or disables function call relaxation. See Section 9.55.4.2 [Function Call Relaxation], page 380.

.begin [no-]longcalls .end [no-]longcalls

Call relaxation is disabled by default unless the '--longcalls' command-line option is specified. The longcalls directive overrides the default determined by the command-line options.

#### 9.55.5.3 transform

This directive enables or disables all assembler transformation, including relaxation (see Section 9.55.4 [Xtensa Relaxation], page 380) and optimization (see Section 9.55.3 [Xtensa Optimizations], page 379).

.begin [no-]transform .end [no-]transform

Transformations are enabled by default unless the '--no-transform' option is used. The transform directive overrides the default determined by the command-line options. An underscore opcode prefix, disabling transformation of that opcode, always takes precedence over both directives and command-line flags.

## 9.55.5.4 literal

The .literal directive is used to define literal pool data, i.e., read-only 32-bit data accessed via L32R instructions.

```
.literal label, value[, value...]
```

This directive is similar to the standard .word directive, except that the actual location of the literal data is determined by the assembler and linker, not by the position of the .literal directive. Using this directive gives the assembler freedom to locate the literal data in the most appropriate place and possibly to combine identical literals. For example, the code:

```
entry sp, 40
.literal .L1, sym
132r a4, .L1
```

can be used to load a pointer to the symbol sym into register a4. The value of sym will not be placed between the ENTRY and L32R instructions; instead, the assembler puts the data in a literal pool.

Literal pools are placed by default in separate literal sections; however, when using the '--text-section-literals' option (see Section 9.55.1 [Command-line Options], page 376), the literal pools for PC-relative mode L32R instructions are placed in the current section.<sup>1</sup> These text section literal pools are created automatically before ENTRY instructions and manually after '.literal\_position' directives (see Section 9.55.5.5 [literal\_position], page 385). If there are no preceding ENTRY instructions, explicit .literal\_position directives must be used to place the text section literal pools; otherwise, as will report an error.

<sup>&</sup>lt;sup>1</sup> Literals for the .init and .fini sections are always placed in separate sections, even when '--text-section-literals' is enabled.

When literals are placed in separate sections, the literal section names are derived from the names of the sections where the literals are defined. The base literal section names are .literal for PC-relative mode L32R instructions and .lit4 for absolute mode L32R instructions (see Section 9.55.5.7 [absolute-literals], page 386). These base names are used for literals defined in the default .text section. For literals defined in other sections or within the scope of a literal\_prefix directive (see Section 9.55.5.6 [literal\_prefix], page 386), the following rules determine the literal section name:

- 1. If the current section is a member of a section group, the literal section name includes the group name as a suffix to the base .literal or .lit4 name, with a period to separate the base name and group name. The literal section is also made a member of the group.
- 2. If the current section name (or literal\_prefix value) begins with ".gnu.linkonce.kind.", the literal section name is formed by replacing ".kind" with the base .literal or .lit4 name. For example, for literals defined in a section named .gnu.linkonce.t.func, the literal section will be .gnu.linkonce.literal.func or .gnu.linkonce.lit4.func.
- 3. If the current section name (or literal\_prefix value) ends with .text, the literal section name is formed by replacing that suffix with the base .literal or .lit4 name. For example, for literals defined in a section named .iram0.text, the literal section will be .iram0.literal or .iram0.lit4.
- 4. If none of the preceding conditions apply, the literal section name is formed by adding the base .literal or .lit4 name as a suffix to the current section name (or literal\_prefix value).

#### 9.55.5.5 literal\_position

When using '--text-section-literals' to place literals inline in the section being assembled, the .literal\_position directive can be used to mark a potential location for a literal pool.

#### .literal\_position

The .literal\_position directive is ignored when the '--text-section-literals' option is not used or when L32R instructions use the absolute addressing mode.

The assembler will automatically place text section literal pools before ENTRY instructions, so the .literal\_position directive is only needed to specify some other location for a literal pool. You may need to add an explicit jump instruction to skip over an inline literal pool.

For example, an interrupt vector does not begin with an ENTRY instruction so the assembler will be unable to automatically find a good place to put a literal pool. Moreover, the code for the interrupt vector must be at a specific starting address, so the literal pool cannot come before the start of the code. The literal pool for the vector must be explicitly positioned in the middle of the vector (before any uses of the literals, due to the negative offsets used by PC-relative L32R instructions). The .literal\_position directive can be used to do this. In the following code, the literal for 'M' will automatically be aligned correctly and is placed after the unconditional jump.

```
.global M
code_start:
j continue
```

```
.literal_position
.align 4
continue:
movi a4, M
```

## 9.55.5.6 literal\_prefix

The literal\_prefix directive allows you to override the default literal section names, which are derived from the names of the sections where the literals are defined.

.begin literal\_prefix [name] .end literal\_prefix

For literals defined within the delimited region, the literal section names are derived from the *name* argument instead of the name of the current section. The rules used to derive the literal section names do not change. See Section 9.55.5.4 [literal], page 384. If the *name* argument is omitted, the literal sections revert to the defaults. This directive has no effect when using the '--text-section-literals' option (see Section 9.55.1 [Command-line Options], page 376).

### 9.55.5.7 absolute-literals

The absolute-literals and no-absolute-literals directives control the absolute vs. PC-relative mode for L32R instructions. These are relevant only for Xtensa configurations that include the absolute addressing option for L32R instructions.

```
.begin [no-]absolute-literals
.end [no-]absolute-literals
```

These directives do not change the L32R mode—they only cause the assembler to emit the appropriate kind of relocation for L32R instructions and to place the literal values in the appropriate section. To change the L32R mode, the program must write the LITBASE special register. It is the programmer's responsibility to keep track of the mode and indicate to the assembler which mode is used in each region of code.

If the Xtensa configuration includes the absolute L32R addressing option, the default is to assume absolute L32R addressing unless the '--no-absolute-literals' command-line option is specified. Otherwise, the default is to assume PC-relative L32R addressing. The absolute-literals directive can then be used to override the default determined by the command-line options.

## 9.56 Z80 Dependent Features

#### 9.56.1 Command-line Options

#### -march=CPU[-EXT...][+EXT...]

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: z80, z180, ez80, gbz80, z80n, r800. In addition to the basic instruction set, the assembler can be told to accept some extention mnemonics. For example, -march=z180+sli+infc extends z180 with SLI instructions and  $IN F_{,}(C)$ . The following extentions are currently supported: full (all known instructions), adl (ADL CPU mode by default, eZ80 only), sli (instruction known as SLI, SLL or SL1), xyhl (instructions with halves of index registers: IXL, IXH, IYL, IYH), xdcb (instructions like RotOp (II+d), R and  $BitOp n_{,}(II+d)$ , R), infc (instruction  $IN F_{,}(C)$  or IN (C)), outc0 (instruction OUT (C), 0). Note that rather than extending a basic instruction set, the extention mnemonics starting with - revoke the respective functionality: -march=z80-full+xyhl first removes all default extentions and adds support for index registers halves only.

If this option is not specified then -march=z80+xyhl+infc is assumed.

#### -local-prefix=prefix

Mark all labels with specified prefix as local. But such label can be marked global explicitly in the code. This option do not change default local label prefix .L, it is just adds new one.

#### -colonless

Accept colonless labels. All symbols at line begin are treated as labels.

-sdcc Accept assembler code produced by SDCC.

#### -fp-s=FORMAT

Single precision floating point numbers format. Default: ieee754 (32 bit).

#### -fp-d=FORMAT

Double precision floating point numbers format. Default: ieee754 (64 bit).

Floating point numbers formats.

- ieee754 Single or double precision IEEE754 compatible format.
- half Half precision IEEE754 compatible format (16 bits).
- single Single precision IEEE754 compatible format (32 bits).
- double Double precision IEEE754 compatible format (64 bits).
- zeda32 32 bit floating point format from z80float library by Zeda.
- math48 48 bit floating point format from Math48 package by Anders Hejlsberg.

## 9.56.2 Syntax

The assembler syntax closely follows the 'Z80 family CPU User Manual' by Zilog. In expressions a single '=' may be used as "is equal to" comparison operator.

Suffices can be used to indicate the radix of integer constants; 'H' or 'h' for hexadecimal, 'D' or 'd' for decimal, 'Q', 'O', 'q' or 'o' for octal, and 'B' for binary.

The suffix 'b' denotes a backreference to local label.

## 9.56.2.1 Special Characters

The semicolon ';' is the line comment character;

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

The Z80 assembler does not support a line separator character.

The dollar sign '\$' can be used as a prefix for hexadecimal numbers and as a symbol denoting the current location counter.

A backslash '\' is an ordinary character for the Z80 assembler.

The single quote ''' must be followed by a closing quote. If there is one character in between, it is a character constant, otherwise it is a string constant.

#### 9.56.2.2 Register Names

The registers are referred to with the letters assigned to them by Zilog. In addition as recognizes 'ixl' and 'ixh' as the least and most significant octet in 'ix', and similarly 'iyl' and 'iyh' as parts of 'iy'.

#### 9.56.2.3 Case Sensitivity

Upper and lower case are equivalent in register names, opcodes, condition codes and assembler directives. The case of letters is significant in labels and symbol names. The case is also important to distinguish the suffix 'b' for a backward reference to a local label from the suffix 'B' for a number in binary notation.

#### 9.56.2.4 Labels

Labels started by .L acts as local labels. You may specify custom local label prefix by -localprefix command-line option. Dollar, forward and backward local labels are supported. By default, all labels are followed by colon. Legacy code with colonless labels can be built with -colonless command-line option specified. In this case all tokens at line begin are treated as labels.

#### 9.56.3 Floating Point

Floating-point numbers of following types are supported:

ieee754 Supported half, single and double precision IEEE754 compatible numbers.

- zeda32 32 bit floating point numbers from z80float library by Zeda.
- math48 48 bit floating point numbers from Math48 package by Anders Hejlsberg.

#### 9.56.4 Z80 Assembler Directives

as for the Z80 supports some additional directives for compatibility with other assemblers.

These are the additional directives in **as** for the Z80:

```
.assume ADL = expression
```

Set ADL status for eZ80. Non-zero value enable compilation in ADL mode else used Z80 mode. ADL and Z80 mode produces incompatible object code. Mixing both of them within one binary may lead problems with disassembler.

```
db expression|string[,expression|string...]
defb expression|string[,expression|string...]
defm string[,string...]
```

For each string the characters are copied to the object file, for each other expression the value is stored in one byte. A warning is issued in case of an overflow. Backslash symbol in the strings is generic symbol, it cannot be used as escape character. See Section 7.5 [.ascii], page 52.

```
dw expression[,expression...]
```

defw expression[,expression...]

For each expression the value is stored in two bytes, ignoring overflow.

```
d24 expression[,expression...]
```

```
def24 expression[,expression...]
```

For each expression the value is stored in three bytes, ignoring overflow.

```
d32 expression[,expression...]
```

```
def32 expression[,expression...]
```

For each expression the value is stored in four bytes, ignoring overflow.

```
ds count[, value]
```

```
defs count[, value]
```

Fill *count* bytes in the object file with *value*, if *value* is omitted it defaults to zero.

#### symbol defl expression

The defl directive is like .set but with different syntax. See Section 7.86 [.set], page 81. It set the value of symbol to expression. Symbols defined with defl are not protected from redefinition.

#### symbol equ expression

The equ directive is like .equiv but with different syntax. See Section 7.30 [.equiv], page 61. It set the value of symbol to expression. It is an error if symbol is already defined. Symbols defined with equ are not protected from redefinition.

```
psect name
```

A synonym for .section, no second argument should be given. See Section 7.85 [.section], page 77.

#### xdef symbol

A synonym for .global, make symbol is visible to linker. See Section 7.41 [.global], page 63.

xref name

A synonym for .extern (Section 7.35 [.extern], page 62).

## 9.56.5 Opcodes

In line with common practice, Z80 mnemonics are used for the Z80, Z80N, Z180, eZ80, Ascii R800 and the GameBoy Z80.

In many instructions it is possible to use one of the half index registers ('ixl','ixh','iyl','iyh') in stead of an 8-bit general purpose register. This yields instructions that are documented on the eZ80 and the R800, undocumented on the Z80 and unsupported on the Z180. Similarly in f,(c) is documented on the R800, undocumented on the Z80 and unsupported on the Z180 and the eZ80.

The assembler also supports the following undocumented Z80-instructions, that have not been adopted in any other instruction set:

out (c), 0 Sends zero to the port pointed to by register C.

sli m Equivalent to m = (m << 1)+1, the operand m can be any operand that is valid for 'sla'. One can use 'sll' as a synonym for 'sli'.

```
op (ix+d), r
```

This is equivalent to

ld r, (ix+d) op r ld (ix+d), r

The operation 'op' may be any of 'res b,', 'set b,', 'rl', 'rlc', 'rr', 'rrc', 'sla', 'sli', 'sra' and 'srl', and the register 'r' may be any of 'a', 'b', 'c', 'd', 'e', 'h' and 'l'.

#### op (iy+d), r

As above, but with 'iy' instead of 'ix'.

The web site at http://www.z80.info is a good starting place to find more information on programming the Z80.

You may enable or disable any of these instructions for any target CPU even this instruction is not supported by any real CPU of this type. Useful for custom CPU cores.

