

XuanTie-C908-UserManual

Nov 06, 2023

Copyright 2021 T-Head Semiconductor Co., Ltd.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

History

Version	Description	Date
01	C908 first version	2021.10.19
02	change pmpaddr table	2021.10.25
03	second version	2023.08.23

1	Introduction	1
1.1	Features	1
1.1.1	Features of C908MP Architectural	1
1.1.2	Features of C908 Core	2
1.1.3	Features of C908 vector computing units	3
1.2	Configuration options	3
1.3	XuanTie extended architecture	6
1.4	Version compatibility	6
1.5	Terms	6
2	C908MP Overview	8
2.1	Structure	8
2.2	In-core subsystems	8
2.2.1	IFU	8
2.2.2	IEU	8
2.2.3	VFPU	10
2.2.4	LSU	10
2.2.5	MMU	10
2.2.6	PMP	11
2.3	Multi-core subsystems	11
2.3.1	CIU	11
2.3.2	L2 cache	11
2.3.3	Master device interface unit	11
2.3.4	DCP	11
2.3.5	LLP	12
2.4	Multi-cluster subsystems	12
2.4.1	PIC	12

2.4.2	Timer	12
2.4.3	Debug system	12
2.5	Interface overview	13
3	Instruction Sets	14
3.1	RV base instruction sets	14
3.1.1	Integer instruction set (RV64I)	14
3.1.2	Multiply/Divide instruction set (RV64M)	17
3.1.3	Atomic instruction set (RV64A)	18
3.1.4	Single-precision floating-point instruction set	19
3.1.5	Compressed instruction set (RV64C)	22
3.1.6	Vector instruction set (RVV)	24
3.1.7	Bit operation instruction set (RV64B)	24
3.2	XuanTie extended instruction sets	24
3.2.1	Arithmetic operation instructions	24
3.2.2	Bit operation instructions	25
3.2.3	Memory access instructions	26
3.2.4	Cache instructions	31
3.2.5	Multi-core synchronization instructions	33
3.2.6	Half-precision floating-point instructions	33
4	CPU Modes and Registers	37
4.1	CPU modes	37
4.2	Register view	38
4.3	General-purpose registers	38
4.4	Floating-point registers	39
4.4.1	Transmit data between floating-point and general-purpose registers	40
4.4.2	Maintain consistency of register precision	40
4.5	Vector registers	40
4.5.1	Transmit data between floating-point and general-purpose registers	40
4.5.2	Transmit data between floating-point and vector registers	41
4.6	System control registers	41
4.6.1	Standard control registers	41
4.6.2	Extended control registers	44
4.7	Data formats	47
4.7.1	Integer data format	47
4.7.2	Floating-point data format	47
4.8	Big-endian and little-endian	47
5	Exceptions and Interrupts	50
5.1	Overview	50
5.2	Exceptions	52
5.2.1	Exception handling	52

5.2.2	Return from exceptions	53
5.3	Interrupts	53
5.3.1	Interrupt priorities	54
5.3.2	Interrupt responses	54
5.3.3	Return from interrupts	55
5.3.4	Asynchronous errors	55
6	Memory Model	56
6.1	Overview	56
6.1.1	Memory attributes	56
6.2	SYSMAP configuration reference	57
6.2.1	Memory ordering model	59
6.3	MMU	60
6.3.1	Overview	60
6.3.2	TLB	60
6.3.3	Page Table Format	61
6.3.4	Address translation process	63
6.3.5	System control registers	64
6.3.5.1	Supervisor address translation and protection register (satp)	64
6.3.5.2	smcir register	65
6.3.5.3	smir register	66
6.3.5.4	MMU EntryHi register (smeh)	67
6.3.5.5	MMU EntryLo register(smeh)	68
6.3.5.5	MMU EntryLo register(smeh)	68
6.4	MMU Parity Checking	68
6.5	PMP	69
6.5.1	Overview	69
6.5.2	PMP control registers	69
6.5.2.1	Physical memory protection configuration register (pmpcfg)	69
6.5.2.2	Physical memory protection address register (pmpaddr)	72
6.6	Memory access order	72
7	Memory Subsystem	74
7.1	Memory Subsystem Overview	74
7.2	L1 I-Cache	74
7.2.1	Overview	74
7.2.2	Instruction prefetching	75
7.2.3	Way prediction	75
7.2.4	Loop acceleration buffer	75
7.2.5	Branch history table	76
7.2.6	Branch and jump target predictor	76
7.2.7	Indirect branch predictor	76
7.2.8	Return address predictor	77

7.2.9	Fast jump target predictor	77
7.2.10	Parity Check Function	78
7.3	L1 D-Cache	78
7.3.1	Overview	78
7.3.2	Cache coherence	79
7.3.3	Exclusive access	79
7.4	L2 Cache	80
7.4.1	Overview	80
7.4.2	Cache coherence	80
7.4.3	Structure	81
7.4.4	RAM latency	81
7.5	Accelerated memory access	83
7.5.1	Instruction prefetch of the L1 I-Cache	83
7.5.2	Multi-channel data prefetch of the L1 D-Cache	84
7.5.3	L1 adaptive write allocation mechanism	84
7.5.4	L2 prefetch mechanism	85
7.6	L1/L2 cache operation instructions and registers	85
7.6.1	Extended registers of the L1 cache	85
7.6.2	Extended registers of the L2 cache	85
7.6.3	L1/L2 cache operation instructions	86
8	Vector Computations	88
8.1	Vector programming model	88
8.2	Vector control registers	88
8.3	Vector exceptions	89
9	Security Design	91
9.1	Security Requirements	91
9.2	Processor Security Model	91
9.3	System Security Architecture	93
9.3.1	Secure memory management	93
9.3.2	Secure interrupts	96
9.3.3	Secure Access Control	98
9.3.4	Secure Debugging	100
10	Interrupt Controllers	101
10.1	Core local interrupt (CLINT) controller	101
10.1.1	CLINT register address mapping	101
10.1.2	Software interrupts	103
10.1.3	CLINT timer	104
10.1.4	Timer interrupts	105
10.2	Platform-level interrupt controller (PLIC)	106
10.2.1	Interrupt arbitration	107

10.2.2	Interrupt request and response	107
10.2.3	Interrupt completion	108
10.2.4	PLIC register address mapping	108
10.2.5	PLIC_PRIO register	112
10.2.6	PLIC_IP register	112
10.2.7	PLIC_IE register	113
10.2.8	PLIC_CTRL register	114
10.2.9	PLIC_TH register	114
10.2.10	PLIC_CLAIM register	115
10.3	Multi-core interrupts	115
10.3.1	Multiple cores respond to external interrupts in parallel	115
10.3.2	Send software interrupts across cores	116
11	Bus Interface	117
11.1	AXI master device interface	117
11.1.1	Features of the AXI master device interface	117
11.1.2	Outstanding capability of the AXI master device interface	117
11.1.3	Supported transmission types of Master Device	119
11.1.4	Supported response types of Master Device	119
11.1.5	CPU behavior in different bus responses	120
11.2	Device coherence port	120
11.2.1	Features of DCP	120
11.2.2	Supported transfer types of DCP	121
11.2.3	Supported response types of DCP	121
11.2.4	Responses issued for different behaviors	121
11.3	Low Latency Port	121
11.3.1	Features of LLP	122
11.3.2	Outstanding capabilities of LLP	122
11.3.3	Supported transfer types of LLP	123
11.3.4	Supported response types of LLP	124
12	Debug	125
12.1	Features of the debug unit	125
12.2	Debugging resource configuration	126
13	Power Management	128
13.1	Power domain	128
13.2	Overview of low-power modes	128
13.3	Core WFI process	129
13.4	Individual-core power-off process	129
13.5	Cluster power-off process (hardware clearing of the L2 cache)	130
13.6	Simplified scenario: overall cluster power-off process (hardware clearing of the L2 cache)	131
13.7	Low power consumption related programming models and interface signals	132

13.7.1	Changes in Programming models	133
13.7.2	Interface signals	133
14	Performance Monitoring Unit	134
14.1	PMU overview	134
14.2	PMU programming model	134
14.2.1	PMU functions	134
14.2.2	PMU event overflow interrupt	135
14.3	PMU related control registers	135
14.3.1	M-mode counter access enable register (mcounteren)	135
14.3.2	S-mode counter access enable register (scounteren)	136
14.3.3	M-mode count inhibit register (mcountinhibit)	137
14.3.4	S-mode count inhibit register (scountinhibit)	138
14.3.5	S-mode write enable register (mcounterwen)	139
14.3.6	Performance monitoring event select register (mhpmevent3-31)	139
14.3.7	Event counters	144
15	Program Examples	147
15.1	Optimal performance configuration for CPU	147
15.2	MMU setting examples	148
15.3	PMP setting examples	152
15.4	Cache setting example	153
15.4.1	Example of enabling Cache	153
15.4.2	Example of synchronization between I-Cache and D-Cache	154
15.4.3	Example of synchronization between TLB and D-Cache	155
15.4.4	Setting of L2 cache partition function	155
15.5	Synchronization primitive setting example	155
15.6	PLIC setting example	156
15.7	PMU setting example	157
16	Appendix A Standard Instructions	158
16.1	Appendix A-1 I instructions	158
16.1.1	ADD: a signed add instruction	158
16.1.2	ADDI: a signed add immediate instruction	159
16.1.3	ADDIW: a signed add immediate instruction that operates on the lower 32 bits	159
16.1.4	ADDW: a signed add instruction that operates on the lower 32 bits	160
16.1.5	AND: a bitwise AND instruction	160
16.1.6	ANDI: an immediate bitwise AND instruction	161
16.1.7	AUIPC: an instruction that adds the immediate in the upper bits to the PC	161
16.1.8	BEQ: a branch-if-equal instruction	161
16.1.9	BGE: a signed branch-if-greater-than-or-equal instruction	162
16.1.10	BGEU: an unsigned branch-if-greater-than-or-equal instruction	163
16.1.11	BLT: a signed branch-if-less-than instruction	164

16.1.12	BLTU: an unsigned branch-if-less-than instruction	164
16.1.13	BNE: a branch-if-not-equal instruction	165
16.1.14	CSRRC: a move instruction that clears control registers	166
16.1.15	CSRRCI: a move instruction that clears immediates in control registers	166
16.1.16	CSRRS: a move instruction for setting control registers	167
16.1.17	CSRRSI: a move instruction for setting immediates in control registers	167
16.1.18	CSRRAW: a move instruction that reads/writes control registers	168
16.1.19	CSRRAWI: a move instruction that reads/writes immediates in control registers	169
16.1.20	EBREAK: a breakpoint instruction	169
16.1.21	ECALL: an environment call instruction	170
16.1.22	FENCE: a memory synchronization instruction	170
16.1.23	FENCE.I: an instruction stream synchronization instruction	171
16.1.24	JAL: an instruction for directly jumping to a subroutine	171
16.1.25	JALR: an instruction for jumping to a subroutine by using an address in a register	172
16.1.26	LB: a sign-extended byte load instruction	172
16.1.27	LBU: an unsign-extended byte load instruction	173
16.1.28	LD: a doubleword load instruction	173
16.1.29	LH: a sign-extended halfword load instruction	174
16.1.30	LHU: an unsign-extended halfword load instruction	174
16.1.31	LUI: an instruction for loading the immediate in the upper bits	175
16.1.32	LW: a sign-extended word load instruction	175
16.1.33	LWU: an unsign-extended word load instruction	176
16.1.34	MRET: an instruction for returning from exceptions in M-mode	176
16.1.35	OR: a bitwise OR instruction	177
16.1.36	ORI: an immediate bitwise OR instruction	177
16.1.37	SB: a byte store instruction	177
16.1.38	SD: a doubleword store instruction	178
16.1.39	SFENCE.VMA: a virtual memory synchronization instruction	178
16.1.40	SH: a halfword store instruction	179
16.1.41	SLL: a logical left shift instruction	180
16.1.42	SLLI: an immediate logical left shift instruction	180
16.1.43	SLLIW: an immediate logical left shift instruction that operates on the lower 32 bits	181
16.1.44	SLLW: a logical left shift instruction that operates on the lower 32 bits	181
16.1.45	SLT: a signed set-if-less-than instruction	182
16.1.46	SLTI: a signed set-if-less-than-immediate instruction	182
16.1.47	SLTIU: an unsigned set-if-less-than-immediate instruction	183
16.1.48	SLTU: an unsigned set-if-less-than instruction	183
16.1.49	SRA: an arithmetic right shift instruction	184
16.1.50	SRAI: an immediate arithmetic right shift instruction	184
16.1.51	SLLIW: an immediate arithmetic right shift instruction that operates on the lower 32 bits	185
16.1.52	SRAW: an arithmetic right shift instruction that operates on the lower 32 bits	185

16.1.53	SRET: an instruction for returning from exceptions in S-mode	186
16.1.54	SRL: a logical right shift instruction	186
16.1.55	SRLI: an immediate logical right shift instruction	186
16.1.56	SRLIW: an immediate logical right shift instruction that operates on the lower 32 bits	187
16.1.57	SRLW: a logical right shift instruction that operates on the lower 32 bits	187
16.1.58	SUB: a signed subtract instruction	188
16.1.59	SUBW: a signed subtract instruction that operates on the lower 32 bits	188
16.1.60	SW: a word store instruction	189
16.1.61	WFI: an instruction for entering the low power mode	189
16.1.62	XOR: a bitwise XOR instruction	190
16.1.63	XORI: an immediate bitwise XOR instruction	190
16.2	Appendix A-2 M instructions	190
16.2.1	DIV: a signed divide instruction	191
16.2.2	DIVU: an unsigned divide instruction	191
16.2.3	DIVUW: an unsigned divide instruction that operates on the lower 32 bits	192
16.2.4	DIVW: a signed divide instruction that operates on the lower 32 bits	192
16.2.5	MUL: a signed multiply instruction	193
16.2.6	MULH: a signed multiply instruction that extracts the upper bits	193
16.2.7	MULHSU: a signed-unsigned multiply instruction that extracts the upper bits	193
16.2.8	MULHU: an unsigned multiply instruction that extracts the upper bits	194
16.2.9	MULW: a signed multiply instruction that operates on the lower 32 bits	194
16.2.10	REM: a signed remainder instruction	195
16.2.11	REMU: an unsigned remainder instruction	195
16.2.12	REMUW: an unsigned remainder instruction that operates on the lower 32 bits	196
16.2.13	REMW: a signed remainder instruction that operates on the lower 32 bits	196
16.3	Appendix A-3 A instructions	197
16.3.1	AMOADD.D: an atomic add instruction	197
16.3.2	AMOADD.W: an atomic add instruction that operates on the lower 32 bits	198
16.3.3	AMOAND.D: an atomic bitwise AND instruction	199
16.3.4	AMOAND.W: an atomic bitwise AND instruction that operates on the lower 32 bits	200
16.3.5	AMOMAX.D: an atomic signed MAX instruction	201
16.3.6	AMOMAX.W: an atomic signed MAX instruction that operates on the lower 32 bits	202
16.3.7	MOMAXU.DA: an atomic unsigned MAX instruction	203
16.3.8	AMOMAXU.W: an atomic unsigned MAX instruction that operates on the lower 32 bits.	204
16.3.9	AMOMIN.D: an atomic signed MIN instruction	205
16.3.10	AMOMIN.W: an atomic signed MIN instruction that operates on the lower 32 bits .	206
16.3.11	AMOMINU.D: an atomic unsigned MIN instruction	207
16.3.12	AMOMINU.W: an atomic unsigned MIN instruction that operates on the lower 32 bits	208
16.3.13	AMOOR.D: an atomic bitwise OR instruction.	209
16.3.14	AMOOR.W: an atomic bitwise OR instruction that operates on the lower 32 bits . .	209
16.3.15	AMOSWAP.D: an atomic swap instruction	210

16.3.16	AMOSWAP.W: an atomic swap instruction that operates on the lower 32 bits	211
16.3.17	AMOXOR.D: an atomic bitwise XOR instruction	212
16.3.18	AMOXOR.W: an atomic bitwise XOR instruction that operates on the lower 32 bits	213
16.3.19	LR.D: a doubleword load-reserved instruction	214
16.3.20	LR.W: a word load-reserved instruction	215
16.3.21	SC.D: a doubleword store-conditional instruction	216
16.3.22	SC.W: a word store-conditional instruction	217
16.4	Appendix A-4 F instructions	218
16.4.1	FADD.S: a single-precision floating-point add instruction	218
16.4.2	FCLASS.S: a single-precision floating-point classify instruction	219
16.4.3	FCVT.L.S: an instruction that converts a single-precision floating-point number into a signed long integer	220
16.4.4	FCVT.LU.S: an instruction that converts a single-precision floating-point number into an unsigned long integer	221
16.4.5	FCVT.S.L: an instruction that converts a signed long integer into a single-precision floating-point number	222
16.4.6	FCVT.S.LU: an instruction that converts an unsigned long integer into a single- precision floating-point number	223
16.4.7	FCVT.S.W: an instruction that converts a signed integer into a single-precision floating-point number	224
16.4.8	FCVT.S.WU: an instruction that converts an unsigned integer into a single-precision floating-point number	225
16.4.9	FCVT.W.S: an instruction that converts a single-precision floating-point number into a signed integer	226
16.4.10	FCVT.WU.S: an instruction that converts a single-precision floating-point number into an unsigned integer	227
16.4.11	FDIV.S: a single-precision floating-point divide instruction	228
16.4.12	FEQ.S: a single-precision floating-point compare equal instruction	229
16.4.13	FLE.S: a single-precision floating-point compare less than or equal to instruction . .	229
16.4.14	FLT.S: a single-precision floating-point compare less than instruction	230
16.4.15	FLW: a single-precision floating-point load instruction	231
16.4.16	FMADD.S: a single-precision floating-point multiply-add instruction	231
16.4.17	FMAX.S: a single-precision floating-point MAX instruction	232
16.4.18	FMIN.S: a single-precision floating-point MIN instruction	233
16.4.19	FMSUB.S: a single-precision floating-point multiply-subtract instruction	233
16.4.20	FMUL.S: a single-precision floating-point multiply instruction	234
16.4.21	FMV.W.X: a single-precision floating-point write move instruction	235
16.4.22	FMV.X.H: a single-precision floating-point read move instruction	236
16.4.23	FNMADD.S: a single-precision floating-point negate-(multiply-add) instruction . . .	236
16.4.24	FNMSUB.S: a single-precision floating-point negate-(multiply-subtract) instruction .	237
16.4.25	FSGNJ.S: a single-precision floating-point sign-injection instruction	238
16.4.26	FSGNJN.S: a single-precision floating-point negate sign-injection instruction	239

16.4.27	FSGNJX.S: a single-precision floating-point XOR sign-injection instruction	239
16.4.28	FSQRT.S: a single-precision floating-point square-root instruction	240
16.4.29	FSUB.S: a single-precision floating-point subtract instruction	241
16.4.30	FSW: a single-precision floating-point store instruction	242
16.5	Appendix A-6 C Instructions	242
16.5.1	C.ADD: a signed add instruction	242
16.5.2	C.ADDI: a signed add immediate instruction	243
16.5.3	C.ADDIWI: an add immediate instruction that operates on the lower 32 bits	243
16.5.4	C.ADDI4SPN: an instruction that adds an immediate scaled by 4 to the stack pointer	244
16.5.5	C.ADDI16SP: an instruction that adds an immediate scaled by 16 to the stack pointer	245
16.5.6	C.ADDW: a signed add instruction that operates on the lower 32 bits	245
16.5.7	C.AND: a bitwise AND instruction	246
16.5.8	C.ANDI: an immediate bitwise AND instruction	247
16.5.9	C.BEQZ: a branch-if-equal-to-zero instruction	248
16.5.10	C.BNEZ: a branch-if-not-equal-to-zero instruction	249
16.5.11	C.EBREAK: a break instruction	250
16.5.12	C.FLD: a floating-point load doubleword instruction	250
16.5.13	C.FLDSP: a floating-point doubleword load stack instruction	251
16.5.14	C.FSD: a floating-point store doubleword instruction	252
16.5.15	C.FSDSP: a floating-point store doubleword stack pointer instruction	253
16.5.16	C.J: a unconditional jump instruction	253
16.5.17	C.JALR: a jump and link register instruction	254
16.5.18	C.JR: a jump register instruction	255
16.5.19	C.LD: a load doubleword instruction	255
16.5.20	C.LDSP: a load doubleword instruction	256
16.5.21	C.LI: a load immediate instruction	257
16.5.22	C.LUI: a load upper immediate instruction	257
16.5.23	C.LW: a load word instruction	258
16.5.24	C.LWSP: a load word stack pointer instruction	259
16.5.25	C.MV: an instruction that copies the value in rs to rd	259
16.5.26	C.NOP: a no-operation instruction	260
16.5.27	C.OR: a bitwise OR instruction	260
16.5.28	C.SD: a store doubleword instruction	261
16.5.29	C.SDSP: a store doubleword stack pointer instruction	262
16.5.30	C.SLLI: an immediate logical left shift instruction	262
16.5.31	C.SRAI: a right shift arithmetic immediate instruction	263
16.5.32	C.SRLI: an immediate right shift instruction	264
16.5.33	C.SW: a store word instruction	265
16.5.34	C.SWSP: a store word stack pointer instruction	265
16.5.35	C.SUB: a signed subtract instruction	266
16.5.36	C.SUBW: a signed subtract instruction that operates on the lower 32 bits	267
16.5.37	C.XOR: a bitwise XOR instruction	268

16.6	Appendix A-8 Pseudo instructions	268
17	Appendix B T-Head Extended Instructions	272
17.1	Appendix B-1 Cache instructions	272
17.1.1	DCACHE.CALL: an instruction that clears all dirty page table entries in the D-Cache	272
17.1.2	DCACHE.CIALL: an instruction that clears all dirty page table entries in the D-Cache and invalidates the D-Cache	273
17.1.3	DCACHE.CIPA: clears dirty page table entries that match the specified physical addresses from the D-Cache and invalidates the the D-Cache	274
17.1.4	DCACHE.CISW: an instruction that clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache	274
17.1.5	DCACHE.CIVA: an instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the D-Cache	275
17.1.6	DCACHE.CPA: an instruction that clears dirty page table entries that match the specified physical addresses from the D-Cache	276
17.1.7	DCACHE.CPAL1: an instruction that clears dirty page table entries that match the specified physical addresses from the L1 D-Cache	276
17.1.8	DCACHE.CVA: an instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache	277
17.1.9	DCACHE.CVAL1: an instruction that clears dirty page table entries that match the specified virtual addresses in the L1 D-Cache	278
17.1.10	DCACHE.IPA: an instruction that invalidates page table entries that match the specified physical addresses in the D-Cache	278
17.1.11	DCACHE.ISW: an instruction that invalidates page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache	279
17.1.12	DCACHE.IVA: an instruction that invalidates the D-Cache based on the specified virtual address	280
17.1.13	DCACHE.IALL: an instruction that invalidates all page table entries in the D-Cache.	280
17.1.14	ICACHE.IALL: an instruction that invalidates all page table entries in the I-Cache .	281
17.1.15	ICACHE.IALLS: an instruction that invalidates all page table entries in the I-Cache through broadcasting	281
17.1.16	ICACHE.IPA: an instruction that invalidates page table entries that match the specified physical addresses in the I-Cache	282
17.1.17	ICACHE.IVA: an instruction that invalidates page table entries that match the specified virtual addresses in the I-Cache	283
17.1.18	L2CACHE.CALL: an instruction that clears all dirty page table entries in the L2 Cache	283
17.1.19	L2CACHE.CIALL: an instruction that clears all dirty page table entries in the L2 Cache and invalidates the L2 Cache	284
17.1.20	L2CACHE.IALL: an instruction that invalidates the L2 Cache	284
17.1.21	DCACHE.CSW: an instruction that clears dirty page table entries in the D-Cache based on the specified set and way	285
17.2	Appendix B-2 Multi-core synchronization instructions	286

17.2.1	SFENCE.VMAS: a broadcast instruction that synchronizes the virtual memory address	286
17.2.2	SYNC: an instruction that performs the synchronization operation	287
17.2.3	SYNC.I: an instruction that synchronizes the clearing operation.	287
17.2.4	SYNC.IS: a broadcast instruction that synchronizes the clearing operation	288
17.2.5	SYNC.S: a broadcast instruction that performs a synchronization operation	288
17.3	Appendix B-3 Arithmetic operation instructions	289
17.3.1	ADDSSL: an add register instruction that shifts registers	289
17.3.2	MULA: a multiply-add instruction	289
17.3.3	MULAH: a multiply-add instruction that operates on the lower 16 bits	290
17.3.4	MULAW: a multiply-add instruction that operates on the lower 32 bits	290
17.3.5	MULS: a multiply-subtract instruction	291
17.3.6	MULSH: a multiply-subtract instruction that operates on the lower 16 bits	291
17.3.7	MULSW: a multiply-subtract instruction that operates on the lower 32 bits	292
17.3.8	MVEQZ: an instruction that sends a message when the register is 0	292
17.3.9	MVNEZ: an instruction that sends a message when the register is not 0	293
17.3.10	SRRI: an instruction that implements a cyclic right shift operation on a linked list	293
17.3.11	SRRIW: an instruction that implements a cyclic right shift operation on a linked list of low 32 bits of registers.	294
17.4	Appendix B-4 Bitwise operation instructions	294
17.4.1	EXT: a signed extension instruction that extracts consecutive bits of a register	294
17.4.2	EXTU: a zero extension instruction that extracts consecutive bits of a register	295
17.4.3	FF0: an instruction that finds the first bit with the value of 0 in a register	295
17.4.4	FF1: an instruction that finds the bit with the value of 1	296
17.4.5	REV: an instruction that reverses the byte order in a word stored in the register	296
17.4.6	RE VW: an instruction that reverses the byte order in a low 32-bit word	297
17.4.7	TST: an instruction that tests bits with the value of 0	297
17.4.8	TSTNBZ: an instruction that tests bytes with the value of 0	298
17.5	Appendix B-5 Storage instructions	299
17.5.1	FLRD: a load doubleword instruction that shifts floating-point registers	299
17.5.2	FLRW: a load word instruction that shifts floating-point registers	299
17.5.3	FLURD: a load doubleword instruction that shifts low 32 bits of floating-point registers	300
17.5.4	FLURW: a load word instruction that shifts low 32 bits of floating-point registers	300
17.5.5	FSRD: a store doubleword instruction that shifts floating-point registers	301
17.5.6	FSRW: a store word instruction that shifts floating-point registers.	301
17.5.7	FSURD: a store doubleword instruction that shifts low 32 bits of floating-point registers	302
17.5.8	FSURW: a store word instruction that shifts low 32 bits of floating-point registers	302
17.5.9	LBIA: a base-address auto-increment instruction that extends signed bits and loads bytes	303
17.5.10	LBIB: a load byte instruction that auto-increments the base address and extends signed bits	304
17.5.11	LBUIA: a base-address auto-increment instruction that extends zero bits and loads bytes	304

17.5.12	LBUIB: a load byte instruction that auto-increments the base address and extends zero bits	305
17.5.13	LDD: an instruction that loads double registers	305
17.5.14	LDIA: a base-address auto-increment instruction that loads doublewords and extends signed bits	306
17.5.15	LDIB: a load doubleword instruction that auto-increments the base address and extends the signed bits	306
17.5.16	LHIA: a base-address auto-increment instruction that loads halfwords and extends signed bits	307
17.5.17	LHIB: a load halfword instruction that auto-increments the base address and extends signed bits	308
17.5.18	LHUIA: a base-address auto-increment instruction that extends zero bits and loads halfwords	308
17.5.19	LHUIB: a load halfword instruction that auto-increments the base address and extends zero bits	309
17.5.20	LRB: a load byte instruction that shifts registers and extends signed bits	309
17.5.21	LRBU: a load byte instruction that shifts registers and extends zero bits	310
17.5.22	LRD: a load doubleword instruction that shifts registers	310
17.5.23	LRH: a load halfword instruction that shifts registers and extends signed bits	310
17.5.24	LRHU: a load halfword instruction that shifts registers and extends zero bits	311
17.5.25	LRW: a load word instruction that shifts registers and extends signed bits	311
17.5.26	LRWU: a load word instruction that shifts registers and extends zero bits	312
17.5.27	LURB: a load byte instruction that shifts low 32 bits of registers and extends signed bits	312
17.5.28	LURBU: a load byte instruction that shifts low 32 bits of registers and extends zero bits	313
17.5.29	LURD: a load doubleword instruction that shifts low 32 bits of registers	313
17.5.30	LURH: a load halfword instruction that shifts low 32 bits of registers and extends signed bits	314
17.5.31	LURHU: a load halfword instruction that shifts low 32 bits of registers and extends zero bits	314
17.5.32	LURW: a load word instruction that shifts low 32 bits of registers and extends signed bits	315
17.5.33	LURWU: a load word instruction that shifts 32 bits of registers and extends zero bits	315
17.5.34	LWD: a load word instruction that loads double registers and extends signed bits	316
17.5.35	LWIA: a base-address auto-increment instruction that extends signed bits and loads words	317
17.5.36	LWIB: a load word instruction that auto-increments the base address and extends signed bits	317
17.5.37	LWUD: a load word instruction that loads double registers and extends zero bits	318
17.5.38	LWUIA: a base-address auto-increment instruction that extends zero bits and loads words	318

17.5.39	LWUIB: a load word instruction that auto-increments the base address and extends zero bits	319
17.5.40	SBIA: a base-address auto-increment instruction that stores bytes	319
17.5.41	SBIB: a store byte instruction that auto-increments the base address	320
17.5.42	SDD: an instruction that stores double registers	320
17.5.43	SDIA: a base-address auto-increment instruction that stores doublewords	321
17.5.44	SDIB: a store doubleword instruction that auto-increments the base address	321
17.5.45	SHIA: a base-address auto-increment instruction that stores halfwords	322
17.5.46	SHIB: a store halfword instruction that auto-increments the base address	322
17.5.47	SRB: a store byte instruction that shifts registers	323
17.5.48	SRD: a store doubleword instruction that shifts registers	323
17.5.49	SRH: a store halfword instruction that shifts registers	323
17.5.50	SRW: a store word instruction that shifts registers	324
17.5.51	SURB: a store byte instruction that shifts low 32 bits of registers	324
17.5.52	SURD: a store doubleword instruction that shifts low 32 bits of registers	325
17.5.53	SURH: a store halfword instruction that shifts low 32 bits of registers	325
17.5.54	SURW: a store word instruction that shifts low 32 bits of registers	326
17.5.55	SWIA: a base-address auto-increment instruction that stores words	326
17.5.56	SWIB: a store word instruction that auto-increments the base address	327
17.5.57	SWD: an instruction that stores the low 32 bits of double registers	327
17.6	Appendix B-6 Half-precision floating-point instructions	328
17.6.1	FADD.H: a half-precision floating-point add instruction	328
17.6.2	FCLASS.H: a half-precision floating-point classification instruction	329
17.6.3	FCVT.D.H: an instruction that converts half-precision floating-point data to double-precision floating-point data	330
17.6.4	FCVT.H.D: an instruction that converts double-precision floating-point data to half-precision floating-point data	330
17.6.5	FCVT.H.L: an instruction that converts a signed long integer into a half-precision floating-point number	331
17.6.6	FCVT.H.LU: an instruction that converts an unsigned long integer into a half-precision floating-point number	332
17.6.7	FCVT.H.S: an instruction that converts single precision floating-point data to half-precision floating-point data	333
17.6.8	FCVT.H.W: an instruction that converts a signed integer into a half-precision floating-point number	334
17.6.9	FCVT.H.WU: an instruction that converts an unsigned integer into a half-precision floating-point number	335
17.6.10	FCVT.L.H: an instruction that converts a half-precision floating-point number to a signed long integer	336
17.6.11	FCVT.LU.H: an instruction that converts a half-precision floating-point number to an unsigned long integer	337

17.6.12	FCVT.S.H: an instruction that converts half-precision floating-point data to single precision floating-point data	338
17.6.13	FCVT.W.H: an instruction that converts a half-precision floating-point number to a signed integer	339
17.6.14	FCVT.WU.H: an instruction that converts a half-precision floating-point number to an unsigned integer	340
17.6.15	FDIV.H: a half-precision floating-point division instruction	341
17.6.16	FEQ.H: an equal instruction that compares two half-precision numbers	342
17.6.17	FLE.H: a less than or equal to instruction that compares two half-precision floating-point numbers	342
17.6.18	FLH: an instruction that loads half-precision floating-point data	343
17.6.19	FLT.H: a less than instruction that compares two half-precision floating-point numbers	344
17.6.20	FMADD.H: a half-precision floating-point multiply-add instruction	344
17.6.21	FMAX.H: a half-precision floating-point maximum instruction	345
17.6.22	FMIN.H: a half-precision floating-point minimum instruction	346
17.6.23	FMSUB.H: a half-precision floating-point multiply-subtract instruction	346
17.6.24	FMUL.H: a half-precision floating-point multiply instruction	347
17.6.25	FMV.H.X: a half-precision floating-point write transmit instruction	348
17.6.26	FMV.X.H: a transmission instruction that reads half-precision floating-point registers	349
17.6.27	FNMADD.H: a half-precision floating-point negate-(multiply-add) instruction	349
17.6.28	FNMSUB.H: a half-precision floating-point negate-(multiply-subtract) instruction . .	350
17.6.29	FSGNJ.H: a half-precision floating-point sign-injection instruction	351
17.6.30	FSGNJN.H: a half-precision floating-point sign-injection negate instruction	352
17.6.31	FSGNJX.H: a half-precision floating-point sign-injection XOR instruction	352
17.6.32	FSH: an instruction that stores half-precision floating point numbers	353
17.6.33	FSQRT.H: an instruction that calculates the square root of the half-precision floating-point number	353
17.6.34	FSUB.H: a half-precision floating-point subtract instruction	354

18 Appendix C Control Registers 356

18.1	Appendix C-1 M-mode control registers	356
18.1.1	M-mode information register group	356
18.1.1.1	Machine vendor ID register (mvendorid)	356
18.1.1.2	Machine architecture ID register (marchid)	356
18.1.1.3	Machine implementation ID register (mimpid)	357
18.1.1.4	Machine hart ID register (mhartid)	357
18.1.2	M-mode exception configuration register group	357
18.1.2.1	Machine status register (mstatus)	357
18.1.2.2	M-mode instruction set architecture register (misa)	360
18.1.2.3	M-mode exception downgrade control register (medeleg)	360
18.1.2.4	M-mode interrupt downgrade control register (mideleg)	361
18.1.2.5	M-mode interrupt-enable register (mie)	361

18.1.2.6	M-mode trap vector base address register (mtvec)	362
18.1.2.7	M-mode counter access enable register (mcounteren)	363
18.1.3	M-mode exception handling register group	363
18.1.3.1	M-mode scratch register (mscratch)	363
18.1.3.2	M-mode exception program counter register (mepc)	363
18.1.3.3	M-mode cause register (mcause)	363
18.1.3.4	M-mode interrupt-pending register (mip)	364
18.1.4	M-mode memory protection registers	365
18.1.4.1	Physical memory protection configuration register (pmpcfg)	365
18.1.4.2	Physical memory address register (pmpaddr)	366
18.1.5	M-mode counter registers	366
18.1.5.1	M-mode cycle counter (mcycle)	366
18.1.5.2	M-mode instructions-retired counter (minstret)	366
18.1.5.3	M-mode event counter (mhpmcounter _n)	366
18.1.6	M-mode counter configuration registers	366
18.1.6.1	M-mode event selector (mhpmevent _n)	367
18.1.7	M-mode CPU control and status extension registers	367
18.1.7.1	M-mode extension status register (mxstatus)	367
18.1.7.2	M-mode hardware configuration register (mhcr)	369
18.1.7.3	M-mode hardware operation register (mcor)	370
18.1.7.4	M-mode L2 Cache control register (mccr ₂)	372
18.1.7.5	M-mode implicit operation register (mhint)	373
18.1.7.6	M-mode reset vector base address register (mrvbr)	375
18.1.7.7	S-mode counter write enable register (mcounterwen)	376
18.1.7.8	M-mode event interrupt enable register (mcounterinten)	376
18.1.7.9	M-mode event overflow mark register (mcounteren)	377
18.1.8	M-mode cache access extension registers	377
18.1.8.1	M-mode cache instruction register (mcins)	378
18.1.8.2	M-mode cache access index register (mcindex)	378
18.1.8.3	M-mode cache data register (mcdata _{0/1})	379
18.1.9	M-mode CPU model registers	379
18.1.9.1	M-mode CPU model register (mcpuid)	380
18.1.9.2	On-chip bus base address register (mapbaddr)	380
18.1.10	Multi-core extension registers	380
18.1.10.1	Snoop listening enable register (msmpr)	380
18.2	Appendix C-2 S-mode control registers	380
18.2.1	S-mode exception configuration registers	380
18.2.1.1	S-mode status register (sstatus)	381
18.2.1.2	S-mode interrupt-enable register (sie)	381
18.2.1.3	S-mode trap vector base address register (stvec)	382
18.2.1.4	S-mode counter access enable register (scounteren)	382
18.2.2	S-mode exception handling registers	382

18.2.2.1	S-mode scratch register (sscratch)	382
18.2.2.2	S-mode exception program counter register (sepc)	382
18.2.2.3	S-mode cause register (scause)	383
18.2.2.4	S-mode interrupt-pending register (sip)	383
18.2.3	S-mode address translation registers	383
18.2.3.1	S-mode address translation register (satp)	383
18.2.4	S-mode CPU control and status extension registers	384
18.2.4.1	S-mode extension status register (sxstatus)	384
18.2.4.2	S-mode hardware control register (shcr)	384
18.2.4.3	S-mode event overflow interrupt enable register (scounterinten)	384
18.2.4.4	S-mode event overflow mark register (scounterof)	384
18.2.4.5	S-mode cycle counter (scycle)	384
18.2.4.6	S-mode instructions-retired counter (sinstret)	385
18.2.4.7	S-mode event counter (shpmcountern)	385
18.2.5	S-mode MMU extension register	385
18.2.5.1	S-mode MMU control register (smcir)	385
18.2.5.2	S-mode MMU control register (smir)	385
18.2.5.3	S-mode MMU control register (smeh)	385
18.2.5.4	S-mode MMU control register (smel)	386
18.3	Appendix C-3 U-mode control registers	386
18.3.1	U-mode floating-point control registers	386
18.3.1.1	Floating-point accrued exceptions register (fflags)	386
18.3.1.2	Floating-point dynamic rounding mode register (frm)	386
18.3.1.3	Floating-point control and status register (fcsr)	386
18.3.2	U-mode counter/timer registers	387
18.3.2.1	User cycle register (cycle)	387
18.3.2.2	U-mode timer register (time)	388
18.3.2.3	User instructions-retired counter (instret)	388
18.3.2.4	User event counter (hpmcountern)	388
18.3.3	U-mode floating-point extension control registers	388
18.3.3.1	U-mode floating-point extension control register (fxcr)	388
18.3.4	Vector extension registers	389
18.3.4.1	Vector start position register (vstart)	389
18.3.4.2	Fixed-point overflow flag bit register (vxsat)	389
18.3.4.3	Fixed-point rounding mode register (vxrm)	389
18.3.4.4	Vector length register (vl)	390
18.3.4.5	Vector data type register (vtype)	390
18.3.4.6	Vector width (unit: byte) register (vlenb)	391
18.4	Appendix C-4 Additional Register description	391

19 Program Examples	395
19.1 Optimal CPU performance configuration	395

19.2	MMU setting example	396
19.3	PMP setting example	400
19.4	Cache examples	401
19.4.1	Cache enabling example	401
19.4.2	Example of synchronization between the instruction and data caches	402
19.4.3	Example of synchronization between the TLB and the data cache	402
19.5	Multi-core startup example	403
19.6	Synchronization primitive examples	403
19.7	PLIC setting example	404
19.8	PMU setting example	405

20 Appendix D XuanTie C900 Multi-core Synchronization Related Instructions and Program Implementations 406

20.1	Overview	406
20.2	RISC-V instructions	406
20.2.1	fence instruction	406
20.2.2	fence.i instruction	407
20.2.3	sfence.vma instruction	407
20.2.4	AMO instruction	407
20.2.5	Load-Reserved/Store-Conditional instruction	408
20.3	T-Head enhancement instruction	409
20.3.1	sync.is	409
20.3.2	dcache.cipa rs1	410
20.3.3	icache.iva rs1	410
20.4	Software example	410
20.4.1	TLB maintenance	410
20.4.1.1	TLB flush	410
20.4.1.2	Flush process related TLB entries based on ASID	410
20.4.1.3	Flush TLB entries based on VA	410
20.4.1.4	Flush TLB entries based on VA and ASID	411
20.4.2	Instruction space synchronization	411
20.4.2.1	Intra-core global instruction space synchronization	411
20.4.2.2	Inter-core global instruction space synchronization	412
20.4.2.3	T-Head inter-core precise instruction space synchronization	412
20.4.3	DMA synchronization	412
20.4.3.1	T-Head inter-core precise DMA synchronization, including three directions	412
20.4.4	AMO implementations for reference	413

C908MP, built on the RISC-V instruction set architecture, is a high-performance multi-core 64-bit processor. It is mainly designed to meet the increasing requirements for image and vision processing in IoT scenarios, such as smart vision, in-vehicle vision, dash cams, and smart interaction. It can also be used in other scenarios, including robot vacuums, unmanned aerial vehicles (UAVs), automatic driving, augmented reality (AR), medical image processing, industrial robot vision, and mobile Internet.

C908MP adopts a homogeneous multi-core architecture and supports multiple clusters. Each cluster can support 1 to 4 cores. Each C908 core adopts the self-developed microsystem architecture and high-performance computing technologies, such as dual-issue in-order processing, multi-mode branch prediction, and multi-channel data prefetching, to implement performance optimization. In addition, C908 core supports real-time detection and shutdown of idle functional modules to reduce the dynamic power consumption of the processor.

1.1 Features

1.1.1 Features of C908MP Architectural

- Adopts a homogeneous multi-core architecture and supports multiple clusters. Each cluster can support 1 to 4 cores.
- Supports one AXI4 or ACE master interface and a 128-bit bus.
- Supports one configurable low latency port (LLP) for the AXI4.0 master interface and a 128-bit bus.

- Supports one configurable device coherence port(DCP) for the AXI4.0 master interface and a 128-bit bus.
- Supports an L1 instruction or data cache size of 16 KB, 32 KB, or 64 KB and a cache line size of 64 bytes. Allows you to configure an error correction code (ECC) or odd-even parity check mechanism.
- Supports an L2 cache size of 128 KB, 256 KB, 512 KB, 1 MB, 1.5 MB, 2 MB, 3 MB, or 4 MB, and a cache line size of 64 bytes. Allows you to configure an ECC check mechanism.
- For L1 cache, supports the Modified, Exclusive, Shared, Invalid (MESI) cache coherence protocol; for L2 cache, supports the Modified, Owned, Exclusive, Shared, Invalid (MOESI) cache coherence protocol.
- Supports the core local interrupt controller (CLINT) and platform-level interrupt controller (PLIC), and supports multi-cluster interrupt distribution.
- Supports RISC-V performance counters and timers.
- Supports Sv39 and Sv48 memory management, and supports the SVNAPOT standard extension and SVPBMT standard extension.
- Supports 8, 16, 32, and 64 PMP entries, and supports ePMP.
- Supports the T-Head TEE extension.
- Allows power-off for each core or the entire cluster.
- Supports the RISC-V debug framework, and supports multi-core multi-cluster debugging.

1.1.2 Features of C908 Core

- RISC-V 64GCB[V] instruction set architecture;
- Both RV64 and RV32 instruction sets are supported in C908 User Mode;
- Little-endian mode supported;
- 9-stage pipelining architecture;
- In-order dual-issue, In-order fetch, dispatch, execute, and retire;
- Two-level translation lookaside buffer (TLB) memory management units for virtual/physical address translation and memory management;
- I-Cache/D-Cache size: 16KB/32KB/64KB, with a cache line size of 64 bytes.
- Parity check can be configured for instruction cache. ECC or parity check can be configured for data cache.
- Instruction prefetch, and automatic detection and dynamic startup of hardware;
- Low-power access technology with I-Cache way prediction;
- Multi-algorithm branch predictor with 2KB/4KB/8KB options;

- Branch target buffer (BTB) with 256 entries;
- 8-layer hardware return address stack supported;
- Indirect branch predictor with 256 entries;
- Loop termination prediction supported;
- Instruction fusion technology supported;
- Dual issue and full in-order execution for load/store instructions;
- Concurrent bus access for memory read/write operations up to 8-way/12-way ;
- Write combining supported;
- 8-channel data prefetch, fixed stride and irregular stride data prefetch supported;

1.1.3 Features of C908 vector computing units

- Support RISC-V V instruction extension
- Computing capability up to 512G Flops (@INT8)/ 256G Flops(@FP16) at maximum configuration of 4 cores and 2GHz
- Vector execution units support FP16/BFP16/FP32 floating points and 8-bit, 16-bit, 32-bit, and 64-bit integral vector computations.
- Support 128-bit/256-bit vector register width (VLEN)
- The number of pipeline stages for vector execution unit operations and data storage be configured as 1 or 2
- Supports 128-bit and 256-bit vector data width for access.
- Support vector load/store segment instructions
- Support performance-optimized unaligned memory access

1.2 Configuration options

Table 1.1 describes configuration options of C908.

Table 1.1: Configurable items of C908

Feature	Options	Description
C908 Cluster		
Number of cores	1, 2, 3, 4	C908MP supports 1-4 C908 cores.
VECTOR_SIMD	Yes/No	You can configure a vector execution unit.

Continued on next page

Table 1.1 – continued from previous page

Feature	Options	Description
Vector register width	128 or 256	You can set VLEN=128 or VLEN=256. Note that if VLEN=128, DP=64; if VLEN=256, DP=128. DP indicates the datapath width.
Double vector execution unit	Yes/No	Indicates whether to configure two sets of vector calculation pipelines and storage access pipelines.
BHT	Pro or Lite	You can set the BHT implementation method to Pro (tage) or Lite (gshare).
MMU	SV39 or SV48	You can set the value to SV39 or SV48. If you set it to SV48, SV39 is also supported.
jTLB entry	512 or 1024	The number of jTLB entries.
Master interface protocol	AXI or ACE	You can set the master interface to support the AXI or ACE protocol.
LLP	Yes/No	You can configure a low latency port (LLP).
DCP	Yes/No	DCP is used for peripheral devices accessing on-chip high-speed data cache to implement data consistency. It can also be used for DMA.
L1 I-Cache	16 KB, 32 KB, or 64 KB	You set the cache size to 16 KB, 32 KB, or 64 KB.
L1 D-Cache	16 KB, 32 KB, or 64 KB	You set the cache size to 16 KB, 32 KB, or 64 KB.
L1 ECC/Parity	Yes/No	The parity check of L1 I-Cache or ECC check of L1 D-Cache
L2 Cache	128KB,256KB, 512KB,1MB, 1.5MB, 2MB, 3 MB or 4 MB	You can set a size of 128 KB, 256 KB, 512 KB, 1 MB, 1.5 MB, 2MB, 3 MB, or 4 MB.
L2 ECC	Yes/No	The ECC check of L2 Tag/Data RAM
Number of PMP regions	8, 16, 32, or 64	You can specify the number of PMP regions.
EPMP	Yes/No	You can configure enhanced PMP features.
TEE	Yes/No You can select TEE extensions.	

Continued on next page

Table 1.1 – continued from previous page

Feature	Options	Description
L2 RAM latency		Tag RAM Setup: No flop/Flop Tag RAM Access: 1~5 cycles Data RAM Setup: No flop/Flop Data RAM Access: 1~8 cycles
Debug resource level	Minimum, typical, or maximum.	The typical configuration is recommended. For more information, see the following table.
pic_top (external interrupt controller)		
Number of interrupts	32-1024, step 32	Number of interrupts
Number of clusters	1-16	The number of clusters in the pic_top.
Number of harts	1-256	The number of harts in the pic_top. The value must be the same as the number of cores. You do not need to create association relationships between harts and clusters.
TEE	Yes/No	You can select TEE extensions.
tdt_dmi_top (debug bridge, JTAG to APB)		
Number of APB interfaces	1-32	You can use the same tdt_dmi_top to debug multiple clusters, and an APB interface is associated with a cluster.
Sys APB Access	Yes/No	Sys APB Access allows CPU to access debug registers through the master port and system bus.

Note:

- FPU is fixed and not configurable.
- The TEE setting of the cluster must be consistent with the TEE setting of pic_top.
- Debugging resource configuration:

Minimum configuration: one trigger. The trigger can be configured to the instruction address type. Congruent matching and low-bit mask matching are supported.

Typical configuration: three triggers. Each trigger can be configured to the instruction address type or memory access address type. Congruent matching and low-bit mask matching are supported.

Maximum configuration: eight triggers. Each trigger can be configured to the instruction address type, instruction data type, memory access address type, or access data type. Congruent matching, low-bit mask matching, Greater Than or Equal To operator, Less Than operator, mask matching with the lower half, and mask matching with the upper half are supported. In this configuration, you can also set two triggers to form a group. If a trigger group is set up, events are triggered only when the conditions of the two triggers are met. There are two interrupt and exception triggers. Each trigger

can be configured to an interrupt trigger or exception trigger.

For more information, see *Debug* .

1.3 XuanTie extended architecture

C908 is compatible with XuanTie C-series extended architecture 1.0, which provides extensions in the following aspects:

- Operation instructions: C908 improves operation capabilities with integer, floating-point, and load/store instructions, well supplementing the RISC-V base instruction sets.
- Cache operations: C908 enables you to easily maintain caches to improve cache efficiency.
- Memory model: C908 manages address attributes efficiently to improve memory access efficiency.
- Control registers: C908 extends the features of control registers based on the standard RISC-V architecture.
- Multi-core synchronization instructions: C908 adopts multi-core synchronization instructions to keep multi-core consistency.

1.4 Version compatibility

C908 is compatible with the following RISC-V standard versions:

- *The RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA, Version 2.2.*
- *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version riscv-privileged-20210915-public-review.*
- *RISC-V “V” Vector Extension, Version 1.0-rc1-20210608.*
- *RISC-V Bit-Manipulation ISA-extensions, Version 1.0.0, 2021-06-12: public review.*
- *RISC-V External Debug Support, Version 0.13.2.*
- *PMP Enhancements for memory access and execution prevention on Machine mode (Smepmp), Version 0.9.3,07/2021.*
- *RISC-V Base Cache Management Operation ISA Extensions Version 1.0-rc2-2c97b28, 2021-11-05: Frozen.*

1.5 Terms

- **Logic 1:** The level value corresponding to the Boolean logic value TRUE.

- **Logic 0:** The level value corresponding to the Boolean logic value FALSE.
- **Set:** The action of setting one or more bits to the level value corresponding to logic 1.
- **Clear:** The action of setting one or more bits to the level value corresponding to logic 0.
- **Reserved bit:** A bit reserved for feature extension. The value of a reserved bit is 0 unless otherwise specified.
- **Signal:** An electrical value used to transfer information based on its state or state transition.
- **Pin:** An external electrical and physical connection. Multiple signals can connect to one pin.
- **Enable:** The action of switching a discrete signal to a valid state:
 - Switch a valid low-level signal from a high level to a low level.
 - Switch a valid high-level signal from a low level to a high level.
- **Disable:** The action of switching the state of an enabled signal:
 - Switch a valid low-level signal from a low level to a high level.
 - Switch a valid high-level signal from a high level to a low level.
- **LSB:** The least significant bit. **MSB:** The most significant bit.
- **Signal, bit field, and control bit:** Expressed based on a general rule.
- **Identifier followed by a value range:** Indicates a group of signals from the most significant bit to the least significant bit.

For example, `addr[4:0]` indicates a group of address buses, where `addr[4]` indicates the most significant bit, and `addr[0]` indicates the least significant bit.

- **Single identifier:** Indicates a single signal.

For example, `pad_cpu_rst_b` indicates a single signal.

In some cases, an identifier followed by a number is used to express a specific meaning. For example, `addr15` indicates the 16th bit of a group of buses.

2.1 Structure

The structure of C908MP is shown in Fig. 2.1 .

2.2 In-core subsystems

In-core subsystems of C908 include the instruction fetch unit (IFU), instruction execution unit (IEU), vector floating-point unit (VFPU), load/store unit (LSU), memory management unit (MMU), and physical memory protection unit (PMP).

2.2.1 IFU

The IFU can fetch and parallel process up to four instructions at a time. It improves access efficiency with a variety of technologies, for example, I-Cache way prediction, instruction registers, and direct/indirect branch prediction. The IFU features low power consumption, high branch prediction accuracy, and high prefetch efficiency.