## 9.57 Z8000 Dependent Features

The Z8000 as supports both members of the Z8000 family: the unsegmented Z8002, with 16 bit addresses, and the segmented Z8001 with 24 bit addresses.

When the assembler is in unsegmented mode (specified with the **unsegm** directive), an address takes up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the **segm** directive), a 24-bit address takes up a long (32 bit) register. See Section 9.57.3 [Assembler Directives for the Z8000], page 392, for a list of other Z8000 specific assembler directives.

#### 9.57.1 Options

-z8001 Generate segmented code by default.

-z8002 Generate unsegmented code by default.

#### 9.57.2 Syntax

#### 9.57.2.1 Special Characters

'!' is the line comment character.

If a '#' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

You can use ';' instead of a newline to separate statements.

#### 9.57.2.2 Register Names

The Z8000 has sixteen 16 bit registers, numbered 0 to 15. You can refer to different sized groups of registers by register number, with the prefix '**r**' for 16 bit registers, '**rr**' for 32 bit registers and '**rq**' for 64 bit registers. You can also refer to the contents of the first eight (of the sixteen 16 bit registers) by bytes. They are named '**rln**' and '**rhn**'.

```
byte registers

rl0 rh0 rl1 rh1 rl2 rh2 rl3 rh3

rl4 rh4 rl5 rh5 rl6 rh6 rl7 rh7

word registers

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15

long word registers

rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14

quad word registers

rq0 rq4 rq8 rq12
```

#### 9.57.2.3 Addressing Modes

as understands the following addressing modes for the Z8000:

rln rhn rn rrn rqn	Register direct: 8bit, 16bit, 32bit, and 64bit registers.
@rn @rrn	Indirect register: $@rrn$ in segmented mode, $@rn$ in unsegmented mode.
addr	Direct: the 16 bit or 24 bit address (depending on whether the assembler is in segmented or unsegmented mode) of the operand is in the instruction.
address(r <i>i</i>	1) Indexed: the 16 or 24 bit address is added to the 16 bit register to produce the final address in memory of the operand.
rn(#imm) rrn(#imm)	Base Address: the 16 or 24 bit register is added to the 16 bit sign extended immediate displacement to produce the final address in memory of the operand.
rn(rm) rrn(rm)	Base Index: the 16 or 24 bit register $rn$ or $rrn$ is added to the sign extended 16 bit index register $rm$ to produce the final address in memory of the operand.
#xx	Immediate data xx.

## 9.57.3 Assembler Directives for the Z8000

The Z8000 port of as includes additional assembler directives, for compatibility with other Z8000 assemblers. These do not begin with '.' (unlike the ordinary as directives).

segm .z8001	Generate code for the segmented Z8001.
unsegm .z8002	Generate code for the unsegmented Z8002.
name	Synonym for .file
global	Synonym for .global
wval	Synonym for .word
lval	Synonym for .long
bval	Synonym for .byte
sval	Assemble a string. sval expects one string I It assembles each byte of the string into conse escape sequence '%xx' (where xx represents a

Assemble a string. sval expects one string literal, delimited by single quotes. It assembles each byte of the string into consecutive addresses. You can use the escape sequence '%xx' (where xx represents a two-digit hexadecimal number) to represent the character whose ASCII value is xx. Use this feature to describe single quote and other characters that may not appear in string literals as themselves. For example, the C statement 'char \*a = "he said \"it's 50% off\"";' is

represented in Z8000 assembly language (shown with the assembler output in hex at the left) as 68652073 sval 'he said %22it%27s 50%25 off%22%00' 61696420 22697427 73203530 25206F66 662200 rsect synonym for .section block  $\operatorname{synonym}$  for <code>.space</code> special case of .align; aligns output to even byte boundary. even

## 9.57.4 Opcodes

For detailed information on the Z8000 machine instruction set, see Z8000 Technical Manual.

## 10 Reporting Bugs

Your bug reports play an essential role in making as reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of **as** work better. Bug reports are your contribution to the maintenance of **as**.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

## 10.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the assembler gets a fatal signal, for any input whatever, that is a **as** bug. Reliable assemblers never crash.
- If as produces an error message for valid input, that is a bug.
- If as does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of assemblers, your suggestions for improvement of as are welcome in any case.

## 10.2 How to Report Bugs

A number of companies and individuals offer support for GNU products. If you obtained **as** from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file **etc/SERVICE** in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for as to https://sourceware.org/bugzilla/.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of a symbol you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the assembler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with. To enable us to fix the bug, you should include all these things:

- The version of as. as announces it if you start it with the '--version' argument. Without this, we will not know whether there is any point in looking for the bug in the current version of as.
- Any patches you may have applied to the **as** source.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile as—e.g. "gcc-2.7".
- The command arguments you gave the assembler to assemble your example and observe the bug. To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.

- A complete input file that will reproduce the bug. If the bug is observed when the assembler is invoked via a compiler, send the assembler source, not the high level language source. Most compilers will produce the assembler source when run with the '-S' option. If you are using gcc, use the options '-v --save-temps'; this will save the assembler source in a file with an extension of .s, and also show you exactly how as is being run.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal."

Of course, if the bug is that **as** gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of **as** is out of sync, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.

• If you wish to suggest changes to the as source, send us context diffs, as generated by diff with the '-u', '-c', or '-p' option. Always send diffs from the old file to the new file. If you even discuss something in the as source, refer to it by context, not by line number.

The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

Here are some things that are not necessary:

• A description of the envelope of the bug.

Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else. Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

• A patch for the bug.

A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

Sometimes with a program as complicated as **as** it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

• A guess about what the bug is or what it depends on.

Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

## 11 Acknowledgements

If you have contributed to GAS and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Nick Clifton (email address nickc@redhat.com).

Dean Elsner wrote the original GNU assembler for the VAX.<sup>1</sup>

Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in messages.c, input-file.c, write.c.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the coff and b.out back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end (tc-mips.c, tc-mips.h), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Renesas H8/300 processors (tc-z8k, tc-h8300), and IEEE 695 object file format (obj-ieee), was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added .include support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g., jsr), while synthetic instructions remained shrinkable (jbsr). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

<sup>&</sup>lt;sup>1</sup> Any more details?

Steve Chamberlain made GAS able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus Support (sparc, and some initial 64-bit support).

Linas Vepstas added GAS support for the ESA/390 "IBM 370" architecture.

Richard Henderson rewrote the Alpha assembler. Klaus Kaempf wrote GAS and BFD support for openVMS/Alpha.

Timothy Wall, Michael Hayes, and Greg Smart contributed to the various tic\* flavors.

David Heine, Sterling Augustine, Bob Wilson and John Ruttenberg from Tensilica, Inc. added support for Xtensa processors.

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Jon Beniston added support for the Lattice Mico32 architecture.

Many others have contributed large or small bugfixes and enhancements. If you have contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we are not intentionally leaving anyone out.

## Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

#### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaT<sub>E</sub>X input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

#### 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

#### 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

#### 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

#### 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

#### 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

#### 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

#### 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

#### 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) year your name. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ''GNU Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# AS Index

# #

#	32
#APP	31
#NO_APP	31

# \$

\$ in symbol names 165, 169, 231, 32	22
\$a1	31
<pre>\$acos math builtin, TIC54X 33 \$asin math builtin, TIC54X 33</pre>	37
\$asin math builtin, TIC54X 33	37
\$atan math builtin, TIC54X 3	
\$atan2 math builtin, TIC54X	37
<pre>\$ceil math builtin, TIC54X 3</pre>	38
\$cos math builtin, TIC54X 33	
\$cosh math builtin, TIC54X 3	
\$cvf math builtin, TIC54X 33	38
<pre>\$cvf math builtin, TIC54X 33 \$cvi math builtin, TIC54X 33</pre>	38
\$d 101, 13	
\$exp math builtin, TIC54X 3	
\$fabs math builtin, TIC54X 33	
\$firstch subsym builtin, TIC54X 3	44
\$floor math builtin, TIC54X	
\$fmod math builtin, TIC54X 3	38
\$int math builtin, TIC54X 3	38
\$iscons subsym builtin, TIC54X3	
\$isdefed subsym builtin, TIC54X 3	
\$ismember subsym builtin, TIC54X	
\$isname subsym builtin, TIC54X3	
\$isreg subsym builtin, TIC54X3	
<pre>\$lastch subsym builtin, TIC54X3</pre>	
\$ldexp math builtin, TIC54X	38
\$log math builtin, TIC54X 3	
\$log10 math builtin, TIC54X	
\$max math builtin, TIC54X 3	
\$min math builtin, TIC54X 3	
\$pow math builtin, TIC54X 3	
\$round math builtin, TIC54X	38
\$sgn math builtin, TIC54X 3	
\$sin math builtin, TIC54X 3	
\$sinh math builtin. TIC54X	39
<pre>\$sinh math builtin, TIC54X 3 \$sqrt math builtin, TIC54X 3</pre>	39
\$structacc subsym builtin, TIC54X	45
\$structsz subsym builtin, TIC54X	
\$symcmp subsym builtin, TIC54X3	44
\$symlen subsym builtin, TIC54X	44
\$t1	
\$tan math builtin, TIC54X	
\$tanh math builtin, TIC54X	
\$trunc math builtin, TIC54X	
\$x10	

# %

%gp	301
'%gpreg'	301
'%pidreg'	301

#### —

	366
	. 22 101
32 option, 1580	101
432' option, x86-64	181
<sup>4</sup> 64' option, i386	181
'64' option, x86-64	181
abi-call0	
abi-windowed	377
absolute-literals	
allow-reg-prefix	322
alternate	. 25
auto-litpools	376
base-size-default-16'	
base-size-default-32'	
big	322
bitwise-or' option, M680x0	213
'compress-debug-sections=' option	7
'disp-size-default-16'	214
disp-size-default-32'	214
'divide' option, i386	181
dsp	322
emulation=crisaout	
command-line option, CRIS	157
emulation=criself	
command-line option, CRIS	157
enforce-aligned-data	
fatal-warnings	. 29
fdpic	
fix-v4bx command-line option, ARM	123
'fixed-special-register-names'	
command-line option, MMIX	249
force-long-branches'	222
'generate-example'	222
'globalize-symbols'	
command-line option, MMIX	249
'gnu-syntax' command-line	
option, MMIX	249
'linker-allocated-gregs'	
command-line option, MMIX	249
listing-cont-lines	
listing-lhs-width	
listing-lhs-width2	
listing-rhs-width	
little	322
longcalls	376
march=architecture	
command-line option, CRIS	157

MD
11
mul-bug-abort command-line
option, CRIS 157
no-absolute-literals
no-auto-litpools
'no-expand' command-line option, MMIX 249
no-longcalls
'no-merge-gregs' command-line
option, MMIX
no-mul-bug-abort
command-line option, CRIS
no-pad-sections
'no-predefined-syms'
command-line option, MMIX 249
'no-pushj-stubs' command-line
option, MMIX
'no-stubs' command-line option, MMIX 249
no-target-align
no-text-section-literals
no-trampolines
no-transform
no-underscore command-line
option, CRIS $\dots \dots 157$
no-warn
'pcrel'
pic command-line option, CRIS 157
'print-insn-syntax'
'print-opcodes'
'register-prefix-optional'
option, M680x0
relax
161dx 022
'relax' command-line option, MMIX 249
'relax' command-line option, MMIX 249 rename-section
<pre>'relax' command-line option, MMIX 249rename-section 377renesas 322</pre>
<pre>'relax' command-line option, MMIX 249rename-section 377renesas 322sectname-subst 78</pre>
<pre>'relax' command-line option, MMIX 249rename-section 377renesas 322sectname-subst 78 'short-branches' 221</pre>
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        statistics       29
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322
<pre>'relax' command-line option, MMIX 249rename-section 377renesas 322sectname-subst 78 'short-branches' 221small 322statistics 29 'strict-direct-mode' 221</pre>
<pre>'relax' command-line option, MMIX 249rename-section 377renesas</pre>
<pre>'relax' command-line option, MMIX 249rename-section</pre>
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        statistics       29         'strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        trampolines       377
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        statistics       29         'strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157
<pre>'relax' command-line option, MMIX 249rename-section 377renesas</pre>
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25         -ac       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25         -ac       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25         -ad       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25         -ad       25         -ad       25         -ad       25         -ah       25
'relax' command-line option, MMIX       249        rename-section       377        renesas       322        sectname-subst       78         'short-branches'       221        small       322        strict-direct-mode'       221        target-align       376        text-section-literals       376        traditional-format       29        transform       377        underscore command-line option, CRIS       157        warn       30         'x32' option, i386       181         'xgate-ramoffset'       221         '-1' option, VAX/VMS       366         -32addr command-line option, Alpha       102         -a       25         -ad       25         -ad       25

-an	-
-as	
-Asparc	
-Asparcfmaf	
-Asparcima	
-Asparclet	
-Asparclite	
-Asparcvis	
-Asparcvis2	
-Asparcvis3	
-Asparcvis3r	
-Avo	
-Av8	
-Avo	
-Av9	
-Av9b	
-Av9c	
-Av9d	
-Av9e	
-Av9m	
-Av9v	
-big option, M32R	
-colonless command-line option, Z80	
-d, VAX option	
-D	
-D, ignored on VAX	
-eabi= command-line option, ARM	
-EB command-line option, AArch64	
-EB command-line option, ARC	
-EB command-line option, ARM	
-EB command-line option, BPF	146
-EB option (MIPS)	233
-EB option, M32R <sup>-</sup>	
'-EB' option, TILE-Gx	
-EL command-line option, AArch64	
-EL command-line option, ARC	110
-EL command-line option, ARM	123
-EL command-line option, BPF	
-EL option (MIPS)	233
-EL option, M32R	209
'-EL' option, TILE-Gx	349
-f	
-F command-line option, Alpha $\ldots\ldots\ldots$	
'-fno-pic' option, RISC-V	
-fp-d command-line option, Z80	
-fp-s command-line option, Z80	
'-fpic' option, RISC-V	
-g command-line option, Alpha	
-G command-line option, Alpha	102
-G option (MIPS)	
'-h' option, VAX/VMS	
	365
'-H' option, VAX/VMS	$\frac{365}{366}$
'-H' option, VAX/VMS	$\frac{365}{366}$
<pre>'-H' option, VAX/VMS -I path '-ignore-parallel-conflicts'</pre>	365 366 . 26
<pre>'-H' option, VAX/VMS -I path '-ignore-parallel-conflicts' option, M32RX</pre>	365 366 . 26 210
<pre>'-H' option, VAX/VMS -I path '-ignore-parallel-conflicts'</pre>	365 366 . 26 210 210

-k command-line option, ARM12	
-К 2	6
-KPIC option, M32R 20	9
-KPIC option, MIPS 23	3
'-1' option, M680x0	
-little option, M32R 20	9
-local-prefix command-line option, Z80 38	<b>57</b>
-L	
'-m[no-]68851' command-line	
option, M680x0	3
'-m[no-]68881' command-line	
option, M680x0	3
'-m[no-]div' command-line option, M680x0 21	3
'-m[no-]emac' command-line option, M680x021	
'-m[no-]float' command-line	-
option, M680x0	3
'-m[no-]mac' command-line option, M680x0 21	
'-m[no-]usp' command-line option, M680x0 21	
-m11/03	
-m11/04	
-m11/05	
-m11/10	
-m11/15	
-m11/20	
-m11/20	
-m11/23	
-m11/24	
-m11/34	
-m11/34a	
-m11/3527	
-m11/30	
-m11/40	
-m11/44	
-m11/45	
-m11/53	
-m11/55	
-m11/60	
-m11/70	
-m11/73	1
-m11/83	1
-m11/85	
-m11/9327	
-m11/94	
'-m16c' option, M16C	
'-m31' option, s390	
'-m32' option, TILE-Gx	
'-m32bit-doubles'	
'-m32r' option, M32R	
'-m32rx' option, M32R2	
'-m32rx' option, M32RX	9
-m4byte-align command-line option, V850 35	9
'-m64' option, s390	13
'-m64' option, TILE-Gx	
'-m64bit-doubles'	
'-m68000' and related options 21	
'-m68hc11'	
'-m68hc12' 22	1

'-m68hcs12'	221
-m8byte-align command-line option, V850	359
-mabi= command-line option, AArch64	
'-mabi=ABI' option, RISC-V	
'-madd-bnd-prefix' option, i386	
'-madd-bnd-prefix' option, x86-64	
'-malign-branch-boundary=' option, i386	184
'-malign-branch-boundary=' option, x86-64	
'-malign-branch-prefix-size='	-
option, i386	184
'-malign-branch-prefix-size='	-
option, x86-64	184
'-malign-branch=' option, i386	184
'-malign-branch=' option, x86-64	184
-mall	275
-mall-enabled command-line option, LM32	
-mall-extensions	275
-mall-opcodes command-line option, AVR	
'-mamd64' option, x86-64	185
-mapcs-26 command-line option, ARM	122
-mapcs-32 command-line option, ARM	122
-mapcs-float command-line option, ARM	122 122
-mapes-reentrant	122
command-line option, ARM	122
'-march-attr' option, RISC-V	
-march= command-line option, AArch64	
	118
'-march=' command-line option, M680x0	213
-march= command-line option, TIC6X	346
-march= command-line option, Z80	387
'-march=' option, i386	181
'-march=' option, i380	303
"-march=' option, x86-64	181
'-march=ISA' option, RISC-V	285
-matpcs command-line option, ARM	
'-mavxscalar=' option, i386	$122 \\ 182$
-mavxscalar=' option, 1580	182
-maxiscalar option, x80-04	162
command-line option, LM32	204
'-mbig-endian'	
'-mbig-endian' option, RISC-V	
'-mbig-obj' option, i386	183
'-mbig-obj' option, x86-64	100
'-mbranches-within-32B-boundaries'	104
option, i386 '-mbranches-within-32B-boundaries'	184
	101
option, x86-64	184
-mbreak-enabled command-line	004
option, LM32	
-mccs command-line option, ARM	
-mcis	
-mcode-density command-line option, ARC	110
-mconstant-gp command-line option, IA-64	
-mcpu command-line option, Alpha	
'-mcpu' option, cpu	
'-mcpu='	
-mcpu= command-line option, AArch64	
-mcpu= command-line option, ARM	117

-mcpu= command-line option, Blackfin142
'-mcpu=' command-line option, M680x0 213
-mcpu=cpu command-line option, ARC 109
-mcsm
'-mcsr-check' option, RISC-V 285
-mdcache-enabled command-line
option, LM32 204
-mdebug command-line option, Alpha 102
-mdivide-enabled command-line
option, LM32
'-mdollar-hex' option, dollar-hex 227
-mdpfp command-line option, ARC 110
-mdsbt command-line option, TIC6X
'-me' option, stderr redirect
-meis
-mepiphany command-line option, Epiphany 172
-mepiphany16 command-line
option, Epiphany 172
'-merrors-to-file' option, stderr redirect 336
'-mesa' option, s390
'-mevexlig=' option, i386
'-mevexlig=' option, x86-64
'-mevexrcig=' option, i386
'-mevexrcig=' option, x86-64
'-mevexwig=' option, i386
"-mevexwig=" option, x86-64
'-mf' option, far-mode
-mf11

-mkd11e	276
-mkd11f	
-mkd11h	
-mkd11k	
-mkd11q	
-mkd11z	
-mkev11	
'-mlfence-after-load=' option, i386	
'-mlfence-after-load=' option, x86-64	184
'-mlfence-before-indirect-branch='	104
option, i386	184
'-mlfence-before-indirect-branch='	
option, x86-64	
'-mlfence-before-ret=' option, i386	185
'-mlfence-before-ret=' option, x86-64	185
-mlimited-eis	
-mlink-relax command-line option, AVR	136
'-mlittle-endian'	300
'-mlittle-endian' option, RISC-V	286
'-mlong' 221,	
'-mlong-double'	
'-mm9s12x'	
'-mm9s12xg'	
-mmcu= command-line option, AVR	
-mmfpt	
-mmicrocode	
'-mmnemonic=' option, i386	
'-mmnemonic=' option, x86-64	
	100
-mmultiply-enabled	904
command-line option, LM32	
-mmutiproc	
-mmxps	
'-mnaked-reg' option, i386	
'-mnaked-reg' option, x86-64	
-mnan= command-line option, MIPS	
'-mno-allow-string-insns'	
'-mno-arch-attr' option, RISC-V	
-mno-cis	
-mno-csm	
'-mno-csr-check' option, RISC-V	286
-mno-dollar-line-separator command	
line option, AVR	136
-mno-dsbt command-line option, TIC6X	346
-mno-eis	
-mno-extensions	275
-mno-fdpic command-line option, Blackfin	142
-mno-fis	275
-mno-fp-11	275
-mno-fpp	275
-mno-fpu	275
-mno-kev11	
-mno-limited-eis	
-mno-link-relax	210
command-line option, AVR	136
-mno-mfpt	
-mno-microcode	
-mno-microcode	
-mno-mxps	270

-mno-pic	
-mno-pic command-line option, TIC6X	
'-mno-regnames' option, s390	
'-mno-relax' option, RISC-V	285
-mno-skip-bug command-line option, AVR	135
-mno-spl	276
-mno-sym32	239
-mno-verbose-error command-line	
option, AArch64	. 96
-mno-wrap command-line option, AVR	135
-mnopic command-line option, Blackfin	142
-mnps400 command-line option, ARC	110
-	183
'-momit-lock-prefix=' option, x86-64	183
-mpic	275
	346
'-mpid'	300
-mpid= command-line option, TIC6X	
	285
"-mreg-prefix=prefix" option, reg-prefix	200
'-mregnames' option, s390	303
-mrelax command-line option, ARC	110
	358
'-mrelax' option, RISC-V	
'-mrelax-relocations=' option, i386	
'-mrelax-relocations=' option, x86-64	184
-mrh850-abi command-line option, V850	
-mrmw command-line option, AVR	
'-mrx-abi'	
'-mshared' option, i386	
'-mshared' option, x86-64	
'-mshort' 221,	
$`\verb+mshort-double' \dots \dots \dots 221,$	373
-msign-extend-enabled	
command-line option, LM32	204
'-msmall-data-limit'	
-msoft-float command-line option, $V850\ldots$	
-mspfp command-line option, $\operatorname{ARC}$	110
-mspl	
'-msse-check=' option, i386	
'-msse-check=' option, x86-64	182
'-msse2avx' option, i386	
'-msse2avx' option, x86-64	182
-msym32	239
'-msyntax=' option, i386	183
'-msyntax=' option, x86-64	183
-mt11	276
-mthumb command-line option, ARM	121
-mthumb-interwork	
command-line option, ARM	121
'-mtune=' option, i386	
'-mtune=' option, x86-64	
-mtune=arch command-line option, Visium	
'-muse-conventional-section-names'	
'-muse-renesas-section-names'	
'-muse-unaligned-vector-move'	
option, i386	182
· · · · · · · · · · · · · · · · · · ·	

'-muse-unaligned-vector-move'	
option, x86-64	182
-muser-enabled command-line	
option, LM32	204
-mv850 command-line option, V850	358
-mv850any command-line option, V850	358
-mv850e command-line option, V850	358
-mv850e1 command-line option, V850	
-mv850e2 command-line option, V850	358
-mv850e2v3 command-line option, V850	
-mv850e2v4 command-line option, V850	
-mv850e3v5 command-line option, V850	358
-mverbose-error command-line	
option, AArch64	. 96
'-mvexwig=' option, i386	
'-mvexwig=' option, x86-64	
-mvxworks-pic option, MIPS	
'-mwarn-areg-zero' option, s390	
-mwarn-deprecated	000
command-line option, ARM	123
-mwarn-syms command-line option, ARM	
'-mx86-used-note=' option, i386	
'-mx86-used-note=' option, x86-64	
'-mzarch' option, s390	
-M	
-N command-line option, CRIS	
'-nIp' option, M32RX	
'-no-bitinst', M32R2	
'-no-ignore-parallel-conflicts'	209
option, M32RX	910
-no-mdebug command-line option, Alpha	
-no-parallel option, M32RX	209
'-no-warn-explicit-parallel-conflicts'	010
option, M32RX	
'-no-warn-unmatched-high' option, M32R	
-nocpp ignored (MIPS)	
-0	
'-0' option, i386	
-0 option, M32RX	
'-0' option, x86-64	
'-00' option, i386	
'-00' option, x86-64	
'-01' option, i386	
'-01' option, x86-64	
'-02' option, i386	
'-02' option, x86-64	
'- <b>Os</b> ' option, i386	
'- <b>Os</b> ' option, x86-64	
-parallel option, M32RX	
-relax command-line option, Alpha	
-replace command-line option, Alpha	
-R	
-S, ignored on VAX	
-sdcc command-line option, Z80	
-t, ignored on VAX	
-T, ignored on VAX	
-v	. 29

<b>-V</b> , redundant on VAX
-version
'-warn-explicit-parallel-conflicts'
option, M32RX 209
'-warn-unmatched-high' option, M32R 210
-W
'-Wnp' option, M32RX 210
'-Wnuh' option, M32RX 210
'-Wp' option, M32RX
-wsigned_overflow
command-line option, V850
'-Wuh' option, M32RX 210
-wunsigned_overflow
command-line option, $V850 \dots 358$
'-x' command-line option, MMIX 249
<b>-z8001</b> command-line option, Z8000
-z8002 command-line option, Z8000

. (symbol)
directive, TILEPro
.arch directive, ARM
.arch_extension directive, AArch64 100 .arch_extension directive, ARM 125 .arm directive, ARM 125
.assume directive, Z80
.bss directive, AArch64
.cantunwind directive, ARM
.cfi_b_key_frame directive, AArch64 101 .code directive, ARM 125 .cpu directive, AArch64 100
.cpu directive, ARM
.eabi_attribute directive, ARM 126 .ehtype directive, TIC6X 347 .endp directive, TIC6X 347
.even directive, AArch64100.even directive, ARM126.extend directive, ARM126
.float16 directive, AArch64