2.2.2 IEU

An IEU can decode and issue two instructions at the same time. IEUs include the arithmetic logic unit (ALU), multiplication (MULT) unit, division (DIV) unit, and branch/jump unit (BJU). The ALU is used

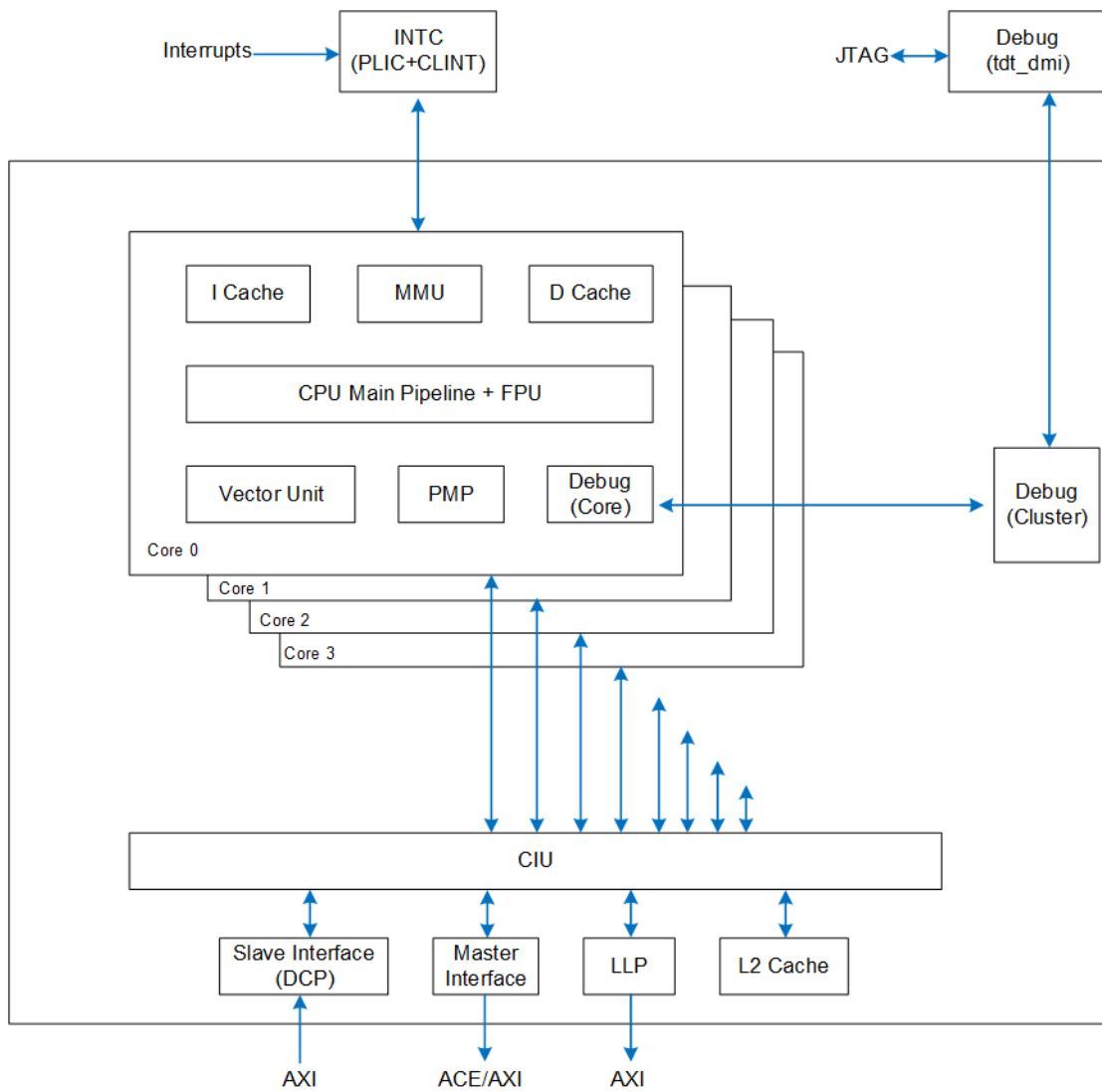


Fig. 2.1: C908MP structure

for 32-bit and 64-bit integers and bit-extension operations. The MULT unit supports 16×16 , 32×32 , and 64×64 integer multiplication. The DIV unit is designed based on the radix-4 and radix-16 SRT algorithms. Its execution cycle varies with operands. The BJU can process branch prediction errors within each cycle. The IEU can retire two instructions at the same time.

2.2.3 VFPU

FPU include the floating-point arithmetic logic unit (FALU), floating-point fused multiply-add unit (FMAU), and floating-point divide and square root unit (FDSU). They support half-precision, single-precision, and double-precision operations. The FALU performs operations such as addition, subtraction, comparison, conversion, register data transmission, sign injection, and classification.

The FMAU performs operations such as common multiplication and fused multiply-add operations. The FDSU performs operations such as floating-point division and square root operations. The vector execution unit is developed by extending the floating-point unit. On the basis of the original scalar floating-point computation, floating-point units can be extended to vector floating-point units. Vector floating-point units include the vector floating-point arithmetic logic unit (VFALU), vector floating-point fused multiply-add unit (VFMAU), and vector floating-point divide and square root unit (VFDSU).

Vector floating-point units support vector floating-point computation of different bits. In addition, vector integer units are added. Vector integer units include the vector arithmetic logic unit (VALU), vector shift unit (VSHIFT), vector multiplication unit (VMUL), vector division unit (VDIVU), vector permutation unit (VPERM), vector reduction unit (VREDU), and vector logical operation unit (VMISC).

2.2.4 LSU

The LSU supports dual issue for scalar store/load instructions, single issue for vector store/load instructions, and full out-of-order execution for all store/load instructions. The LSU supports non-blocking access to caches. It supports byte, halfword, word, doubleword, and quadword store/load instructions, and supports sign/zero extension for byte and halfword load instructions. Store/load instructions can be executed in a pipeline so that only one data entry is accessed per cycle. The LSU supports 8-channel hardware prefetch. It can transfer data to the L1 D-Cache in advance. If the D-Cache is absent, the LSU supports parallel bus access.

2.2.5 MMU

The MMU translates 39-bit and 48-bit virtual addresses to 40-bit physical addresses in compliance with the RISC-V SV39 standard and RISC-V SV48 standard. C908MMU extends software writeback methods and address attributes based on the hardware writeback standards defined in SV39 and SV48.

For more information, see *Memory Model*.

2.2.6 PMP

The PMP unit of C908 complies with the RISC-V standard and supports 8, 16, 32, and 64 entries, but does not support the NA4 mode. The minimum granularity supported by the PMP unit is 4 KB. In addition to the standard PMP features, you can also configure the enhanced PMP (ePMP) features.

For more information, see *Memory Model*.

2.3 Multi-core subsystems

Multi-core subsystems of C908 include the data consistency interface unit (CIU), L2 cache, master device interface unit, device coherence port (DCP) supporting AXI4.0, and low latency port (LLP).

2.3.1 CIU

The CIU ensures data coherence between L1 D-Caches based on the MESI protocol. Two listening buffers are configured to parallel handle multiple listening requests, to fully utilize the listening bandwidth. The CIU adopts an efficient data bypassing mechanism. When a listening request hits an L1 D-Cache under listening, data is directly bypassed to the request initiation core. In addition, the CIU supports broadcasting of invalid TLB/I-Cache requests. This reduces the software costs of maintaining data coherence between TLB/I-Cache and D-Cache.

2.3.2 L2 cache

The L2 cache is tightly coupled to the CIU for synchronous access with L1 D-Caches. The L2 cache adopts a block-based pipelining architecture and can parallel handle two access requests within one cycle. It supports a maximum access bandwidth of 1024 bps. The operating frequency of the L2 cache is the same as that of C908. The tag and data RAM access latency can be configured by using software. L2 cache supports quick-return mechanisms. When L2 cache is hit, data is directly bypassed to the request initiation core.

2.3.3 Master device interface unit

The master device interface unit supports the AXI4 protocol and address access by keyword priority, and can work under different system clock to CPU clock ratios, for example, 1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7, and 1:8.

2.3.4 DCP

The DCP supports the AXI4.0 protocol and can be used for peripheral devices accessing on-chip high-speed data cache.

For more information, see *Interrupt Controllers* .

2.3.5 LLP

The LLP supports the AXI4.0 protocol and can be used as a dedicated port for accessing peripheral devices. The LLP has a separate data channel and is not affected by the bandwidth of the main port.

For more information, see *Interrupt Controllers* .

2.4 Multi-cluster subsystems

Multi-cluster subsystems of C908 and C908 include the programmable interrupt controllers (PICs), timer, and custom multi-cluster multi-core single-port debug framework.

2.4.1 PIC

PICs include the platform-level interrupt controller (PLIC) and core local interrupt (CLINT) controller. The PLIC controls sampling and distribution of up to 1023 external interrupt sources. It supports electrical level interrupts and pulse interrupts. You can set up to 32 interrupt priorities. The CLINT controller is used to process software interrupts and timer interrupts. The designs of PICs of C908 adopt the external type. Therefore, the PICs can process external interrupts and local interrupts occurred on different clusters.

For more information, see *Interrupt Controllers* .

2.4.2 Timer

A multi-cluster multi-core system uses a 64-bit system timer, and each core has a private timer compare value register. Values of the system timer are collected and compared with those in the private timer compare value register of the software to generate timer signals.

For more information, see *Interrupt Controllers* .

2.4.3 Debug system

C908 adopts a multi-cluster multi-core single-port debug framework. It uses a shared JTAG interface to access the hardware assisted debug unit of each cluster, and control the cores to enter or exit the debug mode and the processor resources accessed. The JTAG interface and debug unit (DM) in C908 are in compliance with the RISC-V debug V0.13.2 standard protocol.

For more information, see *Debug* .

2.5 Interface overview

C908 provides the following interfaces by feature: clock reset signal, bus system, interrupt system, debug system, low power system, DFT system, and CPU running monitoring signal.

For more information, see Fig. 2.2.

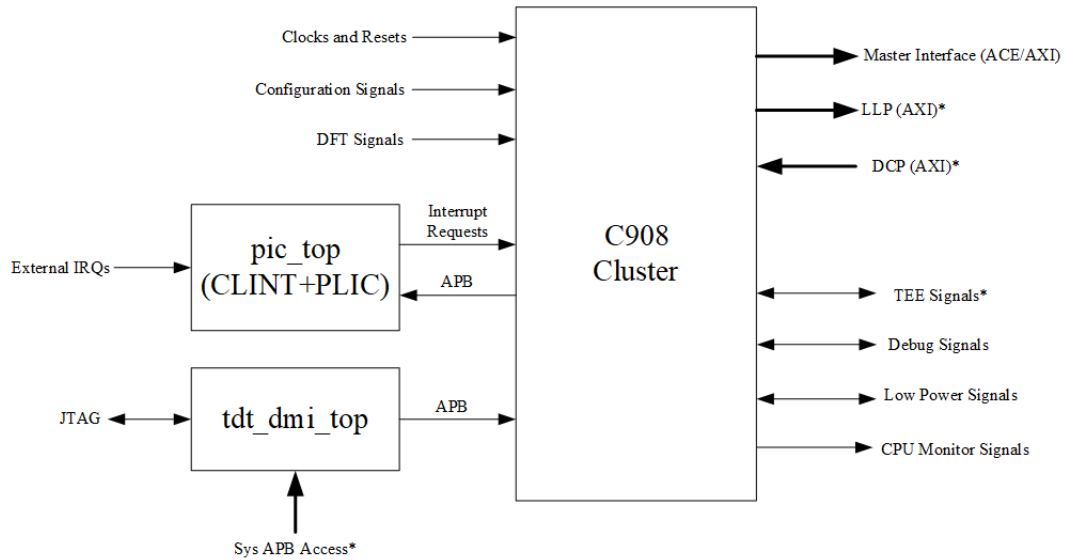


Fig. 2.2: C908MP interfaces

This section describes the instruction sets implemented in C908: RV base instruction sets and XuanTie extended instruction sets.

3.1 RV base instruction sets

3.1.1 Integer instruction set (RV64I)

The integer instruction set includes instructions of the following types by feature:

- Add/Subtract instructions
- Logical operation instructions
- Shift instructions
- Compare instructions
- Data transmission instructions
- Branch and jump instructions
- Memory access instructions
- Control register operation instructions
- Low power instructions
- Exception-return instructions

- Special functional instructions

Table 3.1: RV64I instructions

Instruction	Description	Execution latency
Add/Subtract instructions		
ADD	A signed add instruction	1
ADDW	A signed add instruction that operates on the lower 32 bits	1
ADDI	A signed add immediate instruction	1
ADDIW	A signed add immediate instruction that operates on the lower 32 bits	1
SUB	A signed subtract instruction	1
SUBW	A signed subtract instruction that operates on the lower 32 bits	1
Logic operation instructions		
AND	A bitwise AND instruction.	1
ANDI	An immediate bitwise AND instruction	1
OR	A bitwise OR instruction	1
ORI	An immediate bitwise OR instruction	1
XOR	A bitwise XOR instruction.	1
XORI	An immediate bitwise XOR instruction	1
Shift instructions		
SLL	A logical left shift instruction	1
SLLW	A word logical left shift instruction that operates on the lower 32 bits	1
SLLI	An immediate logical left shift instruction	1
SLLIW	An immediate logical left shift instruction that operates on the lower 32 bits	1
SRL	A logical right shift instruction	1
SRLW	A logical right shift instruction that operates on the lower 32 bits	1
SRLI	An immediate logical right shift instruction	1
SRLIW	An immediate logical right shift instruction that operates on the lower 32 bits	1
SRA	An arithmetic right shift instruction	1
SRAW	An arithmetic right shift instruction that operates on the lower 32 bits	1
SRAI	An immediate arithmetic right shift instruction	1

Continued on next page

Table 3.1 – continued from previous page

Instruction	Description	Execution latency
SRAIW	An immediate arithmetic right shift instruction that operates on the lower 32 bits	1
Compare instructions		
SLT	A signed set-if-less-than instruction	1
SLTU	An unsigned set-if-less-than instruction	1
SLTI	A signed set-if-less-than-immediate instruction	1
SLTIU	An unsigned set-if-less-than-immediate instruction	1
Data transmission instructions		
LUI	A load upper immediate instruction	1
AUIPC	An add upper immediate to PC instruction	1
Branch and jump instructions		
BEQ	A branch-if-equal instruction	1
BNE	A branch-if-not-equal instruction	1
BLT	A signed branch-if-less-than instruction	1
BGE	A signed branch-if-greater-than-or-equal instruction	1
BLTU	An unsigned branch-if-less-than instruction	1
BGEU	An unsigned branch-if-greater-than-or-equal instruction	1
JAL	An instruction for directly jumping to a subroutine	1
JALR	An instruction for jumping to a subroutine by using an address in a register	1
Memory access instructions		
LB	A sign-extended byte load instruction	WEAK ORDER LOAD: >=3 STORE: 1 STRONG ORDER Aperiodic
LBU	An unsign-extended byte load instruction	Same as above
LH	A sign-extended halfword load instruction	Same as above
LHU	An unsign-extended halfword load instruction	Same as above
LW	A sign-extended word load instruction	Same as above
LWU	An unsign-extended word load instruction	Same as above
LD	A doubleword load instruction	Same as above
SB	A byte store instruction	Same as above
SH	A halfword store instruction	Same as above
SW	A word store instruction	Same as above
SD	A doubleword store instruction	Same as above

Continued on next page

Table 3.1 – continued from previous page

Instruction	Description	Execution latency
Control register operation instructions		
CSR _{RR}	A move instruction that reads/writes control registers	Blocked Aperiodic
CSR _{RS}	A move instruction for setting control registers	Same as above
CSR _{RC}	A move instruction that clears control register	Same as above
CSR _{RWI}	A move instruction that reads/writes immediates in control registers	Same as above
CSR _{RSI}	A move instruction for setting immediates in control registers	Same as above
CSR _{RCI}	A move instruction that clears immediates in control registers	Same as above
Low power instructions		
WFI	An instruction for entering the low-power standby mode	Aperiodic
Exception-return instructions		
MRET	An instruction for returning from exceptions in machine mode (M-mode)	Block
SRET	An instruction for returning from exceptions in supervisor mode (S-mode)	Same as above
Special functional instructions		
FENCE	A memory synchronization instruction	Aperiodic
FENCE.I	An instruction stream synchronization instruction	Blocked
SFENCE.VMA	A virtual memory synchronization instruction	Same as above
EBREAK	A breakpoint instruction	1
ECALL	An environment call instruction	1

For more information, see *Appendix A-1 I instructions*.

3.1.2 Multiply/Divide instruction set (RV64M)

Table 3.2: RV64M instructions

Instruction	Description	Execution latency
MUL	A signed multiply instruction	4
MULW	A signed multiply instruction that operates on the lower 32 bits	4
MULH	A signed multiply instruction that extracts upper bits	4
MULHS	A signed-unsigned multiply instruction that extracts upper bits	4
MULHU	An unsigned multiply instruction that extracts upper bits	4

Continued on next page

Table 3.2 – continued from previous page

Instruction	Description	Execution latency
DIV	A signed divide instruction.	3-20
DIVW	A signed divide instruction that operates on the lower 32 bits	3-12
DIVU	An unsigned divide instruction.	3-20
DIVUW	An unsigned divide instruction that operates on the lower 32 bits	3-12
REM	A signed remainder instruction	3-20
REMW	A signed remainder instruction that operates on the lower 32 bits	3-12
REMU	An unsigned remainder instruction.	3-20
REMUW	An unsigned remainder instruction that operates on the lower 32 bits	3-12

For more information, see *Appendix A-2 M instructions*.

3.1.3 Atomic instruction set (RV64A)

Table 3.3: RV64A instructions

Instruction	Description	Execution latency
LR.W	A word load-reserved instruction.	This instruction is split into multiple atomic instructions for execution, but latency is not allowed.
LR.D	A doubleword load-reserved instruction.	
SC.W	A word store-conditional instruction.	
SC.D	A doubleword store-conditional instruction.	
AMOSWAP.W	An atomic swap instruction that operates on the lower 32 bits.	
AMOSWAP.D	An atomic swap instruction.	
AMOADD.W	An atomic add instruction that operates on the lower 32 bits.	
AMOADD.D	An atomic add instruction.	
AMOXOR.W	An atomic bitwise XOR instruction that operates on the lower 32 bits.	
AMOXOR.D	An atomic bitwise XOR instruction.	
AMOAND.W	An atomic bitwise AND instruction that operates on the lower 32 bits.	
AMOAND.D	An atomic bitwise AND instruction.	
AMOOR.W	An atomic bitwise OR instruction that operates on the lower 32 bits.	
AMOOR.D	An atomic bitwise OR instruction	
AMOMIN.W	An atomic signed MIN instruction that operates on the lower 32 bits.	

Continued on next page

Table 3.3 – continued from previous page

Instruction	Description	Execution latency
AMOMIN.D	An atomic signed MIN instruction	
AMOMAX.W	An atomic signed MAX instruction that operates on the lower 32 bits.	
AMOMAX.D	An atomic signed MAX instruction.	
AMOMINU.W	An atomic unsigned MIN instruction that operates on the lower 32 bits.	
AMOMINU.D	An atomic unsigned MIN instruction.	
AMOMAXU.W	An atomic unsigned MAX instruction that operates on the lower 32 bits.	
AMOMAXU.D	An atomic unsigned MAX instruction.	

For more information, see *Appendix A-3 A instructions* .

3.1.4 Single-precision floating-point instruction set

A single-precision floating-point instruction set includes instructions of the following types by feature:

- Operation instructions
- Sign injection instructions
- Data transmission instructions
- Compare instructions
- Data type conversion instructions
- Memory store instructions
- Floating-point classify instructions

Table 3.4: RV64F instructions

Instruction	Description	Latency
Operation instruction		
FADD.S	A single-precision floating-point add instruction.	3
FSUB.S	A single-precision floating-point subtract instruction.	3
FMUL.S	A single-precision floating-point multiply instruction	4
FMADD.S	A single-precision floating-point multiply-add instruction.	5

Continued on next page

Table 3.4 – continued from previous page

Instruction	Description	Latency
FMSUB.S	A single-precision floating-point multiply-subtract instruction.	5
FNMADD.S	A single-precision floating-point negate-(multiply-add) instruction.	5
FNMSUB.S	A single-precision floating-point negate -(multiply-subtract) instruction.	5
FDIV.S	A single-precision floating-point divide instruction.	4-10
FSQRT.S	A single-precision floating-point square-root instruction.	4-10
Sign injection instructions		
FSGNJ.S	A single-precision floating-point sign-injection instruction.	3
FSGNJN.S	A single-precision floating-point negate sign-injection instruction.	3
FSGNJX.S	A single-precision floating-point sign-injection XOR instruction.	3
Data transmission instructions		
FMV.X.W	A single-precision floating-point read move instruction	1+1 in split execution
FMV.W.X	A single-precision floating-point write move instruction.	1+1 in split execution
Compare instructions		
FMIN.S	A single-precision floating-point MIN instruction.	3
FMAX.S	A single-precision floating-point MAX instruction.	3
FEQ.S	A single-precision floating-point compare equal instruction.	3+1 in split execution
FLT.S	A single-precision floating-point compare less than instruction.	3+1 in split execution
FLE.S	A single-precision floating-point compare less than or equal to instruction.	3+1 in split execution
Data type conversion instructions		

Continued on next page

Table 3.4 – continued from previous page

Instruction	Description	Latency
FCVT.W.S	An instruction that converts a single-precision floating-point number into a signed integer.	3+1 in split execution
FCVT.WU.S	An instruction that converts a single-precision floating-point number into an unsigned integer.	3+1 in split execution
FCVT.S.W	An instruction that converts a signed integer into a single-precision floating-point number.	3+1 in split execution
FCVT.S.WU	An instruction that converts an unsigned integer into a single-precision floating-point number.	3+1 in split execution
FCVT.L.S	An instruction that converts a single-precision floating-point number into a signed long integer.	3+1 in split execution
FCVT.LU.S	An instruction that converts a single-precision floating-point number into an unsigned long integer.	3+1 in split execution
FCVT.S.L	An instruction that converts a signed long integer into a single-precision floating-point number.	1+3 in split execution
FCVT.S.LU	An instruction that converts an unsigned long integer into a single-precision floating-point number.	1+3 in split execution
Memory store instruction		
FLW	A single-precision floating-point load instruction.	WEAK ORDER LOAD: ≥ 3 STORE: 1 STRONG ORDER Aperiodic
FSW	A single-precision floating-point store instruction.	Same as above
Floating-point classify instructions		
FCLASS.S	A single-precision floating-point classify instruction.	1+1

For more information, see *Appendix A-4 F instructions* .

3.1.5 Compressed instruction set (RV64C)

The compressed instruction set includes instructions of the following types by feature:

- Add/Subtract instructions
- Logical operation instructions
- Shift instructions
- Data transmission instructions
- Branch and jump instructions
- Immediate offset access instructions

Table 3.5: RV64C instructions

Instruction	Description	Latency
Add/Subtract instructions		
C.ADD	A signed add instruction	1
C.ADDW	A signed add instruction that operates on the lower 32 bits.	1
C.ADDI	A signed add immediate instruction.	1
C.ADDIW	A signed add immediate instruction that operates on the lower 32 bits.	1
C.SUB	A compressed signed subtract instruction.	1
C.SUBW	A signed subtract instruction that operates on the lower 32 bits	1
C.ADDI16SP	An instruction that adds an immediate scaled by 16 to the stack pointer.	1
C.ADDI4SPN	An instruction that adds an immediate scaled by 4 to the stack pointer	1
Logic operation instructions		
C.AND	A bitwise AND instruction	1
C.ANDI	An immediate bitwise AND instruction	1
C.OR	A bitwise OR instruction	1
C.XOR	A bitwise XOR instruction	1
Shift instructions		

Continued on next page

Table 3.5 – continued from previous page

Instruction	Description	Latency
C.SLLI	An immediate logical left shift instruction.	1
C.SRLI	An immediate logical right shift instruction.	1
C.SRAI	An immediate arithmetic right shift instruction.	1
Data transmission instructions		
C.MV	A data move instruction	1
C.LI	An instruction for moving immediates in the lower bits	1
C.LUI	An instruction for moving immediates in the upper bits	1
Branch and jump instructions		
C.BEQZ	A branch- if-equal-to-zero instruction.	1
C.BNEZ	A branch- if-not-equal-to-zero instruction.	1
C.J	An unconditional jump instruction	1
C.JR	A register-based jump instruction	1
C.JALR	An instruction for jumping to a subroutine by using an address in a register	1
Immediate offset access instructions		
C.LW	A word load instruction	Weak order LOAD: ≥ 3 STORE: 1 STRONG ORDER Aperiodic
C.SW	A word store instruction.	Same as above
C.LWSP	A word stack load instruction	Same as above
C.SWSP	A word stack store instruction	Same as above
C.LD	A doubleword load instruction.	Same as above
C.SD	A doubleword store instruction	Same as above
C.LDSP	A doubleword stack load instruction	Same as above

Continued on next page

Table 3.5 – continued from previous page

Instruction	Description	Latency
C.SDSP	A doubleword stack store instruction	Same as above
C.FLD	A double-precision load instruction.	Same as above
C.FSD	A double-precision store instruction.	Same as above
C.FLDSP	A double-precision stack store instruction.	Same as above
C.FSDSP	A double-precision stack load instruction.	Same as above
Special instructions		
C.NOP	A no-operation instruction	1
C.EBREAK	A breakpoint instruction	1

For more information, see *Appendix A-6 C Instructions* .

3.1.6 Vector instruction set (RVV)

For more information, see *RISC-V “V” Vector Extension, Version 1.0-rc1-20210608*

URL: <https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf>

3.1.7 Bit operation instruction set (RV64B)

For more information, see : *RISC-V Bit-Manipulation ISA-extensions, Version 1.0.0, 2021-06-12: public review*

URL: <https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf>

3.2 XuanTie extended instruction sets

C908 provides some extended custom instructions based on the RV64GCB[V] instruction sets. Extended half-precision floating-point instructions of C908 can be directly used. All other extended instruction sets of C908 must be enabled before they can be used; otherwise, illegal instruction errors will occur. To enable an extended instruction set, enable the THEADISAE bit in the MXSTATUS register.

3.2.1 Arithmetic operation instructions

Table 3.6: Arithmetic operation instructions

Instruction	Description	Execution Latency
Add/Subtract instructions		
ADDSL	An add register instruction that shifts registers	1
MULA	A multiply-add instruction	Additive numbers uncorrelated: 4
MULS	A multiply-subtract instruction	Additive numbers uncorrelated: 4
MULAW	A multiply-add instruction that operates on the lower 32 bits	Additive numbers correlated: 1
MULSW	A multiply-subtract instruction that operates on the lower 32 bits.	Additive numbers correlated: 1
MULAH	A multiply-add instruction that operates on the lower 16 bits	Additive numbers correlated: 1
MULSH	A multiply-subtract instruction that operates on the lower 16 bits.	Additive numbers correlated: 1
Shift instructions		
SRRI	A cyclic right shift instruction.	1
SRRIW	A cyclic right shift instruction that operates on the lower 32 bits.	1
Move instructions		
MVEQZ	An instruction for moving values when the register value is 0	1
MVNEZ	An instruction for moving values when the register value is not 0	1

For more information, see [Appendix B-3 Arithmetic operation instructions](#).

3.2.2 Bit operation instructions

Table 3.7: Bit operation instructions

Instruction	Description	Execution latency
Bit operation instructions		
TST	An instruction for testing bits with the value of 0.	1
TSTNBZ	An instruction for testing bytes with the value of 0.	1
REV	An instruction for reversing the byte order.	1
REVV	An instruction for reversing the byte order in the lower 32 bits.	1
FF0	An instruction for fast finding the first bit with the value of 0 in a register.	1
FF1	An instruction for fast finding the first bit with the value of 1 in a register.	1
EXT	A signed extension instruction for extracting consecutive bits of a register.	1
EXTU	A zero extension instruction for extracting consecutive bits of a register.	1

For more information, see *Appendix B-4 Bitwise operation instructions*.

3.2.3 Memory access instructions

Table 3.8: Memory access instructions

Instruction	Description	Execution latency
FLRD	A doubleword load instruction for shifting floating-point registers.	Weak order ≥ 3 STRONG ORDER Aperiodic
FLRW	A word load instruction for shifting floating-point registers.	

Continued on next page

Table 3.8 – continued from previous page

Instruction	Description	Execution latency
FLURD	A doubleword load instruction for shifting the lower 32 bits in floating-point registers.	
FLURW	A word load instruction for shifting the lower 32 bits in floating-point registers.	
LRB	A byte load instruction for shifting registers and extending signed bits.	
LRH	A halfword load instruction for shifting registers and extending signed bits	
LRW	A halfword load instruction for shifting registers and extending signed bits	
LRD	A doubleword load instruction for shifting registers.	
LRBU	A byte load instruction for shifting registers and extending zero bits.	
LRHU	A halfword load instruction for shifting registers and extending zero bits.	
LRWU	A word load instruction for shifting registers and extending zero bits.	
LURB	A byte load instruction for shifting registers and extending signed bits.	
LURH	A halfword load instruction for shifting registers and extending signed bits.	
LURW	A word load instruction for shifting the lower 32 bits in registers and extending signed bits.	

Continued on next page

Table 3.8 – continued from previous page

Instruction	Description	Execution latency
LURD	A doubleword load instruction for shifting the lower 32 bits in floating-point registers.	
LURBU	A byte load instruction for shifting the lower 32 bits in registers and extending zero bits.	
LURHU	A halfword load instruction for shifting the lower 32 bits in registers and extending zero bits.	
LURWU	A word load instruction for shifting the lower 32 bits in registers and extending zero bits.	
LBIA	A base-address auto-increment instruction for loading bytes and extending signed bits.	This instruction is split into the load and ALU instructions for execution.
LBIB	A byte load instruction for auto-incrementing the base address and extending signed bits.	
LHIA	A base-address auto-increment instruction for loading halfwords and extending signed bits.	
LHIB	A halfword load instruction for auto-incrementing the base address and extending signed bits.	
LWIA	A base-address auto-increment instruction for loading words and extending signed bits.	

Continued on next page

Table 3.8 – continued from previous page

Instruction	Description	Execution latency
LWIB	The word load instruction for auto-incrementing the base address and extending signed bits.	
LDIA	A base-address auto-increment instruction for loading doublewords and extending signed bits.	
LDIB	A doubleword load instruction for auto-incrementing the base address and extending signed bits.	
LBUIA	A base-address auto-increment instruction for loading bytes and extending zero bits.	
LBUIB	A byte load instruction for auto-incrementing the base address and extending zero bits.	
LHUIA	An address auto-increment instruction for loading half-words and extending zero bits.	
LHUIB	A halfword load instruction for auto-incrementing the base address and extending zero bits.	
LWUIA	An address auto-increment instruction for loading words and extending zero bits.	
LWUIB	A word load instruction for auto-incrementing the base address and extending zero bits.	
LDD	A double-register load instruction.	This instruction is split into two load instructions for execution.

Continued on next page

Table 3.8 – continued from previous page

Instruction	Description	Execution latency
LWD	A double-register word load instruction for extending signed bits.	
LWUD	A double-register word load instruction for extending zero bits.	
FSRD	A doubleword store instruction for shifting floating-point registers.	Weak order LOAD: ≥ 3 STORE: 1 STRONG ORDER Aperiodic
FSRW	A word store instruction for shifting floating-point registers.	
FSURD	A doubleword store instruction for shifting the lower 32 bits in floating-point registers.	
FSURW	A word store instruction for shifting the lower 32 bits in floating-point registers.	
SRB	A byte store instruction for shifting registers.	
SRW	A word store instruction for shifting registers.	
SRD	A doubleword store instruction for shifting registers.	
SURB	A byte store instruction for shifting the lower 32 bits in registers.	
SURH	A halfword store instruction for shifting the lower 32 bits in registers.	
SURW	A word store instruction for shifting the lower 32 bits in registers.	

Continued on next page

Table 3.8 – continued from previous page

Instruction	Description	Execution latency
SURD	A doubleword store instruction for shifting the lower 32 bits in floating-point registers	
SBIA	A base-address auto-increment instruction for storing bytes	This instruction is split into the store and ALU instructions for execution.
SBIB	A byte store instruction for auto-incrementing the base address.	
SHIA	A base-address auto-increment instruction for storing halfwords.	
SHIB	A halfword store instruction for auto-incrementing the base address.	
SWIA	A base-address auto-increment instruction for storing words.	
SWIB	A word store instruction for auto-incrementing the base address.	
SDIA	A base-address auto-increment instruction for storing doublewords	
SDIB	A doubleword store instruction for auto-incrementing the base address.	
SDD	A double-register store instruction.	This instruction is split into two store instructions for execution.
SWD	An instruction for storing the lower 32 bits in double registers	

For more information, see *Appendix B-5 Storage instructions*.

3.2.4 Cache instructions

Table 3.9: Cache instructions

Instruction	Description	Execution latency (LMUL=1)
DCACHE.CALL	An instruction that clears all dirty page table entries in the D-Cache.	Blocked Aperiodic
DCACHE.CIALL	An instruction that clears all dirty page table entries in the D-Cache and invalidates the entries.	
DCACHE.CIPA	An instruction that clears dirty page table entries that match the specified physical addresses in the D-Cache and invalidating the entries. (This instruction also acts on the L2 cache.)	
DCACHE.CISW	An instruction that clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the entries.	
DCACHE.CIVA	An instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the entries. (This instruction also acts on the L2 cache.)	
DCACHE.CPA	An instruction that clears dirty page table entries that match the specified physical addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.CPAL1	An instruction that clears dirty page table entries that match the specified physical addresses in the L1 D-Cache.	
DCACHE.CSW	An instruction that clears dirty page table entries in the D-Cache based on the specified way and set.	
DCACHE.CVA	An instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.CVAL1	An instruction that clears dirty page table entries that match the specified virtual addresses in the L1 D-Cache.	
DCACHE.IPA	An instruction that invalidates page table entries that match the specified physical addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.ISW	An instruction that invalidates page table entries in the D-Cache based on the specified way and set.	

Continued on next page

Table 3.9 – continued from previous page

Instruction	Description	Execution latency (LMUL=1)
DCACHE.IVA	An instruction that invalidates page table entries that match the specified virtual addresses in the D-Cache. (This instruction also acts on the L2 cache.)	
DCACHE.IALL	An instruction that invalidates all page table entries in the D-Cache	
ICACHE.IALL	An instruction that invalidates all page table entries in the I-Cache	Aperiodic
ICACHE.IALLS	An instruction that invalidates all page table entries in the I-Cache through broadcasting	
ICACHE.IPA	An instruction that invalidates page table entries that match the specified physical addresses in the I-Cache.	
ICACHE.IVA	An instruction that invalidates page table entries that match the specified virtual addresses in the I-Cache.	

For more information, see *Appendix B-1 Cache instructions*.

3.2.5 Multi-core synchronization instructions

Table 3.10: Multi-core synchronization instructions

Instruction	Description
SFENCE.VMAS	A broadcast instruction for synchronizing virtual memory.
SYNC	A synchronization instruction
SYNC.S	A synchronization broadcast instruction
SYNC.I	An instruction for synchronizing the clearing operation.
SYNC.IS	A broadcast instruction for synchronizing the clearing operation.

For more information, see *Appendix B-2 Multi-core synchronization instructions*.

3.2.6 Half-precision floating-point instructions

Table 3.11: Half-precision floating-point instructions

Instruction	Description	Execution latency
Operation instructions		
FADD.H	A half-precision floating-point add instruction.	3
FSUB.H	A half-precision floating-point subtract instruction.	3
FMUL.H	A half-precision floating-point multiply instruction.	3
FMADD.H	A half-precision floating-point multiply-add instruction.	4
FMSUB.H	A half-precision floating-point multiply-subtract instruction.	4
FNMADD.H	A half-precision floating-point negate-(multiply-add) instruction.	4
FNMSUB.H	A half-precision floating-point negate -(multiply-subtract) instruction.	4
FDIV.H	A half-precision floating-point divide instruction.	4-7
FSQRT.H	A half-precision floating-point square-root instruction.	4-7
Sign injection instructions		
FSGNJ.H	A half-precision floating-point sign-injection instruction	3
FSGNJN.H	A half-precision floating-point negate sign-injection instruction	3
FSGNJX.H	A half-precision floating-point XOR sign-injection instruction	3
Data transmission instructions		
FMV.X.H	A half-precision floating-point read move instruction.	1+1
FMV.H.X	A half-precision floating-point write move instruction	1+1
Compare instructions		
FMIN.H	A half-precision floating-point MIN instruction	3
FMAX.H	A half-precision floating-point MAX instruction.	3

Continued on next page

Table 3.11 – continued from previous page

Instruction	Description	Execution latency
FEQ.H	A half-precision floating-point compare equal instruction.	3+1 in split execution
FLT.H	A half-precision floating-point compare less than instruction.	3+1 in split execution
FLE.H	A half-precision floating-point compare less than or equal to instruction.	3+1 in split execution
Data type conversion instructions		
FCVT.S.H	An instruction that converts a half-precision floating-point number into a single-precision floating-point number.	3
FCVT.H.S	An instruction that converts a single-precision floating-point number into a half-precision floating-point number.	3
FCVT.W.H	An instruction that converts a half-precision floating-point number into a signed integer.	3+1 in split execution
FCVT.WU.H	An instruction that converts a half-precision floating-point number into an unsigned integer.	3+1 in split execution
FCVT.H.W	An instruction that converts a signed integer into a half-precision floating-point number	3+1 in split execution
FCVT.H.WU	The instruction that converts an unsigned integer into a half-precision floating-point number.	3+1 in split execution
FCVT.L.H	An instruction that converts a half-precision floating-point number into a signed long integer.	3+1 in split execution
FCVT.LU.H	An instruction that converts a half-precision floating-point number into an unsigned long integer.	3+1 in split execution
FCVT.H.L	An instruction that converts a signed long integer into a half-precision floating-point number.	3+1 in split execution

Continued on next page

Table 3.11 – continued from previous page

Instruction	Description	Execution latency
FCVT.H.LU	An instruction that converts an unsigned long integer into a half-precision floating-point number.	3+1 in split execution
Memory store instructions		
FLH	A half-precision floating-point load instruction	Weak order LOAD: ≥ 3 STORE: 1 STRONG ORDER
FSH	A half-precision floating-point store instruction.	Same as above
Floating-point classify instructions		
FCLASS.H	A single-precision floating-point classify instruction	1+1

For more information, see *Appendix B-6 Half-precision floating-point instructions* .

4.1 CPU modes

C908 supports three RISC-V privilege modes : machine mode (M-mode), supervisor mode (S-mode), and user mode (U-mode). C908 runs programs in M-mode after reset. The three modes correspond to different operation privileges and differ in the following aspects:

1. Register access
2. Use of privileged instructions
3. Memory access

The U-mode provides the lowest privileges.

User programs are allowed to access only the registers specific to the U-mode. This prevents user programs from accessing privileged information. The operating system manages and serves user programs by coordinating their behaviors.

The S-mode provides higher privileges than the U-mode but lower privileges than the M-mode.

Programs running in S-mode are not allowed to access control registers specific to the M-mode and are limited by physical memory protection (PMP). The page-based virtual memory acts as the core of the S-mode.

The M-mode has the highest privileges.

Programs running in M-mode have full access to memory, I/O resources, and underlying features required for starting and configuring the system. By default, the CPU switches to the M-mode to respond to exceptions and interrupts that occur in any mode unless the exceptions and interrupts are delegated.

Most instructions can run in all the three modes. However, some privileged instructions with major impact on the system are available only in S-mode or M-mode. For more information, see *Appendix A Standard Instructions* and *Appendix B T-Head Extended Instructions*.

The privilege mode in which an exception occurs is different from that in which the CPU responds to the exception. The CPU switches to a higher privilege mode to respond to the exception, and switches back to the lower privilege mode after the exception is handled.

4.2 Register view

The register view of C908 is shown in Fig. 4.1.

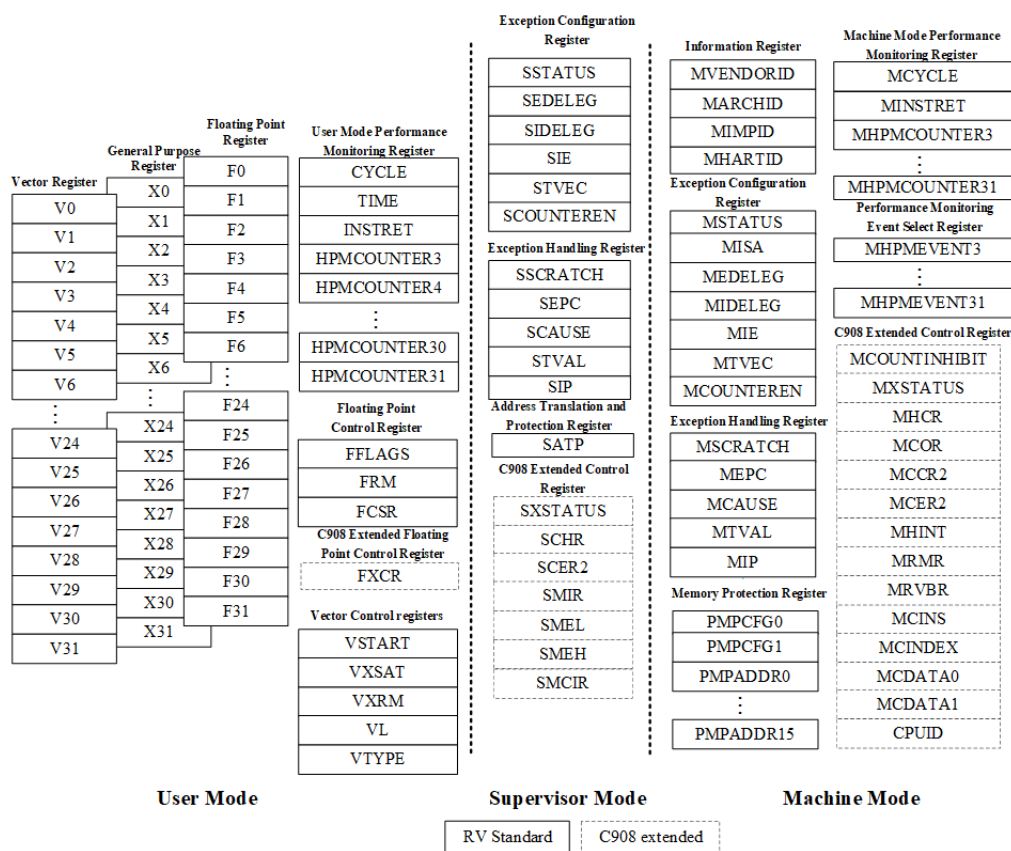


Fig. 4.1: Register view

4.3 General-purpose registers

C908 provides thirty-two 64-bit general-purpose registers that have the same features as those defined in RISC-V. For more information, see Table 4.1.

Table 4.1: General-purpose registers

Register	ABI name	Description
x0	zero	A hardwired zero register.
x1	ra	A return address register.
x2	sp	A stack pointer register.
x3	gp	A global pointer register.
x4	tp	A thread pointer register.
x5	t0	A temporary/standby link register.
x6-7	t1-2	Temporary registers.
x8	s0/fp	A reserved register/frame pointer register.
x9	s1	A reserved register.
x10-11	a0-1	Function argument/Return value registers.
x12-17	a2-7	Function argument registers.
x18-27	s2-11	Reserved registers.
x28-31	t3-6	Temporary registers.

The general-purpose registers are used to store instruction operands, instruction execution results, and address information.

4.4 Floating-point registers

In addition to standard RV64F instructions, C908 also supports floating-point half-precision computing and provides 32 independent floating-point registers. These registers are accessible in U-mode, S-mode, and M-mode.

Table 4.2: Floating-point registers

Register	ABI name	Description
f0-7	ft0-7	Floating-point temporary registers.
f8-9	fs0-1	Floating-point reserved registers.
f10-11	fa0-1	Floating-point argument/return value registers.
f12-17	fa2-7	Floating-point argument registers.
f18-27	fs2-11	Floating-point reserved registers.
f28-31	ft8-11	Floating-point temporary registers.

Unlike x0, f0 is not hardwired to 0, but its bit values are variable like other floating-point registers. A single-precision floating-point number occupies only the lower 32 bits of a 64-bit floating-point register, and the upper 32 bits must be set to 1; otherwise, the number will be considered nonnumeric. A half-precision floating-point number occupies only the lower 16 bits of a 64-bit floating-point register, and the upper 48 bits must be set to 1; otherwise, the number will be considered nonnumeric.

The independent floating-point registers help increase the register capacity and bandwidth, improving performance of the CPU. Along with the floating-point registers, floating-point load and store instructions and instructions for transferring data between floating-point and general-purpose registers are added.

4.4.1 Transmit data between floating-point and general-purpose registers

Data can be transmitted between floating-point and general-purpose registers through floating-point register move instructions. Floating-point register move instructions include:

- FMV.X.H/FMV.H.X: A half-precision data move instruction for floating-point registers.
- FMV.X.W/FMV.W.X: A single-precision data move instruction for floating-point registers.

When half-precision, single-precision data is transmitted from a general-purpose register to a floating-point register, the data format remains unchanged. Therefore, a program can directly use these registers without converting their types.

For more information, see *Appendix A-4 F instructions*.

4.4.2 Maintain consistency of register precision

Floating-point registers can store half-precision, single-precision and integer data. For example, the type of data stored in f1 depends on the last write operation, and may be any one of the four types.

Floating-point units (FPUs) do not detect data formats based on hardware. The hardware parses data formats in a floating-point register only based on the executed floating-point instruction, regardless of the data format in the last write operation in the register. In this case, the consistency of data precision in the register is ensured only by the compiler or program.

4.5 Vector registers

C908 owns 32 independent vector registers that can be configured as 128-bit or 256-bit. These registers are accessible in U-mode, S-mode, and M-mode. Vector registers exchange data with integer general-purpose registers and floating-point registers through vector move instructions.

4.5.1 Transmit data between floating-point and general-purpose registers

Data can be transmitted between vector and general-purpose registers through integer vector register move instructions. Integer vector register move instructions include:

- VMV.V.X: an instruction that moves data from an integer register to a vector register;
- VMV.S.X: an instruction that moves data from an integer register to element 0 of a vector register;
- VEXT.X.V: an integer vector get element instruction.

4.5.2 Transmit data between floating-point and vector registers

Data can be transmitted between vector and floating-point registers through floating-point vector register move instructions. Floating-point vector register move instructions include:

- VFMV.V.F: an instruction that moves data from a floating-point register to a vector register;
- VFMV.F.S: an instruction that moves data from element 0 of a vector register to a floating-point register;
- VFMV.S.F: an instruction that moves data from a floating-point register to element 0 of the vector register.

4.6 System control registers

4.6.1 Standard control registers

This section describes RISC-V standard control registers implemented in C908 by M-mode, S-mode, and U-mode.

The RISC-V standard M-mode control registers implemented in C908 are described in [Table 4.3](#).

Table 4.3: RISC-V standard M-mode control registers

Register	Read/Write permission	ID	Description
M-mode information registers			
mvendorid	Read-only in M-mode	0xF11	A vendor ID register.
marchid	Read-only in M-mode	0xF12	An architecture ID register.
mimpid	Read-only in M-mode	0xF13	An M-mode hardware implementation ID register.
mhartid	Read-only in M-mode	0xF14	An M-mode logical kernel ID register.
M-mode exception configuration registers			
mstatus	Read/Write in M-mode	0x300	An M-mode CPU status register.
misa	Read/Write in M-mode	0x301	An M-mode CPU instruction set attribute register.
medeleg	Read/Write in M-mode	0x302	An M-mode exception delegation control register.
mideleg	Read/Write in M-mode	0x303	An M-mode interrupt delegation control register.
mie	Read/Write in M-mode	0x304	An M-mode interrupt enable control register.

Continued on next page

Table 4.3 – continued from previous page

Register	Read/Write permission	ID	Description
mtvec	Read/Write in M-mode	0x305	An M-mode vector base address register.
mcounteren	Read/Write in M-mode	0x306	An M-mode counter enable control register.
mcountinhibit	Read/Write in M-mode	0x320	An M-mode count inhibit register.
M-mode exception handling registers			
mscratch	Read/Write in M-mode	0x340	An M-mode temporary data backup register upon exceptions.
mepc	Read/Write in M-mode	0x341	An M-mode exception program counter.
mcause	Read/Write in M-mode	0x342	An M-mode exception event cause register.
mtval	Read/Write in M-mode	0x343	An M-mode exception event vector register.
mip	Read/Write in M-mode	0x344	An M-mode interrupt pending state register.
M-mode memory protection registers			
pmpcfg0	Read/Write in M-mode	0x3A0	Physical memory protection configuration register 0.
pmpaddr0	Read/Write in M-mode	0x3B0	Physical memory protection base address register 0.
...			
pmpaddr7	Read/Write in M-mode	0x3B7	Physical memory protection base address register 7.
M-mode counters/timers			
mcycle	Read/Write in M-mode	0xB00	An M-mode cycle counter.
minstret	Read/Write in M-mode	0xB02	An M-mode retired instruction counter.
mhpmcounter3	Read/Write in M-mode	0xB03	Machine-mode counter 3.
...			
mhpmcounter31	Read/Write in M-mode	0xB1F	M-mode counter 31.
M-mode counter configuration registers			
mhpmevent3	Read/Write in M-mode	0x323	M-mode event select register 3.
...			
mhpmevent31	Read/Write in M-mode	0x33F	M-mode event select register 31.

The RISC-V standard S-mode control registers implemented in C908 are described in [Table 4.4](#).

Table 4.4: RISC-V standard S-mode control registers

Register	Read/Write permission	ID	Description
S-mode exception configuration registers			
sstatus	Read/Write in S-mode	0x100	An S-mode CPU status register.
sie	Read/Write in S-mode	0x104	An S-mode interrupt enable control register.
stvec	Read/Write in S-mode	0x105	An S-mode vector base address register.
scounteren	Read/Write in S-mode	0x106	An S-mode counter enable control register.
S-mode exception handling registers			
sscratch	Read/Write in S-mode	0x140	An S-mode temporary data backup register upon exceptions.
sepc	Read/Write in S-mode	0x141	An S-mode exception program counter.
scause	Read/Write in S-mode	0x142	An S-mode exception event cause register.
stval	Read/Write in S-mode	0x143	An S-mode exception event vector register.
sip	Read/Write in S-mode	0x144	An S-mode interrupt pending state register.
S-mode address translation registers			
satp	Read/Write in S-mode	0x180	An S-mode virtual address translation and protection register.

The RISC-V standard user-mode control registers implemented in C908 are described in [Table 4.5](#).

Table 4.5: RISC-V standard U-mode control registers

Register	Read/Write permission	ID	Description
U-mode floating-point control registers			
fflags	Read/Write in U-mode	0x001	A floating-point accrued exception status register.
frm	Read/Write in U-mode	0x002	A floating-point dynamic rounding mode control register.
fcsr	Read/Write in U-mode	0x003	A floating-point control and status register.
U-mode counters/timers			
cycle	Read/Write in U-mode	0xC00	A U-mode cycle counter.
time	Read/Write in U-mode	0xC01	A U-mode timer.
instret	Read/Write in U-mode	0xC02	A U-mode retired instruction counter.
hpmcounter3	Read/Write in U-mode	0xC03	A U-mode counter 3.
...			
hpmcounter31	Read/Write in U-mode	0xC1F	U-mode counter 31.

4.6.2 Extended control registers

This section describes extended control registers implemented in C908 by M-mode, S-mode, and U-mode.

The extended M-mode control registers of C908 are described in [Table 4.6](#).

Table 4.6: Extended M-mode control registers of C908

Register	Read/Write permission	ID	Description
Extended M-mode CPU control and status registers			
mxstatus	Read/Write in M-mode	0x7C0	An extended M-mode status register.
mhcr	Read/Write in M-mode	0x7C1	An M-mode hardware configuration register.
mcor	Read/Write in M-mode	0x7C2	An M-mode hardware operation register.
mccr2	Read/Write in M-mode	0x7C3	An M-mode L2 cache control register.
mhint	Read/Write in M-mode	0x7C5	An M-mode implicit operation register.
mrubr	Read-only in M-mode	0x7C7	An M-mode reset vector base address register.
mcounterwen	Read/Write in M-mode	0x7C9	An S-mode counter write enable register.
mcounterinten	Read/Write in M-mode	0x7CA	An M-mode event interrupt enable register.
mcounterof	Read/Write in M-mode	0x7CB	An M-mode overflow flag register.
Extended M-mode cache access registers			
mcins	Read/Write in M-mode	0x7D2	An M-mode cache instruction register.
mcindex	Read/Write in M-mode	0x7D3	An M-mode cache access index register.
mcdata0	Read/Write in M-mode	0x7D4	An M-mode cache data register 0.
mcdata1	Read/Write in M-mode	0x7D5	An M-mode cache data register 1.
Extended M-mode CPU model registers			
mcuid	Read-only in M-mode	0xFC0	An M-mode CPU model register.
mapbaddr	Read-only in M-mode	0xFC1	An on-chip bus base address register.
Extended multi-core registers			
msmpr	Read/Write in M-mode	0x7F3	A snooping enable register.

For more information, see [Appendix C-1 M-mode control registers](#).

The extended S-mode control registers of C908 are described in [Table 4.7](#).

Table 4.7: Extended S-mode control registers of C908

Register	Read/Write permission	ID	Description
Extended S-mode CPU control and status registers			
sxstatus	Read/Write in S-mode	0x5C0	An extended S-mode status register.
shcr	Read/Write in S-mode	0x5C1	An S-mode hardware control register.
scounterinten	Read/Write in S-mode	0x5C4	An S-mode event interrupt enable register.
scounterof	Read/Write in S-mode	0x5C5	An S-mode event overflow flag register.
scycle	Read/Write in S-mode	0x5E0	An S-mode cycle counter.
shpmcounter3	Read/Write in S-mode	0x5E3	S-mode counter 3.
...			
shpmcounter31	Read/Write in S-mode	0x5FF	S-mode counter 31.
Extended S-mode MMU registers			
smir	Read/Write in S-mode	0x9C0	An S-mode MMU index register.
smel	Read/Write in S-mode	0x9C1	An S-mode MMU EntryLo register.
smeh	Read/Write in S-mode	0x9C2	An S-mode MMU EntryHi register.
smcir	Read/Write in S-mode	0x9C3	An S-mode MMU control register.

For more information, see [Appendix C-2 S-mode control registers](#).

The extended U-mode control registers of C908 are described in [Table 4.8](#).

Table 4.8: Extended U-mode control registers of C908

Register	Read/Write permission	ID	Description
Extended U-mode floating-point control registers			
fxcr	Read/Write in U-mode	0x800	An extended U-mode floating-point control register.

For more information, see [Appendix C-3 U-mode control registers](#).

4.7 Data formats

4.7.1 Integer data format

Values in a register are not distinguished by big-endian or little-endian type, but by signed or unsigned type. Values in a register are arranged from right to left with the least significant bit being the rightmost bit and the most significant bit being the leftmost bit, as shown in Fig. 4.2.

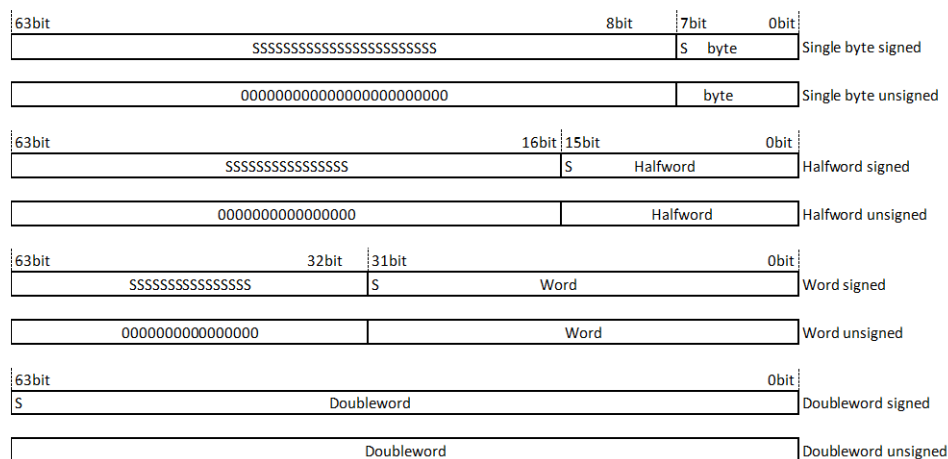


Fig. 4.2: Integer data structure in registers

4.7.2 Floating-point data format

FPU of C908 complies with the RISC-V standard and the ANSI/IEEE 754-2008 standard for floating-point arithmetic, and support half-precision, single-precision computation. The floating-point data format is shown in Fig. 4.3. Single-precision data occupies only the lower 32 bits of a 64-bit floating-point register, and the upper 32 bits must be set to 1; otherwise, the data will be considered nonnumeric. Half-precision data occupies only the lower 16 bits of a 64-bit floating-point register, and the upper 48 bits must be set to 1; otherwise, the data will be considered nonnumeric.