.fnend directive, ARM	
.fnstart directive, ARM	127
.force_thumb directive, ARM	127
.fpu directive, ARM	127
.global	244
.gnu_attribute 4, <i>n</i> directive, MIPS	
.gnu_attribute Tag_GNU_MIPS_ABI_FP,	
<i>n</i> directive, MIPS	244
.handlerdata directive, ARM	
.handlerdata directive, TIC6X	
.insn	
.insn directive, s390	317
.inst directive, AArch64	100
.inst directive, ARM	127
.ldouble directive, ARM	126
.little directive, M32RX	
.long directive, s390	
.ltorg directive, AArch64	
.ltorg directive, ARM	$100 \\ 127$
.ltorg directive, s390	
.m32r directive, M32R	211
.m32r2 directive, M32R2	
.m32rx directive, M32RX	
.machine directive, s390	318
.machinemode directive, s390	
.module	
.module fp=nn directive, MIPS	
.movsp directive, ARM	
.nan directive, MIPS	246
.no_pointers directive, XStormy16	375
.nocmp directive, TIC6X	347
.0	
.object_arch directive, ARM	127
.packed directive, ARM	127
.pacspval directive, ARM	127
.pad directive, ARM	128
.param on HPPA	177
.personality directive, ARM	
.personality directive, TIC6X	347
.personalityindex directive, ARM	128
.personalityindex directive, TIC6X	
.pool directive, AArch64	
.pool directive, ARM	
.quad directive, s390	
.req directive, AArch64	100
.req directive, ARM	
.require_canonical_reg_names	120
directive, TILE-Gx	252
.require_canonical_reg_names	555
	257
directive, TILEPro	
.save directive, ARM	128
.save directive, ARM	$128 \\ 347$
.save directive, ARM .scomm directive, TIC6X .secrel32 directive, ARM	128 347 128
.save directive, ARM .scomm directive, TIC6X .secrel32 directive, ARM .set arch=cpu	128 347 128 243
<pre>.save directive, ARM .scomm directive, TIC6X .secrel32 directive, ARM .set arch=cpu .set at</pre>	128 347 128 243 241
<pre>.save directive, ARM .scomm directive, TIC6X .secrel32 directive, ARM .set arch=cpu .set at .set at</pre>	128 347 128 243 241 241
<pre>.save directive, ARM .scomm directive, TIC6X .secrel32 directive, ARM .set arch=cpu .set at</pre>	128 347 128 243 241 241 241 244

•

	1
.set	doublefloat
.set	dsp
.set	dspr2
.set	dspr3
.set	ginv
.set	hardfloat
.set	insn32
.set	loongson-cam
.set	loongson-ext
.set	loongson-ext2
.set	loongson-mmi
.set	macro
.set	mcu
.set	mdmx
.set	mips16e2
.set	mips3d247
.set	mipsn
.set	msa
.set	mt
.set	noat
.set	noautoextend
.set	nocrc
.set	nodsp
.set	nodspr2 247
.set	nodspr3 247
.set	noginv
.set	noinsn32
.set	noloongson-cam 248
.set	noloongson-ext 248
.set	noloongson-ext2
.set	noloongson-mmi 248
.set	nomacro
.set	nomcu
.set	nomdmx
.set	nomips16e2
.set	nomips3d 247
.set	nomsa
.set	nomt
.set	nosmartmips
.set	nosym32 242
.set	novirt
.set	noxpa
.set	pop 246
.set	push
.set	<b>singlefloat</b>
.set	smartmips 247
.set	softfloat 248
.set	sym32242
.set	virt
.set	xpa
	fp directive, ARM 128
	rt directive, s390
.synt	tax directive, ARM 129
.thur	<b>nb</b> directive, ARM 129
	nb_func directive, ARM 129
.thur	nb_set directive, ARM 129
.tls	descadd directive, AArch64 100
.tls	desccall directive, AArch64 101

.tlsdescldr directive, AArch64 101
.tlsdescseq directive, ARM 129
.unreq directive, AArch64 101
.unreq directive, ARM 129
.unwind_raw directive, ARM 129
.v850 directive, V850
.v850e directive, V850
.v850e1 directive, V850
.v850e2 directive, V850
.v850e2v3 directive, V850 362
.v850e2v4 directive, V850 362
.v850e3v5 directive, V850 362
.variant_pcs directive, AArch64 101
.vsave directive, ARM 129
.xword directive, AArch64 101
. <b>z</b> 8001 392
.z8002 392

#### :

\_

: (label)		33
-----------	--	----

_ opcode prefix	378
DYNAMIC, ARC pre-defined symbol	
GLOBAL_OFFSET_TABLE, ARC	
pre-defined symbol	116

#### 0

@gotoff(symbol), ARC modifier	116
@gotpc(symbol), ARC modifier	116
Chi pseudo-op, XStormy16	375
@lo pseudo-op, XStormy16	375
@pcl(symbol), ARC modifier	116
@plt(symbol), ARC modifier	116
@sda(symbol), ARC modifier	116
@word modifier, D10V	167

# $\setminus$

$" (doublequote character) \dots 34$
$\$ ('\' character)
\b (backspace character) 33
\ddd (octal character code)
\f (formfeed character)
\n (newline character)
\r (carriage return character) 33
\t (tab)
xd (hex character code)

## 

16-bit code, i386	196
16bit_pointers directive, XStormy16	375
16byte directive, Nios II	268
16byte directive, PRU	284

## $\mathbf{2}$

<b>2byte</b> directive	
2byte directive, Nios II	
<b>2byte</b> directive, PRU	

# 

32bit_pointers directive, XStormy16	375
3DNow!, i386	195
3DNow!, x86-64	195

### 

430 support	
4byte directive	
4byte directive, Nios II 268	
4byte directive, PRU	

### 

<b>8byte</b> directive	
Sbyte directive, Nios II 268	
<b>Sbyte</b> directive, PRU	

# A

a.out
a.out symbol attributes
'A_DIR' environment variable, TIC54X
AArch64 floating point (IEEE)
AArch64 immediate character
AArch64 line comment character
AArch64 line separator
AArch64 machine directives
AArch64 opcodes 101
AArch64 options (none)
AArch64 register names
AArch64 relocations
AArch64 support
abort directive
ABORT directive
absolute section
absolute-literals directive
ADDI instructions, relaxation
addition, permitted arguments 48
addresses
addresses, format of
addressing modes, D10V 166
addressing modes, D30V 171
addressing modes, H8/300 173

addressing modes, M680x0	216
addressing modes, M68HC11	222
addressing modes, S12Z	228
addressing modes, SH	323
addressing modes, XGATE	373
addressing modes, Z8000	391
ADR reg, <label> pseudo op, ARM</label>	
ADRL reg, <label> pseudo op, ARM</label>	130
ADRP, ADD, LDR/STR group	
relocations, AArch64	. 99
advancing location counter	
align directive	
align directive, Nios II	
align directive, OpenRISC	273
align directive, PRU	283
align directive, SPARC	
align directive, TIC54X	
aligned instruction bundle	
alignment for NEON instructions	
alignment of branch targets	
alignment of LOOP instructions	
Alpha floating point (IEEE)	
Alpha line comment character	
-	103
Alpha notes	100
Alpha options	102
Alpha registers	102
Alpha relocations	
Alpha support	
Alpha Syntax	
Alpha-only directives	
Altera Nios II support	267
altered difference tables	
alternate syntax for the 680x0	
ARC Branch Target Address	
ARC BTA saved on exception entry	
ARC Build configuration for: BTA Registers	
ARC Build configuration for: Core Registers	
ARC Build configuration for: Interrupts	
ARC Build Configuration Registers Version	
ARC C preprocessor macro separator	
ARC core general registers	111
	$111 \\ 112$
ARC Exception Cause Register	111
ARC Exception Return Address	
ARC extension core registers	
ARC frame pointer	111
-	111
ARC global pointer ARC interrupt link register	111
	111
	111
ARC level 2 interrupt link register	111
	110
ARC line separator	110
0	111
ARC loop counter	111
ARC machine directives	
ARC opcodes	116

ARC options	109
ARC Processor Identification register	111
ARC Program Counter	111
ARC register name prefix character	110
ARC register names	
ARC Saved User Stack Pointer	111
ARC stack pointer	111
ARC Status register	
ARC STATUS32 saved on exception	
ARC Stored STATUS32 register on entry to	
level P0 interrupts	111
ARC support	
ARC symbol prefix character	
ARC word aligned program counter	
arch directive, i386	
arch directive, M680x0	
arch directive, MSP 430	
arch directive, x86-64	
architecture options, IP2022	
architecture options, IP2K	203
architecture options, M16C architecture options, M32C	207
architecture options, M32C	207
architecture options, M32R	
architecture options, M32R2	
architecture options, $M32RX$	
architecture options, M680x0	
$eq:architecture variant option, CRIS \dots \dots \dots \dots$	
architectures, Meta	231
architectures, PowerPC	
architectures, SCORE	
architectures, SPARC	325
arguments for addition	. 48
arguments for subtraction	. 48
arguments in expressions	
arithmetic functions	
arithmetic operands	
ARM data relocations	
ARM floating point (IEEE)	
ARM identifiers	
ARM immediate character	
ARM line comment character	
ARM line separator	
ARM machine directives	
ARM opcodes	
ARM options (none)	
ARM register names	
ARM support	
ascii directive	
asciz directive	
asg directive, TIC54X	
assembler bugs, reporting	
assembler crash	
assembler directive .3byte, RX	
assembler directive .arch, CRIS	
assembler directive . dword, $\mbox{CRIS}\ldots\ldots\ldots$	
assembler directive .far, M68HC11	
assembler directive . fetchalign, $\operatorname{RX} \ldots \ldots$	
assembler directive .interrupt, M68HC11	

assembler directive .mode, M68HC11	224
assembler directive .relax, M68HC11	
assembler directive .syntax, CRIS	
assembler directive .xrefb, M68HC11	225
assembler directive BSPEC, MMIX	254
assembler directive BYTE, MMIX	253
assembler directive ESPEC, MMIX	254
assembler directive GREG, MMIX	252
assembler directive IS, MMIX	252
assembler directive LOC, MMIX	
assembler directive LOCAL, MMIX	
assembler directive OCTA, MMIX	
assembler directive PREFIX, MMIX	
assembler directive TETRA, MMIX	
assembler directive WYDE, MMIX	
assembler directives, CRIS	
assembler directives, M68HC11	
assembler directives, M68HC12	
assembler directives, MMIX	
assembler directives, RL78	
assembler directives, RX	
assembler directives, XGATE	
assembler internal logic error	
assembler version	
assembler, and linker	
assembly listings, enabling	
assigning values to symbols 43	
at register, MIPS	
att_syntax pseudo op, i386	189
att_syntax pseudo op, x86-64	
attributes, symbol	
auxiliary attributes, COFF symbols	
auxiliary symbol information, COFF	
AVR line comment character	
AVR line separator	
AVR modifiers	
AVR opcode summary	
AVR options (none)	
AVR register names	
AVR support	134

### В

backslash $(\backslash \backslash)$
backspace (\b)
balign directive 53
balignl directive 53
balignw directive 53
bes directive, TIC54X 342
bfloat16 directive, i386 195
bfloat16 directive, x86-64 195
big endian output, MIPS 15
big endian output, PJ14
big-endian output, MIPS233
big-endian output, TIC6X 346
bignums 35
binary constants, TIC54X
binary files, including 65

binary integers	34
bit names, IA-64	. 201
bitfields, not supported on VAX	. 368
Blackfin directives	. 144
Blackfin options (none)	. 142
Blackfin support	
Blackfin syntax	
block	. 393
BMI, i386	. 195
BMI, x86-64	
BPF line comment character	. 146
BPF opcodes	. 147
BPF options (none)	. 146
BPF register names	
BPF support	. 146
branch improvement, M680x0	
branch improvement, M68HC11	. 225
branch improvement, VAX	. 366
branch instructions, relaxation	
Branch Target Address, ARC	. 111
branch target alignment	. 379
break directive, TIC54X	
BSD syntax	. 277
bss directive	53
bss directive, TIC54X	. 339
bss section	38, 40
BSS directive	. 286
BTA saved on exception entry, ARC	. 111
bug criteria	. 395
bug reports	
bugs in assembler	
Build configuration for: BTA Registers, ARC.	. 112
Build configuration for: Core Registers, ARC.	
Build configuration for: Interrupts, ARC $\ldots$ .	. 112
Build Configuration Registers Version, ARC.	
Built-in symbols, CRIS	. 158
builtin math functions, TIC54X	. 337
builtin subsym functions, TIC54X	
bundle	
bundle-locked	
bundle_align_mode directive	
bundle_lock directive	
bundle_unlock directive	
bus lock prefixes, i386	. 193
bval	
byte directive	54
byte directive, TIC54X	. 339

### С

call instructions, relaxation	
call instructions, x86-64	191
call_hiadj directive, Nios II	
call_lo directive, Nios II	
carriage return (backslash-r)	
case sensitivity, Z80	
cfi_endproc directive	. 55
cfi_fde_data directive	. 55
cfi_personality directive	. 55
cfi_personality_id directive	. 55
cfi_sections directive	. 54
cfi_startproc directive	. 55
char directive, TIC54X	339
character constant, Z80	
character constants	
character escape codes	
character escapes, Z80	
character, single	
characters used in symbols	
clink directive, TIC54X	. 02
code16 directive, i386	106
code16gcc directive, i386	
code32 directive, i386	
code64 directive, i386	
code64 directive, x86-64	
COFF auxiliary symbol information	
COFF structure debugging	
COFF symbol attributes	
COFF symbol descriptor	. 60
COFF symbol storage class	
COFF symbol type	
COFF symbols, debugging	
COFF value attribute	
COMDAT	
comm directive	
command line conventions	. 22
command-line options ignored, VAX $\ldots\ldots\ldots$	365
command-line options, V850	357
comment character, XStormy16	375
comments	
comments, M680x0	220
comments, removed by preprocessor	
common directive, SPARC	334
common sections	
common variable storage	
comparison expressions	
conditional assembly	
constant, single character	
constants	
constants, bignum	
constants, character	
constants, converted by preprocessor	
constants, floating point	
constants, integer	
constants, number	
constants, number	
constants, sparc	
	- 29
constants, string	

conversion instructions, i386	190
conversion instructions, x86-64	
coprocessor wait, i386	
copy directive, TIC54X	
core general registers, ARC	
cpu directive, ARC	
cpu directive, M680x0	
cpu directive, MSP 430	
CR16 line comment character	
CR16 line separator	
CR16 Operand Qualifiers	155
CR16 support	155
crash of assembler	
CRISemulation=crisaout	
command-line option	157
CRISemulation=criself	
command-line option	157
CRISmarch=architecture	101
command-line option	157
CRISmul-bug-abort	107
command-line option	157
	197
CRISno-mul-bug-abort	155
command-line option	157
CRISno-underscore	
command-line option	
CRISpic command-line option	
$\mathrm{CRIS} \text{underscore command-line option} \ldots .$	
CRIS -N command-line option	157
CRIS architecture variant option	. 157
CRIS assembler directive .arch	
CRIS assembler directive .dword	160
CRIS assembler directive .syntax	. 160
CRIS assembler directives	
CRIS built-in symbols	
CRIS instruction expansion	
CRIS line comment characters	
CRIS options	
CRIS position-independent code	
CRIS pseudo-op .arch	
CRIS pseudo-op .dword	
CRIS pseudo-op .syntax	
CRIS pseudo-ops	
CRIS register names	
CRIS support	
CRIS symbols in position-independent code	
ctbp register, V850	
ctoff pseudo-op, V850	
ctpc register, V850	
ctpsw register, V850	
current address	
current address, advancing	
custom (vendor-defined) extensions, RISC-V $\ldots$	295

#### $\mathbf{D}$

d24 directive, Z80	
d32 directive, Z80	
D10V @word modifier	167
D10V addressing modes	
D10V floating point	167
D10V line comment character	165
D10V opcode summary	167
D10V optimization	. 12
D10V options	164
D10V registers	
D10V size modifiers	
D10V sub-instruction ordering	
D10V sub-instructions	
D10V support	
D10V syntax	
D30V addressing modes	
D30V floating point	
D30V Guarded Execution	
D30V line comment character	
D30V nops	
D30V nops after 32-bit multiply	
D30V opcode summary	
D30V optimization	
D30V options	
D30V registers	
D30V size modifiers	
D30V sub-instruction ordering	
D30V sub-instruction ordering	
D30V support	
D30V support	
data alignment on SPARC	
data and text sections, joining	
data directive	
data directive, TIC54X	
Data directives	
data relocations, ARM	12/
data section	
data section	
data2 directive, M680x0	
db directive, Z80	
dbpc register, V850	
dbpsw register, V850	
dc directive	
dcb directive	
DCCM RAM Configuration Register, ARC	
debuggers, and symbol order	
debugging COFF symbols	
DEC syntax	
decimal integers	
def directive	
def directive, TIC54X	
def24 directive, Z80	
def32 directive, Z80	
defb directive, Z80	
defl directive, Z80	
defm directive, Z80	
defs directive, Z80	388

defw directive, Z80	389
density instructions	
dependency tracking	
deprecated directives	
desc directive	
descriptor, of a.out symbol	. 46
dfloat directive, VAX	366
difference tables altered	. 88
difference tables, warning	. 26
differences, mmixal	
dim directive	. 60
directives and instructions	
directives for PowerPC	
directives for SCORE	320
directives, Blackfin	
directives, M32R	
directives, M680x0	
directives, machine independent	
directives, Xtensa	
directives, Z8000	
Disable floating-point instructions	
Disable single-precision	
floating-point operations	248
displacement sizing character, VAX	
dollar local symbols	
dot (symbol)	
double directive	
double directive, i386	
double directive, M680x0	
double directive, M68HC11	
double directive, RX	
double directive, TIC54X	
double directive, VAX	
double directive, x86-64	195
double directive, XGATE	374
doublequote (\")	
drlist directive, TIC54X	
drnolist directive, TIC54X	
ds directive	
ds directive, Z80	
DTP-relative data directives	
dw directive, Z80	
dword directive, BPF	
dword directive, DFF	
dword directive, PRU	
uworu ullective, 1 100	<u>200</u>

### $\mathbf{E}$

EB command-line option, C-SKY 162
EB command-line option, Nios II 267
ecr register, V850 361
eight-byte integer
eipc register, V850 361
eipsw register, V850
eject directive
EL command-line option, C-SKY 162
EL command-line option, Nios II 267
ELF symbol type

else directive	
elseif directive	
empty expressions	. 47
emsg directive, TIC54X	
emulation	
encoding options, i386	
encoding options, x86-64	190
end directive	. 60
endef directive	
endfunc directive	
endianness, MIPS	. 15
endianness, PJ	
endif directive	
endloop directive, TIC54X	
endm directive	. 72
endm directive, TIC54X	341
endproc directive, OpenRISC	
endstruct directive, TIC54X	343
endunion directive, TIC54X	343
environment settings, TIC54X	336
EOF, newline must precede	. 32
ep register, V850	
Epiphany line comment character	172
Epiphany line separator	172
Epiphany options	
Epiphany support	
equ directive	
equ directive, TIC54X	
equ directive, Z80	
equiv directive	
eqv directive	
err directive	
error directive	
error messages	
error on valid input	395
errors, caused by warnings	
errors, continuing after	
escape codes, character	
eval directive, TIC54X	339
even	
even directive, M680x0	
even directive, TIC54X	
Exception Cause Register, ARC	111
Exception Return Address, ARC	
exitm directive	
expr (internal section)	
expression arguments	
expressions	
expressions, comparison	
expressions, empty	
expressions, integer	
extAuxRegister directive, ARC	
extCondCode directive, ARC	
extCoreRegister directive, ARC	
extend directive M680x0	
extend directive M68HC11	
extend directive XGATE	
extension core registers, ARC	
J / ·····	_

extension instructions, i386 19	1
extension instructions, x86-64 19	1
extern directive	<b>52</b>
extInstruction directive, ARC11	4

### $\mathbf{F}$

fail directive	
<pre>far_mode directive, TIC54X</pre>	340
faster processing (-f)	26
fatal signal	395
fclist directive, TIC54X	340
fcnolist directive, TIC54X	340
fepc register, V850	361
fepsw register, V850	361
ffloat directive, VAX	
field directive, TIC54X	
file directive	
file directive, MSP 430	259
file name, logical	. 62
file names and line numbers, in	
warnings/errors	. 24
files, including	65
files, input	. 23
fill directive	
filling memory	
filling memory with no-op instructions	
filling memory with zero bytes	
FLIX syntax	
float directive	
float directive, i386	
float directive, M680x0	
float directive, M68HC11	
float directive, RX	
float directive, TIC54X	
float directive, VAX	
float directive, x86-64	
float directive, XGATE	
floating point numbers	
floating point numbers (double)	
floating point numbers (single) 63	, 82
floating point, AArch64 (IEEE)	
floating point, Alpha (IEEE)	105
floating point, ARM (IEEE)	125
floating point, D10V	167
floating point, D30V	
floating point, H8/300 (IEEE)	
floating point, HPPA (IEEE)	
floating point, i386	
floating point, M680x0	
floating point, M68HC11	
floating point, MSP 430 (IEEE)	
floating point, OPENRISC (IEEE)	
01 / ( /	290
01	302
	319
floating point, SH (IEEE)	323
floating point, SPARC (IEEE)	334

floating point, V850 (IEEE)
floating point, VAX
floating point, WebAssembly (IEEE) 371
floating point, x86-64 194
floating point, XGATE
floating point, Z80
flonums
force2bsr command-line option, C-SKY 162
format of error messages
format of warning messages
formfeed (\f)
four-byte integer
fpic command-line option, C-SKY 162
frame pointer, ARC 111
func directive
functions, in expressions

### G

gfloat directive, VAX 366
global
global directive 63
global directive, TIC54X
global pointer, ARC 111
got directive, Nios II
got_hiadj directive, Nios II
got_lo directive, Nios II 268
gotoff directive, Nios II 268
gotoff_hiadj directive, Nios II 268
gotoff_lo directive, Nios II
gp register, MIPS
gp register, V850
gprel directive, Nios II 267
grouping data

### $\mathbf{H}$

H8/300 addressing modes 173
H8/300 floating point (IEEE) 174
H8/300 line comment character 173
H8/300 line separator
H8/300 machine directives (none) 175
H8/300 opcode summary 175
H8/300 options 173
H8/300 registers
H8/300 size suffixes
H8/300 support
H8/300H, assembling for 175
half directive, BPF146
half directive, Nios II 268
half directive, SPARC
half directive, TIC54X 341
hex character code (\xd)
hexadecimal integers
hexadecimal prefix, S12Z 227
hexadecimal prefix, Z80
hfloat directive, i386 195
hfloat directive, VAX

hfloat directive, x86-64 195
hi directive, Nios II
hi pseudo-op, V850
hi0 pseudo-op, V850
hiadj directive, Nios II 267
<b>hidden</b> directive
<b>high</b> directive, M32R 210
hilo pseudo-op, V850
HPPA directives not supported 176
HPPA floating point (IEEE) 176
HPPA Syntax 176
HPPA-only directives 177
hword directive

## Ι

i386 16-bit code 196
i 386 arch directive $\ldots \ldots 196$
i386 att_syntax pseudo op 189
i386 conversion instructions 190
i386 extension instructions 191
i386 floating point 194
i386 immediate operands 189
i386 instruction naming 190
i386 instruction prefixes 192
i386 intel_syntax pseudo op 189
i386 jump optimization 194
i386 jump, call, return 189
i386 jump/call operands 189
i386 line comment character 189
i386 line separator 189
i386 memory references 193
i386 mnemonic compatibility 191
i386 mul, imul instructions 198
i386 options 181
i386 register operands 189
i386 registers 191
i386 sections 189
i386 size suffixes 189
i386 source, destination operands 189
i386 support 181
i386 syntax compatibility 189
i80386 support 181
IA-64 line comment character 201
IA-64 line separator
IA-64 options
IA-64 Processor-status-Register bit names 201
IA-64 registers
IA-64 relocations
IA-64 support
IA-64 Syntax
$\texttt{ident} \ directive \dots \dots 64$
identifiers, ARM $\ldots \ldots 124$
identifiers, MSP 430 $\dots \dots 257$
<b>if</b> directive
<b>ifb</b> directive
<b>ifc</b> directive
ifdef directive

ifeq directive	. 64
ifeqs directive	
ifge directive ifgt directive	
ifle directive	
iflt directive	
ifnb directive	
ifnc directive	. 65
ifndef directive	
ifne directive	
ifnes directive	
ifnotdef directive	
immediate character, AArch64	
immediate character, ARM	
immediate character, M680x0	220
immediate character, VAX	
immediate fields, relaxation	
immediate operands, i386	
immediate operands, x86-64	
imul instruction, i386	
imul instruction, x86-64	
incbin directiveinclude directive	
include directive search path	
indirect character, VAX	. 20 368
infix operators	
inhibiting interrupts, i386	. 40 193
input	
input file linenumbers	. 23
insn directive	
INSN directives	
instruction aliases, s390	
instruction bundle	
instruction expansion, CRIS	
instruction expansion, MMIX	250
instruction formats, risc-v	290
instruction formats, s390	
instruction marker, s390	
instruction mnemonics, s390	304
instruction naming, i386	
instruction naming, x86-64	190
instruction operand modifier, s390	
instruction operands, s390	
instruction prefixes, i386	
instruction set, M680x0instruction set, M68HC11	
instruction set, XGATE	
instruction summary, AVR	
instruction summary, D10V	167
instruction summary, D30V	171
instruction summary, H8/300	
instruction summary, LM32 206,	
instruction summary, SH	
instruction summary, Z8000	
instruction syntax, s390	
instructions and directives	. 33
int directive	
<b>int</b> directive, H8/300	175

int directive, i386 195
int directive, TIC54X 341
int directive, x86-64 195
integer expressions 47
integer, 16-byte
integer, 2-byte 89
integer, 4-byte 89
integer, 8-byte
integers
integers, 16-bit
integers, 32-bit
integers, binary 34
integers, decimal
integers, hexadecimal 34
integers, octal
integers, one byte 54
intel_syntax pseudo op, i386189
intel_syntax pseudo op, x86-64 189
internal assembler sections
internal directive
interrupt link register, ARC 111
Interrupt Vector Base address, ARC 111
invalid input
invocation summary1
IP2K architecture options
IP2K line comment character 203
IP2K line separator
IP2K options
IP2K support
irp directive
irpc directive