4.8 Big-endian and little-endian

The concepts of big-endian and little-endian are proposed with respect to the data storage formats of memories. In the big-endian scheme, the most significant byte of an address is stored to the lower bits in physical memory. In the little-endian scheme, the most significant byte of an address is stored to the upper bits in physical memory. The data formats are shown in Fig. 4.4.

C908 supports only the little-endian scheme, and supports binary integers with standard complements. The length of each instruction operand can be explicitly encoded in programs (load/store instructions) or

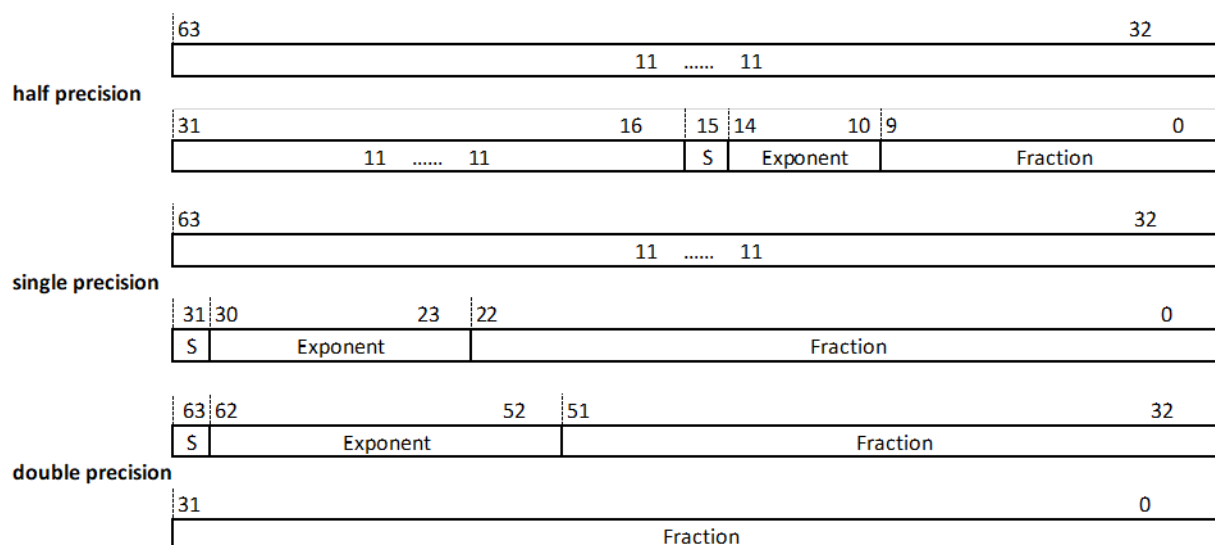


Fig. 4.3: Floating-point data structure in registers

A+7	A+6	A+5	A+4	A+3	A+2	A+1	A	
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	Double word at A
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	Word at A
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	Half word at A
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	Byte at A

little-endian scheme

A+4	A+5	A+6	A+7	A	A+1	A+2	A+3	
Byte3	Byte2	Byte1	Byte0	Byte7	Byte6	Byte5	Byte4	Double word at A
Byte4	Byte5	Byte6	Byte7	Byte3	Byte2	Byte1	Byte0	Word at A
Byte6	Byte7	Byte4	Byte5	Byte1	Byte0	Byte3	Byte2	Half word at A
Byte7	Byte6	Byte5	Byte4	Byte0	Byte1	Byte2	Byte3	Byte at A

big-endian V1 scheme

A+7	A+6	A+5	A+4	A+3	A+2	A+1	A	
Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Double word at A
Byte4	Byte5	Byte6	Byte7	Byte0	Byte1	Byte2	Byte3	Word at A
Byte6	Byte7	Byte4	Byte5	Byte2	Byte3	Byte0	Byte1	Half word at A
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	Byte at A

big-endian V2 scheme

Fig. 4.4: Data structure in memory

implicitly indicated in instruction operations (index operation and byte extraction) Usually, an instruction receives a 64-bit operand and generates a 64-bit result.

5.1 Overview

Exception handling is a core feature of a CPU. Exceptions include instruction exceptions and external interrupts. When some exception events occur, the CPU is enabled to respond to these events. The events include hardware errors, instruction execution errors, and user program request services.

The key of exception handling is to save the operating status of the CPU when an exception occurs and resume the status when the CPU exits exception handling. Exceptions can be identified in all stages of the instruction pipeline. The CPU hardware ensures that subsequent instructions do not change the CPU status. Exceptions are handled at the boundary of an instruction. To be specific, the CPU responds to the exceptions when the instruction retires, and saves the address of the to-be-executed instruction when the CPU exits exception handling. Even if exceptions are identified before an instruction retires, the CPU does not handle the exceptions until the instruction retires. To ensure proper functioning of programs, the CPU does not repeatedly run the executed instructions after exception handling is completed.

In machine mode (M-mode), the CPU responds to an instruction exception or an external interrupt in the following procedure:

Step 1: Save the exception PC to the mepc register.

Step 2: Update the mcause and mtval registers based on the exception type.

Step 3: Save the machine interrupt-enable (MIE) bit in the mstatus register to the MPIE field, clear the MIE field, and prohibit responses to interrupts.

Step 4: Save the privilege mode applied before the exception occurs to the MPP field in the mstatus register, and switch to the M-mode.

Step 5: Obtain the entry address of exception program based on the base address and mode in the mtvec register, and run instructions of the exception program in sequence.

C908 conforms to the exception vector table defined in RISC-V, as shown in [Table 5.1](#).

Table 5.1: Exception and interrupt vector assignment

Interrupt flag	Exception vector ID	Description
1	0	Unavailable.
1	1	A software interrupt in supervisor mode (S-mode).
1	2	Reserved.
1	3	A software interrupt in M-mode.
1	4	Unavailable.
1	5	A timer interrupt in S-mode.
1	6	Reserved.
1	7	The timer interrupt in M-mode.
1	8	Unavailable.
1	9	An external interrupt in S-mode.
1	10	Reserved.
1	11	An external interrupt in M-mode.
1	17	A performance detection overflow interrupt.
1	Others	Reserved.
0	0	Unavailable.
0	1	A fetch instruction access error exception.
0	2	An illegal instruction exception.
0	3	A debug breakpoint exception.
0	4	A load instruction unaligned access exception.
0	5	A load instruction access error exception.
0	6	A store/atomic instruction unaligned access exception.
0	7	A store/atomic instruction access error exception.
0	8	A user-mode (U-mode) environment call exception.
0	9	An S-mode environment call exception.
0	10	Reserved.
0	11	An M-mode environment call exception.
0	12	An instruction fetch page error exception.
0	13	A load instruction page error exception.
0	14	Reserved.
0	15	A store/atomic instruction page error exception.
0	≥ 16	Reserved.

C908 supports exception and interrupt delegation. When an exception or interrupt occurs in S-mode, the CPU switches to the M-mode for handling. This causes performance loss of the CPU. Delegation enables the CPU to respond to exceptions and interrupts in S-mode. Exceptions that occur in M-mode are not delegated, but still handled in M-mode. Interrupts that occur in M-mode can be delegated to the S-mode for handling, except the external interrupts, software interrupts, and timer interrupts that occur in M-mode. In M-mode, the CPU does not respond to delegated interrupts.

In S-mode and U-mode, the CPU can respond to all interrupts and exceptions that meet the specified criteria. The CPU responds to undelegated exceptions and interrupts in M-mode, and updates the machine-mode exception handling registers. The CPU responds to delegated exceptions and interrupts in S-mode, and updates the S-mode exception handling registers.

5.2 Exceptions

5.2.1 Exception handling

In M-mode, the CPU responds to illegal instruction or access error exceptions in the following procedure:

Step 1: Save the exception PC to the mepc register.

Step 2: Set the interrupt flag in the mcause register to 0, write the exception ID to the mcause register, and update the mtval register based on the rules defined in [Table 5.2](#).

Step 3: Save the machine interrupt-enable (MIE) bit in the mstatus register to the MPIE field, clear the MIE field, and prohibit responses to interrupts.

Step 4: Save the privilege mode applied before the exception occurs to the MPP field in the mstatus register, and switch to the M-mode.

Step 5: The PC fetches an instruction from the base address in the mtvec register and executes the instruction. The instruction is usually a jump instruction for jumping to the top-level handler. The handler analyzes the mcause register to obtain the exception ID and calls the handler corresponding to the exception ID.

Table 5.2: Updates to mtval when exceptions occur

Exception vector ID	Exception	mtval update
1	Fetch instruction access error exception	Virtual address accessed by the fetch instruction
2	Illegal instruction exception	Instruction code
3	Debug breakpoint exception	0
4	Load instruction unaligned access exception	Virtual address accessed by the load instruction
5	Load instruction access error exception	0
6	Store/Atomic instruction unaligned access exception	Virtual address accessed by the store/atomic instruction
7	Store/Atomic instruction access error exception	0
8	U-mode environment call exception	0
9	S-mode environment call exception	0
11	M-mode environment call exception	0
12	Fetch instruction page error exception	Virtual address accessed by the fetch instruction
13	Load instruction page access exception	Virtual address accessed by the load instruction
15	Store/Atomic instruction page error exception	Virtual address accessed by the store/atomic instruction

5.2.2 Return from exceptions

You can run the mret instruction to return from an exception. In this case, the CPU performs the following operations:

- Restore the mepc register to the PC. (The mepc register stores the PC applied when the exception occurs. You can adjust the mepc register to skip the exception instruction; otherwise, the exception instruction will be executed again.)
- Restore the value of the MPIE field in the mstatus register to the MIE field in the mstatus register.
- Restore the privilege mode applied before the exception occurs from the MPP field in the mstatus register.

5.3 Interrupts

5.3.1 Interrupt priorities

When receiving multiple interrupt requests, the CPU responds to them by their priorities (in descending order):

- L1 ECC interrupt
- M-mode external interrupt
- M-mode software interrupt
- M-mode timer interrupt
- S-mode external interrupt
- S-mode software interrupt
- S-mode timer interrupt
- PMU overflow interrupt
- L1 ECC interrupt (delegated)
- S-mode external interrupt (delegated)
- S-mode software interrupt (delegated)
- S-mode timer interrupt (delegated)
- PMU overflow interrupt (delegated)

5.3.2 Interrupt responses

In M-mode, the CPU responds to an interrupt in the following procedure:

Step 1: Execute the current instruction and save the PC of the next instruction to the mepc register.

Step 2: Set the interrupt flag in the mcause register to 1, write the interrupt ID to the mcause register, and update the mtval register to 0.

Step 3: Save the machine interrupt-enable (MIE) bit in the mstatus register to the MPIE field, clear the MIE field, and prohibit responses to interrupts.

Step 4: Save the privilege mode applied before the interrupt occurs to the MPP field in the mstatus register, and switch to the M-mode.

Step 5 (The Mode field in the mtvec register is 0, indicating a direct interrupt): The PC fetches an instruction from the base address in the mtvec register and executes the instruction. The instruction is usually a jump instruction for jumping to the top-level handler. The handler analyzes the mcause register to obtain the interrupt ID and calls the handler corresponding to the interrupt ID.

Step 5 (The Mode field in the mtvec register is 1, indicating a vectored interrupt): The PC fetches an instruction from the address calculated in (Base address in the mtvec register + $4 \times$ Interrupt ID)

and executes the instruction. The instruction is usually a jump instruction for jumping to the corresponding interrupt handler.

5.3.3 Return from interrupts

You can run the mret instruction to return from an interrupt. In this case, the CPU performs the following operations:

- Restore the mepc register to the PC. (The mepc register stores the PC of the next instruction and therefore does not need to be adjusted.)
- Restore the value of the MPIE field in the mstatus register to the MIE field in the mstatus register.
- Restore the privilege mode applied before the interrupt occurs from the MPP field in the mstatus register.

5.3.4 Asynchronous errors

In very rare cases, the processor may exhibit behavior known as “asynchronous errors,” which refers to errors caused by an instruction that do not appear at the retirement of that instruction. For example, the CPU executes a load instruction and the bus returns an error. Due to the fast retirement of instructions in the pipeline, the load instruction has already retired by the time the bus produces an error.

It should be noted that the probability of “asynchronous errors” occurring in actual systems is extremely low, and once they occur, it means that the system has encountered a fatal error. In C908, ECC verification errors or bus errors are considered “asynchronous errors” and are reported through interrupts.

6.1 Overview

6.1.1 Memory attributes

C908 supports two memory types: memory and device, which are distinguished by the SO bit. The memory supports speculative execution and out-of-order execution. It is further classified into cacheable memory and non-cacheable memory. The device supports only non-speculative in-order execution and therefore is non-cacheable. It is further classified into bufferable device and non-bufferable device. Bufferable indicates that a response to a write request can be quickly returned on an intermediate node. Non-bufferable indicates that a response to a write request is returned only after the end device completes writing.

To share data among multiple cores, C908 allows you to set the shareable (SH) page attribute. A shareable page is shared among multiple cores, and the hardware maintains data coherence. A non-shareable page is exclusively occupied by a core, and the software, instead of hardware, maintains data coherence among multiple cores.

The SH attribute of the cacheable memory is configurable. The non-cacheable memory and device are shareable by default, and you cannot modify their SH attributes.

In addition, C908 allows you to set the security (SEC) page attribute. If you don't have any special requirements for page security attributes, the default configuration is set to 0.

Table 6.1 describes the page attributes corresponding to each memory type.

Table 6.1: Memory types

Memory type	SO	C	B	SH	SEC
Cacheable memory	0	1	Configurable	Configurable	Reserved
Non-cacheable memory	0	0	Configurable	1	Reserved
Bufferable device	1	0	1	1	Reserved
Non-bufferable device	1	0	0	1	Reserved

The CPU can obtain the page attribute of an address from the `sysmap.h` file or a page table entry (PTE). The two methods are described as follows:

1. Page attributes of addresses are determined by the `sysmap.h` file if virtual addresses are not translated into physical addresses, that is, the machine mode (M-mode) or MMU is disabled.
2. Page attributes of addresses depend on the MAEE field in the `mxstatus` register if virtual addresses are translated into physical addresses, that is, the CPU is not in M-mode and the MMU is enabled. If the MAEE field is enabled, page attributes of addresses are determined by page attributes extended in the corresponding PTEs. If the MAEE field is disabled, page attributes of addresses are determined by the `sysmap.h` file.

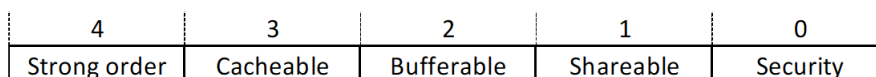
`sysmap.h` is an extended configuration file of C908 that is open to users. You can define page attributes for different address ranges as required.

`sysmap.h` allows you to set page attributes for up to 8 address spaces. The largest address (non-inclusive) of address space i ($i = 0$ to 7) is defined by the `SYSMAP_BASE_ADDRi` ($i = 0$ to 7) macro. The smallest address (inclusive) is defined by the `SYSMAP_BASE_ADDR(i - 1)` macro. That is,

$$\text{SYSMAP_BASE_ADDR}(i - 1) \leq \text{Address of address space } i < \text{SYSMAP_BASE_ADDR}i.$$

The smallest address of address space 0 is `0x0`. Page attributes of memory addresses beyond the eight address spaces defined in the `sysmap.h` file are cacheable/bufferable/shareable/security by default. The upper and lower boundaries of each address space is 4 KB aligned. Therefore, the `SYSMAP_BASE_ADDRi` macro defines the upper 28 bits of an address.

Page attributes of memory addresses within address space i ($i = 0$ to 7) are defined by the `SYSMAP_FLAGi` ($i = 0$ to 7) macro. The attribute layout is shown in :Fig. 6.1 .

Fig. 6.1: Address attributes in the `sysmap.h` file

6.2 SYSMAP configuration reference

- Attribute definition of address space Range 0: `40' h0 <= addr0[39:0] <40' h01000 000, flg0 = 5' b01111`. This address space range includes parts of `INST_RAM` and `DATA_RAM`. Therefore, set the

attribute to cacheable. Attribute definition:

```

`define  SYSMAP_BASE_ADDR0    28'h01000

`define  SYSMAP_FLG0          5'b01111

```

- Attribute definition of address space Range 1: $40' h01000\ 000 \leq \text{addr1}[39:0] < 40' h02000\ 000$, $\text{flg1} = 5' b10000$. This address space range covers the write addresses of special functions (such as the print function) and DATA_RAM. Therefore, set the attribute to so to ensure that operation requests can be sent from the core to the bus. Attribute definition:

```

`define  SYSMAP_BASE_ADDR1    28'h02000

`define  SYSMAP_FLG1          5'b10010

```

- Attribute definition of address space Range 2: $40' h02000\ 000 \leq \text{addr2}[39:0] < 40' h0d000\ 000$, $\text{flg2}=5' b10000$. This address space mainly allocated as APB address space to ensure that operations are strictly executed in order. Attribute definition:

```

`define  SYSMAP_BASE_ADDR2    28'hd0000

`define  SYSMAP_FLG2          5'b10010

```

- Attribute definition of address space Range 3: $40' h0d000\ 000 \leq \text{addr3}[39:0] < 40' heffff\ 000$, $\text{flg3}=5' b01101$. This address space configured as a non-shareable cache space, and only for internal use in SMART SOC. Attribute definition:

```

`define  SYSMAP_BASE_ADDR3    28'heffff

`define  SYSMAP_FLG3          5'b01101

```

- Attribute definition of address space Range 4: $40' heffff\ 000 \leq \text{addr4}[39:0] < 40' hfffff\ 000$, $\text{flg4}=5' b01111$. This address space correspond to an invalid RAM section on the SMART with no practical meaning. Attribute definition:

```

`define  SYSMAP_BASE_ADDR4    28'hfffff

`define  SYSMAP_FLG4          5'b01111

```

- Attribute definition of address space Range 5: $40' h02000\ 000 \leq \text{addr5}[39:0] < 40' h4000000\ 000$, $\text{flg5}=5' b01111$. This address space correspond to an invalid RAM section on the SMART with no practical meaning. Attribute definition:

```

`define  SYSMAP_BASE_ADDR5    28'h4000000

`define  SYSMAP_FLG5          5'b01111

```

- Attribute definition of address space Range 6: $40'h4000000000 \leq \text{addr6}[39:0] < 40'h5000000000$, $\text{flg6}=5'b10000$. This address space correspond to an invalid RAM section on the SMART with no practical meaning. Attribute definition:

```

`define  SYSMAP_BASE_ADDR6    28'h5000000

`define  SYSMAP_FLG6          5'b10010

```

- Attribute definition of address space Range 7: $40'h5000000000 \leq \text{addr7}[39:0] < 40'hffffff000$, $\text{flg7}=5'b01111$. This address space correspond to an invalid RAM section on the SMART with no practical meaning. Attribute definition:

```

`define  SYSMAP_BASE_ADDR7    28'hffffff

`define  SYSMAP_FLG7          5'b01111

```

6.2.1 Memory ordering model

C908MP adopts a weak memory ordering model, which is defined as follows:

- Ordering of access to the same address is maintained among multiple cores, including read after read (RAR), write after write (WAW), write after read (WAR), add read after write (RAW).
- Weak ordering of access to different addresses is allowed among multiple cores, including RAR, WAW, WAR, add RAW.
- Atomic other-multi-copy is ensured. When a core is able to obtain written data of another core, other cores must also be able to obtain the data. However, when a core is able to obtain its own written data, it is not required that other cores be able to obtain the data.

Weak memory ordering causes inconsistency between the actual read/write order among multiple cores and the access order defined by the program. Therefore, C908 provides extended SYNC instructions to enforce memory access ordering in software.

SYNC instructions define the execution order of all instructions, ensuring that all instructions preceding a SYNC instruction are executed before the SYNC instruction. In addition, SYNC instructions can also be used to synchronize instruction memory. After instructions preceding a SYNC instruction are executed, the SYNC instruction clears the pipeline and re-fetches instructions. For more information, see [Table 6.2](#).

Table 6.2: SYNC instructions

Mnemonic	Description	Scope
SYNC.IS	Synchronize data and instruction memory	Shareable
SYNC.I	Synchronize data and instruction memory	Non-shareable
SYNC.S	Synchronize data memory	Shareable
SYNC	Synchronize data memory	Non-shareable

6.3 MMU

6.3.1 Overview

The memory management unit (MMU) of C908 complies with the RISC-V SV39/SV48 standard. It provides the following features:

- **Address translation:** Translates 39-bit virtual addresses to 40-bit physical addresses.
- **Page protection:** Checks the read/write/execution permissions of page visitors.
- **Page attribute management:** Extends address attribute bits and obtains page attributes based on access addresses for further processing by the system.

In C908, SXLEN is fixed as 64 bits, and the MMU performs SV39/SV48 address translation according to 64-bit virtual addresses.

C908 supports U Mode configured as either 32-bit or 64-bit. When UXL is configured as 32 bits:

- The MMU performs SV39/SV48 address translation according to 32-bit virtual addresses.
- The high 32 bits of the virtual address must be 0, or a page fault exception will occur.

6.3.2 TLB

The MMU uses translation lookaside buffers (TLBs) to implement its features. A TLB stores virtual addresses used when the CPU accesses the memory. Before translating a virtual address, the MMU checks the page attributes in the TLB and outputs a physical address corresponding to the virtual address.

The MMU of C908 uses two levels of TLBs: the uTLB at level 1 and the jTLB at level 2. The uTLB includes the I-uTLB and the D-uTLB. After the CPU is reset, the hardware invalidates all entries in the uTLB and the jTLB, without the need of initializing software.

The I-uTLB provides 32 fully associative entries for storing pages in 4 KB, 2 MB, or 1 GB size. When an instruction fetch request hits the I-uTLB, the physical address and the corresponding permission attribute can be obtained in the current cycle.

The D-uTLB provides 17 fully associative entries for storing pages in 4 KB, 2 MB, or 1 GB size. When a load/store request hits the D-uTLB, the physical address and the corresponding permission attribute can be obtained in the current cycle.

The jTLB is a 4-way set-associative cache shared by instructions and data. It provides 1024 entries for storing pages in 4 KB, 2 MB, or 1 GB size. When a request misses the uTLB but hits the jTLB, the physical address and the corresponding permission attribute will be returned within at least three cycles.

6.3.3 Page Table Format

The MMU is used to translate virtual addresses into physical addresses and check corresponding permissions. Specific address mappings and corresponding permissions are configured by the operating system and stored in page tables.

C908 implements address translation through indexing by at most three levels of page tables. The MMU accesses the L1 page table to obtain the base address of an L2 page table and the corresponding permission attributes, accesses the L2 page table to obtain the base address of an L3 page table and the corresponding permission attributes, and accesses the L3 page table to obtain the final physical address and the corresponding permission attributes. The MMU may obtain the final physical address, that is, a leaf table entry, at each level of access. The virtual page number (VPN) consists of 27 bits and is divided into three 9-bit VPN[i]. A part of the VPN is used for indexing in each access.

Content of leaf table entries is cached in the TLB to accelerate address translation. The content includes physical addresses translated from virtual addresses and corresponding permission attributes. If the uTLB is missed, the MMU accesses the jTLB. If the jTLB is missed, the MMU enables a hardware page table walk to access the memory to obtain the final address translation result.

A page table stores entry addresses of next-level page tables or physical information of the final page table. The page table entry (PTE) is shown in Fig. 6.2 and Fig. 6.3 :

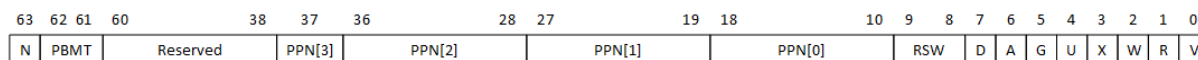


Fig. 6.2: Page table structure(Standard Mode)

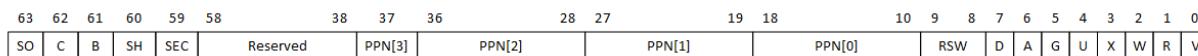


Fig. 6.3: Page table structure(T-Head Mode)

1. Basic Page Attributes

PTE[9:0] are the basic attribute bits of the page table and the functions of each bit are described below.

RSW – Reserved for Software

A bit reserved for software to implement custom page table features. The default value is 2' b00.

D – Dirty

It indicates whether data can be/has been written to the page.

1' b0: indicates that data has not been written/cannot be written to the page.

1' b1: indicates that data has been written/can be written to the page.

When the D bit is 0, a write operation to the page will trigger a page fault (store) exception. You can maintain the meanings of values of the D bit in the exception program through software.

A – Accessed

When the A bit is 1, it indicates that the page is accessible. When the A bit is 0, it indicates that the page is inaccessible. Access to the page will trigger a page fault exception for the corresponding access type.

1' b0: indicates that the page is accessible.

1' b1: indicates that the page is accessible.

G – Global

The global page ID, which indicates whether the page can be shared by multiple processes.

1' b0: indicates that the page is non-shareable and that the ASID is exclusive.

1' b1: indicates that the page is shareable.

U – User

Indicates whether the page is accessible in user mode (U-mode).

1' b0: indicates that the page is inaccessible in U-mode. Access to the page in U-mode will trigger a page fault exception.

1' b1: indicates that the page is accessible in U-mode.

X W R: executable, writable, readable

The meaning of the XWR bit combination is specified as:

Table 6.3: XWR permissions

X	W	R	Meaning
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	Reserved for future use
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	Reserved for future page
1	1	1	Read-write-execute page

V – Valid

Indicates whether the physical page has been mapped to a virtual page. If the V bit of a page is 0, access to the page will cause a page fault exception.

1' b0: indicates that the physical page has not been mapped to a virtual page.

1' b1: indicates that the physical page has been mapped to a virtual page.

2. Physical page number

PTE[37:10] are the physical page number (PPN) of the page table. The physical address of the C908 is fixed at 40 bits, with PPN occupies 28 bits of space, divided into segments every 9 bits. PPN[i] indicates the PPN corresponding to each level of page table.

3. Page extension attributes

The extension attributes of the page table are divided into two modes: standard extension mode and T-Head extension mode.

When mxstatus.mace is 0, the page table organisation follows the standard extension mode. The standard extended mode is compatible with the SVNAPOT and SVPBMT standard extended features, which are implemented by using the N bit of PTE[63] and the PBMT bit of PTE[62:61] respectively. The N bit indicates that the current table entry is a NAPOT size-extended table entry, and the size covered by the table entry is a contiguous area represented by a power-of-two index. The size is shown in the table below.

6.3.4 Address translation process

The address translation process is described as follows:

If the TLB is hit when the CPU attempts to access a virtual address, the CPU directly obtains the physical address and the corresponding attributes from the TLB. If the TLB is missed, the MMU performs the following steps to translate the virtual address:

1. Obtain the access address {satp.PPN, VPN[2], 3' b0} of the L1 page table, and access the D-Cache/memory based on the address to obtain a 64-bit PTE of the L1 page table.
2. Check whether the PTE conforms to the physical memory protection (PMP) permission. If no, generate the corresponding access error exception. If yes, determine whether the X/W/R bit meets the condition of the leaf page table based on the rules shown in `xrw` . If yes, the final physical address has been found. Then go to step 3. If no, obtain the access address {PTE.PPN, next-level VPN, 3' b0} of the next-level page table, and access the D-Cache/memory again.
3. After the leaf page table is found, compare the X/W/R/L bit in the PMP register with the X/W/R bit in the PTE to obtain the minimum permissions, check the permissions, and write the content of the PTE back to jTLB.
4. If permission violation is found in any PMP check, generate the corresponding access error exception based on the access type.
5. Generate a page fault exception in the following three cases: the leaf page table is found but the access type does not conform to the setting of the A/D/X/W/R/U bit, no leaf page table is found after three accesses, or an access error is generated during access to the D-Cache/memory.
6. If the leaf page table is found in less than three accesses, a large page table has been obtained. In this case, check whether the PPN of the large page table is aligned based on the page size. If no, generate a page fault exception.

6.3.5 System control registers

In addition to the standard satp register, the MMU of C908 provides the extended smir, smcir, supervisor-mode (S-mode) entry low (smel), and S-mode entry high (smeh) control registers. You can use the extended registers to directly read, write, probe, and invalidate the TLB.

6.3.5.1 Supervisor address translation and protection register (satp)

The satp register is an MMU control register defined in the SV39/48 standard, with a register width of 64-bit, and it is readable and writable in both M/S modes, but not readable or writable in U mode.

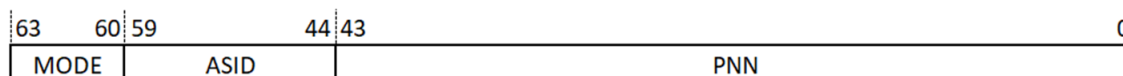


Fig. 6.4: SATP Register Descriptions

Mode: MMU address translation mode

The MODE field indicates the virtual address translation mode, and the standard encoding is shown in [Table 6.4](#) . The C908 implements three modes of SATP MODE field: Bare (4' h0), SV39 (4' h8), and SV48 (4' h9).

When the Linux operating system boots, it determines whether the current system uses the SV39 or SV48 mode. Once determined, there will be no switching between SV39 and SV48, only switching between Bare and SV39/48.

Table 6.4: MMU address translation mode

RV64		
Value	Name	Description
0	Bare	No translation or protection
1-7	-	Reserved
8	Sv39	Page-based 39-bit virtual addressing
9	Sv48	Page-based 48-bit virtual addressing
10	Sv57	Reserved for page-based 57-bit virtual addressing
11	Sv64	Reserved for page-based 64-bit virtual addressing
12-15	-	Reserved

ASID: the current address space identifier (ASID)

Indicates the ASID of the current program. When a jTLB miss occurs during an address translation request and hardware backfilling is required, the jTLB uses the ASID value from the initial miss request, concatenates it with the page table content, and backfills it into the table entry.

PPN: root PPN for hardware writeback

The PPN field defines the root PPN for hardware backfilling, which is used to form the high-order address of the first-level page table initiated by hardware backfilling. For the specific hardware backfilling process, please refer to the *Address Translation Process*.

6.3.5.2 smcir register

The smcir register enables you to probe, read, write, and invalidate the TLB.

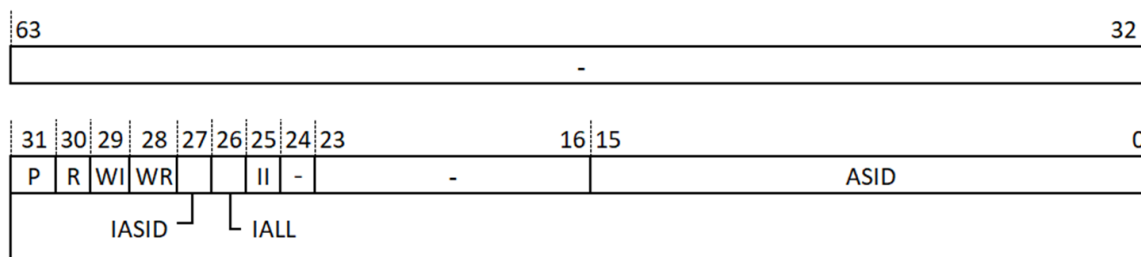


Fig. 6.5: Smcir Register Description

TLBP: TLB probe

Indicates the operation of probing the TLB based on the smeh register.

When the TLB is hit, the value of the smir register is updated to the serial number of the TLB.

TLBR: TLB read

Indicates the operation of reading values of corresponding TLB entries based on indexes in the smir register, and updating the smeh and smel registers based on the values.

TLBWI: TLB indexed write

Indicates the operation of writing values of the smeh and smel registers to corresponding TLB entries based on indexes in the smir register.

TLBWR: TLB random write

Indicates the operation of writing values of the smeh and smel registers to corresponding TLB entries based on indexes in the random register.

TLBIASID: TLB invalidation by ASID

Indicates the operation of invalidating all TLB entries that match the specified ASID.

TLBIALL: TLB initialization

Indicates the operation of invalidating all TLB entries and initializing the TLB.

TLBII: TLB invalidation by index

Indicates the operation of invalidating all TLB entries that match the specified index in the smir register.

TLBIAW: TLB invalidation by world

Indicates the operation of invalidating all TLB entries corresponding to the trustable or non-trustable world.

This field is available only when trusted execution environment (TEE) extension is configured. It has not been implemented in C908.

ASID: the ASID used

Indicates the ASID used for matching in the TLBIASID operation. The smcir register enables you to probe, read, write, and invalidate the TLB.

6.3.5.3 smir register

The smir register is used to index the TLB. In TLB probing, the index of a hit entry is updated. In TLB write indexing, the index field of the smir register is written to write the mapping to the corresponding index in the jTLB.

P – Probe Failure

0: indicates that TLB is hit when the TLBP instruction is executed.

1: indicates that TLB is missed when the TLBP instruction is executed.

Tfatal – Probe multiple

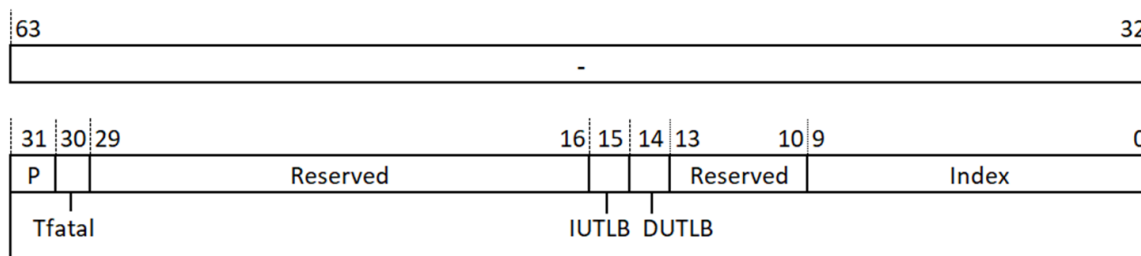


Fig. 6.6: Smir Register Descriptions

Specifies whether multiple matches occur when the TLBP instruction is executed.

0: indicates that no multiple matches occur.

1: indicates that multiple matches occur.

Iutlb - Instruction Micro TLB

Indicates query instructions uTLB.

Dutlb - Data Micro TLB

Indicates query data uTLB.

Index – TLB Index

512-entry configuration: Index [8:7] is the way index, and index [6:0] is the set/entry index (4-way, 128 entries).

1024-entry configuration: Index [9:8] is the way index, and index [7:0] is the set/entry index (4-way, 256 entries).

6.3.5.4 MMU EntryHi register (smeh)

The SMEH register is used to store virtual address information for TLB access. It is updated by hardware during TLB reads and must be pre-written by software before writing TLB entries.

- SMEH provides information about the page to be queried during TLB queries;
- SMEH provides information about the page read during TLB reads;
- SMEH provides information about the page to be written during TLB writes.

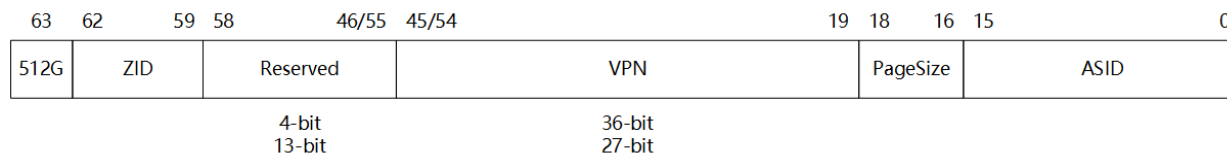


Fig. 6.7: Smeh Register Descriptions

512G:

In SV48, the page size is 512G.

ZID:

When TEE is configured, it represents the ZoneID corresponding to the page. This field is updated by hardware during TLB reads and must be pre-written by software before writing TLB entries.

VPN: the virtual page number

This field is updated by hardware when the TLB is read or a page error exception occurs. Software writes a value to this field before writing values to TLB entries.

Pagesize: the page size

The page size is indicated by using a one-hot, where 100 indicates a size of 4 KB, 010 indicates a size of 2 MB, and 001 indicates a size of 1 GB.

This field is updated by hardware when the TLB is read. Software writes a value to this field before writing values to TLB entries.

ASID:

This field stores the ID of the current address space identified by the operating system. It is used to distinguish between processes.

This field is updated by hardware when the TLB is read. Software writes a value to this field before writing values to TLB entries.

6.3.5.5 MMU EntryLo register(smel)

The smel register stores physical addresses in TLB access and page attributes.

SMEL has a similar format to the page table, and the definitions of related bits can be found in Fig. 6.3. SMEL[57:55] is used to store the PMP R/W/X information of the page from low to high.

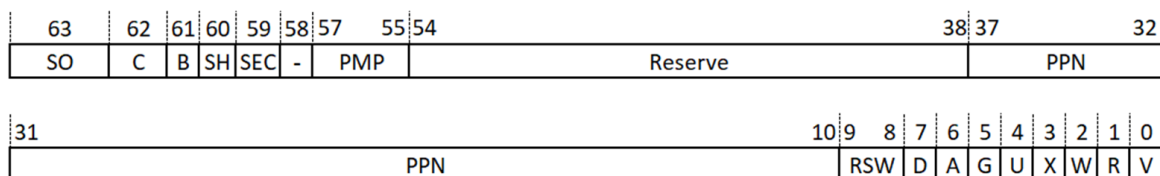


Fig. 6.8: SMEL Register Descriptions

6.4 MMU Parity Checking

The MMU supports configurable parity checking, which can be applied to both TAG and DATA of jTLB. When the parity checking mechanism is enabled, jTLB encodes data with parity during write operations

and checks parity during read operations. When a 1-bit error is detected, the MMU reports the error information and invalidates the cache line containing the error in the jTLB. At the same time, this request is treated as a jTLB miss, and a hardware page table walk is initiated, followed by a fill. Software can query the MCER/SCER registers to obtain relevant error information, such as whether a jTLB parity error has occurred and the location of the error. Control register details can be found in the description of MCER/SCER. Errors of more than 1-bit cannot be detected or corrected. The C908 MMU supports software injection of errors, and the control register details can be found in the description of MEICR.

6.5 PMP

6.5.1 Overview

The PMP unit of C908 complies with the RISC-V standard. The PMP unit checks the access permission on a physical address to determine whether the CPU has the read/write/execution permissions on the address in current mode.

The PMP unit of C908 provides the following features:

- Supports 8/16/32/64 PMP entries, which are identified and indexed by 0 to 63.
- Supports the minimum address split granularity of 4 KB.
- Supports the OFF, top of range (TOR), and naturally aligned power-of-2 region (NAPOT) address matching modes, but not the naturally aligned four-byte region (NA4) mode.
- Supports three permissions: readable, writable, and executable.
- Supports software locks for PMP entries.
- Supports additionally configured with EPMP.

6.5.2 PMP control registers

A PMP entry consists of an 8-bit configuration register and a 64-bit address register. All PMP control registers are accessible in M-mode. Access to PMP control registers in other modes will trigger illegal instruction exceptions.

6.5.2.1 Physical memory protection configuration register (pmpcfg)

The pmpcfg register supports permission configuration for 8 entries.

For more information about the pmpcfg register, see [Table 6.5](#).

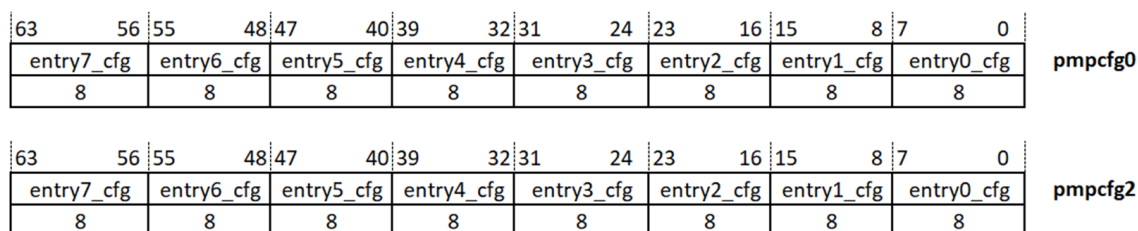


Fig. 6.9: Layout of the pmpcfg register

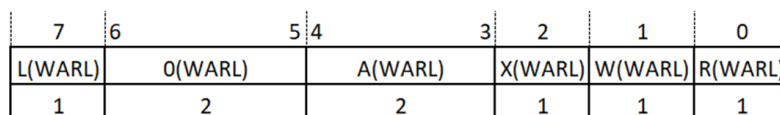


Fig. 6.10: pmpcfg register

Table 6.5: Descriptions of the pmpcfg register

Bit	Name	Description
0	R	The readable attribute of the entry. 0 : indicates that the address matching the entry is non-readable. 1 : indicates that the address matching the entry is readable.
1	W	The writable attribute of the entry. 0 : indicates that the address matching the entry is non-writable. 1 : indicates that the address matching the entry is writable.
2	X	The executable attribute of the entry. 0 : indicates that the address matching the entry is non-executable. 1 : indicates that the address matching the entry is executable.
4:3	A	The address matching mode of the entry. 00 : indicates the OFF mode, in which the entry is invalid. 01 : indicates the TOR mode, in which the address of the adjacent entry is used as the matching range. 10 : indicates the NA4 mode, in which the matching range is 4 bytes. This mode is not supported. 11 : indicates the NAPOT mode, in which the matching range is a power of 2 and is at least 4 KB.
7	L	The lock enable bit of the entry. 0 : indicates that access in M-mode will succeed, and access results in S-mode/U-mode depend on the R/W/X settings. 1 : indicates that the entry is locked and cannot be modified. In TOR mode, the address register of the previous entry cannot be modified either. Access results in all modes depend on the R/W/X settings.

In TOR mode, assuming that the access address is A, the condition for hitting entry i is as follows: $\text{pmpaddr}(i-$

1) $A < \text{pmpaddr}(i)$. The lower boundary of entry 0 is 0.

In NAPOT mode, addresses and corresponding protection region sizes are shown in Table 6.6.

The PMP unit of C908 supports the minimum granularity of 4 KB in NAPOT mode, and does not support the NA4 mode.

Table 6.6: Protection region code

pmpaddr[37:9]	pmpcfg.A	保护区大小	备注
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaa0	NAPOT	4KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aaaa_aaa01	NAPOT	8KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aaaa_a011	NAPOT	16KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aaaa_0111	NAPOT	32KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aaa0_1111	NAPOT	64KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aa01_1111	NAPOT	128KB	Supported
a_aaaa_aaaa_aaaa_aaaa_a011_1111	NAPOT	256KB	Supported
a_aaaa_aaaa_aaaa_aaaa_0111_1111	NAPOT	512KB	Supported
a_aaaa_aaaa_aaaa_aaaa_aaa0_1111_1111	NAPOT	1M	Supported
a_aaaa_aaaa_aaaa_aaaa_aa01_1111_1111	NAPOT	2M	Supported
a_aaaa_aaaa_aaaa_aaaa_a011_1111_1111	NAPOT	4M	Supported
a_aaaa_aaaa_aaaa_aaaa_0111_1111_1111	NAPOT	8M	Supported
a_aaaa_aaaa_aaaa_aaa0_1111_1111_1111	NAPOT	16M	Supported
a_aaaa_aaaa_aaaa_aa01_1111_1111_1111	NAPOT	32M	Supported
a_aaaa_aaaa_aaaa_a011_1111_1111_1111	NAPOT	64M	Supported
a_aaaa_aaaa_aaaa_0111_1111_1111_1111	NAPOT	128M	Supported
a_aaaa_aaaa_aaa0_1111_1111_1111_1111	NAPOT	256M	Supported
a_aaaa_aaaa_aa01_1111_1111_1111_1111	NAPOT	512M	Supported
a_aaaa_aaaa_a011_1111_1111_1111_1111	NAPOT	1G	Supported
a_aaaa_aaaa_0111_1111_1111_1111_1111	NAPOT	2G	Supported
a_aaaa_aaa0_1111_1111_1111_1111_1111	NAPOT	4G	Supported
a_aaaa_aa01_1111_1111_1111_1111_1111	NAPOT	8G	Supported
a_aaaa_a011_1111_1111_1111_1111_1111	NAPOT	16G	Supported
a_aaaa_0111_1111_1111_1111_1111_1111	NAPOT	32G	Supported
a_aaa0_1111_1111_1111_1111_1111_1111	NAPOT	64G	Supported
a_aa01_1111_1111_1111_1111_1111_1111	NAPOT	128G	Supported
a_a011_1111_1111_1111_1111_1111_1111	NAPOT	256G	Supported
a_0111_1111_1111_1111_1111_1111_1111	NAPOT	512G	Supported
0_1111_1111_1111_1111_1111_1111_1111	NAPOT	1T	Supported
1_1111_1111_1111_1111_1111_1111_1111	Reserved	-	-

6.5.2.2 Physical memory protection address register (pmpaddr)

The PMP unit provides 8/16/32/64 pmpaddr (pmpaddr0 ~ pmpaddr7/15/31/63) for storing physical addresses of entries.

As defined in the RISC-V standard, pmpaddr registers store bit [39:2] of physical addresses. The PMP unit of C908 supports the minimum granularity of 4 KB. Therefore, bit [8:0] is not used for address authentication logic.

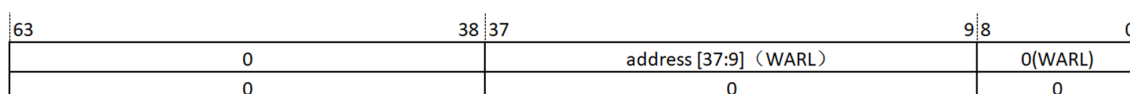


Fig. 6.11: pmpaddr registers

6.6 Memory access order

The following summarizes the processes of accessing an address space by C908 in different scenarios.

Scenario 1: without VA-PA translation

- CPU access PA:
- Obtain the address attribute from the sysmap.h file.
- Perform PMP checks to determine whether the XWR permissions conform to the PMP settings.
- Access the address.

Scenario 2: with VA-PA translation

- CPU access VA:
- Translate the address by using the MMU to obtain the corresponding PTE.
- Obtain the following information from the PTE: the PA, address attribute (Note 1), and XWR permissions.
- Perform PMP checks to determine whether the XWR permissions conform to the PMP settings. (The minimum XWR permissions defined in the PMP register and PTE prevail.)
- Access the address.

Note: When the MAEE field is 1, the address attribute comes from the PTE.

When the MAEE field is 0, the address attribute comes from the sysmap.h file.

7.1 Memory Subsystem Overview

Each core of C908 has its own I-Cache and D-Cache. Eight cores share one L2 cache. Data coherence among multiple cores is maintained by hardware.

7.2 L1 I-Cache

7.2.1 Overview

The L1 I-Cache provides the following features:

- I Cache size is configurable through hardware: 16KB/32KB/64KB
- With a cache line size of 64 bytes, 4-way set-associative;
- Virtually indexed, physically tagged (VIPT);
- Data width for read access: 64 bits; Data width for write access: 128 bits
- First-in, first-out (FIFO);
- Adopts a pseudo-random replacement policy.
- Supports invalidation of the entire I-Cache and individual cache line.
- Supports instruction prefetching.

- Supports way prediction.
- Supports parity checking mechanism.
- A request for an instruction cache miss will snoop the data cache (controlled by a switch bit).

7.2.2 Instruction prefetching

The L1 I-Cache supports instruction prefetching through configuration of the implicit operation register MHINT.IPLD.

- When there is a cache miss in the current cache line access, the prefetching of the next continuous cache line is enabled, and the prefetching result is cached in a prefetch buffer.
- When the instruction access hits the prefetch buffer, the instruction is directly obtained from the buffer and filled back into the instruction cache, thereby reducing the fetch delay.

Instruction prefetching requires that the prefetched cache line and the currently accessed cache line are in the same page, otherwise the instruction prefetching function will be disabled to ensure the security of the fetch address. In addition, the read-sensitive peripheral address space is also prohibited from being allocated to the instruction area.

7.2.3 Way prediction

The C908 I-Cache adopts the 4-way set-associative structure. To reduce power consumption in parallel access to two caches, C908 implements I-Cache way prediction. When way prediction information is valid, access to invalid data ways is disabled, and the CPU accesses data only in the predicted way. You can configure the IWPE field in the mhint register to enable I-Cache way prediction.

Way prediction can be classified into the following two types by instruction fetch behavior:

- **Sequential access:** When the CPU consecutively fetches instructions in a line, the CPU predicts way information of the current access based on the way hit information of the last access.
- **Jump access:** A branch instruction obtains way prediction information of the target cache line along with the jump target address, and accesses one of the caches based on the information.

7.2.4 Loop acceleration buffer

C908 provides a 32-byte loop acceleration buffer to cope with a large number of short loops in programs. When detecting a short-loop instruction sequence, the CPU loads it to the loop acceleration buffer. When a subsequent instruction fetch request hits the buffer, the CPU directly obtains the instruction and jump target address from the buffer and disables access to the I-Cache, branch history table, and branch and jump target predictor, reducing dynamic power consumption of instruction fetch. You can configure the LPE field in the mhint register to enable short-loop acceleration.

7.2.5 Branch history table

C908 uses the branch history table to predict jump directions of conditional branch instructions. The branch history table is 64 KB in size. The bi-mode branch predictor predicts one branch result per cycle. The branch history table consists of predictors and selectors. The predictors are classified into jump and non-jump predictors and are maintained in real time based on branch history information. The branch history table indexes ways based on branch history information and the address of the current branch instruction to predict the jump direction of the branch instruction.

The branch history table predicts jump directions of the following conditional branch instructions:

BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, and C.BNEZ

7.2.6 Branch and jump target predictor

The C908 uses the branch and jump target predictor to predict jump target addresses of branch instructions. The branch and jump target predictor records the historical target addresses of branch instructions. If the current branch instruction hits the branch and jump target predictor, the recorded target address is used as the predicted target address of the current branch instruction.

The branch and jump target predictor provides the following features:

- Supports 1024/2048 entries.
- Adopts the 2-way set-associative structure and supports selection and replacement based on the PC in the lower bits of a branch instruction.
- Maintains I-Cache way prediction information.
- Supports indexing by using a part of the PC of the current branch instruction.

The branch and jump target predictor predicts jump target addresses of the following branch instructions:

- BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, and C.BNEZ
- JAL and C.J

7.2.7 Indirect branch predictor

C908 uses the indirect branch predictor to predict target addresses of indirect branch instructions. Indirect branch instructions obtain target addresses from registers. One indirect branch instruction can contain multiple branch target addresses, which cannot be predicted by using the conventional branch and jump target predictor. Therefore, C908 uses the branch history-based indirect branch predictor to associate historical target addresses of an indirect branch instruction with its branch history information, and discretize different target addresses of one indirect branch instruction based on different branch history information. This makes it possible to predict multiple target addresses.

Indirect branch instructions include:

- JALR: except when the source register is x1 or x5
- C.JALR: except when the source register is x5
- C.JR: except when the source register is x1 or x5

7.2.8 Return address predictor

The return address predictor is used to quickly and accurately predict a return address when a function call ends. When the instruction fetch unit (IFU) obtains a valid function call instruction through decoding, it pushes a function return address to the return address predictor. When the IFU obtains a valid function return instruction through decoding, it pulls a function return address from the return address predictor. The return address predictor supports up to 12 nested function calls. If more than 12 function calls are nested, a target address prediction error will occur.

- Function call instructions include JAL, JALR, and C.JALR.
- Function return instructions include JALR, C.JR, and C.JALR.

For more information, see [Table 7.1](#).

Table 7.1: Instruction features

rd	rs1	rs1=rd	RAS action
!link	!link	-	none
!link	link	-	pop
link	!link	-	push
link	link	0	push and pop
link	link	1	push

7.2.9 Fast jump target predictor

To improve efficiency of the IFU in consecutive jumps, C908 provides a fast jump target predictor at level 1 of the IFU. When the IFU jumps consecutively, the fast jump target predictor records the address of the second jump instruction and the jump target address. If an instruction fetch request hits the fast jump target predictor, the IFU starts to jump at level 1, reducing performance loss of at least one cycle.

The fast jump target predictor predicts jump target addresses of the following branch instructions:

- BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ, and C.BNEZ
- JAL and C.J
- Function return instructions

7.2.10 Parity Check Function

The instruction cache supports a configurable parity check mechanism. The parity check mechanism checks the tag array of the instruction cache with a granularity of 29 bits and checks the data array with a granularity of 34 bits. After the parity check mechanism is enabled, the instruction cache performs odd-even encoding on the data when writing and checks the data when reading. When a 1-bit data error occurs, it can be detected, the current error data is invalidated, a new instruction fetch request is sent to the bus, and the cache is filled back. At the same time, error information is reported, including way information, index information, etc., which can be queried in the MCER/SCER register. The specific control register description can refer to the description of MCER/SCER in Appendix C-1 of the Machine Mode Processor Control and Status Extension Register.

Errors of more than 1 bit cannot be detected or corrected.

The C908 instruction cache supports software injection of errors, and the specific control register description can refer to the description of MEICR in Appendix C-1 of the Machine Mode Processor Control and Status Extension Register.

7.3 L1 D-Cache

7.3.1 Overview

The L1 D-Cache provides the following features:

- D-Cache size is configurable through hardware: 16KB/32KB/64KB
- With a cache line size of 64 bytes, 4-way set-associative;
- Virtually indexed, physically tagged (VIPT);
- Maximum data width per read access: 128 bits, supporting byte, halfword, word, doubleword, and quadword access;
- Maximum data width per write access: 256 bits, supporting access to any combinations of bytes;
- Write policies: write-back with write-allocate, and write-back with write-no-allocate;
- First-in, first-out (FIFO);
- Invalidation and clearing by D-Cache or cache line supported;
- Multi-channel data prefetch for instructions.
- Supports ECC and parity checking mechanism.

7.3.2 Cache coherence

For requests with shareable and cacheable page attributes, data coherence between L1 D-Caches of different cores is maintained by hardware.