#### J

9
2
1
1
1
1
1
9
9

### $\mathbf{L}$

L16SI instructions, relaxation 382
L16UI instructions, relaxation 382
L32I instructions, relaxation 382
L8UI instructions, relaxation 382
label (:)
label directive, TIC54X 341
labels
labels, Z80 388
largecomm directive, ELF 186
lcomm directive
<b>lcomm</b> directive, COFF 186
lcommon directive, ARC $\dots$ 112

ld	
ldouble directive M680x0	218
ldouble directive M68HC11	
ldouble directive XGATE	374
ldouble directive, TIC54X	340
LDR reg,= <expr> pseudo op, AArch64</expr>	101
LDR reg,= <label> pseudo op, ARM</label>	130
LEB128 directives	286
length directive, TIC54X	341
length of symbols	. 32
level 1 interrupt link register, ARC	111
level 2 interrupt link register, ARC	111
lflags directive (ignored)	. 67
line	110
line comment character	
line comment character, AArch64	. 99
line comment character, Alpha	103
line comment character, ARC	
line comment character, ARM	124
line comment character, AVR	136
line comment character, BPF	
line comment character, CR16	
line comment character, D10V	165
line comment character, D30V	168
line comment character, Epiphany	172
line comment character, H8/300	173
line comment character, i386	189
line comment character, IA-64	201
line comment character, IP2K	
line comment character, LM32	206
line comment character, M32C	208
line comment character, M680x0	220
line comment character, M68HC11	222
line comment character, Meta	
line comment character, MicroBlaze $\ldots \ldots$	
line comment character, MIPS	248
line comment character, MSP 430	257
line comment character, Nios II	267
line comment character, NS32K	270
line comment character, $\operatorname{OpenRISC} \ldots \ldots \ldots$	
line comment character, PJ	279
line comment character, PowerPC	
line comment character, $\operatorname{PRU}$	
line comment character, RL78	
line comment character, RX	
line comment character, s390	
line comment character, S12Z	
line comment character, SCORE	321
line comment character, SH	
line comment character, Sparc	
line comment character, TIC54X	
line comment character, TIC6X	346
line comment character, V850	359
line comment character, VAX	
line comment character, Visium	
line comment character, WebAssembly	371
line comment character, XGATE	
line comment character, XStormv16	375

line comment character, Z80 $\ldots\ldots\ldots\ldots$	
line comment character, Z8000	
line comment characters, CRIS	159
line comment characters, MMIX	
line directive	
line directive, MSP 430	
line numbers, in input files	
line separator character	
line separator character, Nios II	
line separator, AArch64	
line separator, Alpha	103
line separator, ARC	110
line separator, ARM $\ldots\ldots\ldots$	124
line separator, AVR	
line separator, CR16 $\dots$	
line separator, Epiphany	
line separator, H8/300	
line separator, i386	
line separator, IA-64	
line separator, IP2K	
line separator, LM32	206
line separator, M32C	
line separator, M680x0	
line separator, M68HC11	
line separator, Meta	231
line separator, MicroBlaze	
line separator, MIPS	
line separator, MSP 430	
line separator, NS32K	270
line separator, OpenRISC	
line separator, PJ	
line separator, PowerPC	
line separator, RL78	
line separator, RX	
line separator, s390	
line separator, S12Z	
line separator, SCORE	
line separator, SH	322
line separator, Sparc	327
line separator, TIC54X	345
line separator, TIC6X	
line separator, V850	359
line separator, VAX	369
line separator, Visium	370
line separator, XGATE	373
line separator, XStormy16	
line separator, Z80	
line separator, Z8000	
lines starting with #	
link register, ARC	
linker	
linker, and assembler	
linkonce directive	
list directive	
list directive, TIC54X	
listing control, turning off	
listing control, turning on	
listing control: new page	. 60

listing control: paper size	. 76
listing control: subtitle	. 77
listing control: title line	
listings, enabling	. 25
literal directive	384
literal pool entries, s390	316
literal_position directive	385
literal_prefix directive	
little endian output, MIPS	
little endian output, PJ	
little-endian output, MIPS	
little-endian output, TIC6X	
LM32 line comment character	
LM32 line separator	
LM32 modifiers	
LM32 opcode summary	
LM32 options (none)	
LM32 register names	
LM32 support	
In directive	
lo directive, Nios II	
lo pseudo-op, V850	
loc directive	
loc_mark_labels directive	
local common symbols	
local directive	
local labels	
local symbol names	
local symbols, retaining in output	
location counter	
location counter, advancing	
location counter, Z80	
logical file name	. 62
logical line number	
logical line numbers	
long directive	
long directive, i386	
long directive, TIC54X	
long directive, x86-64	
longcall pseudo-op, V850	
longcalls directive	
longjump pseudo-op, V850	
Loongson Content Address Memory (CAM)	
generation override	248
Loongson EXTensions (EXT) instructions	
	248
Loongson EXTensions R2 (EXT2) instructions	-
generation override	248
Loongson MultiMedia extensions Instructions	
	248
loop counter, ARC	
loop directive, TIC54X	
LOOP instructions, alignment	
low directive, M32R	
lp register, V850	
lval	
LWP, i386	
LWP, x86-64	

# $\mathbf{M}$

M16C architecture option	207
M32C architecture option	207
M32C line comment character	208
M32C line separator	208
M32C modifiers	
M32C options	
M32C support	
M32R architecture options	
M32R directives	
M32R options	
M32R support	
M32R warnings	
M680x0 addressing modes	216
M680x0 architecture options	
M680x0 branch improvement	
M680x0 directives	
M680x0 floating point	
M680x0 immediate character	
M680x0 line comment character	
M680x0 line separator	
M680x0 opcodes M680x0 options	219
M680x0 pseudo-opcodes	
M680x0 size modifiers	
M680x0 support	
M680x0 syntax	
M68HC11 addressing modes	
M68HC11 and M68HC12 support	
M68HC11 assembler directive .far	
M68HC11 assembler directive .interrupt	
M68HC11 assembler directive .mode	
M68HC11 assembler directive .relax	
M68HC11 assembler directive .xrefb	
M68HC11 assembler directives	224
M68HC11 assembler directives M68HC11 branch improvement	$\begin{array}{c} 224 \\ 225 \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point	$224 \\ 225 \\ 225 \\ 225$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character	224 225 225 222
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separator	224 225 225 222 222 222
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiers	224 225 225 222 222 222 224
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodes	224 225 225 222 222 222 224 225
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 options	224 225 225 222 222 222 224 225 221
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodes	224 225 225 222 222 222 224 225 221
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 options	224 225 225 222 222 224 225 221 225
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodes	224 225 222 222 224 225 221 225 221 225 222
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodesM68HC11 syntax	224 225 222 222 224 225 221 225 221 225 222 224
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodesM68HC11 syntaxM68HC12 assembler directives	$\begin{array}{c} 224\\ 225\\ 225\\ 222\\ 222\\ 224\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109 \end{array}$
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodesM68HC11 syntaxM68HC12 assembler directivesmA6 command-line option, ARC	$\begin{array}{c} 224\\ 225\\ 225\\ 222\\ 224\\ 225\\ 221\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109\\ 109 \end{array}$
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodesM68HC11 syntaxM68HC12 assembler directivesmA6 command-line option, ARCmA7 command-line option, ARC	224 225 225 222 224 225 221 225 222 224 109 109 . 95
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 optionsM68HC11 pseudo-opcodesM68HC11 syntaxM68HC12 assembler directivesmA6 command-line option, ARCmachine dependencies	224 225 225 222 224 225 221 225 222 224 109 109 . 95
M68HC11 assembler directivesM68HC11 branch improvementM68HC11 floating pointM68HC11 line comment characterM68HC11 line separatorM68HC11 modifiersM68HC11 opcodesM68HC11 opcodesM68HC11 pseudo-opcodesM68HC11 syntaxM68HC12 assembler directivesM68HC12 assembler directivesmA6 command-line option, ARCmachine dependenciesmachine directives, AArch64machine directives, ARC	$\begin{array}{c} 224\\ 225\\ 225\\ 222\\ 222\\ 224\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109\\ 109\\ .95\\ 100\\ \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM	224 225 222 222 224 225 221 225 222 224 109 109 109 100 112
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF	224 225 222 222 224 225 221 225 222 224 109 109 . 95 100 112 125
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF machine directives, H8/300 (none)	$\begin{array}{c} 224\\ 225\\ 225\\ 222\\ 224\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109\\ 109\\ .95\\ 100\\ 112\\ 125\\ 146\\ 175 \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives mA6 command-line option, ARC mA7 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF machine directives, MSP 430	$\begin{array}{c} 224\\ 225\\ 222\\ 222\\ 222\\ 224\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109\\ 109\\ 1.95\\ 100\\ 112\\ 125\\ 146\\ 175\\ 259 \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC mA7 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF machine directives, MSP 430 machine directives, Nios II	$\begin{array}{c} 224\\ 225\\ 222\\ 222\\ 222\\ 224\\ 225\\ 221\\ 225\\ 222\\ 224\\ 109\\ 109\\ 1.95\\ 100\\ 112\\ 125\\ 146\\ 175\\ 259\\ 268 \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC mA7 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF machine directives, MSP 430 machine directives, Nios II machine directives, OPENRISC	$\begin{array}{c} 224\\ 225\\ 222\\ 222\\ 224\\ 225\\ 222\\ 224\\ 109\\ 109\\ .95\\ 100\\ 112\\ 125\\ 146\\ 175\\ 259\\ 268\\ 273\\ \end{array}$
M68HC11 assembler directives M68HC11 branch improvement M68HC11 floating point M68HC11 line comment character M68HC11 line separator M68HC11 modifiers M68HC11 opcodes M68HC11 options M68HC11 pseudo-opcodes M68HC11 syntax M68HC12 assembler directives M68HC12 assembler directives mA6 command-line option, ARC mA7 command-line option, ARC machine dependencies machine directives, AArch64 machine directives, ARM machine directives, BPF machine directives, MSP 430 machine directives, Nios II	$\begin{array}{c} 224\\ 225\\ 222\\ 222\\ 224\\ 225\\ 222\\ 224\\ 109\\ 109\\ .95\\ 100\\ 112\\ 125\\ 146\\ 175\\ 259\\ 268\\ 273\\ 283\\ 283\\ \end{array}$

machine directives, SH	. 324
machine directives, SPARC	. 334
machine directives, TIC54X	. 339
machine directives, TIC6X machine directives, TILE-Gx	. 347
machine directives, TILE-Gx	. 352
machine directives, TILEPro	. 357
machine directives, V850	. 362
machine directives, VAX	
machine directives, x86	
machine directives, XStormy16	. 375
machine independent directives	
machine instructions (not covered)	
machine relocations, Nios II	
machine relocations, PRU	
machine-independent syntax	
macro directive	
macro directive, TIC54X	
macros	
macros, count executed	
Macros, MSP 430	
macros, TIC54X	
make rules	
manual, structure and purpose	
marc600 command-line option, ARC	
mARC601 command-line option, ARC	
mARC700 command-line option, ARC	
march command-line option, C-SKY	
march command-line option, Nios II	
math builtins, TIC54X	
Maximum number of continuation lines	
mbig-endian command-line option, C-SKY	
mbranch-stub command-line option, C-SKY	
mcache command-line option, C-SKY	
mcp command-line option, C-SKY	
mcpu command-line option, C-SKY	. 162
mdsp command-line option, C-SKY	
medsp command-line option, C-SKY	
melrw command-line option, C-SKY	
mEM command-line option, ARC	
memory references, i386	
memory references, x86-64	
memory-mapped registers, TIC54X	
merging text and data sections	
messages from assembler	
Meta architectures	
Meta line comment character	
Meta line separator	
Meta options	
Meta registers	
Meta support	
mforce2bsr command-line option, C-SKY	162
mhard-float command-line option, C-SKY	
mHS command-line option, ARC	. 109
MicroBlaze architectures	
MicroBlaze directives	
MicroBlaze line comment character	
MicroBlaze line separator	
MicroBlaze support	
	. 404

minus, permitted arguments
generation override
MIPS architecture options
MIPS big-endian output 233
MIPS CPU override
MIPS cyclic redundancy check (CRC)
instruction generation override
command-line options 243
MIPS DSP Release 1 instruction generation override
MIPS DSP Release 2 instruction
generation override
MIPS DSP Release 3 instruction
generation override
MIPS endianness 15
MIPS eXtended Physical Address (XPA)
instruction generation override
generation override
MIPS IEEE 754 NaN data
encoding selection 246
MIPS ISA 15
MIPS ISA override
MIPS line comment character 248
MIPS line separator
MIPS little-endian output
MIPS MCU instruction generation override 247
MIPS MDMX instruction
generation override
MIPS MIPS-3D instruction
generation override
MIPS MT instruction generation override 247
MIPS option stack
MIPS processor
MIPS SIMD Architecture instruction
generation override
MIPS16e2 instruction generation override 247
mistack command-line option, C-SKY 163
MIT
<code>mjsri2bsr</code> command-line option, C-SKY $\dots$ 162
mlabr command-line option, C-SKY 163
mlaf command-line option, C-SKY 163
mlib directive, TIC54X 342
mlink-relax command-line option, PRU 283
mlist directive, TIC54X
mliterals-after-br command-line
option, C-SKY 163
mliterals-after-func
command-line option, C-SKY 163
mlittle-endian command-line
option, C-SKY 162
mljump command-line option, C-SKY 162
MMIX assembler directive BSPEC
MMIX assembler directive BSFEC
MMIX assembler directive BYTE
MMIX assembler directive ESPEC
MMIX assembler directive GREG $\ldots \ldots 252$

MMIX assembler directive IS	252
MMIX assembler directive LOC	
MMIX assembler directive LOCAL	
MMIX assembler directive OCTA	
MMIX assembler directive PREFIX	
MMIX assembler directive TETRA	
MMIX assembler directive WYDE	253
MMIX assembler directives	
MMIX line comment characters	
MMIX options	
MMIX pseudo-op BSPEC	254
MMIX pseudo-op BYTE	
MMIX pseudo-op ESPEC	
MMIX pseudo-op GREG	
MMIX pseudo-op IS	
MMIX pseudo-op LOC	
MMIX pseudo-op LOCAL	
MMIX pseudo-op DCCTA	252
MMIX pseudo-op PREFIX	200
MMIX pseudo-op TETRA	204
MMIX pseudo-op WYDE	
MMIX pseudo-ops	
MMIX register names	
MMIX support	
mmixal differences	
mmp command-line option, C-SKY	163
mmregs directive, TIC54X	
mmsg directive, TIC54X	
MMX, i386	
MMX, x86-64	
mnemonic compatibility, i386	191
mnemonic suffixes, i386	189
mnemonic suffixes, x86-64	
mnemonics for opcodes, VAX	
mnemonics, AVR	
mnemonics, D10V	
mnemonics, $D30V$	
mnemonics, H8/300	
mnemonics, LM32	206
mnemonics, $OpenRISC \dots$	
mnemonics, SH	
mnemonics, Z8000	393
mno-branch-stub command-line	
option, C-SKY	
<pre>mno-elrw command-line option, C-SKY</pre>	163
mno-force2bsr command-line	
option, C-SKY	162
mno-istack command-line option, C-SKY	163
mno-jsri2bsr command-line option, C-SKY	162
mno-labr command-line option, C-SKY	163
mno-laf command-line option, C-SKY	163
mno-link-relax command-line option, PRU	283
mno-literals-after-func	
command-line option, C-SKY	
mno-ljump command-line option, C-SKY	163
<pre>mno-ljump command-line option, C-SKY mno-lrw command-line option, C-SKY</pre>	163
mno-lrw command-line option, C-SKY	$163 \\ 162$
	163 162 162

mnolist directive, TIC54X
mnoliterals-after-br
command-line option, C-SKY 163
mnolrw command-line option, C-SKY 162
mnps400 command-line option, ARC 110
modifiers, M32C 207
module layout, WebAssembly 371
Motorola syntax for the $680 \mathrm{x} \mathrm{0} \ldots \ldots 217$
MOVI instructions, relaxation 382
MOVN, MOVZ and MOVK group
relocations, AArch64
MOVW and MOVT relocations, ARM $\ldots \ldots 124$
<b>mri</b> directive
MRI compatibility mode 27
MRI mode, temporarily
msecurity command-line option, C-SKY 163
MSP 430 floating point (IEEE) 259
MSP 430 identifiers $\ldots \ldots \ldots 257$
MSP 430 line comment character $\ldots \ldots 257$
MSP 430 line separator $\ldots \ldots 257$
MSP 430 machine directives 259
MSP 430 macros
MSP 430 opcodes 259
MSP 430 options (none) 256
MSP 430 profiling capability 259
MSP 430 register names 258
MSP 430 support
MSP430 Assembler Extensions 258
mspabi_attribute directive, MSP430 259
mtrust command-line option, C-SKY 163
<b>mul</b> instruction, i386 198
<b>mul</b> instruction, x86-64198
mvdsp command-line option, C-SKY163

## $\mathbf{N}$

N32K support
name
named section
named sections 38
names, symbol
naming object file
NDS32 options
NDS32 processor
new page, in listings 60
newblock directive, TIC54X
newline (\n)
newline, required at file end 32
Nios II line comment character
Nios II line separator character
Nios II machine directives
Nios II machine relocations
Nios II opcodes
Nios II options
Nios II support
Nios support
no-absolute-literals directive
no-force2bsr command-line option, C-SKY 162

no-jsri2bsr command-line option, C-SKY 162
no-longcalls directive 384
no-relax command-line option, Nios II 267
no-schedule directive 383
no-transform directive
nodelay directive, OpenRISC 273
<b>nolist</b> directive
nolist directive, TIC54X
<b>nop</b> directive
NOP pseudo op, ARM 130
<b>nops</b> directive
notes for Alpha 102
notes for WebAssembly 371
NS32K line comment character
NS32K line separator
null-terminated strings
number constants $\dots 34$
number of macros executed
numbered subsections
numbers, 16-bit 64
numeric values
nword directive, SPARC

# 0

Object Attribute, RISC-V 295
object attributes
object file
object file format 22
object file name 29
object file, after errors 30
obsolescent directives 89
octa directive
octal character code (\ddd) 34
octal integers 34
offset directive
offset directive, V850
opcode mnemonics, VAX
opcode names, TILE-Gx 349
opcode names, TILEPro 354
opcode names, Xtensa 378
opcode summary, AVR137
opcode summary, D10V 167
opcode summary, D30V 171
opcode summary, H8/300 175
opcode summary, LM32 206
opcode summary, OpenRISC 274
opcode summary, SH 324
opcode summary, Z8000 393
opcodes for AArch64101
opcodes for ARC 116
opcodes for ARM 130
opcodes for BPF147
opcodes for MSP 430 259
opcodes for Nios II
opcodes for PRU
opcodes for V850 362
opcodes, M680x0 219

opcodes, M68HC11	225
opcodes, WebAssembly	
OPENRISC floating point (IEEE)	
OpenRISC line comment character	
OpenRISC line separator	
OPENRISC machine directives	
OpenRISC opcode summary	
OpenRISC registers	
OpenRISC relocations	
OPENRISC support	
OPENRISC syntax	
operand delimiters, i386	
operand delimiters, x86-64	
operand notation, VAX	368
operands in expressions	. 47
operator precedence	. 48
operators, in expressions	
operators, permitted arguments	
optimization, D10V	
optimization, D30V	
optimizations	
Option directive	
option directive	
option directive. TIC54X	
option summary	1
options for AArch64 (none)	. 96
options for Alpha	
options for ARC	
options for ARM (none)	
options for AVR (none)	
options for Blackfin (none)	
options for BPF (none)	
options for C-SKY	162
options for i386	
options for IA-64	200
options for LM32 (none)	
options for Meta	231
options for MSP430 (none)	
options for NDS32	
options for Nios II	
options for PDP-11	
options for PowerPC	
options for PRU	
options for s390	
options for SCORE	200
options for SPARC	
options for TIC6X	
options for V850 (none)	
options for VAX/VMS	
options for Visium	
options for x86-64	
options for Z80	
options, all versions of assembler	
options, command line	. 22
options, CRIS	157
options, D10V	
options, D30V	168
options, Epiphany	172
- / * * *	

ptions, $H8/300$ 17	3
ptions, IP2K	3
ptions, M32C 20	7
ptions, $M32R$ 20	9
pptions, $M680x021$	3
ptions, M68HC11 22	
ptions, MMIX 24	9
ptions, PJ 27	9
ptions, RL78 29	8
ptions, RX 30	0
pptions, $S12Z$ 22	7
pptions, $SH \dots 32$	2
ptions, $TIC54X33$	6
ptions, XGATE 37	3
ptions, Z8000 39	1
rg directive	3
ther attribute, of a.out symbol 4	6
output file         2	3
output section padding	8

### $\mathbf{P}$

p2align directive
p2alignl directive
p2alignw directive
padding the location counter 51
padding the location counter given
a power of two
padding the location counter given
number of bytes 53
page, in listings 60
paper size, for listings
paths for .include
patterns, writing in memory
PDP-11 comments
PDP-11 floating-point register syntax 277
PDP-11 general-purpose register syntax 277
PDP-11 instruction naming 277
PDP-11 line separator 277
PDP-11 support
PDP-11 syntax
pic command-line option, C-SKY 162
PIC code generation for ARM 123
PIC code generation for M32R 209
PIC selection, MIPS 233
PJ endianness 14
PJ line comment character 279
PJ line separator 279
PJ options
PJ support
plus, permitted arguments 48
pmem directive, PRU 283
popsection directive
Position-independent code, CRIS 157
Position-independent code, symbols in, CRIS 159
PowerPC architectures
PowerPC directives
PowerPC line comment character 282

PowerPC line separator	
PowerPC options	
PowerPC support	
precedence of operators	
precision, floating point	
prefix operators	. 47
prefixes, i386	192
preprocessing	. 31
preprocessing, turning on and off	. 31
previous directive	
primary attributes, COFF symbols	. 46
print directive	. 75
proc directive, OpenRISC	273
proc directive, SPARC	
Processor Identification register, ARC	
profiler directive, MSP 430	
profiling capability for MSP 430	
Program Counter, ARC	
protected directive	
PRU line comment character	
PRU machine directives	
PRU machine relocations	
PRU opcodes	
PRU options	
-	
PRU support	
psect directive, Z80	
pseudo map fd, BPF	140
pseudo-op .arch, CRIS	
pseudo-op .dword, CRIS	
pseudo-op .syntax, CRIS	160
pseudo-op BSPEC, MMIX	254
pseudo-op BYTE, MMIX	253
pseudo-op ESPEC, MMIX	
pseudo-op GREG, MMIX	
pseudo-op IS, MMIX	
pseudo-op LOC, MMIX	
pseudo-op LOCAL, MMIX	
pseudo-op OCTA, MMIX	
pseudo-op PREFIX, MMIX	
pseudo-op TETRA, MMIX	253
pseudo-op WYDE, MMIX	253
pseudo-opcodes for XStormy16	375
pseudo-opcodes, M680x0	219
pseudo-opcodes, M68HC11	
pseudo-ops for branch, VAX	366
pseudo-ops, CRIS	
pseudo-ops, machine independent	
pseudo-ops, MMIX	
psize directive	
PSR bits	
pstring directive, TIC54X	342
psw register, V850	
purgem directive	
purpose of GNU assembler	
pushsection directive	
	. 10

# $\mathbf{Q}$

quad directive	. 76
<b>quad</b> directive, i386	195
<b>quad</b> directive, x86-64	195

### $\mathbf{R}$

real-mode code, i386	196
ref directive, TIC54X	341
refsym directive, MSP 430	
register directive, SPARC	
register name prefix character, ARC	
register names, AArch64	. 99
register names, Alpha	
register names, ARC	
register names, ARM	
register names, AVR	
register names, BPF	
register names, CRIS	160
register names, H8/300	
register names, IA-64	201
register names, LM32	204
register names, MMIX	251
register names, MSP 430	258
register names, OpenRISC	271
register names, S12Z	228
register names, Sparc	327
register names, TILE-Gx	350
register names, TILEPro	354
register names, V850	359
register names, VAX	368
register names, Visium	370
register names, Xtensa	378
register names, Z80	388
register naming, s390	
register notation, $S12Z$	229
register operands, i386	189
register operands, x86-64	189
registers, $D10V$	166
registers, $D30V$	
registers, i386	191
registers, Meta	
registers, SH	
registers, TIC54X memory-mapped	345
registers, x86-64	191
registers, Z8000	
$\texttt{relax-all} \ command-line \ option, \ Nios \ II \ \ldots \ldots$	267
relax-section command-line	
option, Nios II	
relaxation	
relaxation of ADDI instructions $\ldots \ldots \ldots$	
$relaxation \ of \ branch \ instructions \ldots \ldots \ldots$	380
relaxation of call instructions	380
$relaxation \ of \ immediate \ fields \ldots \ldots \ldots$	
${\rm relaxation \ of \ jump \ instructions} \ldots \ldots \ldots$	
${\rm relaxation}~{\rm of}~{\tt L16SI}~{\rm instructions}\ldots\ldots\ldots\ldots$	
${\rm relaxation}~{\rm of}~\tt L16UI~{\rm instructions}\ldots\ldots\ldots\ldots$	382
relaxation of L32I instructions	382

relaxation of L8UI instructions
relaxation of MOVI instructions 382
<b>reloc</b> directive
relocation
relocation example
relocations, AArch64
relocations, Alpha 103
relocations, OpenRISC 271
relocations, Sparc
relocations, WebAssembly
repeat prefixes, i386 193
reporting bugs in assembler
<b>rept</b> directive
reserve directive, SPARC
return instructions, i386 189
return instructions, x86-64 189
REX prefixes, i386 193
RISC-V custom (vendor-defined) extensions 295
RISC-V floating point (IEEE)
RISC-V instruction formats
RISC-V machine directives
RISC-V support
RL78 assembler directives
RL78 line comment character 299
RL78 line separator
RL78 modifiers
RL78 options
RL78 support
rsect
RX assembler directive .3byte
RX assembler directive .fetchalign
RX assembler directives
RX floating point
RX line comment character 302
RX line separator
RX modifiers
RX options
RX support

### $\mathbf{S}$

s390 floating point
s390 instruction aliases
s390 instruction formats
s390 instruction marker 316
s390 instruction mnemonics
s390 instruction operand modifier 315
s390 instruction operands
s390 instruction syntax
s390 line comment character 304
s390 line separator
s390 literal pool entries
s390 options
s390 register naming
s390 support
S12Z addressing modes
S12Z line separator
S12Z options