For requests with non-shareable and cacheable page attributes, the CPU does not maintain data coherence between L1 D-Caches. If non-shareable and cacheable pages need to be shared across cores, data coherence must be maintained by software.

C908MP maintains data coherence between L1 D-Caches of different cores based on the MESI protocol. MESI indicates four states of each cache line in the D-Cache:

- M: indicates that the cache line is available only in this D-Cache and has been modified (UniqueDirty).
- E: indicates that the cache line is available only in this D-Cache and has not been modified (UniqueClean).
- S: indicates that the cache line may be available in multiple D-Caches and has not been modified (ShareClean).
- I: indicates that the cache line is not available in this D-Cache (Invalid).

7.3.3 Exclusive access

C908 supports exclusive memory access instructions: LR and SC. You can use the two instructions to constitute a synchronization primitive such as an atomic lock to synchronize data between different processes of a core or between different cores. The LR instruction tags the address to be exclusively accessed. The SC instruction determines whether the tagged address is preempted by other processes. C908 provides a local monitor in the L1 D-Cache and a global monitor in the L2 cache for each core. Each monitor consists of a state machine and an address buffer. The state machine has two states: IDLE and EXCLUSIVE.

Exclusive access to a cacheable page can be implemented with the local monitor. When the LR instruction is executed, it sets the state machine of the local monitor to the EXCLUSIVE state and stores the address to be accessed and the size to the buffer. When the SC instruction is executed, it reads the state of the local monitor, the address, and the size. If the state is EXCLUSIVE and the address exactly matches the size, the write operation is executed, a write success is returned, and the state machine is reset to the IDLE state. If the state or the address/size matching does not meet the requirement or the D-Cache is disabled, the write operation is not executed, a write failure is returned, and the state machine is reset to the IDLE state. When the write operation of another core performs matching against the local monitor at the same cache line address, the state machine is also reset to the IDLE state. The write operation in the current core or exclusive access to a different address does not affect the local monitor. In addition, the local monitor must be cleared when a process is switched.

Exclusive access to a non-cacheable page is implemented with both the local monitor and the global monitor. When the LR instruction is executed, it must set both the local monitor and the global monitor. After the local monitor passes the check, the SC instruction further checks the global monitor. If the global monitor

passes the check, the write operation is executed, a write success is returned, and the state of the state machine is cleared; otherwise, the write operation is not executed, a write failure is returned, and the state of the state machine is cleared. When the write operation of another core performs matching against a global monitor at an address, the state machine of the global monitor is reset to the IDLE state.

In C908-based systems, we recommend that you use the LR and SC instructions to implement atomic locks. If the address attribute of an atomic lock is cacheable (either shareable or non-shareable), no special design is required for the SoC system. This is a typical case. If the address attribute of an atomic lock is non-cacheable, device, or strongly ordered, the system (for example, the slave client) must be integrated with an exclusive monitor. If an operation is performed in other ways, the response will be UNPREDICTABLE.

7.4 L2 Cache

7.4.1 Overview

The L2 cache provides the following features:

- Cache size: 1 MB, with a cache line size of 64 bytes, 16-way set-associative;
- Strictly inclusive of the L1 D-Cache, and non-strictly inclusive of the L1 I-Cache;
- Physically indexed, physically tagged (PIPT);
- Maximum data width per access: 64 bytes;
- Write policies: write-back with write-allocate, and write-back with write-no-allocate;
- First-in, first-out (FIFO);
- Programmable RAM latency;
- Instruction prefetch and TLB prefetch supported;
- Block-based pipelining.

7.4.2 Cache coherence

The L2 cache of C908MP maintains data coherence between D-Caches of different cores based on the MOESI protocol. MOESI indicates five states of each cache line in the D-Cache:

- M: indicates that the cache line is available only in this D-Cache and has been modified (UniqueDirty).
- O: indicates that the cache line may be available in multiple D-Caches and has been modified (ShareDirty).
- E: indicates that the cache line is available only in this D-Cache and has not been modified (Unique-Clean).

- S: indicates that the cache line may be available in multiple D-Caches and has not been modified (ShareClean).
- I: indicates that the cache line is not available in this D-Cache (Invalid)

7.4.3 Structure

The L2 cache of C908MP is built on a block-based pipelining architecture. Access addresses are discretized in two different blocks to allow parallel access and improve access efficiency.

The block mechanism is shown in Fig. 7.1 .

- The tag RAM is divided into two tag sub-blocks by PA[6]: tag bank 0 and tag bank 1, to handle two access requests in parallel within one clock cycle.
- Similarly, the data RAM is divided into two data sub-blocks by PA[6]: data bank 0 and data bank 1. Each data sub-block is further divided into four 128-bit micro blocks, to obtain one cache line in parallel.

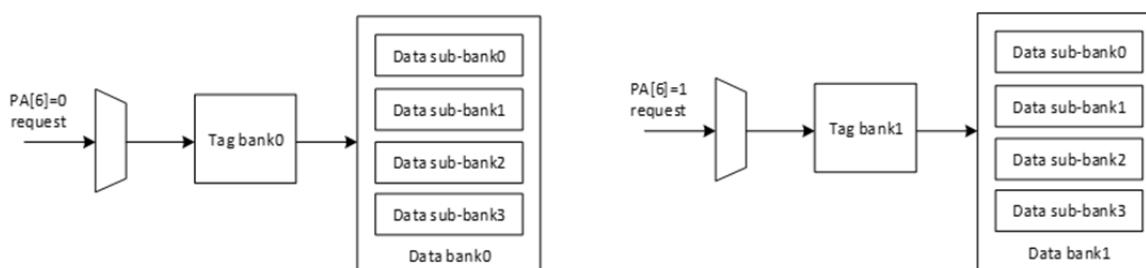


Fig. 7.1: L2 Cache structure

7.4.4 RAM latency

The L2 cache has a long access latency because it is large in size. It usually takes multiple clock cycles to complete access to the L2 cache. C908MP enables you to configure the access latency. You can set the setup time and latency of RAM in different processes. Detailed configurations are shown in Table 7.2 .

Table 7.2: RAM latency configurations

Item	Feature	Description
L2 TAG setup	L2 Cache Tag RAM setup: 1b0 : 0 cycles. Default value 1b1 : 1 cycle.	L2 Cache Tag The RAM setup affects only tags. The RAM access.
L2 TAG latency	L2 Cache Tag RAM latency: 3b000 : 1 cycle. Default value 3b001 : 2 cycles. 3b010 : 3 cycles. 3b011 : 4 cycles. 3b1xx : 5 cycles.	
L2 DATA setup	L2 Cache Data RAM setup: 1b0 : 0 cycles. Default value 1b1 : 1 cycle.	L2 Cache Data The RAM setup affects only data. The RAM access.
L2 DATA latency	L2 Data RAM latency: 3b000 : 1 cycle. Default value 3b001 : 2 cycles. 3b010 : 3 cycles. 3b011 : 4 cycles. 3b100 : 5 cycles. 3b101 : 6 cycles. 3b110 : 7 cycles. 3b111 : 8 cycles.	

You can set the latency based on the time required for accessing the RAM. The default value of setup is 0. When the RAM setup time or winding length is long, you can modify setup to 1.

The number of access cycles with the preceding configurations is shown in [Table 7.3](#).

Table 7.3: Valid access latency of the tag RAM

Tag latency	Valid access latency of the tag RAM	
	TAG setup = 0	TAG setup = 1
/		
000	1	2
001	2	3
010	3	4
011	4	5
1xx	5	5

Table 7.4: Valid access latency of the data RAM

Tag latency	Valid access latency of the data RAM	
	TAG setup = 0	TAG setup = 1
/		
000	1	2
001	2	3
010	3	4
011	4	5
100	5	6
101	6	7
110	7	8
111	8	8

- The maximum valid L2 tag latency is 5 cycles.
- When tag setup is 1, one more cycle is required for access. Before the SRAM is accessed, the SRAM input signal will be flopped.
- The maximum valid L2 data latency is 8 cycles.
- When data setup is 1, one more cycle is required for access. Before the SRAM is accessed, the SRAM input signal will be flopped.

7.5 Accelerated memory access

This section describes the accelerated memory access features of C908 L1 and L2 caches.

7.5.1 Instruction prefetch of the L1 I-Cache

The L1 I-Cache supports instruction prefetch. You can configure the IPLD field in the mhint register to enable this feature. When an instruction access request misses the current cache line, the next consecutive cache line is prefetched and stored to the prefetch buffer. When the instruction access request hits the

prefetch buffer, the instruction is directly obtained from the prefetch buffer and written back to the I-Cache, reducing the instruction fetch latency.

This feature requires that the prefetched cache line and the current accessed cache line be on the same page, to ensure security of the instruction fetch address. In addition, you cannot allocate read-sensitive device address spaces to instruction spaces.

7.5.2 Multi-channel data prefetch of the L1 D-Cache

C908 supports data prefetch to reduce the access latency of large-sized memory such as DDR SDRAMs. C908 detects D-Cache misses to determine a fixed access mode through matching. Then the hardware automatically prefetches cache lines and writes them back to the L1 D-Cache.

C908 supports data prefetch through up to 8 channels and supports two prefetch methods: consecutive prefetch and strided prefetch (stride ≤ 32 cache lines).

C908 also implements forward prefetch and backward prefetch (the stride is negative) to support various possible access modes.

Data prefetch is disabled when the CPU invalidates or clears the D-Cache.

You can configure the DPLD field in the mhint register to enable data prefetch and the DPLD_DIS field to determine the number of cache lines to be prefetched at a time.

The following instructions support data prefetch:

- LB, LBU, LH, LHU, LW, LWU, and LD
- FLW and FLD
- LRB, LRH, LRW, LRD, LRB, LRHU, LRWU, LURB, LURH, LURW, LURD, LURBU, LURHU, LURWU, LBI, LHI, LWI, LDI, LBUI, LHUI, LWUI, LDD, LWD, and LWUD

7.5.3 L1 adaptive write allocation mechanism

C908 implements adaptive write allocation at L1. When the CPU detects consecutive memory write operations, the write allocation attribute of pages is automatically disabled.

You can configure the AMR field in the mhint register to enable L1 adaptive write allocation.

When the CPU invalidates or clears the D-Cache, adaptive write allocation is automatically disabled. After the invalidation or clearing is completed, the CPU detects consecutive memory write operations again.

The following instructions support adaptive write allocation:

- SB, SH, SW, and SD
- FSW and FSD
- SRB, SRH, SRW, SRD, SURB, SURH, SURW, SURD, SBI, SHI, SWI, SDI, SDD, and SWD

7.5.4 L2 prefetch mechanism

The L2 cache supports instruction prefetch and TLB prefetch. It supports the following prefetch features:

- The number of instructions prefetched at a time is software-configurable and can be 0, 1, 2, or 3. All prefetched instructions are written back to the L2 cache.
- Only one entry is prefetched from the TLB at a time.
- The prefetch range is a 4 KB page table, and addresses beyond the range will not be prefetched.
- You can use the machine-mode (M-mode) L2-cache control register (mccr2) to configure the prefetch mechanism.

7.6 L1/L2 cache operation instructions and registers

After the CPU is reset, the I-Cache and D-Cache are automatically invalidated and disabled by default.

Similarly, after the CPU is reset, the L2 cache is automatically invalidated. After the invalidation is completed, the L2 cache is automatically enabled and cannot be disabled. When the L1 cache is disabled, no data is written back to the L2 cache if the L2 cache is missed.

7.6.1 Extended registers of the L1 cache

Extended registers of the C908 L1 cache are classified into the following types by feature:

- Cache enable and mode configuration: The M-mode hardware configuration register (mhcr) allows you to enable/disable the I-Cache/D-Cache and configure the write allocation and writeback modes. The supervisor-mode (S-mode) hardware configuration register (shcr) is a read-only register mapped to the mhcr register.
- Dirty page table entry clearing and invalidation: The M-mode cache operation register (mcor) allows you to clear and invalidate dirty page table entries in the I-Cache and the D-Cache.
- Cache read: The machine-mode cache access instruction register (mcins), M-mode cache access index register (mcindex), and M-mode cache access data register 0/1 (mcddata0/1) allow you to read data from the I-Cache and the D-Cache.

For more information, see *M-mode CPU control and status extension registers* and *M-mode cache access extension registers*.

7.6.2 Extended registers of the L2 cache

Extended registers of the C908 L2 cache are classified into the following types by feature:

- L2 cache enable and latency configuration: The mccr2 register allows you to set the access latency of the L2 cache.
- L2 cache read: The mcins, mcindex, and mcdata0/1 registers allow you to read data from the L2 cache.

For more information, see *M-mode CPU control and status extension registers* and *M-mode cache access extension registers*.

7.6.3 L1/L2 cache operation instructions

C908 provides extended L1/L2 cache operation instructions that invalidate page table entries by address, invalidate all page table entries, clear dirty page table entries by address, clear all dirty page table entries, clear and invalidate dirty page table entries by address, and clear and invalidate all dirty page table entries. For more information, see [Table 7.5](#) .

Table 7.5: L1/L2 cache operation instructions

Instructions	Description
ICACHE.IALL	Invalidates all page table entries in the I-Cache.
ICACHE.IALLS	Invalidates all page table entries in the I-Cache through broadcasting.
ICACHE.IPA	Invalidates page table entries that match the specified physical addresses in the I-Cache.
ICACHE.IVA	Invalidates page table entries that match the specified virtual addresses in the I-Cache.
DCACHE.CALL	Clears all dirty page table entries in the D-Cache.
DCACHE.CIALL	Clears and invalidates all dirty page table entries in the D-Cache.
DCACHE.CIPA	Clears dirty page table entries that match the specified physical addresses in the D-Cache and invalidates the entries.
DCACHE.CISW	Clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the entries.
DCACHE.CIVA	Clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the entries.
DCACHE.CPA	Clears dirty page table entries that match the specified physical addresses in the D-Cache.
DCACHE.CPAL1	Clears dirty page table entries that match the specified physical addresses in the L1 D-Cache.
DCACHE.CVA	Clears dirty page table entries that match the specified virtual addresses in the D-Cache.
DCACHE.CSW	Clears dirty page table entries in the D-Cache based on the specified way and set.
DCACHE.CVAL1	Clears dirty page table entries that match the specified virtual addresses in the L1 D-Cache.
DCACHE.IPA	Invalidates page table entries that match the specified physical addresses in the D-Cache.
DCACHE.ISW	Invalidates page table entries in the D-Cache based on the specified way and set.
DCACHE.IVA	Invalidates page table entries that match the specified virtual addresses in the D-Cache.
DCACHE.IALL	Invalidates all page table entries in the D-Cache.
L2CACHE.CALL	Clears all dirty page table entries in the L2 cache.
L2CACHE.CIALL	Clears all dirty page table entries in the L2 cache and invalidates the entries.
L2CACHE.IALL	Invalidates all dirty page table entries in the L2 cache.

For more information, see *Appendix B-1 Cache instructions*.

C908 is compatible with RISC-V Vector Extension, Version 1.0-rc1-20210608 .

8.1 Vector programming model

C908 supports the following vector extension features:

- 32 independent vector registers from v0 to v31. Vector registers are 128 or 256 bits (VLEN=128/256), which depends on the vector computing capability option.
- Vector floating-point instructions support the FP16 and FP32 elements (SEW=16/32).
- Vector integer instructions support the INT8, INT16, INT32, and INT64 elements (SEW=8/16/32/64).
- Vector register groups are supported to improve the efficiency of vector computations. Four types of vector register groups are supported: 32, 16, 8, or 4 vector groups can be created, each of which contains 1, 2, 4, or 8 vector registers, respectively.

8.2 Vector control registers

Seven non-privileged control and status registers (CSRs) are added for C908:

- vstart

The vstart register specifies the position of the first element when a vector instruction is executed. After a vector instruction is executed, vstart is reset to zero. In most cases, software does not need to modify

vstart. In C908, only vector load/store instructions support non-zero vstart registers. All computational vector instructions support only vstart=0. Otherwise, instruction exceptions occur.

- vxsat

The vxsat register is valid only when the bit is set to 0. This register indicates whether the result of a fixed-point instruction is overflow.

- vxrm

The vxrm register provides four rounding modes: Round up, round to even, round towards zero, and round to odd.

- vcsr

Vector control core status register.

- vl

The vl register specifies the range of elements in the target register to be updated by a vector instruction. A vector instruction updates elements whose numbers are smaller than vl in the target register and resets elements whose numbers are greater than vl to zero. When vstart \geq vl or vl=0, no element in the target register is updated.

- vtype

The vtype register defines basic data properties for vector computations, including: Invalid flag bits, element bits, and vector register groups. The vtype register also includes the EDIV bit. C908 does not support EDIV. Therefore, the EDIV bit is set to 0.

- vlenb (Vector Spec 0.8)

The vector bits of C908 are measured in bytes.

Therefore, C908 supports vector status maintenance (Vector Spec 0.8). The VS bit is defined in mstatus[10:9] to decide whether to save the vector-related register during context switching.

8.3 Vector exceptions

Vector instructions are classified into the following categories:

- Vector load
- Vector computation
- Vector store

Vector computation does not trigger exceptions. Vector store does not trigger exceptions because the bus ignores BRESP faults. Therefore, only vector load triggers exceptions. When an exception is triggered by vector load, the CPU discards the data that it reads and resets vstart to 0. The mepc points to the instruction. When an inexact exception occurs, mepc may point to subsequent instructions.

The CPU handles vector instruction interrupts in the same way as regular instructions. The CPU completes the current instruction and the mepc points to the next instruction. The remaining steps are the same as those in handling regular interrupts.

9.1 Security Requirements

This chapter describes software and hardware security design to meet trusted execution environment (TEE) requirements. System security requirements include:

- Support independent zones.
- Support zone isolation by code execution, memory access, external device, or I/O.
- Support isolation between applications and isolation between applications and the kernel within a zone.
- Support the multi-core SMP architecture.
- Support shared memory access among zones.
- Support the RISC-V 32-bit and 64-bit architectures.
- Supported trustworthy communication among zones.
- Support TEEs that comply with the GP specification.

9.2 Processor Security Model

The RISC-V ISA architecture supports the following privileged modes: M-mode, S-mode, and U-mode. These modes provide different execution and access permissions:

- In U-mode, only non-privileged instructions can be executed. In most cases, user applications are run in this mode.
- In S-mode, root user instructions can be executed and MMU management permissions are granted. In most cases, complex operating systems such as Linux are run in this mode.
- The M-mode provides the most execution and access privilege, including interrupt/exception handling and management, PMP management, and privileged access control management.

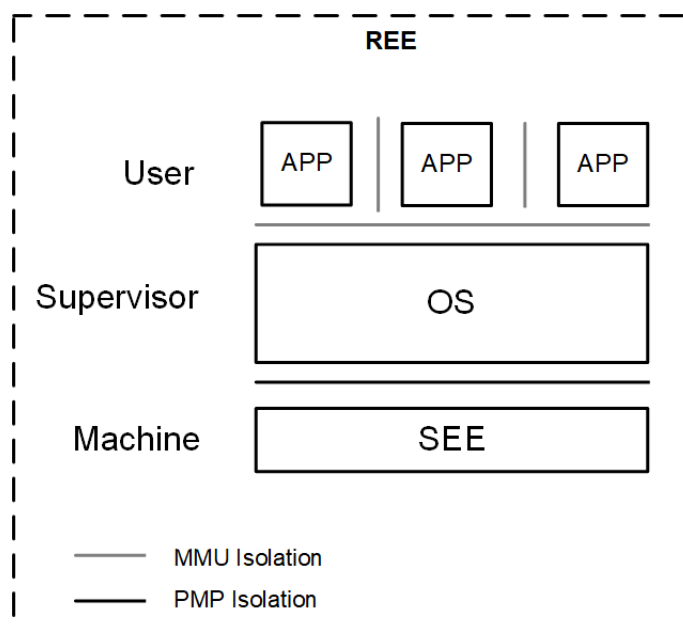


Fig. 9.1: RISC-V privileged modes

The S-mode and U-mode of RISC-V have no difference than other mainstream processor architectures, such as the supervisor and user modes of ARM. In U-mode, only non-privileged instructions can be executed. Applications running in U-mode must transition to S-mode through self-trap and access system resources under the management of the operating system. S-mode supports non-privileged instructions and privileged instructions, and provides permissions to access CSRs in S-mode. In addition, S-mode provides permissions to access MMUs. Memory protection and isolation in user mode and kernel mode are implemented through virtual memory management. The M-mode provides the most execution and access privilege. The RISC-V architecture adds privileged instructions that can be executed only in M-mode and system registers can be accessed only in M-mode, such as PMP. The most important feature of the M-mode is exception interception and handling. During exception handling, the processor transitions all exceptions to the M-mode through self-trap by default. The M-mode exception handler then forwards interrupts to the S-mode. The M-mode is usually used to run trusted firmware to adjust, allocate, and manage software and hardware resources.

To meet the isolation requirements for TEEs, security extensions are added to the Xuantie C series processors based on the RISC-V architecture. These processors can create multiple virtual zones based on software coordination. Fig. 9.2 shows the architecture. An operating system runs independently in each zone and applications run in the operating system. The operating system runs in S-mode and applications run in

U-mode. The processor can run in different zones. When the processor runs in a zone, it occupies the entire physical core in real time. The zone ID of the processor is changed to the ID of the current zone. Zone switching is completed by the trusted firmware in M-mode.

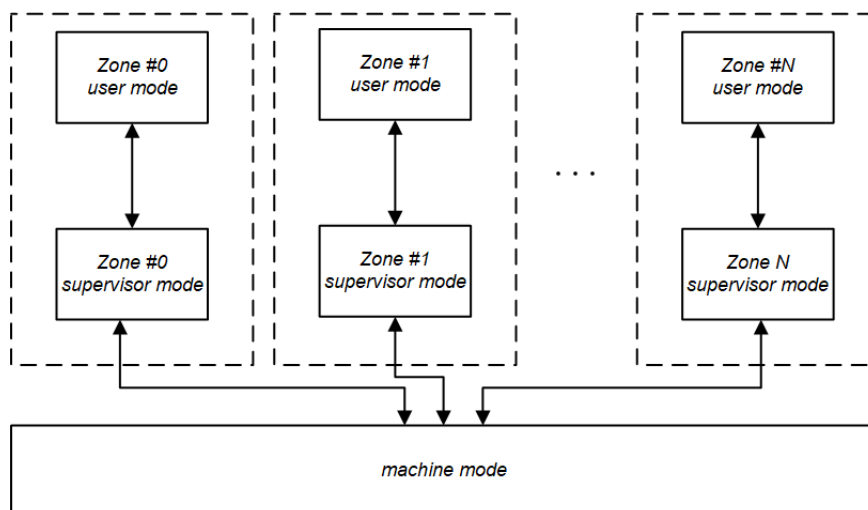


Fig. 9.2: Zones and privileged modes in Xuantie C series processors

9.3 System Security Architecture

9.3.1 Secure memory management

Each hardware thread can run in different zones through time-sharing. When a hardware thread runs in a zone, memory access is isolated in the zone. Other zones are not allowed to access the memory resources in the zone without authorization. In addition, the zone is not allowed to access memory resources in other zones without authorization. Different zones can exchange data through shared memory.

Physical Memory Protection (PMP)

The RISC-V architecture provides the PMP mechanism to isolate memory access in M-mode from memory access in S-mode and U-mode. PMP is configurable only in M-mode. PMP consists of multiple groups (8 to 16 groups in most cases) of address registers and configuration registers. The configuration registers can grant or revoke read, write, and execute permissions in S-mode and U-mode. PMP can also protect memory mapping I/O (MMIO). The M-mode trusted firmware can use PMP to limit the processor from accessing external device I/O.

When a hardware thread switches from one zone to another zone, the PMP configuration is also switched. The M-mode trusted firmware needs to save the PMP configuration in the current zone and loads the PMP configuration in the target zone to update the access permissions on memory and MMIO.

When multiple zones share memory, you can grant permissions to access the shared memory block to these zones by writing the access permissions to the PMP configuration table of each zone. The trusted firmware

will update the PMP table during zone switching. Fig. 9.3 shows the PMP configurations of multiple zones. The SHM area is a memory block shared by the zones.

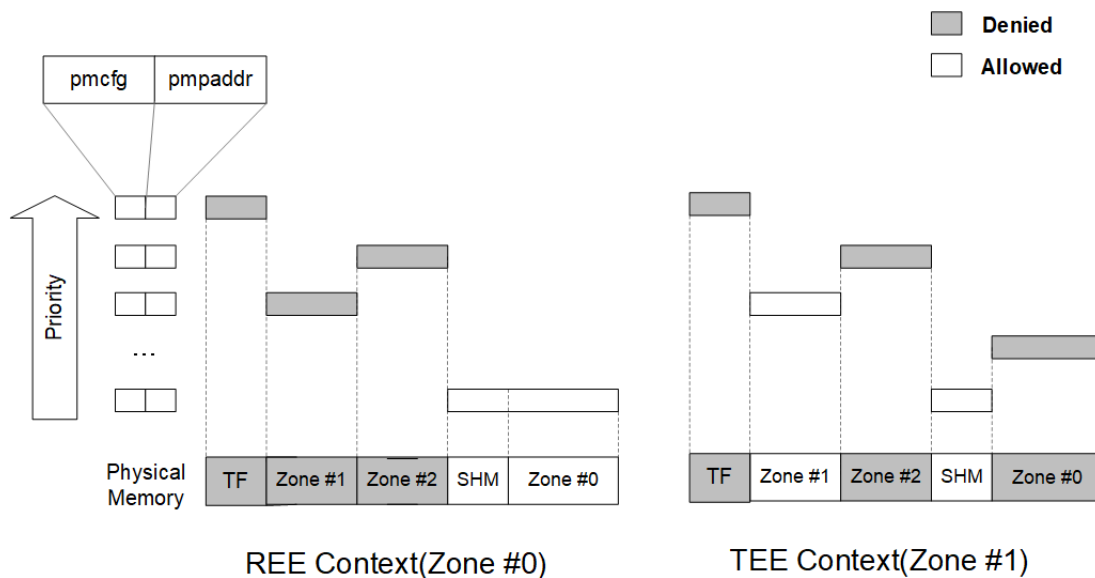


Fig. 9.3: PMP Configurations of Multiple Zones

I/O Physical Memory Protection (IOPMP)

The RISC-V architecture provides a PMP mechanism to protect memory and MMIO access of RISC-V processors in different privileged modes.

Other master devices connected to the bus also require memory access protection: IOPMP. Same as PMP, IOPMP allows you to define access permissions. IOPMP checks whether the reads and writes sent through the bus meet the permission rules. Only legitimate reads and writes are transmitted to the target device. Typically, two methods are used to connect to an IOPMP:

1. Connect the requester to an IOPMP

Add an IOPMP between the bus and each master device, which is similar to PMP of RISC-V. Add IOPMPs for different master devices. These IOPMP are independent of each other. The design is simple but more flexible. However, the IOPMPs cannot be shared among master devices. As shown in Fig. 9.4 :

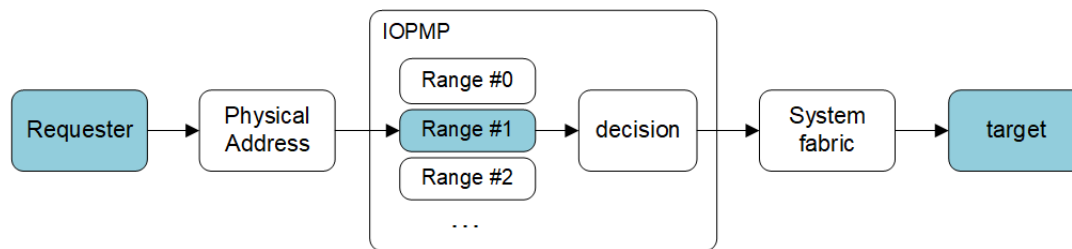


Fig. 9.4: Connect the requester to an IOPMP

2. Connect the destination device to an IOPMP

The IOPMP of the destination device needs to distinguish requests from different master devices. To do this, the requests sent by master devices must carry a master ID. As shown in Fig. 9.5:

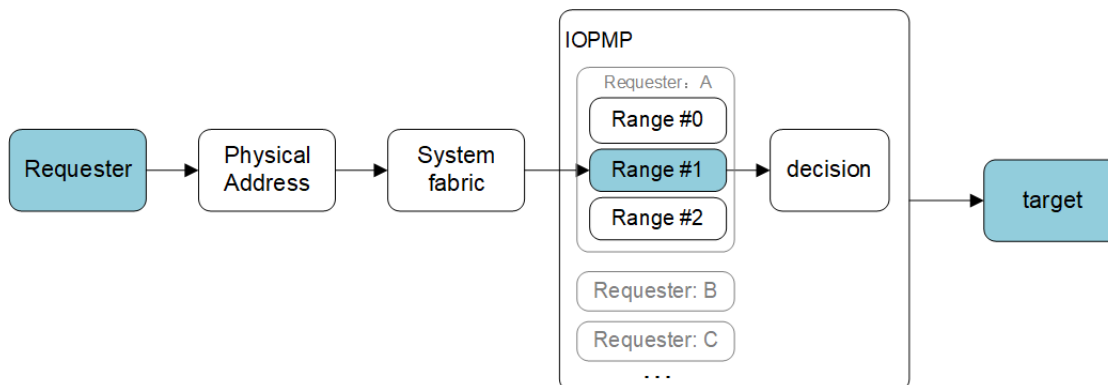


Fig. 9.5: Connect the destination device to an IOPMP

In Fig. 9.6 , the Xuantie processor mounts requesters to IOPMPs to build a secure SoC system framework.

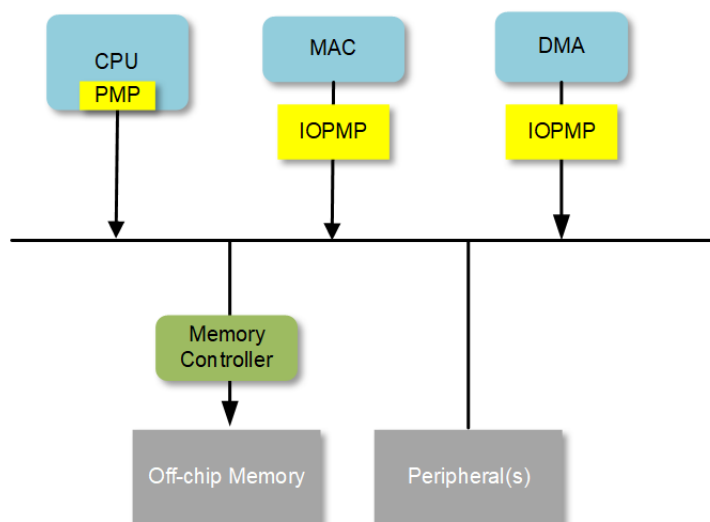


Fig. 9.6: SoC System Framework Based on PMP and IOPMP Isolation.

Memory Management Units (MMUs)

MMUs are used to manage virtual memory in traditional operating systems. MMUs can be used to separate the user space and kernel space. The MMUs of a Xuantie processor integrates a configurable number of TLBs. Each TLB stores translations of virtual addresses to physical addresses and access permissions.

Different zones have separate TLBs to ensure that addresses in different zones are translated separately. Zone switching clears the corresponding TLB (sfence).

Cache

When a processor runs in different zones, each zone has separate PMP configurations. PMP limits the zones from accessing physical memory MMIO and ensures that memory and I/O access among different zones are

not interfered.

In a Xuantie C series RISC-V processor, memory access that hits the cache is also protected by PMP. This means that all access to the cache must be verified by PMP. The access can reach the cache only after it passes the check. Multi-core cache coherence is also protected by PMP.

Device Coherence Port (DCP)

Xuantie C908 provides a DCP. DCP is an AXI slave interface of a processor. External master devices can access internal data with cache coherence through the DCP. This improves the efficiency of data exchange between the processor and external master device. Xuantie C908 does not add protection to the DCP for access from external master devices. To protect external master devices, you need to mount the external master devices that are connected to the DCP to IOPMPs for protection.

9.3.2 Secure interrupts

In the PLIC specification of RISC-V, there are two modes of interrupt sources: M-mode interrupt sources and S-mode interrupt sources. M-mode interrupt sources are handled only by the M-mode. S-mode interrupt sources can be handled by the M-mode or S-mode. The M-mode has permissions to determine whether to send interrupts to the S-mode for handling. The M-mode of the RISC-V architecture provides interrupt interception to help isolate interrupts of different zones. Table 9.1 describes how interrupts of different modes are handled.

Table 9.1: RISC-V interrupt handling model

Target mode of the interrupt source	Processor current mode	Delegation	Whether the interrupt is handled	Mode that handles the interrupt
M-mode	M-mode	Invalid	Yes	M-mode
	S-mode	Invalid	Yes	M-mode
	U-mode	Invalid	Yes	M-mode
S-mode S-mode	M-mode	0	Yes	M-mode
		1	No	
	M-mode	0	Yes	M-mode
		1	Yes	S-mode
	U-mode	0	Yes	M-mode
		1	Yes	S-mode

Interrupts are handled in the following ways based on the interrupt interception feature of the M-mode of RISC-V:

1. M-mode interrupt distribution
2. Interrupt groups

M-mode interrupt distribution

The M-mode supports external interrupt interception. All external interrupts need to transition to the M-mode through self-trap. The M-mode trusted firmware will centrally manage all external interrupts, identify interrupt sources, and forward interrupts to different zones to handle these interrupts. This method meets the requirements for isolating interrupts among different zones. However, interrupts are forwarded by the trusted firmware. The trusted firmware needs to switching the context of the zone during interrupt forwarding, which delays interrupt handling.

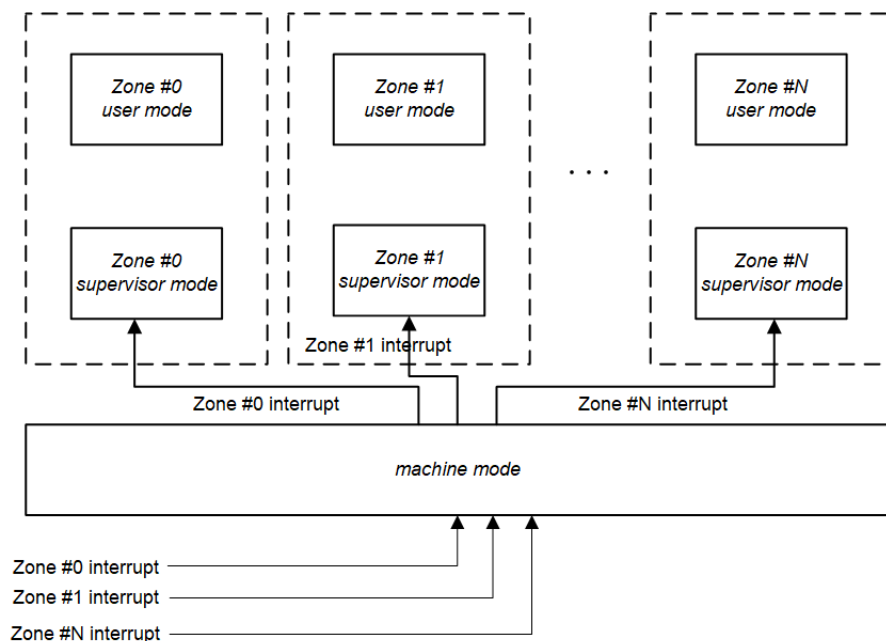


Fig. 9.7: M-mode Interrupt Distribution in Xuantie C series processors

When this method is used, all external interrupts are sent to the M-mode trusted firmware. The trusted firmware first saves the fields of the current zone and reads the number of the external interrupt. Then, the trusted firmware selects a destination zone from the zone interrupt allocation table and obtains the entry of the interrupt handler of the destination zone. The trusted firmware can obtain the address of the interrupt entry by querying the stvec register. Before forwarding the interrupt to the entry, the trusted firmware needs to switch to the PMP configuration of the destination zone, checks the legitimacy of the address of the interrupt handler, and finally executes the mret instruction to switch to the destination zone. After the interrupt is handled, the interrupt handler needs to transition to the M-mode through an ecall. The M-mode trusted firmware will restore the fields of the interrupted zone and switch back to the zone.

Interrupt groups

Interrupt forwarding through the M-mode severely delays interrupt handling. In addition, after an interrupt handler handles the interrupt, it needs to transition to the M-mode through an ecall. This results in incompatibility with the existing interrupt handlers (especially for Linux).

The PLIC provides separate control over each interrupt source and target. This means that the hardware thread to which the interrupts of an interrupt source are forwarded and the mode of the hardware thread can be separately configured. Currently, the execution environments of processors are classified into Rich

Execution Environments (REEs) and Trusted Execution Environments (TEEs). Secure interrupts are handled in TEEs. Most hardware interrupts are regular interrupts. Only a few number of hardware interrupts, such as secure timers, are secure interrupts. To reduce the interrupt handling delay posed by the M-mode, this solution creates interrupt groups. Interrupts from interrupt sources in the current zone are handled in the zone. Interrupts from interrupt sources that do not belong to the current zone are handled in M-mode. Interrupt context scenarios:

- The REE generates regular interrupts.
- The REE generates secure interrupts.
- The TEE generates regular interrupts.
- The TEE generates secure interrupts.

The REE generates regular interrupts or the REE generates secure interrupts

When the processor runs in the REE (Zone #0), the trusted firmware needs to perform the following operations:

1. Enables the S-mode for the interrupt source of regular interrupts.
2. Enables the M-mode for the interrupt source of secure interrupts.
3. Resets the first bit (SSIE_DELEG), fifth bit (STIE_DELEG), and ninth bit (SEIE_DELEG) of the mideleg register. Assume that software interrupts and clock interrupts are configured as regular interrupts.
4. Enables mstatus.MIE and mstatus.SIE, and enables mie.MEIE, mie.MSIE, mie.MTIE, mie.SEIE, mie.SSIE, and mie.STIE.

The TEE generates regular interrupts or the TEE generates secure interrupts

When the processor runs in the TEE (Zone #1), the trusted firmware needs to perform the following operations:

1. Enables the S-mode for the interrupt source of regular interrupts.
2. Enables the M-mode for the interrupt source of secure interrupts.
3. Resets the first bit (SSIE_DELEG), fifth bit (STIE_DELEG), and ninth bit (SEIE_DELEG) of the mideleg register. Assume that software interrupts and clock interrupts are configured as regular interrupts.
4. Enables mstatus.MIE and mstatus.SIE, and enables mie.MEIE, mie.MSIE, mie.MTIE, mie.SEIE, mie.SSIE, and mie.STIE.

9.3.3 Secure Access Control

The M-mode is the most privileged mode that a hardware thread can run in RISC-V. A hardware thread running in M-mode has full access permissions on memory, I/O, and underlying features that are required for

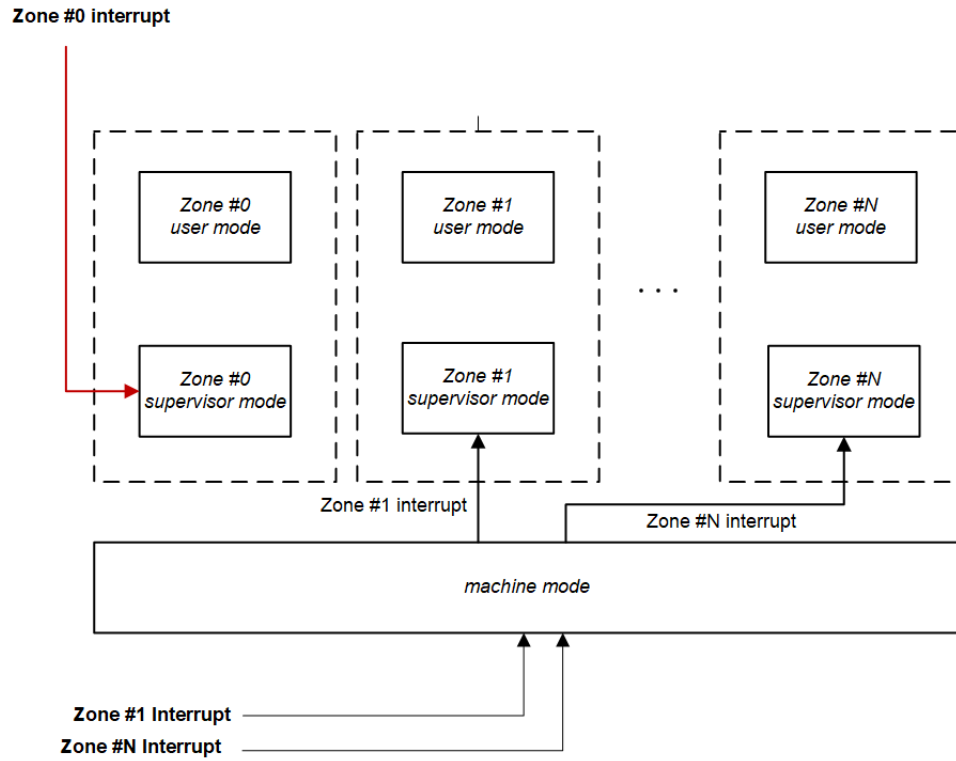


Fig. 9.8: Interrupt Handling Rules When the Processor Is Running in Zone #0

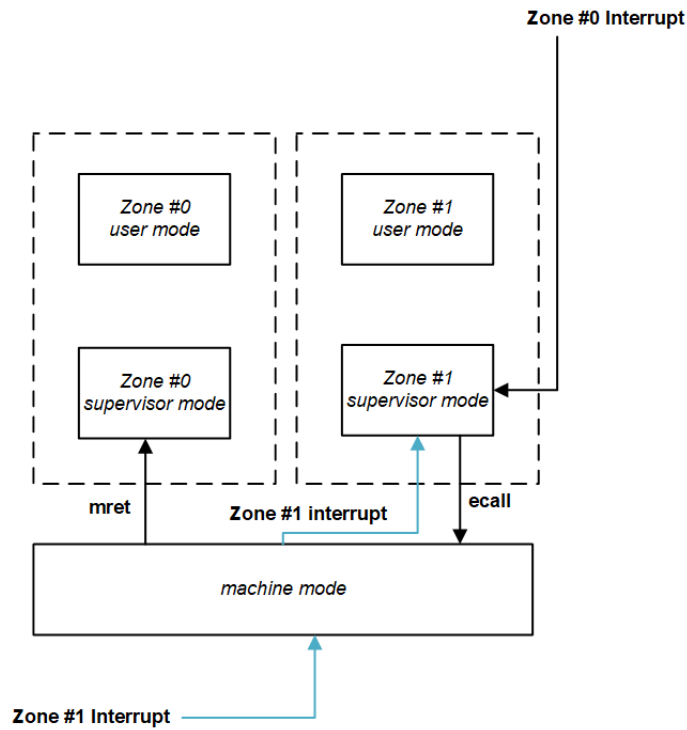


Fig. 9.9: Interrupt Handling Rules When the Processor Is Running in Zone #1

booting and configuring the operating system. The M-mode is a privileged mode that must be implemented by all standard RISC-V processors. Simple RISC-V microcontrollers support only the M-mode.

The most important feature of the M-mode is exception interception and handling. By default, when an exception occurs (regardless of the privileged mode), the control permissions are transferred to the exception handler in M-mode. However, most exceptions in Linux are handled in S-mode. The exception handler in M-mode can redirect exceptions to the S-mode but these operations will severely delay exception handling. To address this issue, RISC-V provides the exception delegation mechanism. This mechanism can selectively transfer interrupts and exceptions to the S-mode for handling and bypass the M-mode. The Machine Interrupt Delegation (mideleg) CSR controls the interrupts or exceptions that are transferred to the S-mode.

Take note that control permissions are not transferred to a mode with less privilege when an interrupt or exception occurs, regardless of the delegation settings. Interrupts and exceptions in M-mode are handled only in M-mode. Interrupts and exceptions in S-mode are handled in M-mode or S-mode depending on the delegation settings. These interrupts and exceptions are never handled in U-mode.

The M-mode is sufficient for simple embedded systems. However, it is applicable only if the entire code repository is trusted because the M-mode provides full access to the hardware platform. In most cases, not all application code can be trusted because it is difficult to verify the security of every application. Therefore, RISC-V provides this mechanism to protect systems against untrusted code and isolate untrusted processes. The untrusted code must be limited to access only the authorized memory block. Processors that support the M-mode and S-mode/U-mode support PMP, which allows the M-mode to specify the memory addresses that the S-mode/U-mode can access. PMP can also limit MMIO access. With the help of PMP, the M-mode can limit untrusted users or super users from accessing the memory and external devices.

9.3.4 Secure Debugging

Currently, C908 does not support individual zone debugging. Only global zone debugging can be enabled or disabled.

10.1 Core local interrupt (CLINT) controller

C908 implements the CLINT controller. It is a memory address mapping module that handles software and timer interrupts.

10.1.1 CLINT register address mapping

The CLINT controller occupies a 64 KB memory space. Addresses in the upper 13 bits depend on the SoC hardware integration. Address mapping in the lower 27 bits is shown in [Table 10.1](#). All registers support only access to word-aligned addresses. The CLINT controller adopts a continuous addressing scheme. For a multi-cluster multi-core architecture, the number of clusters does not affect the addresses of CLINT registers, while the number of cores affects the CLINT register addresses. The address spaces corresponding to the cores are continuous. For example, two clusters are configured. Cluster 0 contains two cores, and Cluster 1 contains four cores. The register addresses of the two cores of Cluster 0 are specified by Cores 0 and 1, and the register addresses of the four cores of Cluster 1 are specified by Cores 2, 3, 4, and 5. For more information, see the following table. A CLINT controller supports up to 256 cores.

Table 10.1: Memory-mapped addresses in CLINT registers

Address	Name	Type	Initial value	Description
0x4000000	MSIP0	Read/Write	0x00000000	The machine-mode (M-mode) software interrupt pending register for core 0. The upper bits are tied to 0, and bit [0] is valid.
0x4000004	MSIP1	Read/Write	0x00000000	The M-mode software interrupt pending register for core 1. The upper bits are tied to 0, and bit [0] is valid.
Reserved	-	-	-	-
0x4000000 +4*n	MSIPn	Read/Write	0x00000000	n=hart_id, n<256
0x4004000	MTIMECMPL0	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 0.
0x4004004	MTIMECMPH0	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 0.
0x4004008	MTIMECMPL1	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the lower 32 bits) for core 0.
0x400400C	MTIMECMPH1	Read/Write	0xFFFFFFFF	The M-mode clock timer compare value register (the upper 32 bits) for core 0.
Reserved	-	-	-	-
0x4004000 +8*n	MTIMECMPLn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
0x4004004 +8*n+4	MTIMECMPHn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
0x400C000	SSIP0	Read/Write	0x00000000	The supervisor-mode (S-mode) software interrupt pending register for core 0. The upper bits are tied to 0, and bit [0] is valid.
0x400C004	SSIP1	Read/Write	0x00000000	The S-mode software interrupt pending register for core 1. The upper bits are tied to 0, and bit [0] is valid.

Continued on next page

Table 10.1 – continued from previous page

Address	Name	Type	Initial value	Description
Reserved	-	-	-	-
0x400C000 +4*n	SSIPn	Read/Write	0x00000000	n=hart_id, n<256
0x400D000	STIMECMPL0	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 0.
0x400D004	STIMECMPH0	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 0.
0x400D008	STIMECMPL1	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the lower 32 bits) for core 0.
0x400D00C	STIMECMPH1	Read/Write	0xFFFFFFFF	The S-mode clock timer compare value register (the upper 32 bits) for core 0.
Reserved	-	-	-	-
0x400D000 +8*n	STIMECMPLn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
0x400D004 +8*n+4	STIMECMPHn	Read/Write	0xFFFFFFFF	n=hart_id, n<256
0x400FFF8	CLINT_S TIMEL	Read/Write	0x00000000	The S-mode clock timer(T-Head Extended Register)
0x400FFFC	CLINT_S TIMEH	Read/Write	0x00000000	The S-mode clock timer(T-Head Extended Register)

10.1.2 Software interrupts

The CLINT controller can generate software interrupts.

Software interrupts are controlled by the software interrupt pending registers configured with address mappings. M-mode software interrupts are controlled by the machine software interrupt pending (MSIP) register. S-mode software interrupts are controlled by the supervisor software interrupt pending (SSIP) register.

You can set the xSIP bit to 1 to generate software interrupts or reset it to 0 clear software interrupts. CLINT S-mode software interrupt requests are valid only when the CLINTEE bit is enabled for the corresponding core.

In M-mode, the CPU is allowed to access and modify all software interrupt registers. In S-mode, the CPU is allowed to access and modify only the SSIP register. In user mode (U-mode), the CPU has no access to software interrupt registers.

The two groups of registers have the same structure. The bit layout and definition of the registers are shown in Fig. 10.1 and Fig. 10.2 .

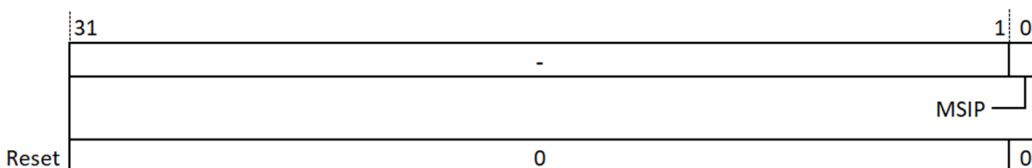


Fig. 10.1: MSIP register

MSIP: the machine software interrupt pending bit

This bit indicates the status of M-mode software interrupts.

- When the MSIP bit is 1, valid M-mode software interrupt requests are available.
- When the MSIP bit is 0, no valid M-mode software interrupt requests are available.

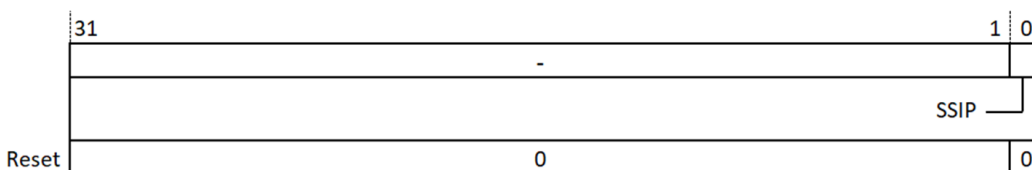


Fig. 10.2: SSIP register

SSIP: the supervisor software interrupt pending bit

This bit indicates the status of S-mode software interrupts.

- When the SSIP bit is 1, valid S-mode software interrupt requests are available.
- When the SSIP bit is 0, no valid S-mode software interrupt requests are available.

10.1.3 CLINT timer

A multi-cluster multi-core system has only one 64-bit system timer that runs in the always-on voltage domain. The system timer does not allow write operations but can be reset. You can obtain the current value of the system timer by reading the values of CLINT_MTIME and CLINT_STIME or by reading the value of the TIME register of the PMU. The system timer is used to provide a unified time reference for multiple cores.

A multi-cluster multi-core system has only one group of 64-bit M-mode timer registers (CLINT_MTIME and CLINT_MTIMEH) and one group of 64-bit S-mode timer registers (CLINT_STIME and CLINT_STIMEH). You can modify the upper or lower 32 bits of the registers to separately read the high-order 32 bits and low-order 32 bits by word-aligned address read.

CLINT_MTIME and CLINT_STIME are self-extended registers of T-Head. The bit layout and definition of the registers are shown in Fig. 10.3 and Fig. 10.4 .

CLINT_MTIMEH and CLINT_MTIMEL are M-mode timer registers for high-order bits and low-order bits separately. The registers store the values of the corresponding timers.

- CLINT_MTIMEH stores the high-order 32 bits of the corresponding timer.
- CLINT_MTIMEL stores the low-order 32 bits of the corresponding timer.

CLINT_STIMEH and CLINT_STIMEL are S-mode timer registers for high-order bits and low-order bits separately. The registers store the values of the corresponding timers.

- CLINT_STIMEH stores the high-order 32 bits of the corresponding timer.
- CLINT_STIMEL stores the low-order 32 bits of the corresponding timer.

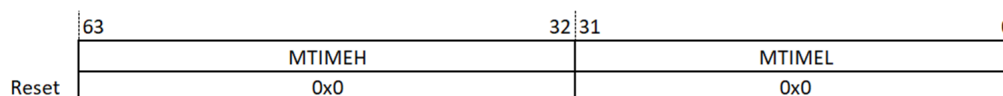


Fig. 10.3: CLINT_MTIME register

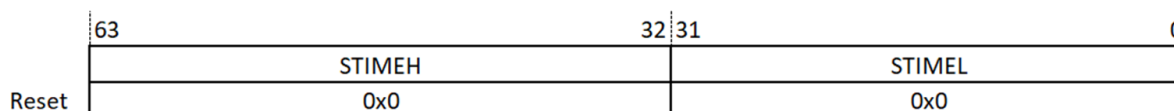


Fig. 10.4: CLINT_STIME register

10.1.4 Timer interrupts

The CLINT controller can generate timer interrupts.

A multi-core system has only one 64-bit system timer, `mtime`. `mtime` must run in the always-on voltage domain. `mtime` cannot be written but can be reset. The current value of `mtime` can be read from the time register of the performance monitoring unit (PMU). `mtime` is used to provide a unified time reference for multiple cores.

Each core has a group of 64-bit M-mode clock timer compare value registers (`mtimecmp1` and `mtimecmpH`) and a group of 64-bit S-mode clock timer compare value registers (`stimecmp1` and `stimecmpH`). You can modify the upper or lower 32 bits of these registers through word-aligned address access.

The CLINT controller compares the value of $\{\text{CMPH}[31:0], \text{CMPL}[31:0]\}$ with the current value of `mtime` to determine whether to generate a timer interrupt. When the value of $\{\text{CMPH}[31:0], \text{CMPL}[31:0]\}$ is greater than the current value of `mtime`, the CLINT controller does not generate an interrupt. When the value of $\{\text{CMPH}[31:0], \text{CMPL}[31:0]\}$ is less than or equal to the current value of `mtime`, the CLINT controller generates a corresponding timer interrupt. You can rewrite the value of the `mtimecmp`/`stimecmp` register to clear the corresponding timer interrupt. S-mode timer interrupt requests are valid only when the CLINTEE bit is enabled for the corresponding core.

In M-mode, the CPU is allowed to access and modify all timer interrupt registers. In S-mode, the CPU is allowed to access and modify only the stimecmpl and stimecmph registers. In U-mode, the CPU has no access to timer interrupt registers.

The two groups of registers have the same structure. The bit layout and definition of the registers are shown in Fig. 10.5 .

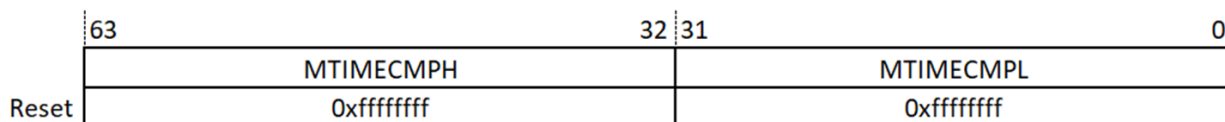


Fig. 10.5: mtimecmph/mtimecmpl registers

mtimecmph/mtimecmpl: the M-mode clock timer compare value registers for the upper bits and the lower bits

These registers store timer compare values.

- mtimecmph: stores the upper 32 bits of timer compare values.
- mtimecmpl: stores the lower 32 bits of timer compare values.

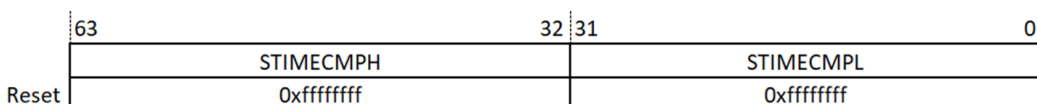


Fig. 10.6: stimecmph/stimecmpl registers

stimecmph/stimecmpl: the S-mode clock timer compare value registers for the upper bits and the lower bits

These registers store timer compare values.

- stimecmph: stores the upper 32 bits of timer compare values.
- stimecmpl: stores the lower 32 bits of timer compare values.

10.2 Platform-level interrupt controller (PLIC)

The PLIC controls sampling, priority arbitration, and distribution of external interrupt sources.

In the PLIC model, the M-mode and S-mode of each core can act as valid interrupt targets.

The PLIC of C908 provides the following features:

- PLIC supports up to 256 cores, each with 2 interrupt targets: machine mode and supervisor mode.
- Sampling of up to 1023 interrupt sources, supporting level and pulse interrupts;
- 32 interrupt priorities;

- Independent enable for each interrupt target;
- Independent interrupt threshold for each interrupt target;
- Configurable access permissions on PLIC registers.

10.2.1 Interrupt arbitration

In the PLIC, only interrupt sources that meet the specified conditions are involved in arbitration on an interrupt target. The conditions include:

- The interrupt source is in the pending state ($IP = 1$).
- The interrupt priority is greater than 0.
- The enable bit for the interrupt target is enabled.

When multiple interrupts for an interrupt target are in the pending state, the PLIC selects the interrupt with the highest priority through arbitration. In the PLIC of C908, M-mode interrupts have higher priorities than S-mode interrupts. In the same privilege mode, a larger value of the priority configuration register indicates a higher priority. Interrupts with a priority of 0 are invalid. If multiple interrupts have the same priority, they will be handled in ascending order of IDs.

The PLIC stores interrupt IDs that are determined based on arbitration results to the interrupt claim/complete register of the corresponding interrupt target.

10.2.2 Interrupt request and response

When the PLIC has a valid interrupt request for an interrupt target and the interrupt priority is higher than the interrupt threshold of the interrupt target, the PLIC sends the interrupt request to the interrupt target. When receiving the interrupt request, the interrupt target sends an interrupt response message to the PLIC if it is able to respond to the interrupt request.

The interrupt response mechanism functions as follows:

- The interrupt target initiates a read operation to the corresponding interrupt claim/complete register. The read operation returns the interrupt ID determined by the PLIC. The interrupt target proceeds to further processing based on the interrupt ID. If the interrupt ID is 0, no valid interrupt request is available, and the interrupt target ends the interrupt handling process.
- After receiving the read operation initiated by the interrupt target and returning the interrupt ID, the PLIC resets the IP bit of the interrupt source corresponding to the interrupt ID, and blocks subsequent sampling on the interrupt source before the current interrupt is completed.

When the L2 ECC feature is configured, the L2 ECC FATAL interrupt number is determined by the customer's decision on the integration of the interrupt controller.

10.2.3 Interrupt completion

After interrupt handling is completed, the interrupt target sends an interrupt completion message to the PLIC. The interrupt completion mechanism functions as follows:

- The interrupt target initiates a write operation to the corresponding interrupt claim/complete register, to write the ID of the completed interrupt to the register. If the interrupt is a level interrupt, the external interrupt source must be cleared before the write operation is initiated.
- After receiving the interrupt completion message, the PLIC does not update the interrupt claim/complete register, but unblocks sampling on the interrupt source corresponding to the interrupt ID to end the interrupt handling process.

10.2.4 PLIC register address mapping

The PLIC occupies a 64 MB memory space. Addresses in the upper 13 bits depend on the SoC hardware integration. Address mapping in the lower 27 bits is shown in Table 10.2.

All registers support only word-aligned address access. That is, you need to use the load word instruction to access PLIC registers, and the result is stored in the low-order 32 bits of the 64-bit GPR. The PLIC adopts a continuous addressing scheme. For a multi-cluster multi-core architecture, the number of clusters does not affect the addresses of PLIC registers, while the number of cores affects the PLIC register addresses. The address spaces corresponding to the cores are continuous. For example, two clusters are configured. Cluster 0 contains two cores, and Cluster 1 contains four cores. The register addresses of the two cores of Cluster 0 are specified by Cores 0 and 1, and the register addresses of the four cores of Cluster 1 are specified by Cores 2, 3, 4, and 5. For more information, see the following table.

Note: Registers not supported by C908 are marked as reserved.

Table 10.2: PLIC register address mapping

Address	Name	Type	Initial value	Description
0x0000000	-	-	-	-
0x0000004	PLIC_PRIO1	R/W	0x0	The priority configuration register for interrupts 1 to 1023.
0x0000008	PLIC_PRIO2	R/W	0x0	
0x000000C	PLIC_PRIO3	R/W	0x0	
...	
0x0000FFC	PLIC_PRIO1023	R/W	0x0	
0x0001000	PLIC_IP0	R/W	0x0	The interrupt pending register for interrupts 1 to 31.

Continued on next page

Table 10.2 – continued from previous page

Address	Name	Type	Initial value	Description
0x0001004	PLIC_IP1	R/W	0x0	The interrupt pending register for interrupts 32 to 63.
...
0x000107C	PLIC_IP31	R/W	0x0	The interrupt pending register for interrupts 992 to 1023.
Reserved	-	-	-	-
0x0002000	PLIC_H0_MIE0	R/W	0x0	The M-mode interrupt enable register for interrupts 1 to 31 in core 0.
0x0002004	PLIC_H0_MIE1	R/W	0x0	The M-mode interrupt enable register for interrupts 32 to 63 in core 0.
...
0x000207C	PLIC_H0_MIE31	R/W	0x0	The M-mode interrupt enable register for interrupts 992 to 1023 in core 0.
0x0002080	PLIC_H0_SIE0	R/W	0x0	The S-mode interrupt enable register for interrupts 1 to 31 in core 0.
0x0002084	PLIC_H0_SIE1	R/W	0x0	The S-mode interrupt enable register for interrupts 32 to 63 in core 0.
...
0x00020FC	PLIC_H0_SIE31	R/W	0x0	The S-mode interrupt enable register for interrupts 992 to 1023 in core 0.
0x0002100	PLIC_H1_MIE0	R/W	0x0	The M-mode interrupt enable register for interrupts 1 to 31 in core 1.
0x0002104	PLIC_H1_MIE1	R/W	0x0	The M-mode interrupt enable register for interrupts 1 to 31 in core 0.
...

Continued on next page

Table 10.2 – continued from previous page

Address	Name	Type	Initial value	Description
0x000217C	PLIC_H1_MIE31	R/W	0x0	The M-mode interrupt enable register for interrupts 1992 to 1023 in core 0.
0x0002180	PLIC_H1_SIE0	R/W	0x0	The S-mode interrupt enable register for interrupts 1 to 31 in core 1.
0x0002184	PLIC_H1_SIE1	R/W	0x0	The S-mode interrupt enable register for interrupts 32 to 63 in core 1.
...
0x00021FC	PLIC_H1_SIE31	R/W	0x0	The S-mode interrupt enable register for interrupts 992 to 1023 in core 1.
Reserved	-	-	-	-
0x0002000 +0x100*n	PLIC_Hn_MIE0	R/W	0x0	The M-mode interrupt enable register for interrupts 1 to 31 in core n. n=hart_id , n<256
0x0002004 +0x100*n	PLIC_Hn_MIE1	R/W	0x0	The M-mode interrupt enable register for interrupts 32 to 63 in core n. n=hart_id , n<256
...
0x000207C +0x100*n	PLIC_Hn_MIE31	R/W	0x0	The M-mode interrupt enable register for interrupts 992 to 1023 in core n. n=hart_id , n<256
0x0002080 +0x100*n	PLIC_Hn_SIE0	R/W	0x0	The S-mode interrupt enable register for interrupts 1 to 31 in core n. n=hart_id , n<256
0x0002084 +0x100*n	PLIC_Hn_SIE1	R/W	0x0	The S-mode interrupt enable register for interrupts 32 to 63 in core n. n=hart_id , n<256

Continued on next page

Table 10.2 – continued from previous page

Address	Name	Type	Initial value	Description
...
0x00020FC +0x100*n	PLIC_Hn_SIE31	R/W	0x0	The S-mode interrupt enable register for interrupts 992 to 1023 in core n. n=hart_id , n<256
0x01FFFFC	PLIC_CTRL	R/W	0x0	The PLIC permission control register.
0x0200000	PLIC_H0_MTH	R/W	0x0	The M-mode interrupt threshold register for core 0.
0x0200004	PLIC_H0_MCLAIMR/W		0x0	The M-mode interrupt claim/complete register for core 0.
Reserved	-	-	-	-
0x0201000	PLIC_H0_STH	R/W	0x0	The S-mode interrupt threshold register for core 0.
0x0201004	PLIC_H0_SCLAIMR/W		0x0	The S-mode interrupt claim/complete register for core 0.
Reserved	-	-	-	-
0x0202000	PLIC_H1_MTH	R/W	0x0	The M-mode interrupt threshold register for core 1.
0x0202004	PLIC_H1_MCLAIMR/W		0x0	The M-mode interrupt claim/complete register for core 1.
Reserved	-	-	-	-
0x0200000 +0x2000*n	PLIC_Hn_MTH	R/W	0x0	The M-mode interrupt threshold register for core n. n=hart_id , n<256
0x0203004 +0x2000*n	PLIC_Hn_MCLAIMR/W		0x0	The M-mode interrupt claim/complete register for core n.
0x0201000 +0x2000*n	PLIC_Hn_STH	R/W	0x0	The S-mode interrupt threshold register for core n. n=hart_id , n<256
0x0201004 +0x2000*n	PLIC_Hn_SCLAIMR/W		0x0	The S-mode interrupt claim/complete register for core n. n=hart_id , n<256

As shown in Fig. 10.7, the total address space occupied by PLIC and CLINT is 128MB, and the base address of this space is determined by *pad_cpu_apb_base* (an input port, please refer to the C908 integration manual). It should be noted that the attribute of this space needs to be set to Strong Ordered.

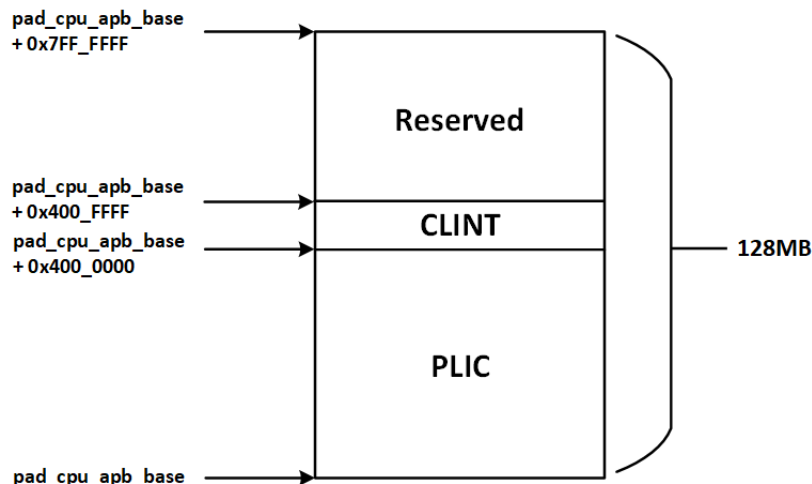


Fig. 10.7: PLIC&CLINT address space

10.2.5 PLIC_PRIO register

This register is used to set the priorities of interrupt sources. You can refer to the description of the privilege control register (PLIC_CTRL) for the read and write permissions of PLIC_PRIO register. The bit layout and definition of PLIC_PRIO are shown in Fig. 10.8 .

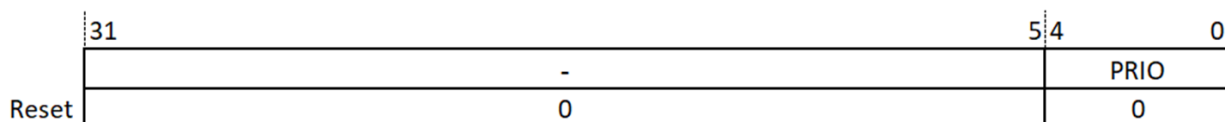


Fig. 10.8: PLIC_PRIO register

PRIO: the interrupt priority

The lower 5 bits of the PLIC_PRIO register are writable. The PLIC_PRIO register supports 32 interrupt priorities. Interrupts with a priority of 0 are invalid.

M-mode interrupts have higher priorities than S-mode interrupts in any conditions. In the same privilege mode, the priority 1 is the lowest priority, and the priority 31 is the highest priority. When multiple interrupts have the same priority, interrupt IDs are further compared, and the one with a smaller ID has a higher priority.

10.2.6 PLIC_IP register

The PLIC can read the PLIC_IP register to obtain the pending state of each interrupt. If the ID of an interrupt is N , the interrupt information is stored in IP y ($y = N \bmod 32$) in the PLIC_IP x ($x = N/32$) register. Bit 0 of the PLIC_IP0 register is tied to 0. You can refer to the description of the privilege control register (PLIC_CTRL) for the read and write permissions of PLIC_PRIO. The bit layout and definition of the register are shown in Fig. 10.9 .

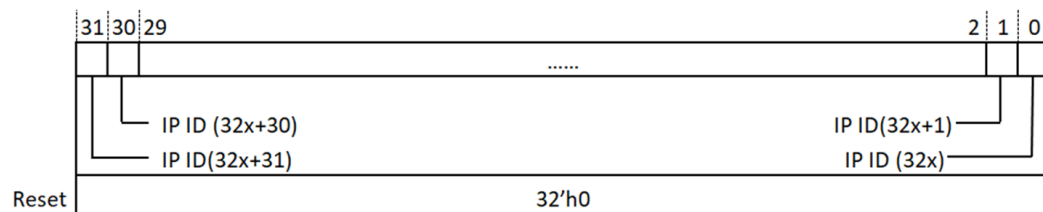


Fig. 10.9: PLIC_IP x register

IP: the interrupt pending state bit

This bit indicates the interrupt pending state of the corresponding interrupt source.

When the IP bit is 1, the interrupt source has pending interrupts. You can run a memory store instruction to set this bit to 1. When the sampling logic of the interrupt source detects valid level or pulse interrupts, this bit is also set to 1.

When the IP bit is 0, the interrupt source has no pending interrupt. You can run a memory store instruction to reset this bit. After an interrupt is handled, PLIC clears the corresponding IP bit.

10.2.7 PLIC_IE register

Each interrupt target has an interrupt enable bit for each interrupt source, to enable the corresponding interrupts. The M-mode interrupt enable register is used to enable M-mode external interrupts. The S-mode interrupt enable register is used to enable S-mode external interrupts.

If the ID of an interrupt is N , the interrupt enable information is stored in IE y ($y = N \bmod 32$) in the PLIC_IE x ($x = N/32$) register. The IE bit corresponding to ID0 is set to 0. For more information about the read and write permissions on the register, see the descriptions of the PLIC_PER register.

The bit layout and definition of the register are shown in Fig. 10.10.

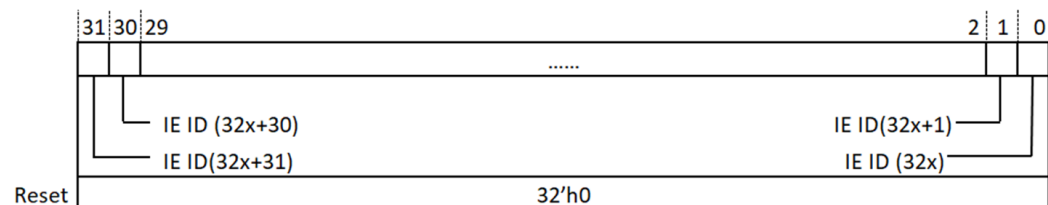


Fig. 10.10: PLIC_IE x register

IE: the interrupt enable state bit

This bit indicates the interrupt enable state of the corresponding interrupt source.

When the IE bit is 1, the interrupt source is enabled for the interrupt target.

When the IE bit is 0, the interrupt source is disabled for the interrupt target.

10.2.8 PLIC_CTRL register

The PLIC_CTRL register is used to control access permissions on PLIC registers in S-mode.

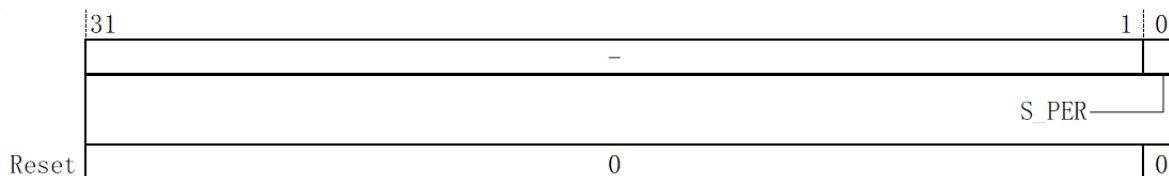


Fig. 10.11: PLIC_CTRL register

S_PER: the access permission control bit

When the S_PER bit is 0, the CPU has access to all PLIC registers only in M-mode. In S-mode, the CPU has access only to the S-mode PLIC_TH register and S-mode PLIC_CLAIM register, but not to the PLIC_PER, PLIC_PRIO, PLIC_IP, or PLIC_IE register. In U-mode, the CPU has no access to any PLIC registers.

When the S_PER bit is 1, the CPU has access to all PLIC registers in M-mode, and has access to all PLIC registers except PLIC_PER in S-mode. In U-mode, the CPU has no access to any PLIC registers.

10.2.9 PLIC_TH register

Each interrupt target has a PLIC_TH register. The PLIC initiates an interrupt request to an interrupt target only when the interrupt request is valid and the interrupt priority is higher than the interrupt threshold of the interrupt target. For more information about the read and write permissions on the register, see the descriptions of the PLIC_PER register.

The bit layout and definition of the register are shown in Fig. 10.12 .

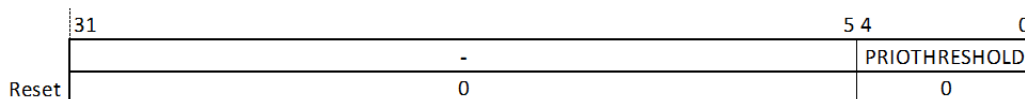


Fig. 10.12: PLIC_TH register

PRIOTHRESHOLD: the priority threshold

This bit indicates the interrupt threshold of the current interrupt target. When the interrupt threshold is 0, all interrupts are allowed.

10.2.10 PLIC_CLAIM register

Each interrupt target has a PLIC_CLAIM register. When the PLIC completes arbitration, this register is updated to the interrupt ID obtained in the current arbitration. For more information about the read and write permissions on the register, see the descriptions of the PLIC_CTRL register.

The bit layout and definition of the register are shown in Fig. 10.13 .

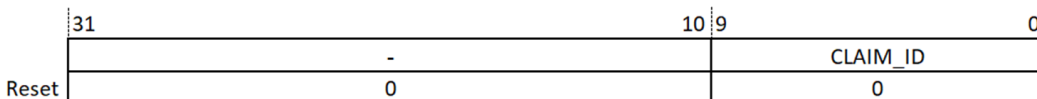


Fig. 10.13: PLIC_CLAIM register

CLAIM_ID: the interrupt request ID

A read operation to the register returns the ID currently stored in the register. The read operation indicates that the interrupt corresponding to the ID is in the process of handling. The PLIC starts the interrupt claim process.

A write operation to the register indicates that the interrupt corresponding to the ID to be written has been handled. The write operation does not update the PLIC_CLAIM register. The PLIC starts the interrupt complete process.

10.3 Multi-core interrupts

This section describes two common multi-core interrupt scenarios.

10.3.1 Multiple cores respond to external interrupts in parallel

In the PLIC model, one interrupt source can be mapped to multiple cores. When the interrupt source generates an interrupt request, the interrupt request is in the pending state with respect to multiple cores. Different cores run in different states, and they respond to the interrupt successively and read the PLIC_CLAIM register to obtain the interrupt ID. Design of the PLIC ensures that only the first core accessing the PLIC_CLAIM register obtains the valid ID, and other cores obtain an invalid ID (ID = 0) and therefore do not handle the interrupt. In this case, the interrupt is handled only once.

Mapping an interrupt to multiple cores reduces the overall interrupt response time because any one of the cores has an opportunity to handle the interrupt. However, bandwidth of the cores that obtain the invalid ID is consumed, wasting additional CPU resources.

Assuming there are two external interrupt sources, and the CPU is configured with 4 cores: Source 1 and Source 2. Source 1 is mapped to Core 0, Core 1 and Source 2 is mapped to Core 1, Core 2, and Core 3. Additionally, it is assumed that the priority of Source 2 is higher.

- When only Source 1 occurs, it may be processed by any core among Core 0, Core 1, and Core 2.

- When only Source 2 occurs, it may be processed by any core among Core 1, Core 2, and Core 3.
- When two interrupts occur simultaneously, there will be priority arbitration in Core 1 and Core 2, and the result is that Source 2 wins. Therefore, Source 2 may be processed by any core among Core 1, Core 2 or Core 3. Source 1 may be processed by Core 0.

10.3.2 Send software interrupts across cores

In the programming model of the CLINT controller, software interrupts are stored in dedicated registers:

- M-mode software interrupts are stored in the MSIP0, MSIP1, MSIP2 and MSIP3 registers.
- S-mode software interrupts are stored in the SSIP0, SSIP1, SSIP2 and SSIP3 registers.

Addresses of the preceding registers are unified and known to all cores. Each core can initiate write operations to the registers to send software interrupts to other cores or itself.

11.1 AXI master device interface

The master device interface of C908MP supports the AMBA 4.0 ACE or AXI protocol. For more information, see *AMBA Specifications —AMBA® AXI™ and ACE™ Protocol Specification*.

11.1.1 Features of the AXI master device interface

The AXI master device interface controls address accesses and data transmission between C908 and the AXI bus. It provides the following features:

- Complies with the AMBA 4.0 ACE or AXI protocol.
- Supports a bus width of 128 bits.
- Supports different frequency ratios between the system clock and the CPU master clock.
- Supports flop-out of all output signals and flop of all input signals to obtain better timing.

11.1.2 Outstanding capability of the AXI master device interface

This section describes the outstanding capability of the AXI master device interface provided by C908.

Table 11.1: Outstanding capability of the AXI master device interface

Parameter	Value	Description
Read Issuing Capability	$8n+26$ $n = \text{Number of cores}$	Each core can issue up to 8 non-cacheable and device read requests. All cores can issue up to 32 non-cacheable and device read requests. All cores can issue up to 26 cacheable read requests.
Write Issuing Capability	$12n+32$ $n = \text{Number of cores}$	Each core can issue up to 12 non-cacheable and device write requests. All cores can issue up to 32 non-cacheable and device write requests. All cores can issue up to 32 cacheable write requests.

Table 11.2: ARID encoding of the AXI master device interface

ARID[7:0]	Scenario	Outstanding requests of each ID
{2' b10, 6' b??????}	Cacheable read requests	Each ID has no outstanding requests. All cacheable write requests are outstanding. A total of 26 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' h00???	Non-cacheable and weak-ordered read requests	Non-cacheable and strong-ordered write requests are outstanding. A total of 31 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' h10}	Non-cacheable and weak-ordered fetch requests	
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable and weak-ordered exclusive lock acquisition and read request	
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable and strong-ordered read requests	

Table 11.3: AWID encoding of the AXI master device interface

AWID[7:0]	Scenario	Outstanding requests of each ID
{3' b111, 5' b?????}	Cacheable write requests	Each ID has no outstanding requests. All cacheable write requests are outstanding. A total of 32 outstanding requests are supported.
{4' b0000, 4' b????}	Non-cacheable and weak-ordered write requests	Each ID has no outstanding requests. All non-cacheable and weak-ordered write requests are outstanding. A total of 16 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable and strong-ordered exclusive lock acquisition and write request	Each ID has no outstanding requests.
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable and strong-ordered write requests	Non-cacheable and strong-ordered write requests are outstanding. A total of 31 outstanding requests are supported.

Note: The ARID and AWID encoding may vary with evolution of the CPU version. Therefore, SoC integration should not depend on specific IDs, but should conform to general-purpose rules of the ACE/AXI protocol.

11.1.3 Supported transmission types of Master Device

The AXI master device interface supports the following transmission types:

- Burst types: INCR and WRAP (Other burst types are not supported);
- Transmission lengths: 1 and 4 (Other transmission lengths are not supported);
- Exclusive access;
- Transmission sizes: quadword, doubleword, word, halfword, and byte (Other sizes are not supported);
- Read/write operation.

Note: The master device interface of C908 implements only a subset of all AXI transmission types. Therefore, SoC integration should not depend on specific transmission types, but should conform to general-purpose rules of the AXI protocol.

11.1.4 Supported response types of Master Device

The AXI master device interface supports the following types of responses from slave devices:

- OKAY
- EXOKAY
- SLVERR
- DECERR

11.1.5 CPU behavior in different bus responses

CPU behavior in different bus responses is shown in Table 11.4.

Table 11.4: Bus exception handling

RRESP/BRESP	Result
OKAY	Indicates that common transfer access succeeds or exclusive transfer access fails. If exclusive read transfer access fails, it indicates that the bus does not support exclusive transfer, and an access error exception is generated. If exclusive write transfer access fails, it indicates that lock preemption fails, and no exception is generated.
EXOKAY	Indicates that exclusive access succeeds.
SLVERR/DECERR	Indicates that an access error occurs. If this error occurs in read transfer, an exception is generated. If this error occurs in write transfer, it is ignored.

11.2 Device coherence port

The device coherence port (DCP) provided by C910 and C908MP is a user-configurable interface that can be used for peripheral devices to access the L2 cache and L1 data cache (D-Cache). The DCP maintains data consistency between peripherals and CPU chips. The DCP supports the AMBA AXI4 protocol. For more information, see *AMBA® AXI™ and ACE™ Protocol Specification*.

11.2.1 Features of DCP

The DCP provides the following basic features:

- Supports the AMBA 4.0 AXI bus protocol.
- Supports a bus width of 128 bits.
- Supports different frequency ratios of the system clock to the CPU master clock.
- Supports flop-out of all output signals and flop-in of all input signals to obtain better timing.
- Supports up to eight concurrent transfers for both read operations and write operations.

11.2.2 Supported transfer types of DCP

The DCP supports the following transfer features:

- Only the INCR transfer mode is supported, and LEN only supports 0 and 3.
- CACHE[3:0] must be 4' b1111, 4' b1011, or 4' b0111. Otherwise, SLVERR is returned.
- SIZE[2:0] must be 3' b100. Otherwise, SLVERR is returned.
- Exclusive access is not supported.
- WSTRB can be a value of any byte when LEN is 0 and must be 1 in all bits when LEN is 3.
- AxADDR is 16B boundary aligned when LEN is 0 and 64B boundary aligned when LEN is 3.
- The 5-bit AxID signal width is supported.
- Read operations and write operations are supported.

11.2.3 Supported response types of DCP

The DCP supports the following response types:

- OKAY
- SLVERR

11.2.4 Responses issued for different behaviors

Table 11.5 describes the types of responses that can be returned from the DCP.

Table 11.5: Responses Returned From DCP

RRESP/BRESP	Result
OKAY	The transfer access was successful and the received request was properly processed.
SLVERR	Access error. An unsupported type of transfer was received.

11.3 Low Latency Port

The low latency port (LLP) provided by C908MP is a user-configurable master interface that can be used to access peripheral devices of the system. The LLP supports the AMBA AXI4 protocol. For more information, see *AMBA® AXI™ and ACE™ Protocol Specification*.

11.3.1 Features of LLP

The LLP provides the following basic features:

- Supports the AXI4.0 protocol.
- Supports a data bus width of 128 bits and an address bus width of 40 bits.
- Supports up to 8 outstanding read requests and 12 outstanding write requests for each core. Four cores can issue up to 32 outstanding read requests and 32 outstanding write requests.
- Supports an 1:N frequency ratio of CPU to LLP (N indicates an integer and ≤ 8).
- Supports all bus responses.
- Supports non-aligned access.

11.3.2 Outstanding capabilities of LLP

This section describes the outstanding capabilities of the LLP provided by C908.

Table 11.6: the outstanding capabilities provided by the LLP

Parameter	Value	Description
Read Issuing Capability	8n n indicates the number of cores.	Each core can issue up to 8 read requests. Four cores can issue up to 32 read requests.
Write Issuing Capability	12n n indicates the number of cores.	Each core can issue up to 12 write requests. Four cores can issue up to 32 write requests.

Table 11.7: AXI LLP ARID encoding

ARID[7:0]	Application scenario	Outstanding capability of each ID
{1' b0, 2' b(coreid), 5' b00??}	Non-cacheable weak-ordered read requests	All non-cacheable outstanding read requests are outstanding. A total of 32 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' h10}	Non-cacheable weak-ordered value-taking requests	
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable weak-ordered lock preemption requests	
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable strong-ordered read requests	

- The same ID on the AR channel will not appear on the master device interface and LLP at the same time.

Table 11.8: AXI LLP AWID encoding

AWID[7:0]	Application scenario	Outstanding capability of each ID
{4' b0000, 4' b????}	Non-cacheable weak-ordered write requests	Each ID has no outstanding requests. All non-cacheable weak-ordered write requests are outstanding. A total of 16 outstanding requests are supported.
{1' b0, 2' b(coreid), 5' h1e}	Non-cacheable weak-ordered lock preemption write requests	Each ID has no outstanding requests.
{1' b0, 2' b(coreid), 5' h1d}	Non-cacheable strong-ordered write requests	Non-cacheable strong-ordered write requests are outstanding. A total of 32 outstanding requests are supported.

- The same ID on an AW channel will not appear on the master device interface and LLP at the same time.

Note: The preceding rules for encoding ARIDs and AWIDs may change with the evolution of CPU versions. Therefore, when you are integrating your SoC, do not specify ID values. Instead, you must follow the general rules of the AXI protocol.

11.3.3 Supported transfer types of LLP

The LLP supports the following transfer features:

- Supports only INCR and does not support FIXED and WRAP.
- Supports 8' b0 for LEN.
- Supports exclusive access.
- Supports the non-cacheable normal memory access and device properties.
- Supports non-aligned access.
- Supports a size of 3' b000 to 3' b100 (from 1B to 16B).
- For normal memory non-cacheable regions, write requests can be merged. wstrb can issue any value, and the Axsiz is fixed at 3' b100.
- For device regions, the Axsiz can be a value ranging from 3' b000 to 3' b011.

Note: The LLP of C908 implements only one subset of AXI transfer types. SoC integration should not depend on specific transfer types and should conform to the general rules of the AXI protocol.

11.3.4 Supported response types of LLP

The LLP supports the following response types:

- OKAY
- EXOKAY
- SLVERR
- DECERR

12.1 Features of the debug unit

C908 is compatible with the RISC-V Debug V0.13.2 protocol. The peripheral debug interface supports two modes: two-wire JTAG (a T-Head custom protocol) and 5-wire JTAG (standard JTAG5).

The debug interface provides an interaction channel between software and the CPU. You can call the debug interface to obtain information stored in registers and memory of the CPU and information about other on-chip devices. You can also call the debug interface to download programs.

The debug interface provides the following features:

- Supports two-wire JTAG and 5-wire JTAG.
- Supports multi-cluster debugging.
- Supports synchronous and asynchronous debug, enabling the CPU to enter the debug mode in extreme conditions.
- Supports software breakpoints.
- Supports multiple memory breakpoints.
- Enables you to check and set the values of CPU registers.
- Enables you to check and modify memory values.
- Enables the CPU to run an instruction in a single step or multiple steps.
- Enables you to quickly download programs.

- Enables the CPU to enter the debug mode after it is reset.

Debug of C908 is jointly completed by the debug software, debug proxy, debugger, and debug interface. The location of the debug interface in the CPU debug environment is shown in Fig. 12.1 . The debug software is connected to the debug proxy over network. The debug proxy is connected to the debugger through a USB interface. The debugger communicates with the debug interface in JTAG mode.

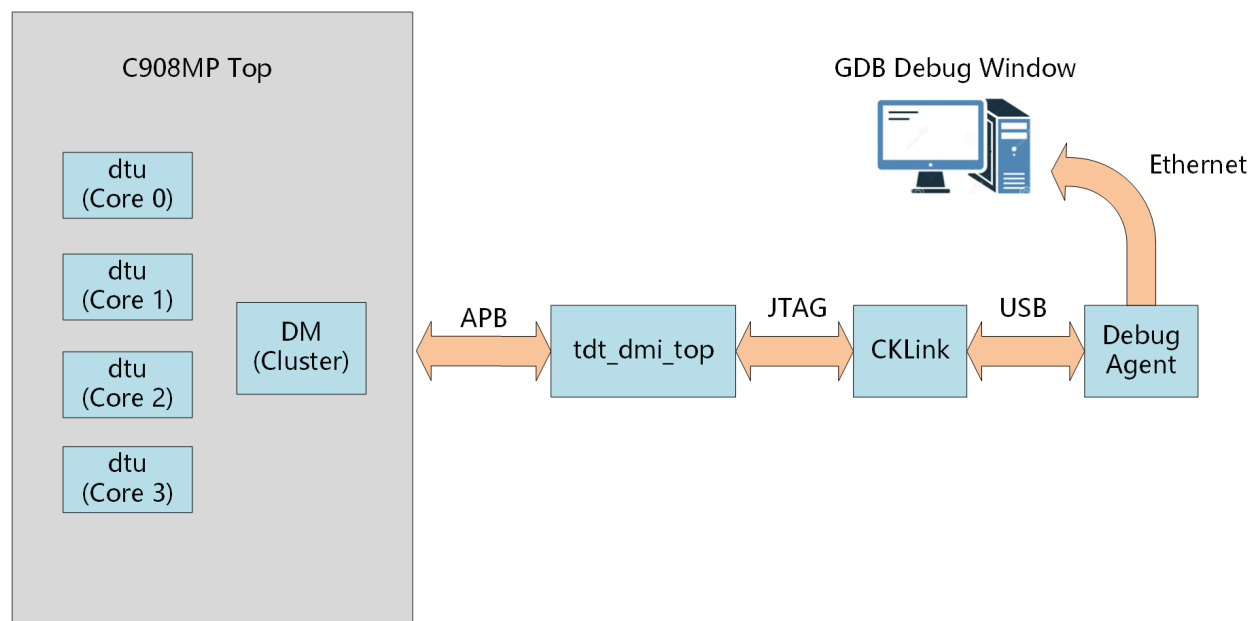


Fig. 12.1: Location of the debug interface in CPU debug environment

12.2 Debugging resource configuration

C908 supports three types of debug resource configurations:

- Minimum configuration: 1 program buffer with implicit EBREAK implemented and 1 hardware breakpoint.
- Typical configuration: 2 program buffers with implicit EBREAK implemented, 3 hardware breakpoints, and 8 PCFIFO entries for recording historical PC jump streams.
- Maximum configuration: 2 program buffers with implicit EBREAK implemented, 8 hardware breakpoints that can form a trigger chain, and 16 PCFIFO entries for recording historical PC jump streams, and an independent debug AHB interface for independent memory space access.

The RISC-V Debug protocol defines multi-function triggers for implementing breakpoints and watchpoints. Supported trigger types:

- Instruction address type: matches the instruction address (that is the instruction PC). This type functions similarly to traditional breakpoints.

- Instruction data type: matches the instruction code.
- Access address type: matches the memory address specified in the access instruction. This type functions similarly to traditional watchpoints.
- Access data type: matches the access data specified in the access instruction.

The RISC-V Debug protocol also defines 6 matching modes. The matching modes and the trigger types are orthogonal.

1. Congruent matching: When the CPU PC value is the same as the trigger value, the trigger is fired.
2. Low-bit mask matching: It can be set to do not compare low-order bits. In this case, when the CPU PC value is the same as the trigger value, the trigger is fired.
3. Greater than or equal to comparison: When the CPU PC value is greater than or equal to the trigger value, the trigger is fired.
4. Less than comparison: When the CPU PC value is less than the trigger value, the trigger is fired.
5. Mask matching with the lower half: The trigger value is split into two halves: the upper half [63:32] and the lower half [31:0]. The upper half is used as the mask, and the lower half is used as the template. When the [31:0] part of the trigger value is equal to the [31:0] part of the CPU value and the [63:32] part of the trigger value, the trigger is fired.
6. Mask matching with the upper half: The trigger value is split into two halves: the upper half [63:32] and the lower half [31:0]. The upper half is used as the mask, and the lower half is used as the template. When the [31:0] part of the trigger value is equal to the [63:32] part of the CPU value and the [63:32] part of the trigger value, the trigger is fired.

For more information, see *5.2.9 match control* in *RISC-V External Debug Support Version 0.13.2*.

In addition to the preceding features, each configuration supports software breakpoints, abstract command registers, entering debug in asynchronous mode or after reset, single-step instruction execution, and other debug resources and methods.

C908 supports flexible power consumption management. It supports multiple power domains, single-core power-off, multi-cluster power-off, retention mode of L2 SRAM (when SRAM with the retention feature is used in L2), clearing L2 cache through external hardware interfaces, and other operations. This chapter describes the power consumption management features of C908.

13.1 Power domain

C908 supports up to 6 power domains:

- PDC 0-3: Each core is a power domain, including the computing unit, control logic, and cache RAM of the core.
- PDL2RAM: All RAM units of L2 cache are contained in this independent power domain. (When SRAM with the retention feature is used in L2, this power domain exists.)
- PDL2SYS: The PDL2SYS power domain covers other domains in a cluster except the PDC 0-3 and PDL2RAM domains, including CIU, L2C CTRL, Debug, SYSIO, and other sub-modules.

13.2 Overview of low-power modes

C908 supports the following low-power modes:

- Normal mode: The cores and L2 are running properly.

- Core WFI mode: Some cores are in the wait for interrupt (WFI) mode.
- Individual-core power-off: Some cores are powered off.
- L2 RAM retention mode: Except L2 RAM, other modules in a cluster, including the four cores and L2 control logic, are power off.
- Cluster power-off: All modules in a cluster are power off.

13.3 Core WFI process

By executing the WFI low power instruction, a core enters WFI mode and outputs signal `core(x)_pad_lpmdb[1:0]=2'b00`, which indicates that the core has entered WFI mode. The L2 subsystem will disable the global ICG of this core inside the cluster.

The core will be woken up and exit WFI mode upon the occurrence of the following events:

- Reset
- Interrupt request: external interrupt, software interrupt, or timer interrupt requests sent by the PLIC or CLINT submodules.
- Debug request

When one of the following events occurs, the core is temporarily woken up to process the event. It reenters low power mode after the event is processed. The core does not exit WFI mode during the entire process.

- Snoop request: Snoop requests sent by other cores.

13.4 Individual-core power-off process

The system can shut down the power of a core to completely terminate the static power of the core.

The process for powering off a core:

- Notifies SoC that the individual-core power-off process is to be executed. The implementation of this step is subject to the SoC design.
- Masks all interrupt requests, including external interrupts, software interrupts, and timer interrupts, and then disables the interrupt enable bit (MIE/SIE) of the `mstatus/status` register and the interrupt enable bit of the `mie/sie` register. If the power-off process is executed in M-mode, the interrupt enable bits of the `mstatus` and `mie` registers are disabled. If the power-off process is executed in S-mode, the interrupt bits of the `sstatus` and `sie` registers are disabled.
- Disables data prefetch
- Executes D-Cache INV&CLR ALL to write dirty lines back to the L2 cache.
- Disables D-Cache (no store instruction allowed between the clear cache and disable cache operations).

- Disables the SMPEN bit to mask snoop requests.
- Executes the fence iorw, iorw instruction.
- Executes the WFI instruction to enter WFI mode.

The system performs the following operations:

- Detects a valid low-power output signal `core(x)_pad_lpm�_b` sent from the core.
- Sets `pad_tdt_dm_core_unavail[x]` to 1 to mask debug requests bound for the core to be powered off.
- Activates the output signal clamp bit of the core to be powered off.
- Sets the reset signal `pad_core(x)_rst_b` to 0 for the core to be powered off.
- Shuts down the power to the core.

A powered-off core can restart only by reset. The process of powering on a core again:

- The system detects a specific event and determines to wake up the core.
- The system sets the reset address of the core.
- The reset signal of the core is set to 0.
- The power is turned on and the reset signal remains unreleased.
- The output signal clamp bit of the core is released.
- The reset signal of the core is released.
- The core executes the initialization program, enables the SMPEN bit, or performs initialization operations, such as enabling the MMU or D-Cache.

13.5 Cluster power-off process (hardware clearing of the L2 cache)

Ensure that the power is shut down for all cores except the main core in the cluster. The main core is the last core to be powered off. It can be any one of the eight cores.

The main core performs the following operations:

- Notifies SoC that the cluster power-off process is to be executed. The implementation is subject to the SoC design.
- Masks all interrupt requests, including external interrupts, software interrupts, and timer interrupts, and then disables the interrupt enable bit (MIE/SIE) of the `mstatus/status` register and the interrupt enable bit of the `mie/sie` register.
- Disables data prefetch
- Executes the D-Cache INV&CLR ALL operation.
- Disables D-Cache (no store instruction allowed between the clear cache and disable cache operations).

- Disables the SMPEN bit.
- Executes the fence iorw, iorw instruction.
- Executes the WFI instruction to enter WFI mode.

The system performs the following operations:

- Detects a valid low-power output signal `core(x)_pad_lpmnd_b` sent from the main core.
- Sets `pad_tdt_dm_core_unavail[x]` to 1 to mask debug requests bound for the main core.
- Activates the output signal clamp bit of the main core.
- Sets the reset signal `pad_core(x)_rst_b` to 0 for the main core.
- Shuts down the power of the main core.
- Sets `pad_cpu_l2cache_flush_req` to 1 to start clearing the L2 cache.
- Waits for C908 to return `cpu_pad_l2cache_flush_done = 1`.
- Sets `pad_cpu_l2cache_flush_req` to 0. (Then C908 will set `cpu_pad_l2cache_flush_done` to 0.)
- Ensures that there are no new requests from DCP (if configured).
- Waits for C908 to return `cpu_pad_no_op = 1`.
- Activates the output signal clamp bit of the top level.
- Sets the reset signal `pad_cpu_rst_b` of the L2 cache to 0.
- Shuts down the power of the top level.

The cluster is powered on again by reset. The process of powering on the cluster again:

- The reset signal is set to 0 for all cores in the cluster and top level.
- The power is turned on, the reset signal remains unreleased, and the PLL is stable.
- The output signal clamp bits of the cores and the top level are released.
- The reset signals of the cores and the top level are released.
- The reset exception service program is executed to recover the CPU.

13.6 Simplified scenario: overall cluster power-off process (hardware clearing of the L2 cache)

In some systems, SoC designers may take a simple way to divide power domains. That is, take the entire C908 cluster (8 cores and one L2 subsystem) as a power domain and power off the cluster as a whole, instead of powering off each core separately. The cluster can be powered off (hardware clearing of the L2 cache) through the following steps:

The system performs the following operations:

- Notifies SoC that the overall cluster power-off process is to be executed. The implementation is subject to the SoC design.
- Ensures that all existing transfers on DCP (if configured) are completed, and no new read/write requests are sent to DCP.

The core (no need to distinguish the main core and secondary core, as the process is the same for them) performs the following operations:

- Masks all interrupt requests including external interrupts, software interrupts, and timer interrupts, and disables the interrupt enable bit (MIE/SIE) of the mstatus/sstatus register, as well as the interrupt enable bit of the mie/sie register.
- Disables data prefetch
- Executes INV&CLR D-Cache ALL to write dirty lines back to the L2 cache.
- Disables D-Cache (no store instruction allowed between the clear cache and disable cache operations).
- Disables the SMPEN bit to mask snoop requests.
- Executes the fence iorw, iorw instruction.
- Executes the WFI instruction.

The system performs the following operations:

- Waits for `core(x)_pad_lpmdb[1:0]==2' b00`, which means all CPU cores enter the low power state.
- Sets `ad_tdt_dm_core_unavail[x]` to 1 for all cores to mask debug requests.
- Sets `pad_cpu_l2cache_flush_req` to 1 to start hardware clearing for the L2 cache.
- Waits for C908 to return `cpu_pad_l2cache_flush_done=1`, which means the L2 cache is cleared.
- Sets `pad_cpu_l2cache_flush_req` to 0. (Then, C908 will set `cpu_pad_l2cache_flush_done` to 0.)
- Waits for `cpu_pad_no_op==1' b1`, which means the L2 cache enters the idle state. (All CPU cores are still in the low power state.)
- Activates the output signal clamp of the cluster.
- Asserts all reset signals.
- Powers off the entire cluster.

13.7 Low power consumption related programming models and interface signals

13.7.1 Changes in Programming models

M-mode reset register (MRMR)

This register has been deprecated. If you continue to access this register, read operations return zero, write operations are invalid, and no exceptions are triggered. The impact of this change is as follows: The reset signal of each core is no longer controlled by MRMR. SoC can independently control the reset and reset release bits of each core through `pad_core(x)_rst_b`.

M-mode snoop enable register (MSMPR)

This register is 64 bits wide. Only bit [0] has a definition (`=SMPEN`) and its default value is 0. This register controls whether cores can accept snoop requests.

- When `MSMPR.SMPEN` is 0, the cores cannot process snoop requests, and the L2 subsystem masks snoop requests bound for the cores.
- When `MSMPR.SMPEN` is 1, the cores can process snoop requests, and the L2 subsystem sends snoop requests to the cores.

The corresponding `SMPEN` bit must be set to 0 before a core is powered off. After the core is powered on, `SMPEN` must be set to 1 before the software enables the D-Cache and MMU. When a core is in normal running mode, its `SMPEN` bit must be set to 1.

M-mode reset vector base address register (mrvbr)

Each core has an `mrvbr` register for determining the restart address of the core. The access permission for `mrvbr` registers is MRO. The initial value of the `mrvbr` register of a core is determined by the hardware signal `pad_core(x)_rvba[39:1]`.

13.7.2 Interface signals

C908 communicates with the power management unit of SoC by using the following signals:

- `core(x)_pad_lpmd_b`:
Indicates whether a core is in WFI mode. 2' b11 indicates normal mode, and 2' b00 indicates WFI mode.
- `cpu_pad_no_op`:
Indicates whether the L2 cache is idle. This signal is valid (a high level) when all cores enter low power mode and the L2 cache finishes all transmissions.
- `pad_cpu_l2cache_flush_req` and `cpu_pad_l2cache_flush_done`:
Clear the L2 cache under the control of SoC. These signals are used in the cluster power-off process. The req signal is driven by SoC, and the done signal is driven by C908.
First, SoC sets the req signal to 1 to start the L2 cache clearing process. Then C908 finishes clearing the L2 cache and returns `done = 1`. Finally, SoC sets the req signal to 0 and sets the done signal to 0.

14.1 PMU overview

The performance monitoring unit (PMU) of C908 complies with the RISC-V standard and collects software and hardware information during a program operation for software developers to optimize their programs.

The software and hardware information collected by the PMU includes the following:

- Number of running clocks and the time
- Instruction statistics
- Statistics of key components of the CPU

14.2 PMU programming model

14.2.1 PMU functions

Basic functions of the PMU are:

- Prohibits the counting of all events by using the `mcountinhibit` register.
- Resets the PMU counters, including `mcycle`, `minstret`, and `mhpmcounter3` to `mhpmcounter31`.
- Configures the corresponding events for each PMU counter. In C908, the mappings between events and counters are fixed. Therefore, events must be configured for the PMU counters based on a fixed pattern. For example, `0x1` must be written to `mhpmevent5`, which means that `mhpmcounter5` counts

the number of 0x1 events (L1 ICache access count), and 0x2 must be written to mhpmevent4, which means that mhpmevent4 counts the number of 0x2 events (L1 ICache miss count), and so forth.

- Grants access permissions. The mcounteren register determines whether PMU counters can be accessed in S-mode, and scounteren determines whether PMU counters can be accessed in U-mode.
- Discharges the prohibition by using the mcountinhibit register and starts counting.

For more information, see *PMU setting example*.

14.2.2 PMU event overflow interrupt

C908 implements the M-mode event overflow mark register (mcounterof) and M-mode event interrupt enable register (mcounterinten). For more information about register functions and read/write permissions, see *Appendix C-1 M-mode control register*. In the mcounterof register, the bits and event counters are in one-to-one correspondence, indicating whether the event counters overflow. In the mcounterinten register, the bits and event counters are in one-to-one correspondence, indicating whether to initiate an interrupt request when an event counter overflows.

The unified interrupt vector number of overflow interrupts initiated by the PMU is 17. The interrupt enabling and processing process is the same as that of common interrupts. For more information, see *Exceptions and Interrupts*.

14.3 PMU related control registers

14.3.1 M-mode counter access enable register (mcounteren)

The mcounteren register determines whether U-mode counters can be accessed in S-mode.

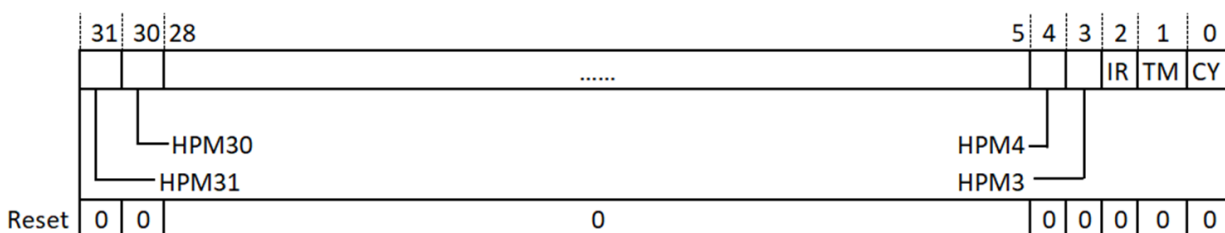


Fig. 14.1: M-mode counter access enable register (mcounteren)

Table 14.1: Description of the M-mode counter access enable register

Bit	Read/Write	Name	Description
31:3	Read/Write	HPM n	The access bit of the shpmcountern/hpmcountern register in S-mode. 0: An illegal instruction exception will occur for accesses to the shpmcountern/ hpmcountern register in S-mode. 1: The shpmcountern/hpmcountern register can be normally accessed in S-mode.
2	Read/Write	IR	The access bit of the sinstret/instret register in S-mode. 0: An instruction exception will occur for accesses to the minstret sinstret/instret register in S-mode. 1: The sinstret/instret register can be normally accessed in S-mode.
1	Read/Write	TM	The access bit of stime/time/stimecmp in S-mode. 0: An illegal instruction exception will occur for accesses to the stime/time/stimecmp in S-mode. 1: When the corresponding bit of the mcounteren register is 1, stime/time/stimecmp can be normally accessed in S-mode. Otherwise, an illegal instruction exception will occur.
0	Read/Write	CY	The access bit of the scycle/cycle register in S-mode. 0: An illegal instruction exception will occur for accesses to the cycle scycle/cycle register in S-mode. 1: The scycle/cycle register can be normally accessed in S-mode.

14.3.2 S-mode counter access enable register (scounteren)

The scounteren register determines whether U-mode counters can be accessed in U-mode.

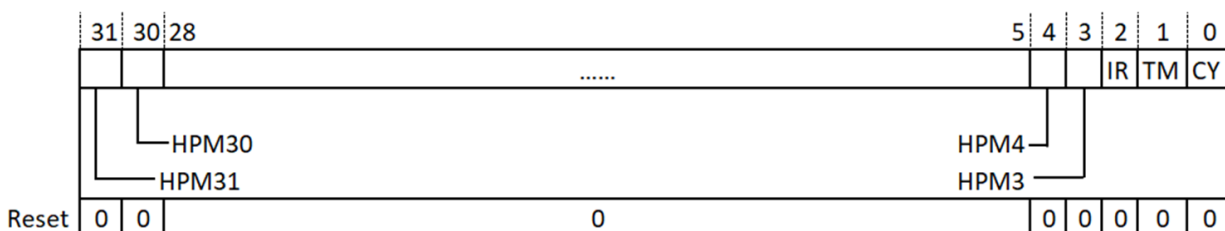


Fig. 14.2: S-mode counter access enable register (scounteren)

Table 14.2: Description of the scounteren register

Bit	Read/Write	Name	Description
31:3	Read/Write	HPM n	The access bit of the hpmcounter n register in U-mode. 0: An illegal instruction exception will occur for accesses to the hpmcounter n register in U-mode. 1: When the corresponding bit of the scounteren register is 1, the hpmcounter register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
2	Read/Write	IR	The access bit of the instret register in U-mode. 0: An illegal instruction exception will occur for accesses to the instret register in U-mode. 1: When the corresponding bit of the scounteren register is 1, the instret register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
1	Read/Write	TM	The access bit of the time register in U-mode. 0: An illegal instruction exception will occur for accesses to the time register in U-mode. 1: When the corresponding bit of the scounteren register is 1, the time register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.
0	Read/Write	CY	The access bit of the cycle register in U-mode. 0: An illegal instruction exception will occur for accesses to the cycle register in U-mode. 1: When the corresponding bit of the scounteren register is 1, the cycle register can be normally accessed in U-mode. Otherwise, an illegal instruction exception will occur.

14.3.3 M-mode count inhibit register (mcountinhibit)

The mcountinhibit register inhibits counting of M-mode counters. When performance analysis is not required, counters can be disabled to reduce the power consumption of the CPU.