S12Z support	
S12Z syntax	
Saved User Stack Pointer, ARC	111
sblock directive, TIC54X	342
sbttl directive	77
schedule directive	383
scl directive	77
SCORE architectures	320
SCORE directives	
SCORE line comment character	321
SCORE line separator	. 321
SCORE options	
SCORE processor	
sdaoff pseudo-op, V850	
search path for .include	
sect directive, TIC54X	
section directive (COFF version)	
section directive (ELF version)	11
section directive (EEF version)	362
section name substitution	
section override prefixes, i386	
Section Stack	18, 84
section-relative addressing	
sections	
sections in messages, internal	
sections, i386	
sections, named	
sections, x86-64	189
seg directive, SPARC	
segm	
set at directive, Nios II	
set break directive, Nios II	
set directive	
set directive, Nios II	
set directive, TIC54X	
set no_warn_regname_label directive, PRU.	284
set noat directive, Nios II	268
set nobreak directive, Nios II	269
set norelax directive, Nios II	
set relaxall directive, Nios II	
set relaxsection directive, Nios II	
SH addressing modes	
SH floating point (IEEE)	
SH line comment character	
SH line separator	
SH machine directives	
SH opcode summary	
SH options	
SH registers	
SH support	
shigh directive, M32R	
short directive	
short directive, TIC54X	
signatures, WebAssembly	
SIMD, i386	
SIMD, x86-64	
single character constant	
single directive	82

<b>single</b> directive, i386 198	
single directive, x86-64 198	5
single quote, Z80	8
sixteen bit integers	
sixteen byte integer	
size directive (COFF version)	2
size directive (ELF version)	
size modifiers, D10V 164	1±
size modifiers, D30V 168	ð
size modifiers, M680x0	
size prefixes, i386 195	2
size suffixes, H8/300 175	
size, translations, Sparc 333	3
sizes operands, i386 189	9
sizes operands, x86-64 189	9
skip directive	
skip directive, M680x0 218	
skip directive, SPARC	
sleb128 directive	
small data, MIPS	
SmartMIPS instruction generation override 24'	
SOM symbol attributes 46	
source program 23	
source, destination operands; i386 189	
source, destination operands; x86-64 189	9
sp register	8
sp register, V850	9
space directive	
space directive, TIC54X	
space used, maximum for assembly	
Spare constants	n
Sparc line comment character	
Sparc line separator	
Sparc registers	
Sparc relocations	
Sparc size translations 333	
SPARC architectures 328	
SPARC data alignment 32'	
SPARC floating point (IEEE)	4
SPARC machine directives	4
SPARC options	
	5
SPARC support 329	
SPARC support	5
SPARC syntax	5 7
SPARC syntax	5 7 0
SPARC syntax	5 7 0 8
SPARC syntax	5 7 0 8 2
SPARC syntax	5 7 0 8 2 2
SPARC syntax	5 7 0 8 2 3
SPARC syntax	5 7 0 8 2 3
SPARC syntax	5 7 0 8 2 3 3
SPARC syntax	5708223333
SPARC syntax.32'special characters, M680x0.220special purpose registers, MSP 430.250sslist directive, TIC54X.34'ssnolist directive, TIC54X.34'stabd directive.8'stabh directive.8'	57082233333
SPARC syntax	57082233331
SPARC syntax	57082233317
SPARC syntax	570822333172
SPARC syntax.32'special characters, M680x0.220special purpose registers, MSP 430.250sslist directive, TIC54X.34'stabd directive.80stabd directive.80stabs directive.80stabs directive.80stabx directives.80stack pointer, ARC11standard assembler sections3'statement separator character3'	5708223331722
SPARC syntax.32'special characters, M680x0.220special purpose registers, MSP 430.250sslist directive, TIC54X.34'stabd directive, TIC54X.34'stabd directive.80stabd directive.80stabs directive.80stabs directive.80stabx directives.80stack pointer, ARC11standard assembler sections.3'statement separator character3'statement separator, AArch64.90	57082233317229
SPARC syntax.32'special characters, M680x0.220special purpose registers, MSP 430.250sslist directive, TIC54X.34'stabd directive.80stabd directive.80stabs directive.80stabs directive.80stabx directives.80stack pointer, ARC11standard assembler sections3'statement separator character3'	570822333172293

statement separator, ARM 124	1
statement separator, AVR	
statement separator, BPF146	3
statement separator, CR16 156	
statement separator, Epiphany 172	2
statement separator, H8/300 173	
statement separator, i386 189	
statement separator, IA-64	
statement separator, IP2K 203	
statement separator, LM32 206	
statement separator, M32C 208	
statement separator, M68HC11 222	
statement separator, Meta 231	
statement separator, MicroBlaze 232	
statement separator, MIPS 248	
statement separator, MSP 430 257	
statement separator, NS32K 270	
statement separator, OpenRISC271	l
statement separator, PJ	)
statement separator, PowerPC 282	2
statement separator, RL78 299	)
statement separator, RX 302	
statement separator, s390 304	1
statement separator, S12Z 228	
statement separator, SCORE 321	l
statement separator, SH 322	2
statement separator, Sparc	7
statement separator, TIC54X 348	5
statement separator, TIC6X 346	3
statement separator, V850 359	)
statement separator, VAX 369	)
statement separator, Visium 370	)
statement separator, XGATE 373	3
statement separator, XStormy16 375	
statement separator, Z80	3
statement separator, Z8000 391	l
statements, structure of 32	
statistics, about assembly 29	
Status register, ARC 111	
STATUS32 saved on exception, ARC 111	L
stopping the assembly	l
Stored STATUS32 register on entry to level	
P0 interrupts, ARC 111	
string constants	3
string directive	
string directive on HPPA 178	
string directive, TIC54X	2
string literals 52	
string, copying to object file	1
string16 directive	1
string16, copying to object file 84	1
<b>string32</b> directive	
string32, copying to object file 84	1
string64 directive	
string64, copying to object file	
string8 directive	1
string8, copying to object file	1
struct directive	1

struct directive, TIC54X	
structure debugging, COFF	
sub-instruction ordering, D10V	
sub-instruction ordering, D30V	
sub-instructions, D10V	
sub-instructions, D30V	
subexpressions	
subsection directive	. $84$
subsym builtins, TIC54X	
subtitles for listings	. 77
subtraction, permitted arguments	. 48
summary of options	
support	
supporting files, including	
suppressing warnings	
sval	
symbol attributes	
symbol attributes, a.out	
symbol attributes, COFF	
symbol attributes, SOM	
symbol descriptor, COFF	
symbol modifiers 136, 205, 207,	224
symbol modifiers, TILE-Gx	350
symbol modifiers, TILEPro	355
symbol names	
symbol names, '\$' in 165, 169, 231,	
symbol names, local	
symbol names, temporary	
symbol prefix character, ARC	110
symbol storage class (COFF)	. 77
symbol type	. 46
symbol type, COFF	. 86
symbol type, ELF	. 86
symbol value	
symbol value, setting	
symbol values, assigning	
symbol versioning	
symbol, common	
symbol, making visible to linker	
symbolic debuggers, information for	
symbols	
symbols with uppercase, VAX/VMS	
symbols, assigning values to	
Symbols, built-in, CRIS	
Symbols, CRIS, built-in	
symbols, local common	
symver directive	
syntax compatibility, i386	
syntax compatibility, x86-64	189
syntax, AVR	136
syntax, Blackfin	142
syntax, D10V	164
syntax, D30V	
syntax, LM32	
syntax, M680x0	
syntax, M68HC11	
syntax, machine-independent	· 01

syntax, OPENRISC271
syntax, RL78
syntax, RX 301
syntax, S12Z 227
syntax, SPARC
syntax, TILE-Gx 349
syntax, TILEPro
syntax, XGATE
syntax, Xtensa assembler 377

#### $\mathbf{T}$

tab (\t)	. 33
tab directive, TIC54X	343
tag directive	
tag directive, TIC54X	
TBM, i386	
TBM, x86-64	
tdaoff pseudo-op, V850	
temporary symbol names	44
text and data sections, joining	
text directive	
text section	
tfloat directive, i386	
tfloat directive, x86-64	
Thumb support	117
TIC54X builtin math functions	
TIC54X line comment character	
TIC54X line separator	345
TIC54X machine directives	
TIC54X memory-mapped registers	345
TIC54X options	336
TIC54X subsym builtins	344
TIC54X support	336
TIC54X-specific macros	344
TIC6X big-endian output	346
TIC6X line comment character	. 346
TIC6X line separator	346
TIC6X little-endian output	346
TIC6X machine directives	. 347
TIC6X options	. 346
TIC6X support	346
TILE-Gx machine directives	352
TILE-Gx modifiers	350
TILE-Gx opcode names	349
TILE-Gx register names	
TILE-Gx support	
TILE-Gx syntax	349
TILEPro machine directives	
TILEPro modifiers	
TILEPro opcode names	
TILEPro register names	
TILEPro support	
TILEPro syntax	
time, total for assembly	. 29
title directive	86
tls_common directive	
tls_gd directive, Nios II	
<b>~</b>	

tls_ie directive, Nios II 268
tls_ldm directive, Nios II 268
tls_ldo directive, Nios II 268
tls_le directive, Nios II 268
TMS320C6X support
tp register, V850
transform directive
trusted compiler
turning preprocessing on and off 31
two-byte integer
type directive (COFF version) 86
type directive (ELF version) 86
type of a symbol 46

### U

ualong directive, SH
uaquad directive, SH
uaword directive, SH
ubyte directive, TIC54X 339
uchar directive, TIC54X 339
uhalf directive, TIC54X 341
uint directive, TIC54X 341
<b>uleb128</b> directive
ulong directive, TIC54X 341
undefined section
union directive, TIC54X 343
unsegm
usect directive, TIC54X 343
ushort directive, TIC54X
uword directive, TIC54X 341

#### $\mathbf{V}$

V850 command-line options
V850 floating point (IEEE)
V850 line comment character 359
V850 line separator
V850 machine directives
V850 opcodes
V850 options (none) 357
V850 register names 359
V850 support
val directive
value attribute, COFF 87
value directive 186
value of a symbol 45
var directive, TIC54X 344
Vax-11 C compatibility 365
VAX bitfields not supported 368
VAX branch improvement
VAX command-line options ignored 365
VAX displacement sizing character
VAX floating point
VAX immediate character
VAX indirect character
VAX line comment character
VAX line separator

VAX machine directives
VAX opcode mnemonics
VAX operand notation
VAX register names
VAX support
VAX/VMS options
version directive
version directive, TIC54X
version of assembler
versions of symbols
Virtualization instruction
generation override
visibility
Visium line comment character
Visium line separator
Visium options
Visium registers
Visium support
VMS (VAX) options
vtable_entry directive
vtable_inherit directive 87

#### $\mathbf{W}$

warning directive
warning for altered difference tables
warning messages
warnings, causing error
warnings, M32R 211
warnings, suppressing 29
warnings, switching on 30
weak directive
weakref directive
WebAssembly floating point (IEEE)
WebAssembly line comment character 371
WebAssembly module layout
WebAssembly notes
WebAssembly opcodes 371
WebAssembly relocations 371
WebAssembly signatures 371
WebAssembly support
WebAssembly Syntax
whitespace
whitespace, removed by preprocessor 31
wide floating point directives, VAX
width directive, TIC54X 341
Width of continuation lines of
disassembly output
Width of first line disassembly output 27
Width of source line output 27
wmsg directive, TIC54X 340
word aligned program counter, ARC 111
word directive
word directive, $BPF146$
word directive, H8/300 175
word directive, i386 195
word directive, Nios II 268
word directive, OpenRISC 273

word directive, PRU	283
word directive, SPARC	335
word directive, TIC54X	341
word directive, x86-64	195
writing patterns in memory	63
wval	392

## Х

x86 machine directives 186
x86-64 arch directive
x86-64 att_syntax pseudo op 189
x86-64 conversion instructions 190
x86-64 extension instructions 191
$x86-64 \text{ floating point} \dots 194$
x86-64 immediate operands 189
x86-64 instruction naming 190
x86-64 intel_syntax pseudo op 189
x86-64 jump optimization 194
x86-64 jump, call, return
x86-64 jump/call operands 189
$x86-64 \text{ memory references} \dots 193$
x86-64 options 181
$x86-64 \text{ register operands} \dots 189$
x86-64 registers 191
x86-64 sections 189
$x86-64 \text{ size suffixes} \dots 189$
x86-64 source, destination operands 189
x86-64 support
x86-64 syntax compatibility 189
<b>xdef</b> directive, Z80
xfloat directive, TIC54X
XGATE addressing modes
XGATE assembler directives
XGATE floating point
XGATE line comment character 373
XGATE line separator
XGATE opcodes
XGATE options 373
XGATE support
XGATE syntax
xlong directive, TIC54X 341

xref directive, Z80	390
XStormy16 comment character	375
XStormy16 line comment character	375
XStormy16 line separator	375
XStormy16 machine directives	375
XStormy16 pseudo-opcodes	375
XStormy16 support	375
Xtensa architecture	376
Xtensa assembler syntax	377
Xtensa directives	383
Xtensa opcode names	378
Xtensa register names	378
xword directive, SPARC	335

## $\mathbf{Z}$

Z80 \$ 388
Z80 ' 388
Z80 floating point
Z80 labels
Z80 line comment character 388
Z80 line separator
Z80 options
Z80 registers
Z80 support
Z80 Syntax
Z80, \
Z80, case sensitivity
Z80-only directives
Z800 addressing modes
Z8000 directives
Z8000 line comment character
Z8000 line separator
Z8000 opcode summary 393
Z8000 options
Z8000 registers
Z8000 support
<b>zdaoff</b> pseudo-op, V850 364
<b>zero</b> directive
<b>zero</b> register, V850 359
zero-terminated strings