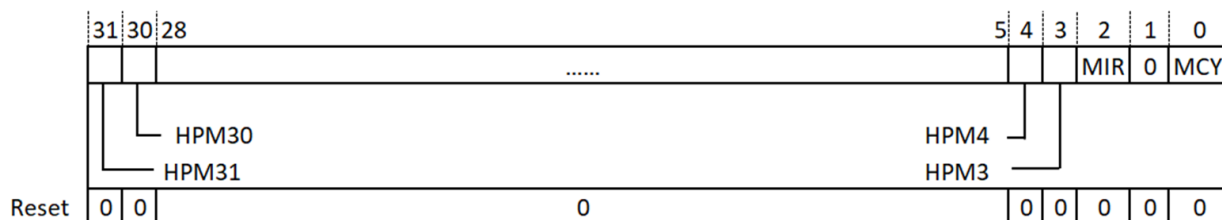


Fig. 14.3: M-mode count inhibit register (mcountinhibit)

Table 14.3: Description of the M-mode count inhibit register

Bit	Read/Write	Name	Description
31:3	Read/Write	MHPM n	n Count inhibit bit of the mhpcounter register 0: normal counting 1: counting inhibited
2	Read/Write	MIR	Count inhibit bit of the minstret register 0: normal counting 1: counting inhibited
1	-	-	-
0	Read/Write	MCY	Count inhibit bit of the mcycle register 0: normal counting 1: counting inhibited

14.3.4 S-mode count inhibit register (scountinhibit)

The scountinhibit register inhibits counting of S-mode counters. When performance analysis is not required, counters can be disabled to reduce the power consumption of the CPU.

When `mcounterwen.bit[n] = 1`, `scountinhibit[n]` can be read/written in S-mode.

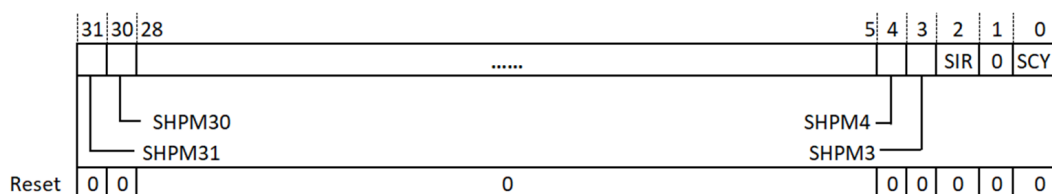


Fig. 14.4: S-mode count inhibit register (scountinhibit)

Table 14.4: Description of the S-mode count inhibit register

Bit	Read/Write	Name	Description
31:3	Read/Write	SHPM n	n Count inhibit bit of the shpmcounter register 0: normal counting 1: counting inhibited
2	Read/Write	SIR	Count inhibit bit of the sinstret register 0: normal counting 1: counting inhibited
1	-	-	-
0	Read/Write	SCY	Count inhibit bit of the scycle register 0: normal counting 1: counting inhibited

14.3.5 S-mode write enable register (mcounterwen)

The mcounterwen register determines whether S-mode event counters can be written in S-mode. This register is an M-mode extension register. For the register description, see *Appendix C-1 M-mode control registers*.

14.3.6 Performance monitoring event select register (mhpmevent3-31)

The mhpmevent3-31 register selects the counting event corresponding to a counter. In C908, a counter corresponds to an event, which cannot be modified. Therefore, only the corresponding event ID can be written to each event selector. An event counter performs counting normally only after the index value of the corresponding event is written to the event selector, and the event counter is initialized by using the csrw instruction.

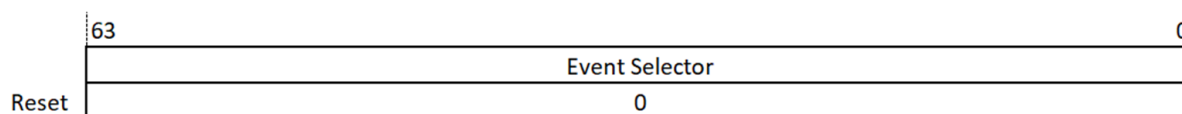


Fig. 14.5: M-mode performance monitoring event select register (mhpmevent)

Table 14.5 describes the M-mode performance monitoring event select register.

Table 14.5: Description of the M-mode performance monitoring event select register

Bit	Read/Write	Name	Description
63:0	Read/Write	Event index	Performance monitoring event index 0: no event 0x1 to 0xB4: performance monitoring events implemented by hardware. For more information, see Table 14.6 . >0xB4: performance monitoring events that are not defined by hardware. These events are customized for use by the software.

Table 14.6 describes the correspondence between event selectors, events, and counters.

Table 14.6: List of correspondence between counters and events

Index	Event
0x1	L1-icache Access
0x2	L1-icache Miss
0x3	iTLB Miss
0x4	Not defined
0x5	jTLB Miss

Continued on next page

Table 14.6 – continued from previous page

Index	Event
0x6	Condition Branch Mispred
0x7	Condition Branch
0x8	Indirect Branch Miss
0x9	Indirect Branch
0xA	Not defined
0xB	Store Instruction
0xC	L1-dcache load access
0xD	L1-dcache load miss
0xE	L1-dcache store access
0xF	L1-dcache store miss
0x10	Not defined
0x11	Not defined
0x12	Not defined
0x13	Not defined
0x14	Not defined
0x15	Not defined
0x16	Issue Instruction
0x17	Not defined
0x18	Not defined
0x19	Not defined
0x1A	Not defined
0x1B	IFU Branch Target Mispred
0x1C	IFU Branch Target Instruction
0x1D	ALU Instruction
0x1E	Not defined
0x1F	Vector SIMD Instruction
0x20	CSR Instruction
0x21	ATOMIC Instruction
0x22	Not defined
0x23	Interupt Numer
0x24	Not defined
0x25	Environment Call
0x26	Long Jump
0x27	Stalled Cycles Frontend
0x28	Stalled Cycles Backend
0x29	SYNC Stall
0x2A	Float Point Instruction

Continued on next page

Table 14.6 – continued from previous page

Index	Event
0x2B	M Mode Cycles
0x2C	S Mode Cycles
0x2D	U Mode Cycles
0x2E	Exception Number
0x2F	Flush Number
0x30	LOAD Instruction
0x31	Fused Instruction
0x32	MULT Instruction
0x33	DIV Instruction
0x34	Mult Inner Forward
0x35	DIV Buffer Hit
0x36	Branch Instruction
0x37	Uncondition Branch
0x38	Branch Mispred
0x39	Uncondition Branch Mispred
0x3A	Taken Branch Mispred
0x3B	Taken Condition Branch
0x3C	Taken Condition Branch Mispred
0x3D	Unalign LOAD Instruction
0x3E	Unalign STORE Instruction
0x3F	LR Instruction
0x40	SC Instruction
0x41	AMO Instruction
0x42	Barrier Instruction
0x43	Failed SC Instruction
0x44	Bus Barrier
0x45	FP DIV Instruction
0x46	FP LOAD Instruction
0x47	FP STORE Instruction
0x48	Vector DIV Instruction
0x49	Vector LOAD Instruction
0x4A	Vector STORE Instruction
0x4B	Vector Micro Op
0x4C	ECC Interrupt
0x4D	Async Abort Interrupt
0x4E	IF Stall
0x4F	IP Stall

Continued on next page

Table 14.6 – continued from previous page

Index	Event
0x50	IB Stall
0x51	IF Refill Stall
0x52	IF Mmu Stall
0x53	IB Mispred Stall
0x54	IB Fifo Stall
0x55	IB Ind Btb Rd Stall
0x56	IB Vsetvl Stall
0x57	ID Stall
0x58	RF Stall
0x59	EU Stall
0x5A	ID Inst Pipedown
0x5B	RF Inst Pipedown
0x5C	ID One Inst Pipedown
0x5D	ID CSR Before Fence Stall
0x5E	ID VSETVL Fof Stall
0x5F	ID Flush Stall
0x60	ID Misprediction Stall
0x61	ID IID Not Vld Stall
0x62	RF One Inst Pipedown
0x63	RF RAW Stall
0x64	RF WAW Stall
0x65	RF Structure Stall
0x66	RF CSR After Fence Stall
0x67	EU IU Full
0x68	EU IU Control Full
0x69	EU CP0 Full
0x6A	EU LSU LOAD Full
0x6B	EU LSU STORE Full
0x6C	EU VFPU Full
0x6D	EU BJU Full
0x6E	IU Dp Stall Pipe0
0x6F	IU MULT Stall Pipe0
0x70	IU DIV EX1 Stall Pipe0
0x71	IU Dp Stall Pipe1
0x72	IU MULT Stall Pipe1
0x73	IU DIV EX1 Stall Pipe1
0x74	IU DP Wb Conflict Pipe0

Continued on next page

Table 14.6 – continued from previous page

Index	Event
0x75	IU DP WAW Stall Pipe0
0x76	IU DP Uncommit Pipe0
0x77	IU DP Wb Conflict Pipe1
0x78	IU DP WAW Stall Pipe1
0x79	IU DP Uncommit Pipe1
0x7A	IU MULT Uncommit
0x7B	IU MULT Wb Stall
0x7C	IU DIV Uncommit
0x7D	IU DIV Wb Stall
0x7E	LSU LOAD WAW Stall
0x7F	LSU LOAD Commit Stall
0x80	LSU LOAD RAW Stall
0x81	LSU STORE Commit Stall
0x82	Vidu Rf No Pipedown
0x83	VPU Stall Pipe0
0x84	VPU Stall Pipe1
0x85	VPU Struct Hazard Stall Pipe0
0x86	VPU Uncommit Stall Pipe0
0x87	VPU VLSU Stall Pipe0
0x88	VPU Struct Hazard Stall Pipe1
0x89	VPU Uncommit Stall Pipe1
0x8A	VPU VLSU Stall Pipe1
0x8B	VFPU FDIV/VDIV Busy
0x8C	BJU CP0 Stall
0x8D	BJU IBUF Stall
0x8E	BJU Wb Stall
0x8F	BJU Pipedown Stall
0x90	RTU Flush
0x91	RTU IU Not No OP
0x92	RTU BJU Not No OP
0x93	RTU LSU Not No OP
0x94	RTU CP0 Not No OP
0x95	RTU VFPU Not No OP
0x96	RTU Only IU Not No OP
0x97	RTU Only BJU Not No OP
0x98	RTU Only LSU Not No OP
0x99	RTU Only CP0 Not No OP

Continued on next page

Table 14.6 – continued from previous page

Index	Event
0x9A	RTU Only VFPU Not No OP
0x9B	L1 Dcache Access
0x9C	L1 Dcache Miss
0x9D	L1 Dcache Exclusive Eviction
0x9E	Icache Prefetch
0x9F	Dcache Amr Active
0xA0	Icache Prefetch Miss
0xA1	Dcache Refill Casued by Prefetch
0xA2	Dcache Hit Caused by Prefetch
0xA3	Store Dtlb Miss
0xA4	Load Dtlb Miss
0xA5	L2 Access
0xA6	L2 Miss
0xA7	Snb Read Create Vld
0xA8	Snb Read Create Stall
0xA9	Snb Write Create Vld
0xAA	Snb Write Create Stall
0xAB	Iq Full
0xAC	Vidu Vec0 Stall
0xAD	Vidu Vec1 Stall
0xAE	Vidu Vec0 Depend Stall
0xAF	Vidu Vec0 Struct Hazard Stall
0xB0	Vidu Vec1 Depend Stall
0xB1	Vidu Vec1 Struct Hazard Stall
0xB2	Vidu Total Cycle
0xB3	Vidu Vec0 Cycle
0xB4	Vidu Vec1 Cycle
>0xB4	Not defined

14.3.7 Event counters

Event counters are divided into three groups: M-mode event counters, U-mode event counters, and S-mode event counters (extended in C908). For more information, see [Table 14.7](#).

Table 14.7: M-mode event counter list

Name	Index	Read/Write	Initial value	Description
MCYCLE	0xB00	MRW	0x0	The cycle counter.
MINSTRET	0xB02	MRW	0x0	The instructions-retired counter.
MHPMCOUNTER3	0xB03	MRW	0x0	A performance-monitoring counter.
MHPMCOUNTER4	0xB04	MRW	0x0	A performance-monitoring counter.
...
MHPMCOUNTER31	0xB1F	MRW	0x0	A performance-monitoring counter.

Table 14.8 lists the U-mode event counters.

Table 14.8: U-mode event counter list

Name	Index	Read/Write	Initial value	Description
CYCLE	0xC00	URO	0x0	The cycle counter.
TIME	0xC01	URO	0x0	The timer.
INSTRET	0xC02	URO	0x0	The instructions-retired counter.
HPMCOUNTER3	0xC03	URO	0x0	A performance-monitoring counter.
HPMCOUNTER4	0xC04	URO	0x0	A performance-monitoring counter.
...
HPMCOUNTER31	0xC1F	URO	0x0	A performance-monitoring counter.
CYCLEH	0xC80	URO	0x0	bits[63:32] of cycle counter, RV32 only
TIMEH	0xC81	URO	0x0	bits[63:32] of timer, RV32 only
INSTRETH	0xC82	URO	0x0	bits[63:32] of instructions- retired counter, RV32 only
HPMCOUNTER3H	0xC83	URO	0x0	bits[63:32] of performance- monitoring counter, RV32 only
HPMCOUNTER4H	0xC84	URO	0x0	bits[63:32] of performance- monitoring counter, RV32 only
...
HPMCOUNTER31H	0xC9F	URO	0x0	bits[63:32] of performance- monitoring counter, RV32 only

Table 14.9: S-mode event counter list

Name	Index	Read/Write	Initial value	Description
SCYCLE	0x5E0	SRO	0x0	The cycle counter.
SINSTRET	0x5E2	SRO	0x0	The instructions-retired counter.
SHPMCOUNTER3	0x5E3	SRO	0x0	A performance-monitoring counter.
SHPMCOUNTER4	0x5E4	SRO	0x0	A performance-monitoring counter.
...
SHPMCOUNTER31	0x5FF	SRO	0x0	A performance-monitoring counter.

The U-mode CYCLE, INSTRET, and HPMCOUNTER_n counters are read-only mappings of the corresponding M-mode event counters. The timer is the read-only mapping of the MTIME register.

The S-mode SCYCLE, SINSTRET, and SHPMCOUNTERn counters are mappings of corresponding M-mode event counters.

This chapter mainly introduces various program examples, including MMU setting, PMP setting, cache setting, synchronization primitive, PLIC setting and PMU setting examples.

15.1 Optimal performance configuration for CPU

Using the following configuration can achieve the optimal performance of C908.

- MHCR = 0x11ff
- MHINT = 0x6e30c
- MCCR2 = 0xe000009 (note: MCCR2 contains settings for RAM delay. In this example, all delays are set to 0. You need to set appropriate RAM delay according to the actual situation.)
- MXSTATUS = 0x638000
- MSMPR = 0x1

```
# mhcr
li x3, 0x11ff
csrs mhcr,x3

#mhint
li x3, 0x6e30c
csrs mhint,x3
```

(continues on next page)

(continued from previous page)

```

# mxstatus
li    x3, 0x638000
csrs  mxstatus,x3

# msmpr
csrsi msmpr,0x1

# mCCR2
li    x3, 0xe0000009
csrs  mCCR2,x3

```

15.2 MMU setting examples

```

/*****

* Function: An example of setting C908MP MMU.
* Memory space: Virtual address <-> physical address.
*
* Pagesize 4K: vpn: {vpn2,vpn1,vpn0} <-> ppn: {ppn2,ppn1,ppn0}
* Pagesize 2M: vpn: {vpn2,vpn1} <-> ppn:{ppn2,ppn1}
* Pagesize 1G: vpn: {vnp2} <-> ppn: {ppn2}
*
*****/

/*C908 will invalidate all MMU TLB entries automatically when reset*/
/*You can use sfence.vma to invalid all MMU TLB entries if necessary*/
sfence.vma x0, x0

/* Pagesize 4K: vpn: {vpn2, vpn1, vpn0} <-> ppn: {ppn2, ppn1, ppn0}*/
/* First-level page addr base: PPN (defined in satp)*/
/* Second-level page addr base: BASE2 (self define)*/
/* Third-level page addr base: BASE3 (self define)*/
/* 1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff
and x3, x3, x4

```

(continues on next page)

(continued from previous page)

```

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{ 44' b BASE2, 10' b1}
↪ */
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44' b BASE2, 10' b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3' b0}, second-level page pte:{ 44' b BASE3, 10'
b1} */
/*Get second-level page addr*/
/* VPN1*/
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, {44' b BASE3, 10' b1}
sd x6, 0(x5)

/*4. Config third-level page*/
/*Third-level page addr: {BASE3, vpn0, 3' b0}, third-level page pte:{
theadflag, ppn2, ppn1, ppn0, 9' b flags, 1' b1} */
/*Get second-level page addr*/
/* VPN0*/
li x4, VPN
li x5, 0x1ff
and x4, x4, x5

```

(continues on next page)

(continued from previous page)

```

srlr x4, x4, 3
/*BASE3*/
li x5, BASE3
srlr x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, {theadflag, ppn2, ppn1, ppn0, 9' b flags, 1' b1}
sd x6, 0(x5)

/* Pagesize 2M: vpn: {vpn2, vpn1} <-> ppn: {ppn2, ppn1}*/
/*First-level page addr base: PPN (defined in satp)*/
/*Second-level page addr base: BASE2 (self define)*/

/*1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff
and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{ 44' b
BASE2, 10' b1}*/
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srlr x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44' b BASE2, 10' b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3' b0}, second-level page pte:{
theadflag, ppn2, ppn1, 9' b0, 9' b flags, 1' b1} */
/*Get second-level page addr*/
/*VPN1*/

```

(continues on next page)

(continued from previous page)

```
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, { theadflag, ppn2, ppn1, 9' b0, 9' b flags,1' b1}
sd x6, 0(x5)

/* Pagesize 1G: vpn: {vpn2} <-> ppn: {ppn2}*/
/*First-level page addr base: PPN (defined in satp)*/
/*1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff
and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{
theadflag, ppn2, 9' b0, 9' b0, 9' b flags,1' b1}*/
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, { theadflag, ppn2, 9' b0, 9' b0, 9' b flags,1' b1}
sd x6, 0(x5)
```

15.3 PMP setting examples

```

/
↳*****
* Function: An example of setting C908MP PMP.
* 0x0 ~ 0xf0000000, TOR 模式, RWX
* 0xf0000000 ~ 0xf8000000, NAPOT 模式, RW
* 0xffff73000 ~ 0xffff74000, NAPOT 模式, RW
* 0xfffc0000 ~ 0xfffc2000, NAPOT 模式, RW
* The four regions above are configured with different execution permissions. In addition,
↳ PMP needs to be configured to prevent the CPU from speculating into unsupported
↳ address regions in different modes, especially in the M-mode with all execution
↳ permissions by default.
Specifically, after configuring the address regions that require execution permissions,
↳ the remaining address regions should be configured with no permissions as shown in the
↳ following example.
*****
↳

# pmpaddr0,0x0 ~ 0xf0000000, TOR mode, read/write/executable permission
li x3, (0xf0000000 >> 2)
csrw pmpaddr0, x3

# pmpaddr1,0xf0000000 ~ 0xf8000000, NAPOT mode, read/write permission
li x3, ( 0xf0000000 >> 2 | (0x8000000-1) >> 3 )
csrw pmpaddr1, x3

# pmpaddr2,0xffff73000 ~ 0xffff74000, NAPOT mode, read/write permission
li x3, ( 0xffff73000 >> 2 | (0x1000-1) >> 3 )
csrw pmpaddr2, x3

# pmpaddr3,0xfffc0000 ~ 0xfffc2000, NAPOT mode, read/write permission
li x3, ( 0xfffc0000 >> 2 | (0x2000-1) >> 3 )
csrw pmpaddr3, x3

```

(continues on next page)

(continued from previous page)

```

# pmpaddr4,0xf0000000 ~ 0x100000000, NAPOT mode, no permission
li x3, ( 0xf0000000 >> 2 | (0x10000000-1) >> 3 )
csrw pmpaddr4, x3

# pmpaddr5,0x100000000 ~ 0xffffffff, TOR NAPOT mode, no permission
li x3, (0xffffffff >> 2)
csrw pmpaddr5, x3

# PMPCFG0, configure execution permissions/mode/lock bit of PMPCFG register for each
↪PMP entries. When lock = 1, this entry is in M-mode.
li x3,0x88989b9b9b8f
csrw pmpcfg0, x3

# pmpaddr5,0x100000000 ~ 0xffffffff, TOR mode. When 0x100000000 <= addr <
↪0xffffffff, pmpaddr5 will be hit. But address space 0xffffffff000 ~ 0xffffffffcan
↪not hit pmpaddr5(The PMP unit of C908 supports the minimum granularity of 4 KB). If
↪you need to mask the last 4KB space of the 1TB space, another NAPOT mode page entry
↪needs to be configured.

```

15.4 Cache setting example

15.4.1 Example of enabling Cache

```

/*C908 will invalidate all I-cache automatically when reset*/
/*You can invalidate I-cache by yourself if necessary*/
/*Invalidate I-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x11
csrs mcor, x3
// You can also use icache instructions to replace the invalidate sequence
// if theadisae is enabled.
//icache.iall
//sync.is

/*Enable I-cache*/
li x3, 0x1
csrs mhcr, x3

```

(continues on next page)

(continued from previous page)

```

/*C908 will invalidate all D-cache automatically when reset*/
/*You can invalidate D-cache by yourself if necessary*/
/*Invalidate D-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x12
csrs mcor, x3

// You can also use dcache instructions to replace the invalidate sequence
// if theadisae is enabled.
// dcache.iall
// sync.is

/*Enable D-cache*/
li x3, 0x2
csrs mhcr, x3

```

15.4.2 Example of synchronization between I-Cache and D-Cache

CPU0

```

sd x3,0(x4) // a new instruction defined in x3
             // is stored to program memory address defined in x4.
dcache.cval1 r0 // clean the new instruction to the shared L2 cache.
sync.s        // ensure completion of clean operation.
             // the dcache clean is not necessarily if INSDC is not enabled.
icache.iva r0 // invalid icache according to shareable configuration.
sync.s/fence.i // ensure completion in all CPUs.
sd x5,0(x6)   // set flag to signal operation completion.
sync.is
jr x4 // jmp to new code

```

CPU1~CPU3

```

WAIT_FINISH:
ld x7,0(x6)
bne x7,x5, WAIT_FINISH // wait CPU0 modification finish.
sync.is
jr x4                  // jmp to new code

```

15.4.3 Example of synchronization between TLB and D-Cache

CPU0

```
sd x4,0(x3) // update a new translation table entry
sync.is/fence.i // ensure completion of update operation.
sfence.vma x5,x0 // invalid the TLB by va
sync.is/fence.i // ensure completion of TLB invalidation and
                // synchronises context
```

15.4.4 Setting of L2 cache partition function

1. Set MCCR2.PAE = 1 to enable partition function.
2. Configure the ML2WP register and set which group each ID is allowed to be placed in. This register is shared by all cores.

For example, pid0 is only allowed to placed in group0, pid1 is only allowed to place in group1:

```
ML2WP[63:56]=8' h80
```

```
ML2WP[55:48]=8' h40
```

...

```
ML2WP[7:0]=8' h01
```

It should be noted that at least one group must be enabled for each id, otherwise the register will show that this id can enabled each group after assignment.

3. Configure the ML2PID register. Each core has one ML2PID, which indicating the PID of the current core. A simple way to configure it is:

```
core0, ML2PID[2:0]=0
```

```
core1, ML2PID[2:0]=1
```

and so on.

Finally, you can decide whether to enable SL2WP and SL2PID by configuring MXSTATUS[9] (SPCE bit).

15.5 Synchronization primitive setting example

CPU0

```
li x1, 0x1
li x6, 0x0
```

(continues on next page)

(continued from previous page)

```

ACQUIRE_LOCK:           // (x3) is the lock address. 0: Free; 1: Busy.
lr x4, 0(x3)             // Read lock
bnez x4, ACQUIRE_LOCK   // Try again if the lock is in use
sc x5, x1, 0(x3)        // Attempt to store new value
bne x6, x5, ACQUIRE_LOCK // Try again if fail
sync.s

...                      // Critical section code

```

CPU1

```

sync.s/fence.i          // Ensure all operations are observed before clearing the lock.
sd x0, 0(x3)            // Clear the lock.

```

15.6 PLIC setting example

```

//Init id 1 machine mode int for hart 0
/*1.set hart threshold if needed*/
li x3, (plic_base_addr + 0x200000) // h0 mthreshold addr
li x4, 0xa //threshold value
sw x4,0x0(x3) // set hart0 threshold as 0xa

/*2.set priority for int id 1*/
li x3, (plic_base_addr + 0x0) // int id 1 prio addr
li x4, 0x1f // prio value
sw x4,0x4(x3) // init id1 priority as 0x1f

/*3.enable m-mode int id1 to hart*/
li x3, (plic_base_addr + 0x2000) // h0 mie0 addr
li x4, 0x2
sw x4,0x0(x3) // enable int id1 to hart0

/*4.set ip or wait external int*/
/*following code set ip*/
li x3, (plic_base_addr + 0x1000) // h0 mthreshold addr
li x4, 0x2 // id 1 pending
sw x4, 0x0(x3) // set int id1 pending

```

(continues on next page)

(continued from previous page)

```

/*5.core enters interrupt handler, read PLIC_CLAIM and get ID*/

/*6.core takes interrupt*/

/*7.core needs to clear external interrupt source if LEVEL(not PULSE)
configured, then core writes ID to PLIC_CLAIM and exits interrupt*/

```

15.7 PMU setting example

```

/*1.inhibit counters counting*/
li x3, 0xffffffff
csrw mcountinhibit, x3

/*2.C908 will initial all pmu counters when reset*/
/*you can initial pmu counters manually if necessarily*/
csrw mcycle, x0
csrw minstret, x0
csrw mhpmcounter3, x0
.....
csrw mhpmcounter31, x0

/*3.configure mhpmevent*/
li x3, 0x1
csrw mhpmevent3, x3 // mhpmcounter3 count event: L1 ICache Access Counter
li x3, 0x2
csrw mhpmevent4, x3 // mhpmcounter4 count event: L1 ICache Miss Counter
.....
li x3, 0x13
csrw mhpmevent21, x3 // mhpmcounter21 count event: L2 Cache write miss Counter

/*4. configure mcounteren and scounteren*/
li x3, 0xffffffff
csrw mcounteren, x3 // enable super mode to read hpmcounter
li x3, 0xffffffff
csrw scounteren, x3 // enable user mode to read hpmcounter

/*5. enable counters to count when you want*/
csrw mcountinhibit, x0

```

Appendix A Standard Instructions

C908 implements the RV64IMAFCB[V] instruction set architecture. The instructions are described in the following sections by instruction set.

16.1 Appendix A-1 I instructions

The following describes the RISC-V I instructions implemented by C908. The instructions are sorted in alphabetic order.

The instructions are 32 bits wide by default. However, in specific cases, the system assembles some instructions into 16-bit compressed instructions. For more information about compressed instructions, see *Appendix A-6 C Instructions*.

16.1.1 ADD: a signed add instruction

Syntax:

add rd, rs1, rs2

Operation:

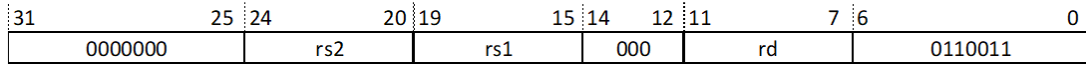
$rd \leftarrow rs1 + rs2$

Permission:

Machine mode (M-mode)/Supervisor mode (S-mode)/User mode (U-mode)

Exception:

None

Instruction format:**16.1.2 ADDI: a signed add immediate instruction****Syntax:**

```
addi rd, rs1, imm12
```

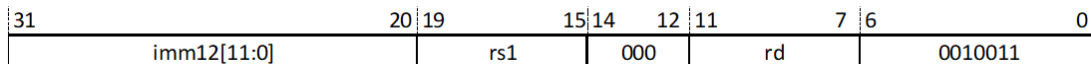
Operation:

$$rd \leftarrow rs1 + \text{sign_extend}(imm12)$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:**16.1.3 ADDIW: a signed add immediate instruction that operates on the lower 32 bits****Syntax:**

```
addiw rd, rs1, imm12
```

Operation:

$$tmp[31:0] \leftarrow rs1[31:0] + \text{sign_extend}(imm12)[31:0]$$

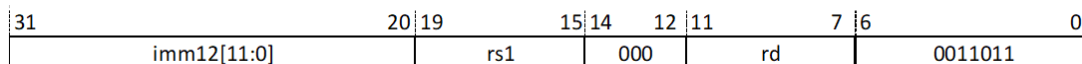
$$rd \leftarrow \text{sign_extend}(tmp[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.4 ADDW: a signed add instruction that operates on the lower 32 bits

Syntax:

addw rd, rs1, rs2

Operation:

$\text{tmp}[31:0] \leftarrow \text{rs1}[31:0] + \text{rs2}[31:0]$

$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$

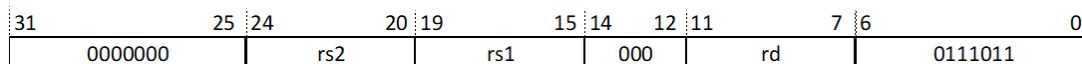
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.5 AND: a bitwise AND instruction

Syntax:

and rd, rs1, rs2

Operation:

$\text{rd} \leftarrow \text{rs1} \& \text{rs2}$

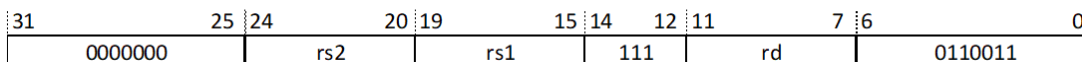
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.6 ANDI: an immediate bitwise AND instruction

Syntax:

```
andi rd, rs1, imm12
```

Operation:

$$rd \leftarrow rs1 \& \text{sign_extend}(imm12)$$

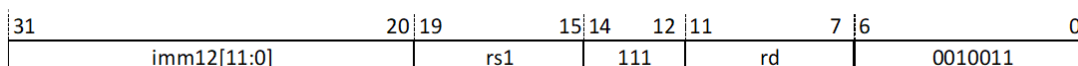
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.7 AUIPC: an instruction that adds the immediate in the upper bits to the PC

Syntax:

```
auipc rd, imm20
```

Operation:

$$rd \leftarrow \text{current pc} + \text{sign_extend}(imm20 \ll 12)$$

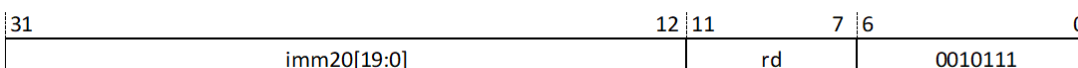
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.8 BEQ: a branch-if-equal instruction

Syntax:

```
beq rs1, rs2, label
```

Operation:

```

if (rs1 == rs2)
    next pc = current pc + sign_extend(imm12 << 1)
else
    next pc = current pc + 4

```

Permission:

M mode/S mode/U mode

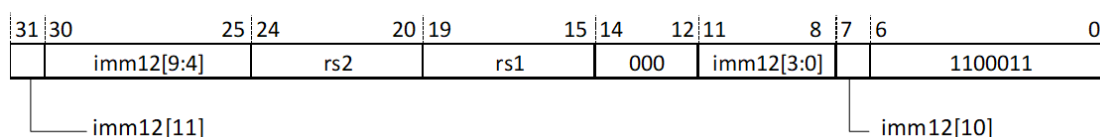
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:**16.1.9 BGE: a signed branch-if-greater-than-or-equal instruction****Syntax:**

```
bge rs1, rs2, label
```

Operation:

```

if (rs1 >= rs2)
    next pc = current pc + sign_extend(imm12 << 1)
else
    next pc = current pc + 4

```

Permission:

M mode/S mode/U mode

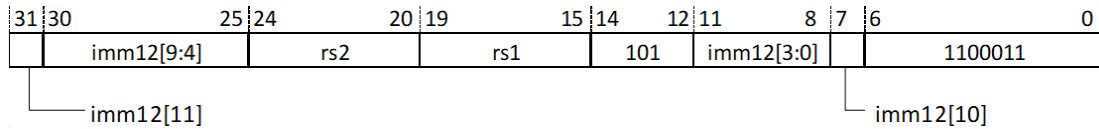
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:**16.1.10 BGEU: an unsigned branch-if-greater-than-or-equal instruction****Syntax:**

```
bgeu rs1, rs2, label
```

Operation:

```
if (rs1 >= rs2)
```

```
    next pc = current pc + sign_extend(imm12<<1)
```

```
else
```

```
    next pc = current pc + 4
```

Permission:

M mode/S mode/U mode

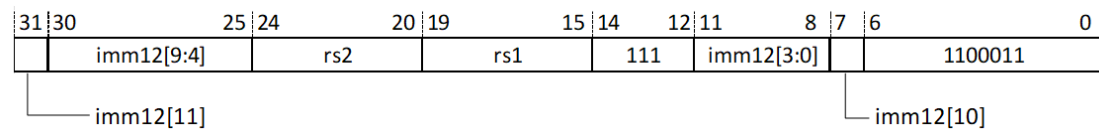
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:

16.1.11 BLT: a signed branch-if-less-than instruction

Syntax:

blt rs1, rs2, label

Operation:

if (rs1 < rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Permission:

M mode/S mode/U mode

Exception:

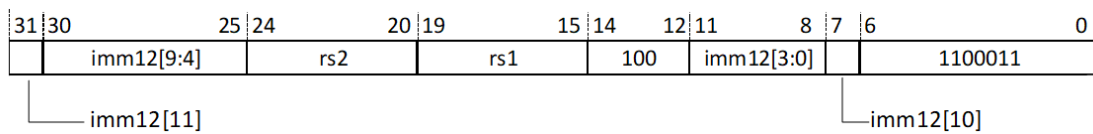
None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:



16.1.12 BLTU: an unsigned branch-if-less-than instruction

Syntax:

bltu rs1, rs2, label

Operation:

if (rs1 < rs2)

next pc = current pc + sign_extend(imm12<<1)

else

next pc = current pc + 4

Permission:

M mode/S mode/U mode

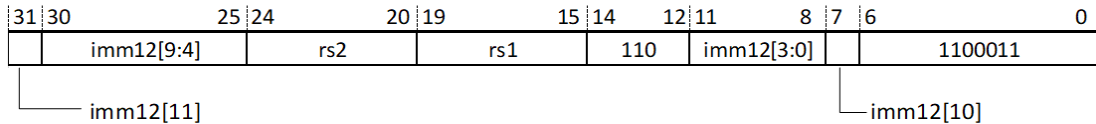
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:**16.1.13 BNE: a branch-if-not-equal instruction****Syntax:**

```
bne rs1, rs2, label
```

Operation:

```
if (rs1 != rs2)
```

```
    next pc = current pc + sign_extend(imm12<<1)
```

```
else
```

```
    next pc = current pc + 4
```

Permission:

```
M mode/S mode/U mode
```

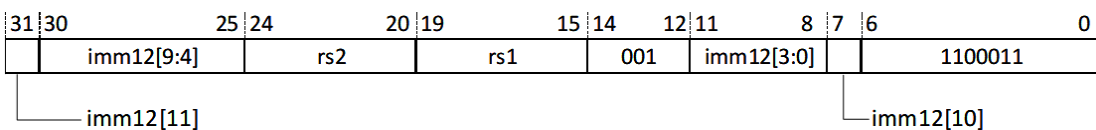
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 4 KB address space.

Instruction format:

16.1.14 CSRRC: a move instruction that clears control registers

Syntax:

```
csrrc rd, csr, rs1
```

Operation:

$$rd \leftarrow csr$$

$$csr \leftarrow csr \& (\sim rs1)$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:

31	20	19	15	14	12	11	7	6	0
csr			rs1		011		rd		1110011

16.1.15 CSRRCI: a move instruction that clears immediates in control registers

Syntax:

```
csrrci rd, csr, imm5
```

Operation:

$$rd \leftarrow csr$$

$$csr \leftarrow csr \& \sim \text{zero_extend}(imm5)$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:

31	20	19	15	14	12	11	7	6	0	
csr			imm5		111		rd		1110011	

16.1.16 CSRRS: a move instruction for setting control registers

Syntax:

`csrrs rd, csr, rs1`

Operation:

$rd \leftarrow csr$

$csr \leftarrow csr \mid rs1$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:

31	20	19	15	14	12	11	7	6	0	
csr			rs1		010		rd		1110011	

16.1.17 CSRRSI: a move instruction for setting immediates in control registers

Syntax:

`csrrsi rd, csr, imm5`

Operation:

$$rd \leftarrow csr$$

$$csr \leftarrow csr \mid \text{zero_extend}(\text{imm5})$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:

31	20	19	15	14	12	11	7	6	0
csr			imm5		110		rd		1110011

16.1.18 CSRRW: a move instruction that reads/writes control registers

Syntax:

$$\text{csrrw } rd, csr, rs1$$
Operation:

$$rd \leftarrow csr$$

$$csr \leftarrow rs1$$
Permission:

M mode/S mode/U mode

Exception:

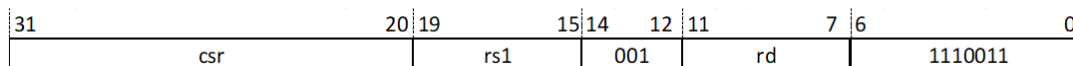
Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:



16.1.19 CSRRWI: a move instruction that reads/writes immediates in control registers

Syntax:

csrrwi rd, csr, imm5

Operation:

$rd \leftarrow csr$

$csr[4:0] \leftarrow imm5$

$csr[63:5] \leftarrow csr[63:5]$

Permission:

M mode/S mode/U mode

Exception:

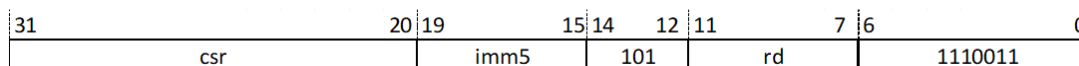
Illegal instruction.

Notes:

Accessible control registers vary under different privileges. For more information, see the descriptions of control registers.

When $rs1 = x0$, this instruction does not initiate write operations and therefore does not cause write-related exceptions.

Instruction format:



16.1.20 EBREAK: a breakpoint instruction

Syntax:

ebreak

Operation:

Generates breakpoint exceptions or enables the core to enter the debug mode.

Permission:

M mode/S mode/U mode

Exception:

Breakpoint exceptions

Instruction format:

31	20	19	15	14	12	11	7	6	0
000000000001		00000		000		00000		1110011	

16.1.21 ECALL: an environment call instruction**Syntax:**

ecall

Operation:

Generates environment call exceptions.

Permission:

M mode/S mode/U mode

Exception:

U-mode, S-mode, and M-mode environment call exceptions

Instruction format:

31	20	19	15	14	12	11	7	6	0
000000000000		00000		000		00000		1110011	

16.1.22 FENCE: a memory synchronization instruction**Syntax:**

fence iorw, iorw

Operation:

Ensures that all memory or device read/write instructions before this instruction are observed earlier than those after this instruction.

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

When the PI and SO bits are both 1, the instruction syntax is fence i,o, and so on.

Instruction format:

31	28	27	26	25	24	23	22	21	20	19	15	14	12	11	7	6	0	
0000				pi	po	pr	pw	si	so	sr	sw	00000			000	00000		0001111

16.1.23 FENCE.I: an instruction stream synchronization instruction

Syntax:

fence.i

Operation:

Clears the I-Cache to ensure that the data access results before this instruction can be accessed by fetch operations after the instruction.

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	28	27	24	23	20	19	15	14	12	11	7	6	0
0000		0000		0000		00000		001	00000		0001111		

16.1.24 JAL: an instruction for directly jumping to a subroutine

Syntax:

jal rd, label

Operation:

next pc \leftarrow current pc + sign_extend(imm20 << 1)

rd \leftarrow current pc + 4

Permission:

M mode/S mode/U mode

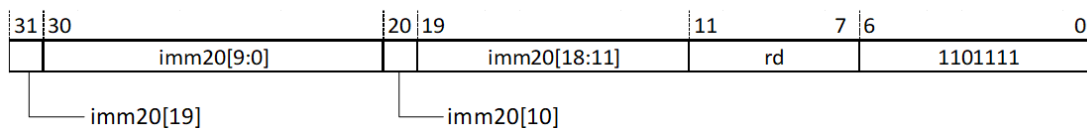
Exception:

None

Notes:

The compiler calculates immediate 20 based on the label.

The jump range of the instruction is ± 1 MB address space.

Instruction format:

16.1.25 JALR: an instruction for jumping to a subroutine by using an address in a register

Syntax:

```
jalr rd, rs1, imm12
```

Operation:

$$\text{next pc} \leftarrow (\text{rs1} + \text{sign_extend}(\text{imm12})) \& 64' \text{ h'ffffffffffffe}$$

$$\text{rd} \leftarrow \text{current pc} + 4$$
Permission:

M mode/S mode/U mode

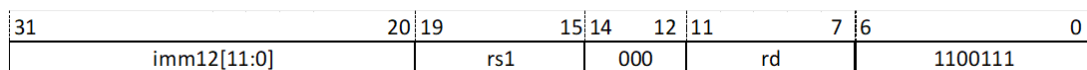
Exception:

None

Notes:

When the CPU runs in M-mode or the MMU is disabled, the jump range of the instruction is the entire 1 TB address space.

When the CPU does not run in M-mode and the MMU is enabled, the jump range of the instruction is the entire 512 GB address space.

Instruction format:

16.1.26 LB: a sign-extended byte load instruction

Syntax:

```
lb rd, imm12(rs1)
```

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{mem}[\text{address}])$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		000		rd		0000011

16.1.27 LBU: an unsign-extended byte load instruction**Syntax:**

lbu rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← zero_extend(mem[address])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		100		rd		0000011

16.1.28 LD: a doubleword load instruction**Syntax:**

ld rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← mem[(address+7):address]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		011		rd		0000011

16.1.29 LH: a sign-extended halfword load instruction**Syntax:**

lh rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← sign_extend(mem[(address+1):address])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		001		rd		0000011

16.1.30 LHU: an unsign-extended halfword load instruction**Syntax:**

lhu rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← zero_extend(mem[(address+1):address])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		101		rd		0000011

16.1.31 LUI: an instruction for loading the immediate in the upper bits

Syntax:

lui rd, imm20

Operation:

$rd \leftarrow \text{sign_extend}(\text{imm20} \ll 12)$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	12	11	7	6	0
imm20[19:0]			rd		0110111

16.1.32 LW: a sign-extended word load instruction

Syntax:

lw rd, imm12(rs1)

Operation:

$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$

$rd \leftarrow \text{sign_extend}(\text{mem}[(\text{address}+3):\text{address}])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		010		rd		0000011

16.1.33 LWU: an unsign-extended word load instruction

Syntax:

lwu rd, imm12(rs1)

Operation:

address \leftarrow rs1 + sign_extend(imm12)

rd \leftarrow zero_extend(mem[(address+3):address])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:

31	20	19	15	14	12	11	7	6	0	
imm12[11:0]			rs1		110		rd		000011	

16.1.34 MRET: an instruction for returning from exceptions in M-mode

Syntax:

mret

Operation:

next pc \leftarrow mepc

mstatus.mie \leftarrow mstatus.mpie

mstatus.mpie \leftarrow 1

Permission:

M mode

Exception:

Illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0011000			00010		00000		000		00000		1110011	

16.1.35 OR: a bitwise OR instruction

Syntax:

or rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \mid rs2$

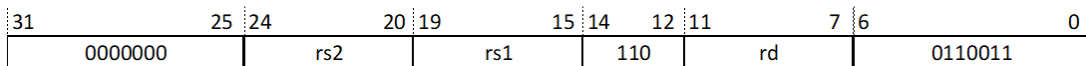
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.36 ORI: an immediate bitwise OR instruction

Syntax:

ori rd, rs1, imm12

Operation:

$rd \leftarrow rs1 \mid \text{sign_extend}(\text{imm12})$

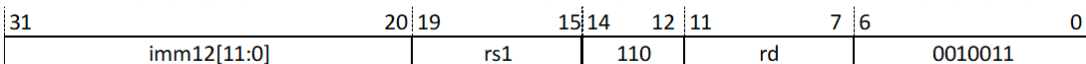
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.37 SB: a byte store instruction

Syntax:

sb rs2, imm12(rs1)

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{mem}[:\text{address}] \leftarrow \text{rs2}[7:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
imm12[11:5]		rs2		rs1		000		imm12[4:0]		0100011	

16.1.38 SD: a doubleword store instruction**Syntax:**

$$\text{sd rs2, imm12(rs1)}$$
Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{mem}[(\text{address}+7):\text{address}] \leftarrow \text{rs2}$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
imm12[11:5]		rs2		rs1		011		imm12[4:0]		0100011	

16.1.39 SFENCE.VMA: a virtual memory synchronization instruction**Syntax:**

$$\text{sfence.vma rs1,rs2}$$
Operation:

Invalidates and synchronizes virtual memory.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

When the TVM bit in the mstatus is 1, running this instruction in S-mode will trigger an illegal instruction exception.

rs1 is the virtual address, and rs2 is the address space identifier (ASID).

- When rs1 and rs2 are both x0, all TLB entries are invalidated.
- When rs1! and rs2 are both x0, all TLB entries that hit the virtual address specified by rs1 are invalidated.
- When rs1 and rs2! are both x0, all TLB entries that hit the process ID specified by rs2 are invalidated.
- When rs1! and rs2! are both x0, all TLB entries that hit the virtual address specified by rs1 and the process ID specified by rs2 are invalidated.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001001			rs2		rs1		000		00000		1110011

16.1.40 SH: a halfword store instruction**Syntax:**

sh rs2, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

mem[(address+1):address] ← rs2[15:0]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
imm12[11:5]			rs2		rs1		001		imm12[4:0]		0100011

16.1.41 SLL: a logical left shift instruction

Syntax:

sll rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \ll rs2[5:0]$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		001		rd		0110011

16.1.42 SLLI: an immediate logical left shift instruction

Syntax:

slli rd, rs1, shamt6

Operation:

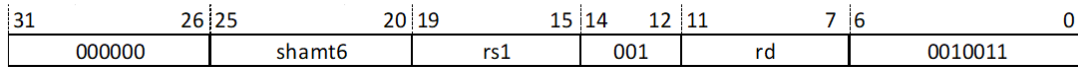
$rd \leftarrow rs1 \ll shamt6$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

16.1.43 SLLIW: an immediate logical left shift instruction that operates on the lower 32 bits

Syntax:

slliw rd, rs1, shamt5

Operation:

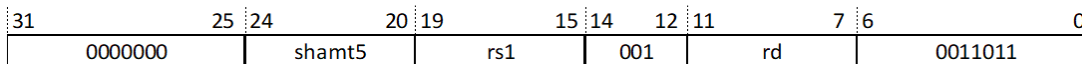
$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] \ll \text{shamt5})[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

16.1.44 SLLW: a logical left shift instruction that operates on the lower 32 bits

Syntax:

sllw rd, rs1, rs2

Operation:

$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] \ll \text{rs2}[4:0])[31:0]$$

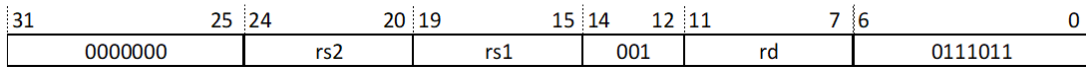
$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.45 SLT: a signed set-if-less-than instruction

Syntax:

slt rd, rs1, rs2

Operation:

if (rs1 < rs2)

rd ← 1

else

rd ← 0

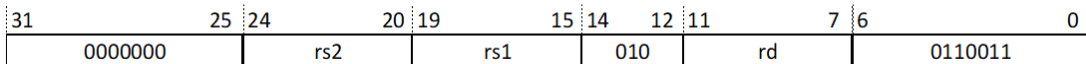
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.46 SLTI: a signed set-if-less-than-immediate instruction

Syntax:

slti rd, rs1, imm12

Operation:

if (rs1 < sign_extend(imm12))

rd ← 1

else

rd ← 0

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1	010	rd	0010011			

16.1.47 SLTIU: an unsigned set-if-less-than-immediate instruction**Syntax:**

```
sltiu rd, rs1, imm12
```

Operation:

```
if (rs1 < zero_extend(imm12))
```

```
    rd ← 1
```

```
else
```

```
    rd ← 0
```

Permission:

```
M mode/S mode/U mode
```

Exception:

None

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1	011	rd	0010011			

16.1.48 SLTU: an unsigned set-if-less-than instruction**Syntax:**

```
sltu rd, rs1, rs2
```

Operation:

```
if (rs1 < rs2)
```

```
    rd ← 1
```

```
else
```

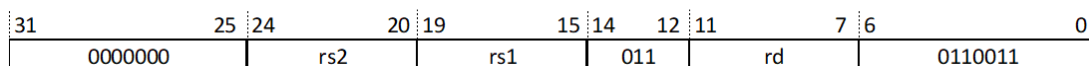
```
    rd ← 0
```

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:**16.1.49 SRA: an arithmetic right shift instruction****Syntax:**

sra rd, rs1, rs2

Operation:

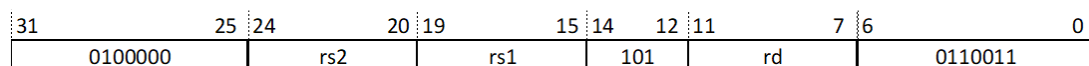
$rd \leftarrow rs1 \ggg rs2[5:0]$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:**16.1.50 SRAI: an immediate arithmetic right shift instruction****Syntax:**

srai rd, rs1, shamt6

Operation:

$rd \leftarrow rs1 \ggg shamt6$

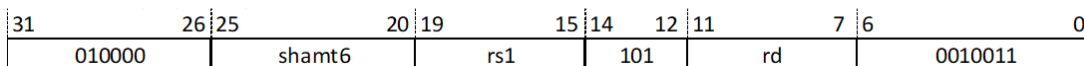
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.51 SLLIW: an immediate arithmetic right shift instruction that operates on the lower 32 bits

Syntax:

sraiw rd, rs1, shamt5

Operation:

$tmp[31:0] \leftarrow (rs1[31:0] \ggg shamt5)[31:0]$

$rd \leftarrow sign_extend(tmp[31:0])$

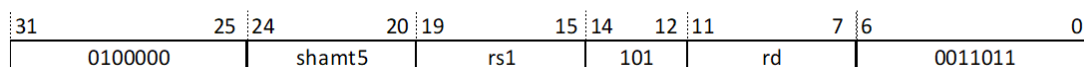
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.52 SRAW: an arithmetic right shift instruction that operates on the lower 32 bits

Syntax:

sraw rd, rs1, rs2

Operation:

$tmp \leftarrow (rs1[31:0] \ggg rs2[4:0])[31:0]$

$rd \leftarrow sign_extend(tmp)$

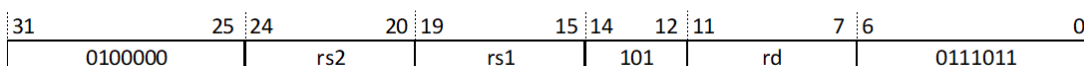
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.53 SRET: an instruction for returning from exceptions in S-mode

Syntax:

sret

Operation:

next pc ← sepc

sstatus.sie ← sstatus.spie

sstatus.spie ← 1

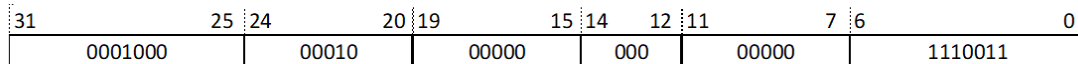
Permission:

S mode

Exception:

Illegal instruction.

Instruction format:



16.1.54 SRL: a logical right shift instruction

Syntax:

srl rd, rs1, rs2

Operation:

rd ← rs1 >> rs2[5:0]

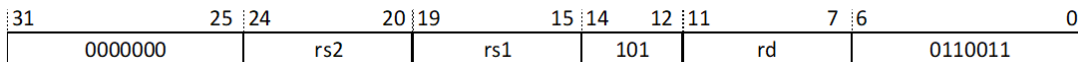
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.1.55 SRLI: an immediate logical right shift instruction

Syntax:

srl rd, rs1, sham6

Operation:

$$rd \leftarrow rs1 \gg \text{shamt6}$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
000000		shamt6			rs1		101		rd		0010011

16.1.56 SRLIW: an immediate logical right shift instruction that operates on the lower 32 bits

Syntax:

$$\text{srlw } rd, rs1, \text{shamt5}$$
Operation:

$$\text{tmp}[31:0] \leftarrow (rs1[31:0] \gg \text{shamt5})[31:0]$$

$$rd \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		shamt5			rs1		101		rd		0011011

16.1.57 SRLW: a logical right shift instruction that operates on the lower 32 bits

Syntax:

$$\text{srlw } rd, rs1, rs2$$
Operation:

$$\text{tmp} \leftarrow (rs1[31:0] \gg rs2[4:0])[31:0]$$

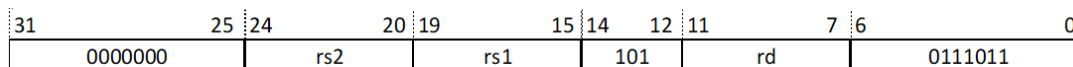
$$rd \leftarrow \text{sign_extend}(\text{tmp})$$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:**16.1.58 SUB: a signed subtract instruction****Syntax:**

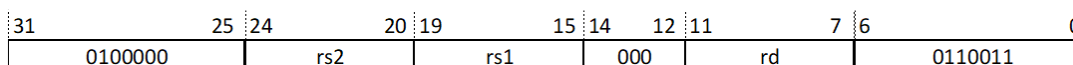
sub rd, rs1, rs2

Operation: $rd \leftarrow rs1 - rs2$ **Permission:**

M mode/S mode/U mode

Exception:

None

Instruction format:**16.1.59 SUBW: a signed subtract instruction that operates on the lower 32 bits****Syntax:**

subw rd, rs1, rs2

Operation: $tmp[31:0] \leftarrow rs1[31:0] - rs2[31:0]$ $rd \leftarrow sign_extend(tmp[31:0])$ **Permission:**

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0100000			rs2		rs1		000		rd		0111011

16.1.60 SW: a word store instruction

Syntax:

sw rs2, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

mem[(address+3):address] ← rs2[31:0]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
imm12[11:5]			rs2		rs1		010		imm12[4:0]		0100011

16.1.61 WFI: an instruction for entering the low power mode

Syntax:

wfi

Operation:

Triggers the CPU to enter the low power mode. In this mode, the CPU clock and most device clocks are disabled.

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001000			00101		00000		000		00000		1110011

16.1.62 XOR: a bitwise XOR instruction

Syntax:

xor rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \wedge rs2$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		100		rd		0110011

16.1.63 XORI: an immediate bitwise XOR instruction

Syntax:

xori rd, rs1, imm12

Operation:

$rd \leftarrow rs1 \& \text{sign_extend}(\text{imm12})$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		100		rd		0010011

16.2 Appendix A-2 M instructions

The following describes the RISC-V M instructions implemented by C908. The instructions are 32 bits wide and sorted in alphabetic order.

16.2.1 DIV: a signed divide instruction

Syntax:

div rd, rs1, rs2

Operation:

$rd \leftarrow rs1 / rs2$

Permission:

Machine mode (M-mode)/Supervisor mode (S-mode)/User mode (U-mode)

Exception:

None

Notes:

When the divisor is 0, the division result is 0xffffffffffff.

When overflow occurs, the division result is 0x8000000000000000.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001			rs2		rs1		100		rd		0110011

16.2.2 DIVU: an unsigned divide instruction

Syntax:

divu rd, rs1, rs2

Operation:

$rd \leftarrow rs1 / rs2$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

When the divisor is 0, the division result is 0xffffffffffff.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001			rs2		rs1		101		rd		0110011

16.2.3 DIVUW: an unsigned divide instruction that operates on the lower 32 bits

Syntax:

```
divuw rd, rs1, rs2
```

Operation:

$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] / \text{rs2}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Notes:

When the divisor is 0, the division result is 0xffffffffffff.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		rs2		rs1		101		rd		0111011	

16.2.4 DIVW: a signed divide instruction that operates on the lower 32 bits

Syntax:

```
divw rd, rs1, rs2
```

Operation:

$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] / \text{rs2}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

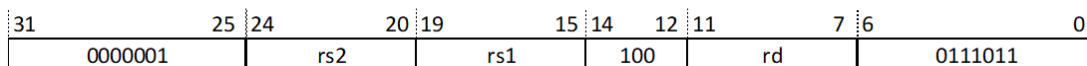
None

Notes:

When the divisor is 0, the division result is 0xffffffffffff.

When overflow occurs, the division result is 0xffffffff80000000.

Instruction format:



16.2.5 MUL: a signed multiply instruction

Syntax:

mul rd, rs1, rs2

Operation:

$rd \leftarrow (rs1 * rs2)[63:0]$

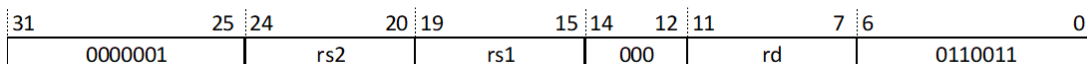
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.2.6 MULH: a signed multiply instruction that extracts the upper bits

Syntax:

mulh rd, rs1, rs2

Operation:

$rd \leftarrow (rs1 * rs2)[127:64]$

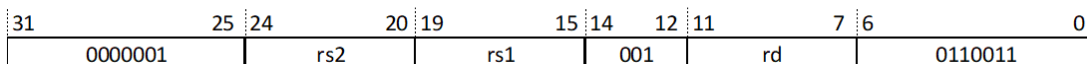
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



16.2.7 MULHSU: a signed-unsigned multiply instruction that extracts the upper bits

Syntax:

mulusu rd, rs1, rs2

Operation:

$$rd \leftarrow (rs1 * rs2)[127:64]$$
Permission:

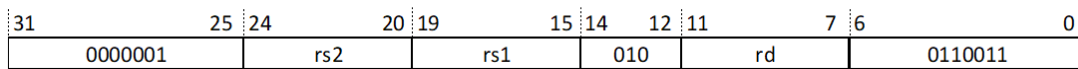
M mode/S mode/U mode

Exception:

None

Notes:

rs1 indicates a signed number, and rs2 indicates an unsigned number.

Instruction format:

16.2.8 MULHU: an unsigned multiply instruction that extracts the upper bits

Syntax:

mulhu rd, rs1, rs2

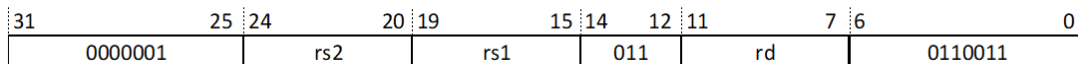
Operation:

$$rd \leftarrow (rs1 * rs2)[127:64]$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

16.2.9 MULW: a signed multiply instruction that operates on the lower 32 bits

Syntax:

mulw rd, rs1, rs2

Operation:

$$\text{tmp} \leftarrow (\text{rs1}[31:0] * \text{rs2}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001			rs2		rs1		000		rd		0111011

16.2.10 REM: a signed remainder instruction**Syntax:**

rem rd, rs1, rs2

Operation:

$$\text{rd} \leftarrow \text{rs1} \% \text{rs2}$$
Permission:

M mode/S mode/U mode

Exception:

None

Notes:

When the divisor is 0, the remainder operation result is the dividend.

When overflow occurs, the remainder operation result is 0x0.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001			rs2		rs1		110		rd		0110011

16.2.11 REMU: an unsigned remainder instruction**Syntax:**

remu rd, rs1, rs2

Operation:

$$\text{rd} \leftarrow \text{rs1} \% \text{rs2}$$

Permission:

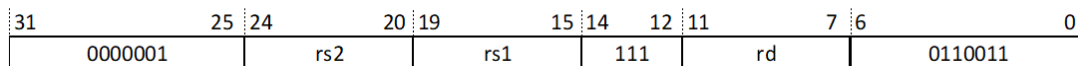
M mode/S mode/U mode

Exception:

None

Notes:

When the divisor is 0, the remainder operation result is the dividend.

Instruction format:

16.2.12 REMUW: an unsigned remainder instruction that operates on the lower 32 bits

Syntax:

remw rd, rs1, rs2

Operation:

$$\text{tmp} \leftarrow (\text{rs1}[31:0] \% \text{rs2}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp})$$
Permission:

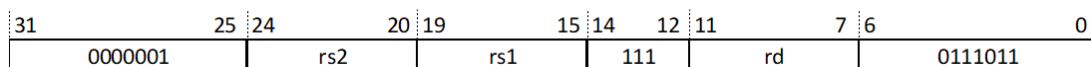
M mode/S mode/U mode

Exception:

None

Notes:

When the divisor is 0, the remainder operation result is obtained by extending the signed bit [31] of the dividend.

Instruction format:

16.2.13 REMW: a signed remainder instruction that operates on the lower 32 bits

Syntax:

remw rd, rs1, rs2

Operation:

$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] \% \text{rs2}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

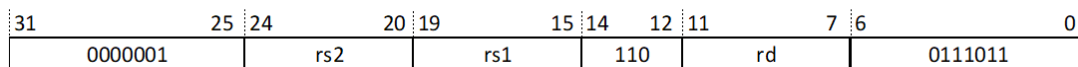
Exception:

None

Notes:

When the divisor is 0, the remainder operation result is obtained by extending the signed bit [31] of the dividend.

When overflow occurs, the remainder operation result is 0x0.

Instruction format:

16.3 Appendix A-3 A instructions

The following describes the RISC-V A instructions implemented by C908. The instructions are 32 bits wide and sorted in alphabetic order.

16.3.1 AMOADD.D: an atomic add instruction

Syntax:

$$\text{amoadd.d.aqrl rd, rs2, (rs1)}$$
Operation:

$$\text{rd} \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}]$$

$$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}] + \text{rs2}$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is `amoadd.d rd, rs2, (rs1)`.
- When aq is 0 and rl is 1, the corresponding assembler instruction is `amoadd.d.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is `amoadd.d.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is `amoadd.d.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
00000			aq	rl	rs2		rs1		011		rd		0101111	

16.3.2 AMOADD.W: an atomic add instruction that operates on the lower 32 bits**Syntax:**

`amoadd.w.aqrl rd, rs2, (rs1)`

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+3:\text{rs1}] + \text{rs2}[31:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `amoadd.w rd, rs2, (rs1)`.
- When `aq` is 0 and `rl` is 1, the corresponding assembler instruction is `amoadd.w.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When `aq` is 1 and `rl` is 0, the corresponding assembler instruction is `amoadd.w.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When `aq` and `rl` are both 1, the corresponding assembler instruction is `amoadd.w.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00000		aq	rl	rs2		rs1		010		rd		0101111	

16.3.3 AMOAND.D: an atomic bitwise AND instruction**Syntax:**

```
amoand.d.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← mem[rs1+7:rs1]
mem[rs1+7:rs1] ← mem[rs1+7:rs1] & rs2
```

Permission: M mode/S mode/U mode**Exception:**

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The `aq` and `rl` bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `amoand.d rd, rs2, (rs1)`.
- When `aq` is 0 and `rl` is 1, the corresponding assembler instruction is `amoand.d.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.

- When `aq` is 1 and `rl` is 0, the corresponding assembler instruction is `amoand.d.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When `aq` and `rl` are both 1, the corresponding assembler instruction is `amoand.d.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
01100			aq	rl	rs2		rs1		011		rd		0101111	

16.3.4 AMOAND.W: an atomic bitwise AND instruction that operates on the lower 32 bits

Syntax:

```
amoand.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+3:\text{rs1}] \& \text{rs2}[31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

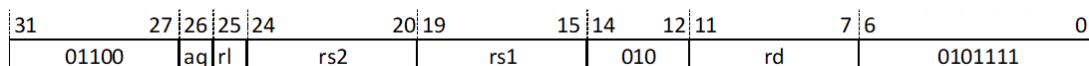
None

Notes:

The `aq` and `rl` bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `amoand.w rd, rs2, (rs1)`.
- When `aq` is 0 and `rl` is 1, the corresponding assembler instruction is `amoand.w.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.

- When `aq` is 1 and `rl` is 0, the corresponding assembler instruction is `amoand.w.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When `aq` and `rl` are both 1, the corresponding assembler instruction is `amoand.w.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:**16.3.5 AMOMAX.D: an atomic signed MAX instruction****Syntax:**

`amomax.d.aqrl rd, rs2, (rs1)`

Operation:

$rd \leftarrow \text{mem}[rs1+7:rs1]$

$\text{mem}[rs1+7:rs1] \leftarrow \max(\text{mem}[rs1+7:rs1], rs2)$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The `aq` and `rl` bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `amomax.d rd, rs2, (rs1)`.
- When `aq` is 0 and `rl` is 1, the corresponding assembler instruction is `amomax.d.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When `aq` is 1 and `rl` is 0, the corresponding assembler instruction is `amomax.d.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.

- When aq and rl are both 1, the corresponding assembler instruction is amomax.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10100		aq	rl	rs2		rs1		011		rd		0101111	

16.3.6 AMOMAX.W: an atomic signed MAX instruction that operates on the lower 32 bits

Syntax:

```
amomax.w.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← sign_extend( mem[rs1+3: rs1] )
mem[rs1+3:rs1] ← max(mem[rs1+3:rs1], rs2[31:0])
```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amomax.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amomax.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amomax.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amomax.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.

tion is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10100		aq	rl	rs2		rs1		010		rd		0101111	

16.3.7 MOMAXU.DA: an atomic unsigned MAX instruction

Syntax:

amomaxu.d.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{mem}[rs1+7:rs1]$

$\text{mem}[rs1+7:rs1] \leftarrow \max(\text{mem}[rs1+7:rs1], rs2)$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amomaxu.d rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amomaxu.d.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amomaxu.d.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amomaxu.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
11100				aq	rl	rs2		rs1		011	rd		0101111	

16.3.8 AMOMAXU.W: an atomic unsigned MAX instruction that operates on the lower 32 bits.

Syntax:

amomaxu.w.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{zero_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{max}(\text{mem}[\text{rs1}+3:\text{rs1}], \text{rs2}[\text{31:0}])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amomaxu.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amomaxu.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amomaxu.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amomaxu.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
11100				aq	rl	rs2		rs1		010	rd		0101111	

16.3.9 AMOMIN.D: an atomic signed MIN instruction

Syntax:

```
amomin.d.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← mem[rs1+7:rs1]
mem[rs1+7:rs1] ← min(mem[rs1+7:rs1],rs2)
```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amomin.d rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amomin.d.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amomin.d.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amomin.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000		aq	rl	rs2		rs1		011		rd		0101111	

16.3.10 AMOMIN.W: an atomic signed MIN instruction that operates on the lower 32 bits

Syntax:

```
amomin.w.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← sign_extend(mem[rs1+3: rs1])
mem[rs1+3:rs1] ← min(mem[rs1+3:rs1], rs2[31:0])
```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amomin.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amomin.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amomin.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amomin.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000		aq	rl	rs2		rs1		010		rd		0101111	

16.3.11 AMOMINU.D: an atomic unsigned MIN instruction

Syntax:

```
amominu.d.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{mem}[rs1+7:rs1]$$

$$\text{mem}[rs1+7:rs1] \leftarrow \min(\text{mem}[rs1+7:rs1], rs2)$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is `amominu.d rd, rs2, (rs1)`.
- When aq is 0 and rl is 1, the corresponding assembler instruction is `amominu.d.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is `amominu.d.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is `amominu.d.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11000		aq	rl	rs2		rs1		011		rd		0101111	

16.3.12 AMOMINU.W: an atomic unsigned MIN instruction that operates on the lower 32 bits

Syntax:

```
amominu.w.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← sign_extend(mem[rs1+3: rs1])
mem[rs1+3:rs1] ← min(mem[rs1+3:rs1], rs2[31:0])
```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amominu.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amominu.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amominu.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amominu.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11000		aq	rl	rs2		rs1		010		rd		0101111	

16.3.13 AMOOR.D: an atomic bitwise OR instruction.

Syntax:

amoor.d.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{mem}[rs1+7:rs1]$

$\text{mem}[rs1+7:rs1] \leftarrow \text{mem}[rs1+7:rs1] \mid rs2$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amoor.d rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amoor.d.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amoor.d.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amoor.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
01000			aq	rl	rs2			rs1			011	rd	0101111	

16.3.14 AMOOR.W: an atomic bitwise OR instruction that operates on the lower 32 bits

Syntax:

```
amoor.w.aqrl rd, rs2, (rs1)
```

Operation:

```
rd ← sign_extend(mem[rs1+3: rs1])
mem[rs1+3:rs1] ← mem[rs1+3:rs1] | rs2[31:0]
```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

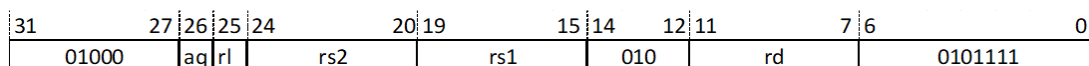
Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amoor.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amoor.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amoor.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amoor.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:**16.3.15 AMOSWAP.D: an atomic swap instruction****Syntax:**

```
amoswap.d.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{mem}[rs1+7: rs1]$$

$$\text{mem}[rs1+7:rs1] \leftarrow rs2$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits: None

Notes:

The *aq* and *rl* bits determine the sequences of executing the memory access instructions before and after this instruction.

- When *aq* and *rl* are both 0, the corresponding assembler instruction is `amoswap.d rd, rs2, (rs1)`.
- When *aq* is 0 and *rl* is 1, the corresponding assembler instruction is `amoswap.d.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When *aq* is 1 and *rl* is 0, the corresponding assembler instruction is `amoswap.d.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When *aq* and *rl* are both 1, the corresponding assembler instruction is `amoswap.d.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00001		aq	rl	rs2		rs1		011		rd		0101111	

16.3.16 AMOSWAP.W: an atomic swap instruction that operates on the lower 32 bits**Syntax:**

$$\text{amoswap.w.aqrl rd, rs2, (rs1)}$$
Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+3: rs1])$$

$$\text{mem}[rs1+3:rs1] \leftarrow rs2[31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits: None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is `amoswap.w rd, rs2, (rs1)`.
- When aq is 0 and rl is 1, the corresponding assembler instruction is `amoswap.w.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is `amoswap.w.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is `amoswap.w.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00001		aq	rl	rs2		rs1		010		rd		0101111	

16.3.17 AMOXOR.D: an atomic bitwise XOR instruction**Syntax:**

`amoxor.d.aqrl rd, rs2, (rs1)`

Operation:

$rd \leftarrow \text{mem}[rs1+7:rs1]$

$\text{mem}[rs1+7:rs1] \leftarrow \text{mem}[rs1+7:rs1] \hat{=} rs2$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is amoxor.d rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is amoxor.d.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is amoxor.d.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is amoxor.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		aq	rl	rs2		rs1		011		rd		0101111	

16.3.18 AMOXOR.W: an atomic bitwise XOR instruction that operates on the lower 32 bits

Syntax:

```
amoxor.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+3:\text{rs1}] \hat{=} \text{rs2}[31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The `aq` and `rl` bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `amoxor.w rd, rs2, (rs1)`.
- When `aq` is 0 and `rl` is 1, the corresponding assembler instruction is `amoxor.w.rl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When `aq` is 1 and `rl` is 0, the corresponding assembler instruction is `amoxor.w.aq rd, rs2, (rs1)`. All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When `aq` and `rl` are both 1, the corresponding assembler instruction is `amoxor.w.aqrl rd, rs2, (rs1)`. Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		aq	rl	rs2		rs1		010		rd		0101111	

16.3.19 LR.D: a doubleword load-reserved instruction

Syntax:

`lr.d.aqrl rd, (rs1)`

Operation:

$rd \leftarrow \text{mem}[rs1+7: rs1]$

`mem[rs1+7:rs1]` is reserved

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The `aq` and `rl` bits determine the sequences of executing the memory access instructions before and after this instruction.

- When `aq` and `rl` are both 0, the corresponding assembler instruction is `lr.d rd, (rs1)`.

- When aq is 0 and rl is 1, the corresponding assembler instruction is lr.d.rl rd, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is lr.d.aq rd, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is lr.d.aqrl rd, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00010			aq	rl	00000			rs1	011		rd	0101111	

16.3.20 LR.W: a word load-reserved instruction**Syntax:**

lr.w.aqrl rd, (rs1)

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3: \text{rs1}])$

mem[rs1+3:rs1] is reserved

Permission:

M mode/S mode/U mode

Exception: Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits: None

Notes: The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is lr.w rd, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is lr.w.rl rd, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is lr.w.aq rd, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is lr.w.aqrl rd, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00010			aq	rl	00000			rs1	010		rd	0101111	

16.3.21 SC.D: a doubleword store-conditional instruction

Syntax:

sc.d.aqrl rd, rs2, (rs1)

Operation:

If(mem[rs1+7:rs1] is reserved)

mem[rs1+7: rs1] \leftarrow rs2

rd \leftarrow 0

else

rd \leftarrow 1

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

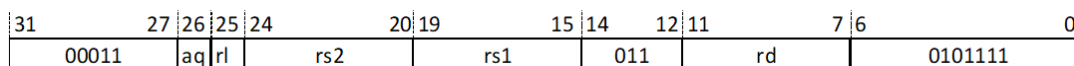
None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is sc.d rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is sc.d.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is sc.d.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is sc.d.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:



16.3.22 SC.W: a word store-conditional instruction

Syntax:

sc.w.aqrl rd, rs2, (rs1)

Operation:

if(mem[rs1+3:rs1] is reserved)

mem[rs1+3:rs1] ← rs2[31:0]

rd ← 0

else

rd ← 1

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

None

Notes:

The aq and rl bits determine the sequences of executing the memory access instructions before and after this instruction.

- When aq and rl are both 0, the corresponding assembler instruction is sc.w rd, rs2, (rs1).
- When aq is 0 and rl is 1, the corresponding assembler instruction is sc.w.rl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed.
- When aq is 1 and rl is 0, the corresponding assembler instruction is sc.w.aq rd, rs2, (rs1). All memory access instructions after this instruction can be executed only after execution of this instruction is completed.
- When aq and rl are both 1, the corresponding assembler instruction is sc.w.aqrl rd, rs2, (rs1). Results of all memory access instructions before this instruction must be observed before this instruction is executed, and all memory access instructions after this instruction can be executed only after execution of this instruction is completed.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
00011			aq	rl	rs2		rs1		010		rd		0101111	

16.4 Appendix A-4 F instructions

The following describes the RISC-V F instructions implemented by C908. The instructions are 32 bits wide and sorted in alphabetic order.

For single-precision floating-point instructions, if the upper 32 bits in the source register are not all 1, the single-precision data is treated as qNaN.

When the fs bit in the mstatus register is 2' b00, running any instruction listed in this appendix will trigger an illegal instruction exception. When the fs bit in the mstatus register is not 2' b00, it is set to 2' b11 after any instruction listed in this appendix is executed.

16.4.1 FADD.S: a single-precision floating-point add instruction

Syntax:

fadd.s fd, fs1, fs2, rm

Operation:

frd \leftarrow fs1 + fs2

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is fadd.s fd, fs1, fs2, rne.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is fadd.s fd, fs1, fs2, rtz.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is fadd.s fd, fs1, fs2, rdn.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fadd.s fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fadd.s fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the floating-point control and status register (FCSR), `fcsr`. The corresponding assembler instruction is `fadd.s fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			fs2		fs1		rm		fd		1010011

16.4.2 FCLASS.S: a single-precision floating-point classify instruction**Syntax:**

```
fclass.s rd, fs1
```

Operation:

```
if ( fs1 = -inf)
```

```
    rd ← 64' h1
```

```
if ( fs1 = -norm)
```

```
    rd ← 64' h2
```

```
if ( fs1 = -subnorm)
```

```
    rd ← 64' h4
```

```
if ( fs1 = -zero)
```

```
    rd ← 64' h8
```

```
if ( fs1 = +zero)
```

```
    rd ← 64' h10
```

```
if ( fs1 = +subnorm)
```

```
    rd ← 64' h20
```

```
if ( fs1 = +norm)
```

```
    rd ← 64' h40
```

```
if ( fs1 = +Inf)
```



```

rd ← 64' h80
if ( fs1 = sNaN)
    rd ← 64' h100
if ( fs1 = qNaN)
    rd ← 64' h200

```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1110000			00000		fs1		001		rd		1010011

16.4.3 FCVT.L.S: an instruction that converts a single-precision floating-point number into a signed long integer

Syntax:

```
fcvt.l.s rd, fs1, rm
```

Operation:

```
rd ← single_convert_to_signed_long(fs1)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.l.s rd, fs1, rne`.

- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.l.s rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.l.s rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.l.s rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.l.s rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.l.s rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100000			00010		fs1		rm		rd		1010011

16.4.4 FCVT.LU.S: an instruction that converts a single-precision floating-point number into an unsigned long integer

Syntax:

```
fcvt.lu.s rd, fs1, rm
```

Operation:

```
rd ← single_convert_to_unsigned_long(fs1)
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

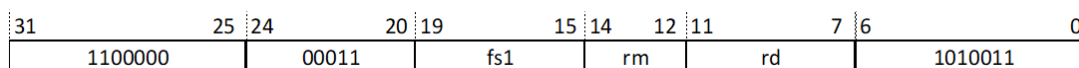
Floating-point status bits `NV` and `NX`

Notes:

`RM` determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.lu.s rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.lu.s rd, fs1, rtz`.

- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.lu.s rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.lu.s rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.lu.s rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.lu.s rd, fs1`.

Instruction format:

16.4.5 FCVT.S.L: an instruction that converts a signed long integer into a single-precision floating-point number

Syntax:

```
fcvt.s.l fd, rs1, rm
```

Operation:

```
fd ← signed_long_convert_to_single(fs1)
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.s.l fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.s.l fd, rs1, rtz`.

- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.s.l fd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.s.l fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.s.l fd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.s.l fd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101000			00010		rs1		rm		fd		1010011

16.4.6 FCVT.S.LU: an instruction that converts an unsigned long integer into a single-precision floating-point number

Syntax:

```
fcvt.s.l fd, fs1, rm
```

Operation:

```
fd ← unsigned_long_convert_to_single_fp(fs1)
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.s.lu fd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.s.lu fd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.s.lu fd, fs1, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.s.lu fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.s.lu fd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.s.lu fd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101000			00011		<code>rs1</code>	<code>rm</code>		<code>fd</code>	1010011		

16.4.7 FCVT.S.W: an instruction that converts a signed integer into a single-precision floating-point number

Syntax:

```
fcvt.s.w fd, rs1, rm
```

Operation:

```
fd ← signed_int_convert_to_single(fs1)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit `NX`

Notes:

`RM` determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.s.w fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.s.w fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.s.w fd, rs1, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.s.w fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.s.w fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.s.w fd, rs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101000			00000		rs1		rm		fd		1010011

16.4.8 FCVT.S.WU: an instruction that converts an unsigned integer into a single-precision floating-point number

Syntax:

```
fcvt.s.wu fd, rs1, rm
```

Operation:

```
fd ← unsigned_int_convert_to_single_fp(fs1)
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.s.wu fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.s.wu fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.s.wu fd, rs1, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.s.wu fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.s.wu fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.s.wu fd, rs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101000			00001		rs1		rm		fd		1010011

16.4.9 FCVT.W.S: an instruction that converts a single-precision floating-point number into a signed integer

Syntax:

```
fcvt.w.s rd, fs1, rm
```

Operation:

```
tmp ← single_convert_to_signed_int(fs1)
rd ← sign_extend(tmp)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.w.s rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.w.s rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.w.s rd, fs1, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.w.s rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.w.s rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.w.s rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100000			00000		fs1		rm		rd		1010011

16.4.10 FCVT.WU.S: an instruction that converts a single-precision floating-point number into an unsigned integer

Syntax:

```
fcvt.wu.s rd, fs1, rm
```

Operation:

```
tmp ← single_convert_to_unsigned_int(fs1)
rd ← sign_extend(tmp)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.wu.s rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.wu.s rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.wu.s rd, fs1, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.wu.s rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.wu.s rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.wu.s rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100000			00001		fs1		rm		rd		1010011

16.4.11 FDIV.S: a single-precision floating-point divide instruction**Syntax:**

`fdiv.s fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 / fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2, rup`.

- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fdiv.s fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001100			fs1		fs2		rm		fd		1010011

16.4.12 FEQ.S: a single-precision floating-point compare equal instruction**Syntax:**

`feq.s rd, fs1, fs2`

Operation:

if(`fs1 == fs2`)

`rd` ← 1

else

`rd` ← 0

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1010000			fs2		fs1		010		rd		1010011

16.4.13 FLE.S: a single-precision floating-point compare less than or equal to instruction**Syntax:**

`fle.s rd, fs1, fs2`

Operation:

```

if(fs1 <= fs2)
    rd ← 1
else
    rd ← 0

```

Permission:

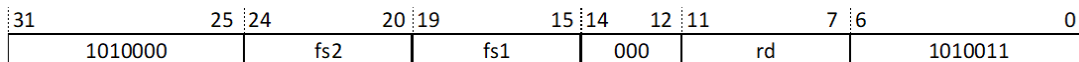
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

16.4.14 FLT.S: a single-precision floating-point compare less than instruction

Syntax:

flt.s rd, fs1, fs2

Operation:

```

if(fs1 < fs2)
    rd ← 1
else
    rd ← 0

```

Permission:

M mode/S mode/U mode

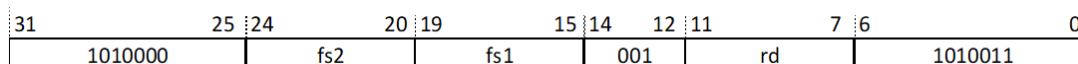
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



16.4.15 FLW: a single-precision floating-point load instruction

Syntax:

flw fd, imm12(rs1)

Operation:

address \leftarrow rs1 + sign_extend(imm12)

fd[31:0] \leftarrow mem[(address+3):address]

fd[63:32] \leftarrow 32' hfffffff

Permission:

M mode/S mode/U mode

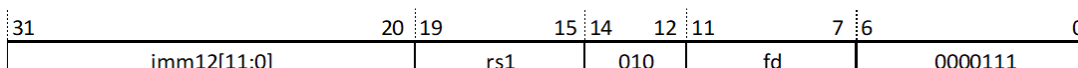
Exception:

Unaligned access, access error, page error, or Illegal instruction.

Affected flag bits:

None

Instruction format:



16.4.16 FMADD.S: a single-precision floating-point multiply-add instruction

Syntax:

fmadd.s fd, fs1, fs2, fs3, rm

Operation:

rd \leftarrow fs1 * fs2 + fs3

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmadd.s fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			00		fs2		fs1		rm		fd		1000011

16.4.17 FMAX.S: a single-precision floating-point MAX instruction**Syntax:**

```
fmmax.s fd, fs1, fs2
```

Operation:

```
if(fs1 >= fs2)
```

```
    fd ← fs1
```

```
else
```

```
    fd ← fs2
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0010100		fs2		fs1		001		fd		1010011	

16.4.18 FMIN.S: a single-precision floating-point MIN instruction

Syntax:

fmin.s fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs2

else

fd ← fs1

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0010100		fs2		fs1		000		fd		1010011	

16.4.19 FMSUB.S: a single-precision floating-point multiply-subtract instruction

Syntax:

fmsub.s fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 - fs3

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			00		fs2		fs1		rm		fd		1000111

16.4.20 FMUL.S: a single-precision floating-point multiply instruction**Syntax:**

`fmul.s fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 * fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmul.s fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmul.s fd, fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmul.s fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmul.s fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmul.s fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmul.s fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001000			fs2		fs1		rm		fd		1010011

16.4.21 FMV.W.X: a single-precision floating-point write move instruction**Syntax:**

`fmv.w.x fd, rs1`

Operation:

$fd[31:0] \leftarrow rs[31:0]$

$fd[63:32] \leftarrow 32' \text{ hfffffff}$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
1111000			00000		rs1		000		fd		1010011	

16.4.22 FMV.X.H: a single-precision floating-point read move instruction

Syntax:

fmv.x.w rd, fs1

Operation:tmp[31:0] \leftarrow fs1[31:0]rd \leftarrow sign_extend(tmp[31:0])**Permission:**

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
1110000			00000		fs1		000		rd		1010011	

16.4.23 FNMADD.S: a single-precision floating-point negate-(multiply-add) instruction

Syntax:

fnmadd.s fd, fs1, fs2, fs3, rm

Operation:fd \leftarrow -(fs1*fs2 + fs3)**Permission:**

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fnmadd.s fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			00		fs2		fs1		rm		fd		1001111

16.4.24 FNMSUB.S: a single-precision floating-point negate-(multiply-subtract) instruction

Syntax:

`fnmsub.s fd, fs1, fs2, fs3, rm`

Operation:

$fd \leftarrow -(fs1 * fs2 - fs3)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmsub.s fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3		00		fs2		fs1		rm		fd		1001011	

16.4.25 FSGNJ.S: a single-precision floating-point sign-injection instruction

Syntax:

`fsgnj.s fd, fs1, fs2`

Operation:

$fd[30:0] \leftarrow fs1[30:0]$

$fd[31] \leftarrow fs2[31]$

$fd[63:32] \leftarrow 32' \text{ hfffffff}$

Permission:

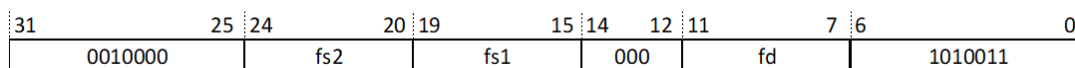
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

16.4.26 FSGNJN.S: a single-precision floating-point negate sign-injection instruction

Syntax:

fsgnjn.s fd, fs1, fs2

Operation: $fd[30:0] \leftarrow fs1[30:0]$ $fd[31] \leftarrow ! fs2[31]$ $fd[63:32] \leftarrow 32' \text{ hfffffff}$ **Permission:**

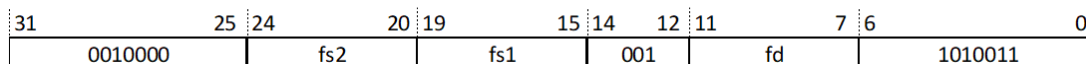
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

16.4.27 FSGNJX.S: a single-precision floating-point XOR sign-injection instruction

Syntax:

fsgnjx.s fd, fs1, fs2

Operation: $fd[30:0] \leftarrow fs1[30:0]$ $fd[31] \leftarrow fs1[31] \wedge fs2[31]$ $fd[63:32] \leftarrow 32' \text{ hfffffff}$ **Permission:**

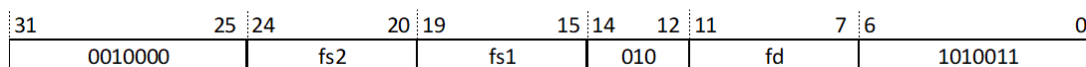
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

16.4.28 FSQRT.S: a single-precision floating-point square-root instruction

Syntax:

fsqrt.s fd, fs1, rm

Operation:

$fd \leftarrow \text{sqrt}(fs1)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is fsqrt.s fd, fs1, rne.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is fsqrt.s fd, fs1, rtz.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is fsqrt.s fd, fs1, rdn.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is fsqrt.s fd, fs1, rup.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is fsqrt.s fd, fs1, rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.

- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is `fsqrt.s fd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0		
0101100			00000			fs1		rm		fd		1010011	

16.4.29 FSUB.S: a single-precision floating-point subtract instruction**Syntax:**

`fsub.s fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 - fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fsub.f, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fsub.s fd, fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fsub.s fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fsub.s fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fsub.s fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is `fsub.s fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000100			fs2		fs1		rm	fd		1010011	

16.4.30 FSW: a single-precision floating-point store instruction

Syntax:

fsw fs2, imm12(rs1)

Operation:

address \leftarrow rs1 + sign_extend(imm12)

mem[(address+31):address] \leftarrow fs2[31:0]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
imm12[11:5]			fs2		rs1		010		imm12[4:0]		0100111

16.5 Appendix A-6 C Instructions

This section describes RISC-V C instructions implemented by C908. The instructions are 16 bits wide and sorted in alphabetic order.

16.5.1 C.ADD: a signed add instruction

Syntax:

c.add rd, rs2

Operation:

rd \leftarrow rs1 + rs2

Permission:

M mode/S mode/U mode

Exception:

None

Notes: $rs1 = rd \neq 0$ $rs2 \neq 0$ **Instruction format:**

15	13	12	11	7	6	2	1	0
100		1	rs1/rd	rs2		10		

16.5.2 C.ADDI: a signed add immediate instruction**Syntax:**

c.addi rd, nzimm6

Operation: $rd \leftarrow rs1 + \text{sign_extend}(nzimm6)$ **Permission:**

M mode/S mode/U mode

Exception:

None

Notes: $rs1 = rd \neq 0$ $nzimm6 \neq 0$ **Instruction format:**

15	13	12	11	7	6	2	1	0
000			rs1/rd	nzimm6[4:0]		01		

└─ nzimm6[5]

16.5.3 C.ADDIW: an add immediate instruction that operates on the lower 32 bits**Syntax:**

c.addiw rd, imm6

Operation:

Chapter 16. Appendix A Standard Instructions

$$\text{tmp}[31:0] \leftarrow \text{rs1}[31:0] + \text{sign_extend}(\text{imm6})$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$

Permission:

M mode/S mode/U mode

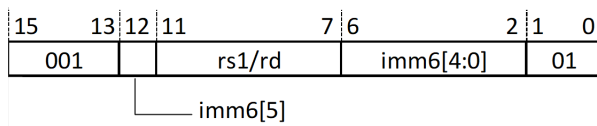
Exception:

None

Notes:

$$\text{rs1} = \text{rd} \neq 0$$

Instruction format:



16.5.4 C.ADDI4SPN: an instruction that adds an immediate scaled by 4 to the stack pointer

Syntax:

$$\text{c.addi4spn rd, sp, nzuimm8} \ll 2$$

Operation:

$$\text{rd} \leftarrow \text{sp} + \text{zero_extend}(\text{nzuimm8} \ll 2)$$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

$$\text{nzuimm8} \neq 0$$

Typical rd code registers are:

- 000 x8
- 001 x9
- 010 x10
- 011 x11

- 100 x12
- 101 x13
- 110 x14
- 111 x15

Instruction format:

15	13	12		5	4	2	1	0
000		nzuimm8[3:2 7:4 0 1]			rd		00	

16.5.5 C.ADDI16SP: an instruction that adds an immediate scaled by 16 to the stack pointer

Syntax:

c.addi16sp sp, nzuimm6<<4

Operation:

$sp \leftarrow sp + \text{sign_extend}(nzuimm6 \ll 4)$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

15	13	12	11		7	6		2	1	0
011		00010						01		
nzuimm6[5]					nzuimm6[0 2 4:3 1]					

16.5.6 C.ADDW: a signed add instruction that operates on the lower 32 bits

Syntax:

c.addw rd, rs2

Operation:

$\text{tmp}[31:0] \leftarrow \text{rs1}[31:0] + \text{rs2}[31:0]$

$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

$$rs1 = rd$$

Typical rd/rs1 and rs2 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	1	11	rs1/rd	01	rs2	01						

16.5.7 C.AND: a bitwise AND instruction**Syntax:**

$$c.and\ rd,\ rs2$$
Operation:

$$rd \leftarrow rs1 \& rs2$$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

$$rs1 = rd$$

Typical rd/rs1 and rs2 code registers are:

- 000: x8

- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	0	11	rs1/rd			11	rs2			01		

16.5.8 C.ANDI: an immediate bitwise AND instruction

Syntax:

`c.andi rd, imm6`

Operation:

$rd \leftarrow rs1 \& \text{sign_extend}(imm6)$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

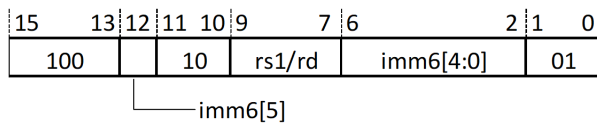
$rs1 = rd$

Typical rd/rs1 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14

- 111: x15

Instruction format:



16.5.9 C.BEQZ: a branch-if-equal-to-zero instruction

Syntax:

`c.beqz rs1, label`

Operation:

if (`rs1 == 0`)

`next pc = current pc + imm8<<1;`

else

`next pc = current pc + 2;`

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

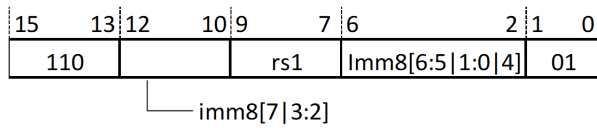
Typical rs1 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

The compiler calculates immediate 8 based on the label.

The jump range of the instruction is ± 256 B address space.

Instruction format:



16.5.10 C.BNEZ: a branch-if-not-equal-to-zero instruction

Syntax:

c.bnez rs1, label

Operation:

if (rs1 != 0)

next pc = current pc + imm8<<1;

else

next pc = current pc + 2;

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

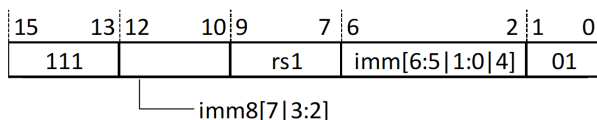
Typical rs1 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

The compiler calculates immediate 12 based on the label.

The jump range of the instruction is ± 256 B address space.

Instruction format:



16.5.11 C.EBREAK: a break instruction

Syntax:

c.ebreak

Operation:

Generates breakpoint exceptions or enables the core to enter the debug mode.

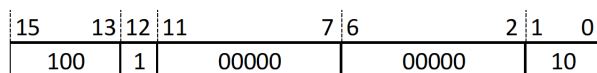
Permission:

M mode/S mode/U mode

Exception:

Breakpoint exceptions

Instruction format:



16.5.12 C.FLD: a floating-point load doubleword instruction

Syntax:

c.fld fd, uimm5<<3(rs1)

Operation:

address \leftarrow rs1 + zero_extend(uimm5<<3)

fd \leftarrow mem[address+7:address]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

Typical rs1 code registers are:

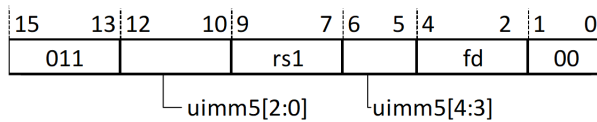
- 000: x8

- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Typical fd code registers are:

- 000: f8
- 001: f9
- 010: f10
- 011: f11
- 100: f12
- 101: f13
- 110: f14
- 111: f15

Instruction format:



16.5.13 C.FLDSP: a floating-point doubleword load stack instruction

Syntax:

`c.fldsp fd, uimm6<<3(sp)`

Operation:

$\text{address} \leftarrow \text{sp} + \text{zero_extend}(\text{uimm6} \ll 3)$

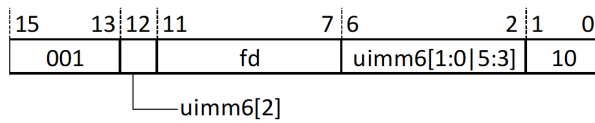
$\text{fd} \leftarrow \text{mem}[\text{address} + 7 : \text{address}]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:**16.5.14 C.FSD: a floating-point store doubleword instruction****Syntax:**

`c.fsd fs2, uimm5<<3(rs1)`

Operation:

$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{uimm5} \ll 3)$

$\text{mem}[\text{address}+7:\text{address}] \leftarrow \text{fs2}$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Notes:

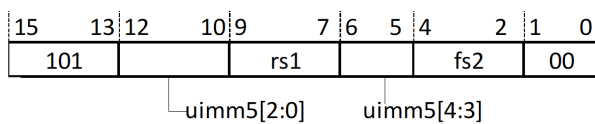
Typical fs1 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Typical rs2 code registers are:

- 000: f8
- 001: f9

- 010: f10
- 011: f11
- 100: f12
- 101: f13
- 110: f14
- 111: f15

Instruction format:**16.5.15 C.FSDSP: a floating-point store doubleword stack pointer instruction****Syntax:**

c.fsdsp fs2, uimm6<<3(sp)

Operation:

address \leftarrow sp + zero_extend(uimm6<<3)

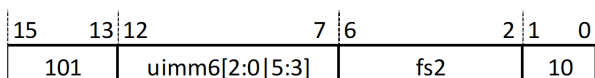
mem[address+7:address] \leftarrow fs2

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:**16.5.16 C.J: a unconditional jump instruction****Syntax:**

c.j label

Operation:

next pc \leftarrow current pc + sign_extend(imm<<1);

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

The compiler calculates immediate 11 based on the label.

The jump range of the instruction is ± 2 KB address space.**Instruction format:**

15	13	12	2	1	0
101	imm11[10 3 8:7 9 5 6 2:0 4]			01	

16.5.17 C.JALR: a jump and link register instruction**Syntax:**

c.jalr rs1

Operation:next pc \leftarrow rs1;x1 \leftarrow current pc + 2;**Permission:**

M mode/S mode/U mode

Exception:

None

Notes:rs1 \neq 0.

When MMU is enabled, the jump range is the entire 512 GB address space.

When MMU is disabled, the jump range is the entire 1 TB address space.

Instruction format:

15	13	12	11	7	6	2	1	0
100	1	rs1			00000	10		

16.5.18 C.JR: a jump register instruction

Syntax:

```
c.jr rs1
```

Operation:

```
next pc = rs1;
```

Permission:

```
M mode/S mode/U mode
```

Exception:

None

Notes:

```
rs1 != 0.
```

When MMU is enabled, the jump range is the entire 512 GB address space.

When MMU is disabled, the jump range is the entire 1 TB address space.

Instruction format:

15	13	12	11	7	6	2	1	0
100	0	rs1			00000	10		

16.5.19 C.LD: a load doubleword instruction

Syntax:

```
c.ld rd, uimm5<<3(rs1)
```

Operation:

```
address ← rs1 + zero_extend(uimm5<<3)
```

```
rd ← mem[address+7:address]
```

Permission:

```
M mode/S mode/U mode
```

Exception:

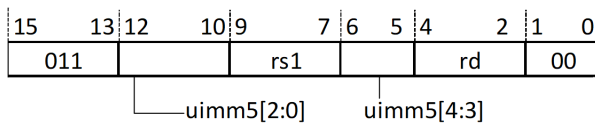
Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

Typical rs1/rd code registers are:

- 000: x8

- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:**16.5.20 C.LDSP: a load doubleword instruction****Syntax:**

c.ldsp rd, uimm6<<3(sp)

Operation:

address \leftarrow sp+ zero_extend(uimm6<<3)

rd \leftarrow mem[address+7:address]

Permission:

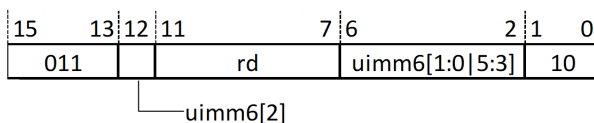
M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

rd != 0

Instruction format:

16.5.21 C.LI: a load immediate instruction

Syntax:

c.li rd, imm6

Operation:

$rd \leftarrow \text{sign_extend}(\text{imm6})$

Permission:

M mode/S mode/U mode

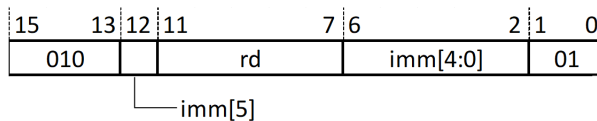
Exception:

None

Notes:

rd != 0

Instruction format:



16.5.22 C.LUI: a load upper immediate instruction

Syntax:

c.lui rd, nzimm6

Operation:

$rd \leftarrow \text{sign_extend}(\text{nzimm6} \ll 12)$

Permission:

M mode/S mode/U mode

Exception:

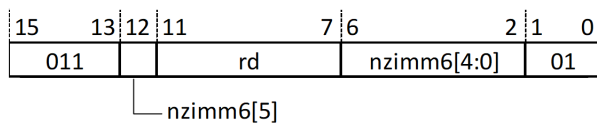
None

Notes:

rd != 0.

Nzimm6 != 0.

Instruction format:



16.5.23 C.LW: a load word instruction

Syntax:

```
c.lw rd, uimm5<<2(rs1)
```

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{uimm5} \ll 2)$$

$$\text{tmp}[31:0] \leftarrow \text{mem}[\text{address}+3:\text{address}]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$

Permission:

M mode/S mode/U mode

Exception:

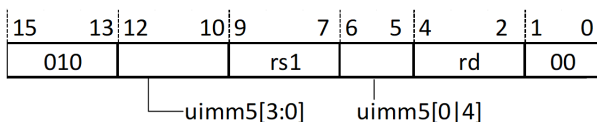
Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.5.24 C.LWSP: a load word stack pointer instruction

Syntax:

c.lwsp rd, uimm6<<2(sp)

Operation:

address \leftarrow sp+ zero_extend(uimm6<<2)

tmp[31:0] \leftarrow mem[address+3:address]

rd \leftarrow sign_extend(tmp[31:0])

Permission:

M mode/S mode/U mode

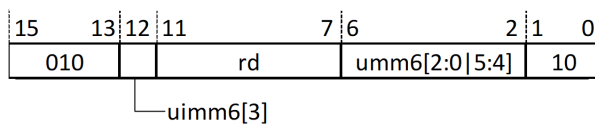
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

rd \neq 0

Instruction format:



16.5.25 C.MV: an instruction that copies the value in rs to rd

Syntax:

c.mv rd, rs2

Operation:

rd \leftarrow rs2;

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

rs2 \neq 0, rd \neq 0.

Instruction format:

15	13	12	11	7	6	2	1	0
100	0	rd			rs2		10	

16.5.26 C.NOP: a no-operation instruction

Syntax:

c.nop

Operation:

No operations

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

15	13	12	11	7	6	2	1	0
000	0	00000			00000		01	

16.5.27 C.OR: a bitwise OR instruction

Syntax:

c.or rd, rs2

Operation:

$rd \leftarrow rs1 \mid rs2$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

rs1 = rd

Typical rd/rs1 code registers are:

- 000: x8
- 001: x9

- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	0	11	rs1/rd			10	rs2			01		

16.5.28 C.SD: a store doubleword instruction

Syntax:

`c.sd rs2, uimm5<<3(rs1)`

Operation:

$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{uimm5} \ll 3)$

$\text{mem}[\text{address}+7:\text{address}] \leftarrow \text{rs2}$

Permission:

M mode/S mode/U mode

Exception:

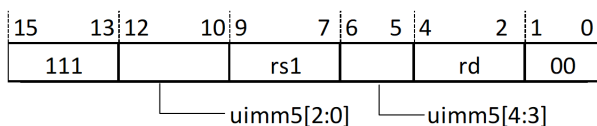
Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Notes:

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.5.29 C.SDSP: a store doubleword stack pointer instruction

Syntax:

`c.fsdsp rs2, uimm6<<3(sp)`

Operation:

$address \leftarrow sp + zero_extend(uimm6 \ll 3)$

$mem[address+7:address] \leftarrow rs2$

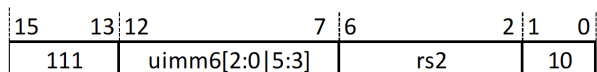
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:



16.5.30 C.SLLI: an immediate logical left shift instruction

Syntax:

`c.slli rd, nzuimm6`

Operation:

$rd \leftarrow rs1 \ll nzuimm6$

Permission:

M mode/S mode/U mode

Exception:

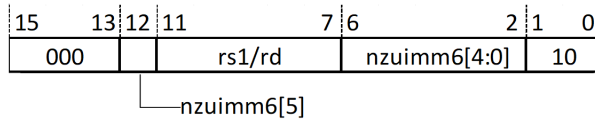
None

Notes:

$rs1 == rd$

$rd/rs1 \neq 0, nzuimm6 \neq 0$

Instruction format:



16.5.31 C.SRAI: a right shift arithmetic immediate instruction

Syntax:

`c.srli rd, nzuimm6`

Operation:

$rd \leftarrow rs1 \ggg nzuimm6$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

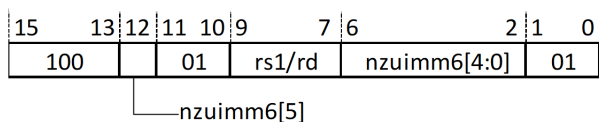
$nzuimm6 \neq 0$

$rs1 == rd$

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.5.32 C.SRLI: an immediate right shift instruction

Syntax:

c.srli rd, nzuimm6

Operation:

rd \leftarrow rs1 \gg nzuimm6

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

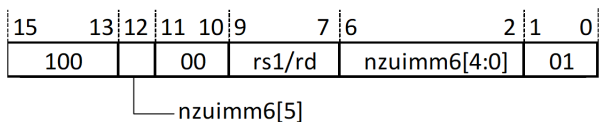
nzuimm6 \neq 0

rs1 == rd

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.5.33 C.SW: a store word instruction

Syntax:

c.sw rs2, uimm5<<2(rs1)

Operation:

address \leftarrow rs1 + zero_extend(uimm5<<2)

mem[address+3:address] \leftarrow rs2

Permission:

M mode/S mode/U mode

Exception:

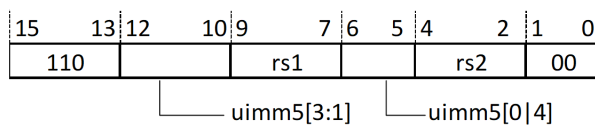
Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Notes:

Typical rs1/rs2 code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



16.5.34 C.SWSP: a store word stack pointer instruction

Syntax:

c.swsp rs2, uimm6<<2(sp)

Operation:

$$\text{address} \leftarrow \text{sp} + \text{zero_extend}(\text{uimm6} \ll 2)$$

$$\text{mem}[\text{address}+3:\text{address}] \leftarrow \text{rs2}$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:

15	13	12	7	6	2	1	0
110		uimm6[3:0 5:4]			rs2		10

16.5.35 C.SUB: a signed subtract instruction**Syntax:**

$$\text{c.sub rd, rs2}$$
Operation:

$$\text{rd} \leftarrow \text{rs1} - \text{rs2}$$
Permission:

M mode/S mode/U mode

Exception:

None

Notes:

$$\text{rs1} == \text{rd}$$

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	0	11	rs1/rd			00	rs2			01		

16.5.36 C.SUBW: a signed subtract instruction that operates on the lower 32 bits

Syntax:

c.subw rd, rs2

Operation:

$\text{tmp}[31:0] \leftarrow \text{rs1}[31:0] - \text{rs2}[31:0]$

$\text{rd} \leftarrow \text{sign_extend}(\text{tmp})$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

rs1 == rd

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	1	11	rs1/rd			00	rs2			01		

16.5.37 C.XOR: a bitwise XOR instruction

Syntax:

c.xor rd, rs2

Operation:

$rd \leftarrow rs1 \wedge rs2$

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

rs1 == rd

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:

15	13	12	11	10	9	7	6	5	4	2	1	0
100	0	11	rs1/rd	01	rs2	01						

16.6 Appendix A-8 Pseudo instructions

RISC-V implements a series of pseudo instructions. The instructions listed in this section are for reference only and are sorted in alphabetic order.

Pseudo instruction	Base instruction	Meaning
beqz rs, offset	beq rs, x0, offset	Takes the branch if rs is zero.

Continued on next page

Table 16.1 – continued from previous page

Pseudo instruction	Base instruction	Meaning
bnez rs, offset	bne rs, x0, offset	Takes the branch if rs is not zero.
blez rs, offset	bge x0,rs, offset	Takes the branch if rs is less than or equal to zero.
bgez rs, offset	bge rs, x0, offset	Takes the branch if rs is greater than or equal to zero.
bltz rs, offset	blt rs, x0, offset	Takes the branch if rs is less than zero.
bgtz rs, offset	blt x0, rs, offset	Takes the branch if rs is greater than zero.
bgt rs, rt, offset	blt rt, rs, offset	Takes the branch if rs is greater than rt.
ble rs, rt, offset	bge rt, rs, offset	Takes the branch if rs is less than or equal to rt.
bgtu rs, rt, offset	bltu rt, rs, offset	Takes the branch if rs is greater than rt, using unsigned comparison.
bleu rs, rt, offset	bgeu rt, rs, offset	Takes the branch if rs is less than or equal to rt, using unsigned comparison.
call offset	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]	Calls far-away subroutine.
csrc csr, rs	csrrc x0, csr, rs	Clears bits in the control/status register (CSR).
csrci csr, imm	csrrci x0, csr, imm	Clears bits in the CSR, immediate.
csrs csr, rs	csrrs x0, csr, rs	Sets bits in the CSR.
csrsi csr, imm	csrrsi x0, csr, imm	Sets bits in the CSR, immediate
csrw csr, rs	csrrw x0, csr, rs	Writes the CSR.
csrwi csr, imm	csrrwi x0, csr, imm	Writes the CSR, immediate.
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Calculates the double-precision floating point (FP) absolute value.
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Calculates the single-precision FP absolute value.
fence	fence iorw, iorw	Fences on all memory and I/O.
fl{w d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w d} rd, symbol[11:0](rt)	An FP load global instruction.

Continued on next page

Table 16.1 – continued from previous page

Pseudo instruction	Base instruction	Meaning
fmv.d rd, rs	fsgnj.d rd, rs, rs	A double-precision FP copy instruction.
fmv.s rd, rs	fsgnj.s rd, rs, rs	A single-precision FP copy instruction.
fneg.d rd, rs	fsgnjn.d rd, rs, rs	A double-precision FP negate instruction.
fneg.s rd, rs	fsgnjn.s rd, rs, rs	A single-precision FP negate instruction.
frcsr rd	csrrs x0, fcsr, x0	Reads FP CSR.
frflags rd	csrrs rd, fflags, x0	Reads FP exception flags.
frfm rd	csrrs rd, frm, x0	Reads FP rounding mode.
fscsr rs	csrrw x0, fcsr, rs	Writes FP CSR.
fscsr rd, rs	csrrs rd, fcsr, rs	Swaps FP CSR.
fsflags rs	csrrw x0, fcsr, rs	Writes FP exception flags.
fsflags rd, rs	csrrs rd, fcsr, rs	Swaps FP exception flags.
fsflagsi imm	csrrwi x0, fflags, imm	Writes FP exception flags, immediate.
fsflagsi rd, imm	csrrwi rd, fflags, imm	Swaps FP exception flags, immediate.
fsrm rs	csrrw x0, frm, rs	Writes FP rounding mode.
fsrm rd, rs	csrrs rd, frm, rs	Swaps FP rounding mode.
fsrmi imm	csrrwi x0, frm, imm	Writes FP rounding mode, immediate.
fsrmi rd, imm	csrrwi rd, frm, imm	Swaps FP rounding mode, immediate.
fs{w d} rd, symbol,rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0](rt)	An FP store global instruction.
j offset	jal x0, offset	A jump instruction.
jal offset	jal x1, offset	Jumps to subroutine and link.
jalr rs	jalr x1, rs, 0	Jumps to subroutine and links register.
jr rs	jalr x0, rs, 0	A jump register instruction.
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	A load address instruction.
li rd, immediate	Split into multiple instructions based on the size of the immediate	A load immediate instruction

Continued on next page

Table 16.1 – continued from previous page

Pseudo instruction	Base instruction	Meaning
$l\{b h w d\}$ rd,symbol, rt	auipc rt, symbol[31:12] $l\{b h w d\}$ rd,symbol[11:0](rt)	A load global instruction.
mv rd, rs	addi rd, rs, 0	A instruction that copies the value in rs to rd.
neg rd, rs	sub rd, x0, rs	A register negate instruction.
negw rd, rs	subw rd, x0, rs	Negates the lower 32 bits of registers.
nop	addi x0,x0,0	A no operation instruction.
not rd, rs	xori rd, rs, -1	A register NOT instruction.
rdcycle[h] rd	csrrs rd, cycle[h], x0	A read cycle counter instruction.
rdinstret[h] rd	csrrs rd, instret[h], x0	Reads instructions-retired counter.
rdtime[h] rd	csrrs rd, time[h], x0	Reads real-time clock.
ret	jalr x0, x1,0	Returns from subroutine.
$s\{b h w d\}$ rd, symbol, rt	auipc rt,symbol[31:12] $s\{b h w d\}$ rd,symbol[11:0](rt)	A store global instruction.
seqz rd, rs	sltiu rd, rs, 1	Sets 0 in registers to 1.
sextw rd, rs	addiw rd, rs, 0	A sign extend word instruction.
sgtz rd, rs	slt rd, rs, x0, rs	Sets rd to 1 if rs is greater than zero.
sltz rd, rs	slt rd, rs, rs, x0	Sets rd to 1 if rs is less than zero.
snez rd, rs	sltu rd, rs, x0, rs	Sets rd to 1 if rs is not equal to zero.
tail offset	auipc x6,offset[31:12] jalr x0, x6,offset[11:0]	Tail call far-away subroutine.

Appendix B T-Head Extended Instructions

Apart from the GC instruction sets defined in the standard, C908 provides custom instruction sets, including the cache instruction set, synchronization instruction set, arithmetic operation instruction set, bitwise operation instruction set, storage instruction set, and half-precision floating-point instruction set.

Among these instruction sets, the cache instructions, synchronization instructions, arithmetic operation instructions, bitwise operation instructions, and storage instructions can be executed only when the value of `mxstatus.theadisaee` is 1. Otherwise, an instruction exception will occur. Half-precision floating-point instructions can be executed only when the value of `mstatus.fs ! is 2' b00`. Otherwise, an illegal instruction exception will occur. The following describes each instruction in these instruction sets.

17.1 Appendix B-1 Cache instructions

You can use the cache instruction set to manage caches. Each instruction has 32 bits.

Arithmetic operation instructions in this instruction set are described in alphabetical order.

17.1.1 DCACHE.CALL: an instruction that clears all dirty page table entries in the D-Cache

Syntax:

```
dcache.call
```

Operation:

Clears all page table entries in the L1 D-Cache and writes all dirty page table entries back into the next-level storage. You can perform this operation only on the current core.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		00001		00000		000		00000		0001011	

17.1.2 DCACHE.CIALL: an instruction that clears all dirty page table entries in the D-Cache and invalidates the D-Cache

Syntax:

`dcache.ciall`

Operation:

Writes all dirty page table entries in the L1 D-Cache back into the next-level storage and invalidates all these page table entries.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		00011		00000		000		00000		0001011	

17.1.3 DCACHE.CIPA: clears dirty page table entries that match the specified physical addresses from the D-Cache and invalidates the the D-Cache

Syntax:

```
dcache.cipa rs1
```

Operation:

Writes page table entries that match the specified physical addresses of the D-Cache or L2 Cache of rs1 back into the next-level storage and invalidates these page table entries. You can perform this operation on all cores and the L2 Cache.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of illegal instruction.

If the value of mxstatus.theadisaee is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		01011		rs1		000		00000		0001011	

17.1.4 DCACHE.CISW: an instruction that clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache

Syntax:

```
dcache.cisw rs1
```

Operation:

Writes the dirty page table entry that matches the specified way and set from the L1 Cache of rs1 back into the next-level storage and invalidates this page table entry. You can perform this operation only on the current core.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

C908 D-Cache is a 2-way set-associative cache. $rs1[31]$ specifies the way and $rs1[s:6]$ specifies the set. When the size of the D-Cache is 32 KB, w denotes 13. When the size of the D-Cache is 64 KB, w denotes 14, and so forth.

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00011		rs1		000		00000		0001011	

17.1.5 DCACHE.CIVA: an instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the D-Cache

Syntax:

```
dcache.civa rs1
```

Operation:

Writes the page table entry that matches the specified virtual address from the D-Cache or L2 Cache of `rs1` back into the next-level storage and invalidates this page table entry. You can perform this operation on the current core and the L2 Cache. The sharing attribute of the virtual address determines whether you can perform this operation on other cores.

Permission:

M mode/S mode/U mode

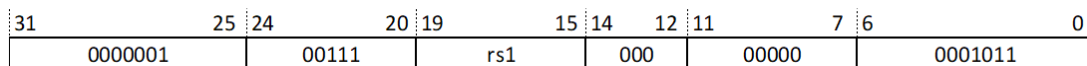
Exception:

Illegal instruction or error page during instruction loading.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1 and the value of `mxstatus.ucme` is 1, this instruction can be executed in U mode.
- If the value of `mxstatus.theadisaee` is 1 and the value of `mxstatus.ucme` is 0, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:



17.1.6 DCACHE.CPA: an instruction that clears dirty page table entries that match the specified physical addresses from the D-Cache

Syntax:

dcache.cpa rs1

Operation:

Writes the page table entry that matches the specified physical address from the D-Cache or L2 Cache of rs1 back into the next-level storage. You can perform this operation on all cores and the L2 Cache.

Permission:

M mode/S mode

Exception:

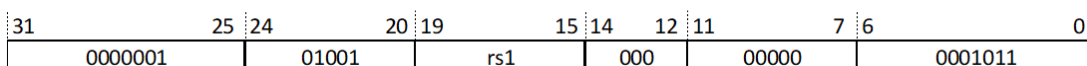
Illegal instruction.

Notes:

If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of illegal instruction.

If the value of mxstatus.theadisaee is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:



17.1.7 DCACHE.CPAL1: an instruction that clears dirty page table entries that match the specified physical addresses from the L1 D-Cache

Syntax:

dcache.cpal1 rs1

Operation: Writes the page table entry that matches the specified physical address from the D-Cache of rs1 back into the next-level storage. You can perform this operation on all cores and the L1 Cache.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		01000		rs1		000		00000		0001011	

17.1.8 DCACHE.CVA: an instruction that clears dirty page table entries that match the specified virtual addresses in the D-Cache

Syntax:

`dcache.cva rs1`

Operation:

Writes the page table entry that matches the specified virtual address from the D-Cache or L2 Cache of `rs1` back into the next-level storage. You can perform this operation on the current core and the L2 Cache. The sharing attribute of the virtual address determines whether you can perform this operation on other cores.

Permission:

M mode/S mode

Exception:

Illegal instruction or error page during instruction loading.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00101		rs1		000		00000		0001011	

17.1.9 DCACHE.CVAL1: an instruction that clears dirty page table entries that match the specified virtual addresses in the L1 D-Cache

Syntax:

```
dcache.cval1 rs1
```

Operation:

Writes the page table entry that matches the specified virtual address from the D-Cache of s1 back into the next-level storage. You can perform this operation on all cores and the L1 Cache.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction or error page during instruction loading.

Notes:

If the value of mxstatus.theadisae is 0, executing this instruction causes an exception of illegal instruction.

If the value of mxstatus.theadisae is 1 and the value of mxstatus.ucme is 0, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00100		rs1		000		00000		0001011	

17.1.10 DCACHE.IPA: an instruction that invalidates page table entries that match the specified physical addresses in the D-Cache

Syntax:

```
dcache.ipa rs1
```

Operation:

Invalidates the page table entry that matches the specified physical address in the D-Cache or L2 Cache of rs1. You can perform this operation on all cores and the L2 Cache.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000001			01010		rs1		000		00000		0001011	

17.1.11 DCACHE.ISW: an instruction that invalidates page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache

Syntax:

```
dcache.isw rs1
```

Operation:

Invalidates the page table entry in the D-Cache based on the specified set and way. You can perform this operation only on the current core.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

C908 D-Cache is a 2-way set-associative cache. `rs1[31]` specifies the way and `rs1[s:6]` specifies the set. When the size of the D-Cache is 32 KB, `w` denotes 13. When the size of the D-Cache is 64 KB, `w` denotes 14, and so forth.

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000001			00010		rs1		000		00000		0001011	

17.1.12 DCACHE.IVA: an instruction that invalidates the D-Cache based on the specified virtual address

Syntax:

```
dcache.iva rs1
```

Operation:

Invalidates the page table entry that matches the specified virtual address from the D-Cache or L2 Cache of rs1. You can perform this operation on the current core and the L2 Cache. The sharing attribute of the virtual address determines whether you can perform this operation on other cores.

Permission:

M mode/S mode

Exception:

Illegal instruction or error page during instruction loading.

Notes:

- If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of illegal instruction.
- If the value of mxstatus.theadisaee is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00110		rs1		000		00000		0001011	

17.1.13 DCACHE.IALL: an instruction that invalidates all page table entries in the D-Cache.

Syntax:

```
dcache.iall
```

Operation:

Invalidates all page table entries in the L1 Cache. You can perform this operation only on the current core.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			00010		00000		000		00000		0001011	

17.1.14 ICACHE.IALL: an instruction that invalidates all page table entries in the I-Cache

Syntax:

```
icache.iall
```

Operation:

Invalidates all page table entries in the I-Cache. You can perform this operation only on the current core.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			10000		00000		000		00000		0001011	

17.1.15 ICACHE.IALLS: an instruction that invalidates all page table entries in the I-Cache through broadcasting

Syntax:

```
icache.ialls
```

Operation:

Invalidates all page table entries in the I-Cache and invalidates all page table entries in the I-Cache of other cores through broadcasting. You can perform this operation on all cores.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		10001		00000		000		00000		0001011	

17.1.16 ICACHE.IPA: an instruction that invalidates page table entries that match the specified physical addresses in the I-Cache

Syntax:

`icache.ipa rs1`

Operation:

Invalidates the page table entry that matches the specified physical address in the I-Cache of `rs1`. You can perform this operation on all cores.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		11000		rs1		000		00000		0001011	

17.1.17 ICACHE.IVA: an instruction that invalidates page table entries that match the specified virtual addresses in the I-Cache

Syntax:

```
icache.iva rs1
```

Operation:

Invalidates the page table entry that matches the specified virtual address in the I-Cache of rs1. You can perform this operation only on the current core. The sharing attribute of the virtual address determines whether you can perform this operation on other cores.

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction or error page during instruction loading.

Notes:

If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of illegal instruction.

If the value of mxstatus.theadisaee is 1 and the value of mxstatus.ucme is 1, this instruction can be executed in U mode.

If the value of mxstatus.theadisaee is 1 and the value of mxstatus.ucme is 0, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		10000		rs1		000		00000		0001011	

17.1.18 L2CACHE.CALL: an instruction that clears all dirty page table entries in the L2 Cache

Syntax:

```
l2cache.call
```

Operation:

Writes all dirty page table entries from the L2 Cache back into the next-level storage.

Permission:

```
M mode/S mode
```

Exception:

Illegal instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			10101		00000		000		00000		0001011	

17.1.19 L2CACHE.CIALL: an instruction that clears all dirty page table entries in the L2 Cache and invalidates the L2 Cache

Syntax:

```
l2cache.ciall
```

Operation:

Writes all dirty page table entries from the L2 Cache back into the next-level storage and invalidates all page table entries in the L2 Cache.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			10111		00000		000		00000		0001011	

17.1.20 L2CACHE.IALL: an instruction that invalidates the L2 Cache

Syntax:

```
l2cache.iall
```

Operation:

Invalidates all page table entries in the L2 Cache.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

- If the value of `mxstatus.cskisayee` is 0, executing this instruction causes an exception of illegal instruction.
- If the value of `mxstatus.cskisayee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		10110		00000		000		00000		0001011	

17.1.21 DCACHE.CSW: an instruction that clears dirty page table entries in the D-Cache based on the specified set and way

Syntax:

`dcache.csw rs1`

Operation:

Writes the dirty page table entry from the D-Cache back into the next-level storage device based on the specified set and way.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

C908 D-Cache is a 2-way set-associative cache. `rs1[31]` specifies the way and `rs1[s:6]` specifies the set. When the size of the D-Cache is 32 KB, `w` denotes 13. When the size of the D-Cache is 64 KB, `w` denotes 14, and so forth.

If the value of `mxstatus.theadisae` is 0, executing this instruction causes an exception of illegal instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

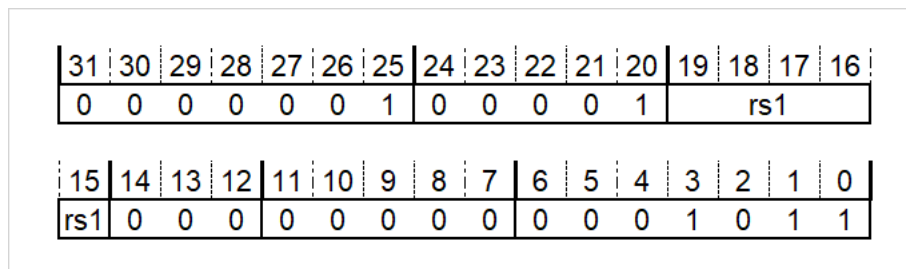


Fig. 17.1: DCACHE.CSW

17.2 Appendix B-2 Multi-core synchronization instructions

This synchronization instruction set extends multi-core synchronization instructions. Each instruction has 32 bits. Synchronization instructions in this instruction set are described in alphabetical order.

17.2.1 SFENCE.VMAS: a broadcast instruction that synchronizes the virtual memory address

Syntax:

`sfence.vmas rs1,rs2`

Operation:

Invalidates and synchronizes page table entries in the virtual memory and broadcasts them to other cores in the cluster.

Permission:

M mode/S mode

Exception:

Illegal instruction.

Notes:

`rs1` is the virtual address, and `rs2` is the address space identifier (ASID).

- If the value of `rs1` is `x0` and the value of `rs2` is `x0`, invalidate all page table entries in the TLB and broadcast them to other cores in the cluster.
- When `rs1!` and `rs2` are both `x0`, all TLB entries that hit the virtual address specified by `rs1` are invalidated and broadcast to other cores in the cluster.

- When rs1 and rs2! are both x0, all TLB entries that hit the process ID specified by rs2 are invalidated and broadcast to other cores in the cluster.
- When rs1! and rs2! are both x0, all TLB entries that hit the virtual address specified by rs1 and the process ID specified by rs2 are invalidated and broadcast to other cores in the cluster.

If the value of mxstatus.theadisae is 0, executing this instruction causes an exception of illegal instruction.

If the value of mxstatus.theadisae is 1, executing this instruction in U mode causes an exception of illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000010		rs2		rs1		000		00000		0001011	

17.2.2 SYNC: an instruction that performs the synchronization operation

Syntax:

sync

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		11000		00000		000		00000		0001011	

17.2.3 SYNC.I: an instruction that synchronizes the clearing operation.

Syntax:

sync.i

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction, and clears the pipeline when this instruction retires.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		11010		00000		000		00000		0001011	

17.2.4 SYNC.IS: a broadcast instruction that synchronizes the clearing operation**Syntax:**

sync.is

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction. Clears the pipeline when this instruction retires and broadcasts the request to other cores.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		11011		00000		000		00000		0001011	

17.2.5 SYNC.S: a broadcast instruction that performs a synchronization operation**Syntax:**

sync.s

Operation:

Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction, and broadcasts the request to other cores.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			11001		00000		000		00000		0001011	

17.3 Appendix B-3 Arithmetic operation instructions

The arithmetic operation instruction set extends arithmetic operation instructions. Each instruction has 32 bits.

Arithmetic operation instructions in this instruction set are described in alphabetical order.

17.3.1 ADDSL: an add register instruction that shifts registers

Syntax:

addsl rd rs1, rs2, imm2

Operation:

$rd \leftarrow rs1 + rs2 \ll imm2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
00000			imm2		rs2		rs1		001		rd		0001011	

17.3.2 MULA: a multiply-add instruction

Syntax:

mula rd, rs1, rs2

Operation:

$rd \leftarrow rd + (rs1 * rs2)[63:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		00		rs2		rs1		001		rd		0001011	

17.3.3 MULA: a multiply-add instruction that operates on the lower 16 bits**Syntax:**

mulah rd, rs1, rs2

Operation:

$$\text{tmp}[31:0] \leftarrow \text{rd}[31:0] + (\text{rs1}[15:0] * \text{rs}[15:0])$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00101		00		rs2		rs1		001		rd		0001011	

17.3.4 MULAW: a multiply-add instruction that operates on the lower 32 bits**Syntax:**

mulaw rd, rs1, rs2

Operation:

$$\text{tmp}[31:0] \leftarrow \text{rd}[31:0] + (\text{rs1}[31:0] * \text{rs}[31:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		10		rs2	rs1		001		rd		0001011		

17.3.5 MULS: a multiply-subtract instruction

Syntax:

mul_s rd, rs1, rs2

Operation:

$rd \leftarrow rd - (rs1 * rs2)[63:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		01		rs2	rs1		001		rd		0001011		

17.3.6 MULSH: a multiply-subtract instruction that operates on the lower 16 bits

Syntax:

mul_{sh} rd, rs1, rs2

Operation:

$tmp[31:0] \leftarrow rd[31:0] - (rs1[15:0] * rs2[15:0])$

$rd \leftarrow \text{sign_extend}(tmp[31:0])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00101		01		rs2	rs1		001		rd		0001011		

17.3.7 MULSW: a multiply-subtract instruction that operates on the lower 32 bits

Syntax:

```
mulaw rd, rs1, rs2
```

Operation:

$$\text{tmp}[31:0] \leftarrow \text{rd}[31:0] - (\text{rs1}[31:0] * \text{rs2}[31:0])$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp}[31:0])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		11		rs2		rs1		001		rd		0001011	

17.3.8 MVEQZ: an instruction that sends a message when the register is 0

Syntax:

```
mveqz rd, rs1, rs2
```

Operation: if (rs2 == 0)

$$\text{rd} \leftarrow \text{rs1}$$

else

$$\text{rd} \leftarrow \text{rd}$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		00		rs2		rs1		001		rd		0001011	

17.3.9 MVNEZ: an instruction that sends a message when the register is not 0

Syntax:

```
mvnez rd, rs1, rs2
```

Operation:

```
if (rs2 != 0)
```

```
    rd ← rs1
```

```
else
```

```
    rd ← rd
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		01		rs2		rs1		001		rd		0001011	

17.3.10 SRRI: an instruction that implements a cyclic right shift operation on a linked list

Syntax:

```
srri rd, rs1, imm6
```

Operation:

```
rd ← rs1 >>>> imm6
```

Shifts the original value of rs1 to the right, disconnects the last value on the list, and re-attaches the value to the start of the linked list.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
000100		imm6		rs1		001		rd		0001011	

17.3.11 SRRIW: an instruction that implements a cyclic right shift operation on a linked list of low 32 bits of registers.

Syntax:

```
srriw rd, rs1, imm5
```

Operation:

$$rd \leftarrow \text{sign_extend}(rs1[31:0] \gg \gg \gg \text{imm5})$$

Shifts the original value of $rs1[31:0]$ to the right, disconnects the last value on the list, and re-attaches the value to the start of the linked list.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0001010	imm5	rs1	001	rd	0001011	

17.4 Appendix B-4 Bitwise operation instructions

The bitwise operation instruction set extends bitwise operation instructions. Each instruction has 32 bits.

Arithmetic operation instructions in this instruction set are described in alphabetical order.

17.4.1 EXT: a signed extension instruction that extracts consecutive bits of a register

Syntax:

```
ext rd, rs1, imm1,imm2
```

Operation:

$$rd \leftarrow \text{sign_extend}(rs1[\text{imm1}:\text{imm2}])$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

If imm1 is smaller than imm2, the action of this instruction is not predictable.

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
imm1		imm2			rs1		010		rd		0001011

17.4.2 EXTU: a zero extension instruction that extracts consecutive bits of a register

Syntax:

extu rd, rs1, imm1,imm2

Operation:

$rd \leftarrow \text{zero_extend}(rs1[imm1:imm2])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Notes:

If imm1 is smaller than imm2, the action of this instruction is not predictable.

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
imm1		imm2			rs1		011		rd		0001011

17.4.3 FF0: an instruction that finds the first bit with the value of 0 in a register

Syntax:

ff0 rd, rs1

Operation:

Finds the first bit with the value of 0 from the highest bit of rs1 and writes the result back into the rd register. If the highest bit of rs1 is 0, the result 0 is returned. If all the bits in rs1 are 1, the result 64 is returned.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
10000				10		00000		rs1		001		rd		0001011	

17.4.4 FF1: an instruction that finds the bit with the value of 1

Syntax:

ff1 rd, rs1

Operation:

Finds the first bit with the value of 1 from the highest bit of rs1 and writes the index of this bit back into rd. If the highest bit of rs1 is 1, the result 0 is returned. If all the bits in rs1 are 1, the result 64 is returned.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
10000				11		00000		rs1		001		rd		0001011	

17.4.5 REV: an instruction that reverses the byte order in a word stored in the register

Syntax:

rev rd, rs1

Operation:

$rd[63:56] \leftarrow rs1[7:0]$

$rd[55:48] \leftarrow rs1[15:8]$

$rd[47:40] \leftarrow rs1[23:16]$

$rd[39:32] \leftarrow rs1[31:24]$

$rd[31:24] \leftarrow rs1[39:32]$

$rd[23:16] \leftarrow rs1[47:40]$

$rd[15:8] \leftarrow rs1[55:48]$

$rd[7:0] \leftarrow rs1[63:56]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000		01		00000		rs1		001		rd		0001011	

17.4.6 REVW: an instruction that reverses the byte order in a low 32-bit word**Syntax:**

revw rd, rs1

Operation:

tmp[31:24] ←rs1[7:0]

tmp [23:16] ←rs1[15:8]

tmp [15:8] ←rs1[23:16]

tmp [7:0] ←rs1[31:24]

rd ←sign_extend(tmp[31:0])

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10010		00		00000		rs1		001		rd		0001011	

17.4.7 TST: an instruction that tests bits with the value of 0**Syntax:**

tst rd, rs1, imm6

Operation:

if(rs1[imm6] == 1)

rd ← 1

else

$$rd \leftarrow 0$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
100010		imm6			rs1		001		rd		0001011

17.4.8 TSTNBZ: an instruction that tests bytes with the value of 0

Syntax:

tstnbz rd, rs1

Operation:

$$rd[63:56] \leftarrow (rs1[63:56] == 0) ? 8'hff : 8'h0$$

$$rd[55:48] \leftarrow (rs1[55:48] == 0) ? 8'hff : 8'h0$$

$$rd[47:40] \leftarrow (rs1[47:40] == 0) ? 8'hff : 8'h0$$

$$rd[39:32] \leftarrow (rs1[39:32] == 0) ? 8'hff : 8'h0$$

$$rd[31:24] \leftarrow (rs1[31:24] == 0) ? 8'hff : 8'h0$$

$$rd[23:16] \leftarrow (rs1[23:16] == 0) ? 8'hff : 8'h0$$

$$rd[15:8] \leftarrow (rs1[15:8] == 0) ? 8'hff : 8'h0$$

$$rd[7:0] \leftarrow (rs1[7:0] == 0) ? 8'hff : 8'h0$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000		00		00000		rs1		001		rd		0001011	

17.5 Appendix B-5 Storage instructions

The storage instruction set extends storage instructions. Each instruction has 32 bits.

Arithmetic operation instructions in this instruction set are described in alphabetical order.

17.5.1 FLRD: a load doubleword instruction that shifts floating-point registers

Syntax:

flrd rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{mem}[(rs1+rs2 \ll imm2)+7: (rs1+rs2 \ll imm2)]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

If the value of `mxstatus.theadisae` is 1' b0 or the value of `mstatus.fs` is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100		imm2		rs2		rs1		110		rd		0001011	

17.5.2 FLRW: a load word instruction that shifts floating-point registers

Syntax:

flrw rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{one_extend}(\text{mem}[(rs1+rs2 \ll imm2)+3: (rs1+rs2 \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is `1' b0` or the value of `mstatus.fs` is `2' b00`, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		imm2		rs2		rs1		110		rd		0001011	

17.5.3 FLURD: a load doubleword instruction that shifts low 32 bits of floating-point registers

Syntax:

`flurd rd, rs1, rs2, imm2`

Operation:

$rd \leftarrow \text{mem}[(rs1+rs2[31:0] \ll imm2)+7: (rs1+rs2[31:0] \ll imm2)]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

`rs2[31:0]` specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

If the value of `mxstatus.theadisaee` is `1' b0` or the value of `mstatus.fs` is `2' b00`, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01110		imm2		rs2		rs1		110		rd		0001011	

17.5.4 FLURW: a load word instruction that shifts low 32 bits of floating-point registers

Syntax:

`flurw rd, rs1, rs2, imm2`

Operation:

$rd \leftarrow \text{one_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)+3: (rs1+rs2[31:0] \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

If the value of mxstatus.theadisaee is 1' b0 or the value of mstatus.fs is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01010		imm2		rs2		rs1		110		rd		0001011	

17.5.5 FSRD: a store doubleword instruction that shifts floating-point registers**Syntax:**

```
fsrd rd, rs1, rs2, imm2
```

Operation:

$$\text{mem}[(\text{rs1} + \text{rs2} \ll \text{imm2}) + 7 : (\text{rs1} + \text{rs2} \ll \text{imm2})] \leftarrow \text{rd}[63:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

If the value of mxstatus.theadisaee is 1' b0 or the value of mstatus.fs is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100		imm2		rs2		rs1		111		rd		0001011	

17.5.6 FSRW: a store word instruction that shifts floating-point registers.**Syntax:**

```
fsrw rd, rs1, rs2, imm2
```

Operation:

$$\text{mem}[(\text{rs1} + \text{rs2} \ll \text{imm2}) + 3 : (\text{rs1} + \text{rs2} \ll \text{imm2})] \leftarrow \text{rd}[31:0]$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

If the value of `mxstatus.theadisaee` is 1' b0 or the value of `mstatus.fs` is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		imm2		rs2		rs1		111		rd		0001011	

17.5.7 FSURD: a store doubleword instruction that shifts low 32 bits of floating-point registers

Syntax:

fsurd rd, rs1, rs2, imm2

Operation:

$$\text{mem}[(\text{rs1} + \text{rs2}[31:0] \ll \text{imm2}) + 7: (\text{rs1} + \text{rs2}[31:0] \ll \text{imm2})] \leftarrow \text{rd}[63:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

`rs2[31:0]` specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

If the value of `mxstatus.theadisaee` is 1' b0 or the value of `mstatus.fs` is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01110		imm2		rs2		rs1		111		rd		0001011	

17.5.8 FSURW: a store word instruction that shifts low 32 bits of floating-point registers

Syntax:

fsurw rd, rs1, rs2, imm2

Operation:

$$\text{mem}[(\text{rs1} + \text{rs2}[31:0] \ll \text{imm2}) + 3 : (\text{rs1} + \text{rs2}[31:0] \ll \text{imm2})] \leftarrow \text{rd}[31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

If the value of mxstatus.theadisaee is 1' b0 or the value of mstatus.fs is 2' b00, executing this instruction causes an exception of illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01010		imm2		rs2		rs1		111		rd		0001011	

17.5.9 LBIA: a base-address auto-increment instruction that extends signed bits and loads bytes

Syntax:

lbia rd, (rs1), imm5,imm2

Operation:

$$\text{rd} \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}])$$

$$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00011		imm2		imm5		rs1		100		rd		0001011	

17.5.10 LBIB: a load byte instruction that auto-increments the base address and extends signed bits

Syntax:

lbib rd, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1])$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00001		imm2		imm5		rs1		100		rd		0001011	

17.5.11 LBUIA: a base-address auto-increment instruction that extends zero bits and loads bytes

Syntax:

lbuia rd, (rs1), imm5,imm2

Operation:

$$rd \leftarrow \text{zero_extend}(\text{mem}[rs1])$$

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
10011				imm2		imm5			rs1		100		rd		0001011	

17.5.12 LBUIB: a load byte instruction that auto-increments the base address and extends zero bits

Syntax:

lbuib rd, (rs1), imm5,imm2

Operation:

$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$

$rd \leftarrow \text{zero_extend}(\text{mem}[rs1])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
10001				imm2		imm5			rs1		100		rd		0001011	

17.5.13 LDD: an instruction that loads double registers

Syntax:

ldd rd1,rd2, (rs1),imm2

Operation:

$\text{address} \leftarrow rs1 + \text{zero_extend}(imm2 \ll 4)$

$rd1 \leftarrow \text{mem}[\text{address}+7:\text{address}]$

$rd2 \leftarrow \text{mem}[\text{address}+15:\text{address}+8]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd1, rd2, and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
11111				imm2		rd2			rs1		100		rd1		0001011	

17.5.14 LDIA: a base-address auto-increment instruction that loads doublewords and extends signed bits

Syntax:

ldia rd, (rs1), imm5,imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+7:\text{rs1}])$

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
01111				imm2		imm5			rs1		100		rd		0001011	

17.5.15 LDIB: a load doubleword instruction that auto-increments the base address and extends the signed bits

Syntax:

ldib rd, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+7:rs1])$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01101		imm2		imm5		rs1		100		rd		0001011	

17.5.16 LHIA: a base-address auto-increment instruction that loads halfwords and extends signed bits

Syntax:

lhia rd, (rs1), imm5,imm2

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+1:rs1])$$

$$rs1 \leftarrow rs1 + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00111		imm2		imm5		rs1		100		rd		0001011	

17.5.17 LHIB: a load halfword instruction that auto-increments the base address and extends signed bits

Syntax:

lhib rd, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+1:rs1])$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00101		imm2		imm5		rs1		100		rd		0001011	

17.5.18 LHUIA: a base-address auto-increment instruction that extends zero bits and loads halfwords

Syntax:

lhui rd, (rs1), imm5,imm2

Operation:

$$rd \leftarrow \text{zero_extend}(\text{mem}[rs1+1:rs1])$$

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10111		imm2		imm5		rs1		100		rd		0001011	

17.5.19 LHUIB: a load halfword instruction that auto-increments the base address and extends zero bits

Syntax:

lhuib rd, (rs1), imm5,imm2

Operation:

$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$

$rd \leftarrow \text{zero_extend}(\text{mem}[rs1+1:rs1])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10101		imm2		imm5		rs1		100		rd		0001011	

17.5.20 LRB: a load byte instruction that shifts registers and extends signed bits

Syntax:

lrb rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2 \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
00000				imm2	rs2		rs1		100		rd		0001011	

17.5.21 LRBU: a load byte instruction that shifts registers and extends zero bits

Syntax:

lrbu rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2 \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
10000				imm2	rs2		rs1		100		rd		0001011	

17.5.22 LRD: a load doubleword instruction that shifts registers

Syntax:

lrd rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{mem}[(rs1+rs2 \ll imm2)+7: (rs1+rs2 \ll imm2)]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
01100				imm2	rs2		rs1		100		rd		0001011	

17.5.23 LRH: a load halfword instruction that shifts registers and extends signed bits

Syntax:

l_{rh} rd, rs1, rs2, imm2

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2 \ll \text{imm2})+1: (rs1+rs2 \ll \text{imm2})])$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		imm2		rs2		rs1		100		rd		0001011	

17.5.24 LRHU: a load halfword instruction that shifts registers and extends zero bits**Syntax:**

l_{rh} rd, rs1, rs2, imm2

Operation:

$$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2 \ll \text{imm2})+1: (rs1+rs2 \ll \text{imm2})])$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10100		imm2		rs2		rs1		100		rd		0001011	

17.5.25 LRW: a load word instruction that shifts registers and extends signed bits**Syntax:**

l_{rw} rd, rs1, rs2, imm2

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2 \ll \text{imm2})+3: (rs1+rs2 \ll \text{imm2})])$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		imm2		rs2		rs1		100		rd		0001011	

17.5.26 LRWU: a load word instruction that shifts registers and extends zero bits**Syntax:**

lrwu rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2 \ll imm2)+3: (rs1+rs2 \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11000		imm2		rs2		rs1		100		rd		0001011	

17.5.27 LURB: a load byte instruction that shifts low 32 bits of registers and extends signed bits**Syntax:**

lurb rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00010		imm2		rs2		rs1		100		rd		0001011	

17.5.28 LURBU: a load byte instruction that shifts low 32 bits of registers and extends zero bits

Syntax:

lurbu rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10010		imm2		rs2		rs1		100		rd		0001011	

17.5.29 LURD: a load doubleword instruction that shifts low 32 bits of registers

Syntax:

lurd rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{mem}[(rs1+rs2[31:0] \ll imm2)+7: (rs1+rs2[31:0] \ll imm2)]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01110		imm2		rs2		rs1		100		rd		0001011	

17.5.30 LURH: a load halfword instruction that shifts low 32 bits of registers and extends signed bits

Syntax:

lurh rd, rs1, rs2, imm2

Operation:

rd \leftarrow sign_extend(mem[(rs1+rs2[31:0]<<imm2)+1:
(rs1+rs2[31:0]<<imm2)])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00110		imm2		rs2		rs1		100		rd		0001011	

17.5.31 LURHU: a load halfword instruction that shifts low 32 bits of registers and extends zero bits

Syntax:

lurhu rd, rs1, rs2, imm2

Operation:

rd \leftarrow zero_extend(mem[(rs1+rs2[31:0]<<imm2)+1:
(rs1+rs2[31:0]<<imm2)])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10110		imm2		rs2		rs1		100		rd		0001011	

17.5.32 LURW: a load word instruction that shifts low 32 bits of registers and extends signed bits

Syntax:

lurw rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2[31:0] \ll \text{imm2})+3:$

$(rs1+rs2[31:0] \ll \text{imm2}))$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01010		imm2		rs2		rs1		100		rd		0001011	

17.5.33 LURWU: a load word instruction that shifts 32 bits of registers and extends zero bits

Syntax:

lwd rd1, rd2, (rs1),imm2

Operation:

Chapter 17. Appendix B T-Head Extended Instructions

```

address ← rs1 + zero_extend(imm2 << 3)
rd1 ← sign_extend(mem[address+3: address])
rd2 ← sign_extend(mem[address+7: address+4])

```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd1, rd2, and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11010		imm2		rs2		rs1		100		rd		0001011	

17.5.34 LWD: a load word instruction that loads double registers and extends signed bits

Syntax:

```
lwd rd, imm7(rs1)
```

Operation:

```

address ← rs1 + sign_extend(imm7)
rd ← sign_extend(mem[address+31: address])
rd+1 ← sign_extend(mem[address+63: address+32])

```

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11100		imm2		rd2		rs1		100		rd1		0001011	

17.5.35 LWIA: a base-address auto-increment instruction that extends signed bits and loads words

Syntax:

lwia rd, (rs1), imm5,imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01011		imm2		imm5		rs1		100		rd		0001011	

17.5.36 LWIB: a load word instruction that auto-increments the base address and extends signed bits

Syntax:

lwib rd, (rs1), imm5,imm2

Operation:

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
01001				imm2		imm5			rs1		100		rd		0001011	

17.5.37 LWUD: a load word instruction that loads double registers and extends zero bits

Syntax:

lwud rd1,rd2, (rs1),imm2

Operation:

address ← rs1 + zero_extend(imm2 << 3)

rd1 ← zero_extend(mem[address+3: address])

rd2 ← zero_extend(mem[address+7: address+4])

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd1, rd2, and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
11110				imm2		rd2			rs1		100		rd1		0001011	

17.5.38 LWUIA: a base-address auto-increment instruction that extends zero bits and loads words

Syntax:

lwuia rd, (rs1), imm5,imm2

Operation:

rd ← zero_extend(mem[rs1+3:rs1])

rs1 ← rs1 + sign_extend(imm5 << imm2)

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
11011				imm2		imm5			rs1		100		rd		0001011	

17.5.39 LWUIB: a load word instruction that auto-increments the base address and extends zero bits

Syntax:

lwuib rd, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

$$rd \leftarrow \text{zero_extend}(\text{mem}[rs1+3:rs1])$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

The values of rd and rs1 must not be the same.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0			
11001				imm2		imm5			rs1		100		rd		0001011	

17.5.40 SBIA: a base-address auto-increment instruction that stores bytes

Syntax:

sbia rs2, (rs1), imm5,imm2

Operation:

$$\text{mem}[rs1] \leftarrow rs2[7:0]$$

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
00011				imm2		imm5		rs1		101		rs2		0001011	

17.5.41 SBIB: a store byte instruction that auto-increments the base address**Syntax:**

sbib rs2, (rs1), imm5,imm2

Operation: $rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$ $\text{mem}[rs1] \leftarrow rs2[7:0]$ **Permission:**

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
00001				imm2		imm5		rs1		101		rs2		0001011	

17.5.42 SDD: an instruction that stores double registers**Syntax:**

sdd rd1,rd2, (rs1),imm2

Operation: $\text{address} \leftarrow rs1 + \text{zero_extend}(imm2 \ll 4)$ $\text{mem}[\text{address}+7:\text{address}] \leftarrow rd1$ $\text{mem}[\text{address}+15:\text{address}+8] \leftarrow rd2$ **Permission:**

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11111		imm2		rd2		rs1		101		rd1		0001011	

17.5.43 SDIA: a base-address auto-increment instruction that stores doublewords**Syntax:**

sdia rs2, (rs1), imm5,imm2

Operation:

$$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{rs2}[63:0]$$

$$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01111		imm2		imm5		rs1		101		rs2		0001011	

17.5.44 SDIB: a store doubleword instruction that auto-increments the base address**Syntax:**

sdib rs2, (rs1), imm5,imm2

Operation:

$$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$

$$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{rs2}[63:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01101		imm2		imm5		rs1		101		rs2		0001011	

17.5.45 SHIA: a base-address auto-increment instruction that stores halfwords

Syntax:

shia rs2, (rs1), imm5,imm2

Operation:

$\text{mem}[\text{rs1}+1:\text{rs1}] \leftarrow \text{rs2}[15:0]$

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00111		imm2		imm5		rs1		101		rs2		0001011	

17.5.46 SHIB: a store halfword instruction that auto-increments the base address

Syntax:

shib rs2, (rs1), imm5,imm2

Operation:

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

$\text{mem}[\text{rs1}+1:\text{rs1}] \leftarrow \text{rs2}[15:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00101		imm2		imm5		rs1		101		rs2		0001011	

17.5.47 SRB: a store byte instruction that shifts registers

Syntax:

srb rd, rs1, rs2, imm2

Operation:

$\text{mem}[(\text{rs1}+\text{rs2}\ll\text{imm2})] \leftarrow \text{rd}[7:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00000		imm2		imm5		rs1		101		rd		0001011	

17.5.48 SRD: a store doubleword instruction that shifts registers

Syntax:

srd rd, rs1, rs2, imm2

Operation:

$\text{mem}[(\text{rs1}+\text{rs2}\ll\text{imm2})+7: (\text{rs1}+\text{rs2}\ll\text{imm2})] \leftarrow \text{rd}[63:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100		imm2		rs2		rs1		101		rd		0001011	

17.5.49 SRH: a store halfword instruction that shifts registers

Syntax:

srh rd, rs1, rs2, imm2

Operation:

$$\text{mem}[(\text{rs1}+\text{rs2}\ll\text{imm2})+1: (\text{rs1}+\text{rs2}\ll\text{imm2})] \leftarrow \text{rd}[15:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100		imm2		rs2		rs1		101		rd		0001011	

17.5.50 SRW: a store word instruction that shifts registers**Syntax:**

srw rd, rs1, rs2, imm2

Operation:

$$\text{mem}[(\text{rs1}+\text{rs2}\ll\text{imm2})+3: (\text{rs1}+\text{rs2}\ll\text{imm2})] \leftarrow \text{rd}[31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		imm2		rs2		rs1		101		rd		0001011	

17.5.51 SURB: a store byte instruction that shifts low 32 bits of registers**Syntax:**

surb rd, rs1, rs2, imm2

Operation:

$$\text{mem}[(\text{rs1}+\text{rs2}[31:0]\ll\text{imm2})] \leftarrow \text{rd}[7:0]$$
Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00010		imm2		rs2		rs1		101		rd		0001011	

17.5.52 SURD: a store doubleword instruction that shifts low 32 bits of registers**Syntax:**

surd rd, rs1, rs2, imm2

Operation:

$\text{mem}[(\text{rs1} + \text{rs2}[31:0] \ll \text{imm2}) + 7: (\text{rs1} + \text{rs2}[31:0] \ll \text{imm2})] \leftarrow \text{rd}[63:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01110		imm2		rs2		rs1		101		rd		0001011	

17.5.53 SURH: a store halfword instruction that shifts low 32 bits of registers**Syntax:**

surh rd, rs1, rs2, imm2

Operation:

$\text{mem}[(\text{rs1} + \text{rs2}[31:0] \ll \text{imm2}) + 1: (\text{rs1} + \text{rs2}[31:0] \ll \text{imm2})] \leftarrow \text{rd}[15:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00110		imm2		rs2		rs1		101		rd		0001011	

17.5.54 SURW: a store word instruction that shifts low 32 bits of registers

Syntax:

surw rd, rs1, rs2, imm2

Operation:

$\text{mem}[(\text{rs1} + \text{rs2}[31:0] \ll \text{imm2}) + 3: (\text{rs1} + \text{rs2}[31:0] \ll \text{imm2})] \leftarrow \text{rd}[31:0]$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Notes:

rs2[31:0] specifies an unsigned value. 0s are added to the high bits [63:32] for address calculation.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01010		imm2		rs2		rs1		101		rd		0001011	

17.5.55 SWIA: a base-address auto-increment instruction that stores words

Syntax:

swia rs2, (rs1), imm5,imm2

Operation:

$\text{mem}[\text{rs1} + 3: \text{rs1}] \leftarrow \text{rs2}[31:0]$

$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01011		imm2		imm5		rs1		101		rs2		0001011	

17.5.56 SWIB: a store word instruction that auto-increments the base address

Syntax:

swib rs2, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

$$\text{mem}[rs1+3:rs1] \leftarrow rs2[31:0]$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01001		imm2		imm5		rs1		101		rs2		0001011	

17.5.57 SWD: an instruction that stores the low 32 bits of double registers

Syntax:

swd rd1,rd2,(rs1),imm2

Operation:

$$\text{address} \leftarrow rs1 + \text{zero_extend}(imm2 \ll 3)$$

$$\text{mem}[\text{address}+3:\text{address}] \leftarrow rd1[31:0]$$

$$\text{mem}[\text{address}+7:\text{address}+4] \leftarrow rd2[31:0]$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11100		imm2		rd2		rs1		101		rd1		0001011	

17.6 Appendix B-6 Half-precision floating-point instructions

You can use instructions in this instruction set to process floating-point half-precision data. Each instruction has 32 bits. Instructions in this instruction set are described in alphabetical order.

17.6.1 FADD.H: a half-precision floating-point add instruction

Syntax:

fadd.h fd, fs1, fs2, rm

Operation:

$fd \leftarrow fs1 + fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is fadd.h fd, fs1, fs2, rne.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is fadd.h fd, fs1, fs2, rtz.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is fadd.h fd, fs1, fs2, rdn.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is fadd.h fd, fs1, fs2, rup.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is fadd.h fd, fs1, fs2, rmm.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is fadd.h fd, fs1, fs2.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000010			fs2		fs1		rm		fd		1010011

17.6.2 FCLASS.H: a half-precision floating-point classification instruction

Syntax:

fclass.h rd, fs1

Operation:

if (fs1 = -inf)

rd ← 64' h1

if (fs1 = -norm)

rd ← 64' h2

if (fs1 = -subnorm)

rd ← 64' h4

if (fs1 = -zero)

rd ← 64' h8

if (fs1 = +zero)

rd ← 64' h10

if (fs1 = +subnorm)

rd ← 64' h20

if (fs1 = +norm)

rd ← 64' h40

if (fs1 = +inf)

rd ← 64' h80

if (fs1 = sNaN)

rd ← 64' h100

if (fs1 = qNaN)

rd ← 64' h200

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
1110010	00000	fs1	001	rd	1010011	

17.6.3 FCVT.D.H: an instruction that converts half-precision floating-point data to double-precision floating-point data

Syntax:

fcvt.d.h fd, fs1

Operation:fd \leftarrow half_convert_to_double(fs1)**Permission:**

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0100001	00010	fs1	000	fd	1010011	

17.6.4 FCVT.H.D: an instruction that converts double-precision floating-point data to half-precision floating-point data

Syntax:

fcvt.h.d fd, fs1, rm

Operation:fd \leftarrow double_convert_to_half(fs1)**Permission:**

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.d fd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.d fd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.d fd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.d fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.d fd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.h.d fd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0100010	00001	fs1	rm	fd	1010011	

17.6.5 FCVT.H.L: an instruction that converts a signed long integer into a half-precision floating-point number

Syntax:

`fcvt.h.l fd, rs1, rm`

Operation:

$fd \leftarrow \text{signed_long_convert_to_half}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

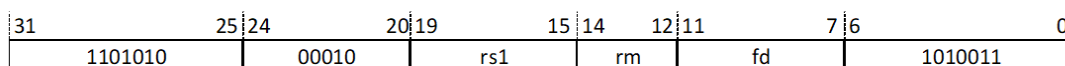
Affected flag bits:

Floating-point status bits NX and OF

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.l fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.l fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.l fd, rs1, fdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.l fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.l fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.h.l fd, rs1`.

Instruction format:

17.6.6 FCVT.H.LU: an instruction that converts an unsigned long integer into a half-precision floating-point number

Syntax:

```
fcvt.h.lu fd, rs1, rm
```

Operation:

```
fd ← unsigned_long_convert_to_half_fp(rs1)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

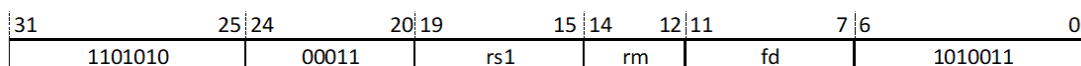
Affected flag bits:

Floating-point status bits NX and OF

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.lu fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.lu fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.lu fd, rs1, fdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.lu fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.lu fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.h.lu fd, rs1`.

Instruction format:

17.6.7 FCVT.H.S: an instruction that converts single precision floating-point data to half-precision floating-point data

Syntax:

`fcvt.h.s fd, fs1, rm`

Operation:

`fd` ← `single_convert_to_half(fs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.s fd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.s fd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.s fd, fs1, fdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.s fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.s fd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcscr` register. The corresponding assembler instruction is `fcvt.h.s fd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0100010	00000	fs1	rm	fd	1010011	

17.6.8 FCVT.H.W: an instruction that converts a signed integer into a half-precision floating-point number

Syntax:

`fcvt.h.w fd, rs1, rm`

Operation:

`fd` ← `signed_int_convert_to_half(rs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NX and OF

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.w fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.w fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.w fd, rs1, fdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.w fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.w fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.h.w fd, rs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
1101010	00000	rs1	rm	fd	1010011	

17.6.9 FCVT.H.WU: an instruction that converts an unsigned integer into a half-precision floating-point number

Syntax:

`fcvt.h.wu fd, rs1, rm`

Operation:

`fd` ← `unsigned_int_convert_to_half_fp(rs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

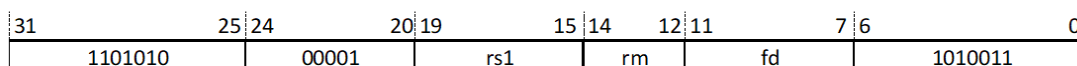
Affected flag bits:

Floating-point status bits NX and OF

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.h.wu fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.h.wu fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.h.wu fd, rs1, fdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.h.wu fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.h.wu fd, rs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.h.wu fd, rs1`.

Instruction format:

17.6.10 FCVT.L.H: an instruction that converts a half-precision floating-point number to a signed long integer

Syntax:

`fcvt.l.h rd, fs1, rm`

Operation:

`rd` ← `half_convert_to_signed_long(fs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

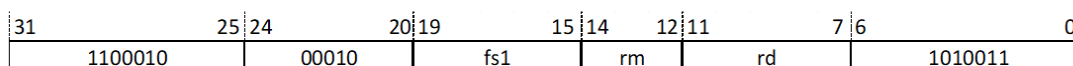
Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.l.h rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.l.h rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.l.h rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.l.h rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.l.h rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.l.h rd, fs1`.

Instruction format:

17.6.11 FCVT.LU.H: an instruction that converts a half-precision floating-point number to an unsigned long integer

Syntax:

`fcvt.lu.h rd, fs1, rm`

Operation:

$rd \leftarrow \text{half_convert_to_unsigned_long}(fs1)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.lu.h rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.lu.h rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.lu.h rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.lu.h rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.lu.h rd, fs1, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fcvt.lu.h rd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
1100010	00011	fs1	rm	rd	1010011	

17.6.12 FCVT.S.H: an instruction that converts half-precision floating-point data to single precision floating-point data

Syntax:

`fcvt.s.h fd, fs1`

Operation:

`fd` ← `half_convert_to_single(fs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0100000		00010		fs1		000		fd		1010011	

17.6.13 FCVT.W.H: an instruction that converts a half-precision floating-point number to a signed integer

Syntax:

`fcvt.w.h rd, fs1, rm`

Operation:

`tmp ← half_convert_to_signed_int(fs1)`

`rd ← sign_extend(tmp)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.w.h rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.w.h rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.w.h rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.w.h rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.w.h rd, fs1, rmm`.

- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is `fcvt.w.h rd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
1100010	00000	fs1	rm	rd	1010011	

17.6.14 FCVT.WU.H: an instruction that converts a half-precision floating-point number to an unsigned integer

Syntax:

```
fcvt.wu.h rd, fs1, rm
```

Operation:

```
tmp ← half_convert_to_unsigned_int(fs1)
```

```
rd ← sign_extend(tmp)
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fcvt.wu.h rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.wu.h rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.wu.h rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.wu.h rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.wu.h rd, fs1, rmm`.

- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is `fcvt.wu.h rd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
1100010	00001	fs1	rm	rd	1010011	

17.6.15 FDIV.H: a half-precision floating-point division instruction**Syntax:**

`fdiv.h fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 / fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fdiv.h fd fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fdiv.h fd fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fdiv.h fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fdiv.h fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fdiv.h fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.

- 3' b111: Dynamically rounds off based on the rm bit of the fcsr register. The corresponding assembler instruction is `fdiv.h fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001110			fs2		fs1		rm		fd		1010011

17.6.16 FEQ.H: an equal instruction that compares two half-precision numbers**Syntax:**

```
feq.h rd, fs1, fs2
```

Operation:

```
if(fs1 == fs2)
```

```
    rd ← 1
```

```
else
```

```
    rd ← 0
```

Permission:

```
M mode/S mode/U mode
```

Exception:

```
Illegal instruction.
```

Affected flag bits:

```
Floating-point status bit NV
```

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1010010			fs2		fs1		010		rd		1010011

17.6.17 FLE.H: a less than or equal to instruction that compares two half-precision floating-point numbers**Syntax:**

```
fle.h rd, fs1, fs2
```

Operation:

```
if(fs1 <= fs2)
```

```
    rd ← 1
```

else

$rd \leftarrow 0$

Permission:

M mode/S mode/U mode

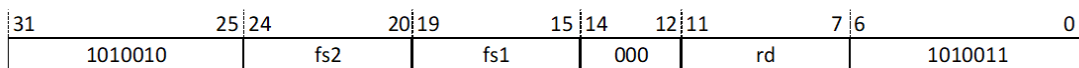
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



17.6.18 FLH: an instruction that loads half-precision floating-point data

Syntax:

flh fd, imm12(rs1)

Operation:

$address \leftarrow rs1 + sign_extend(imm12)$

$fd[15:0] \leftarrow mem[(address+1):address]$

$fd[63:16] \leftarrow 48' hfffffff$

Permission:

M mode/S mode/U mode

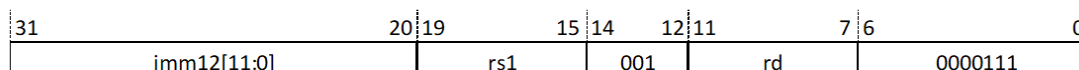
Exception:

Unaligned access, access error, page error, or illegal instruction.

Affected flag bits:

None

Instruction format:



17.6.19 FLT.H: a less than instruction that compares two half-precision floating-point numbers

Syntax:

flt.h rd, fs1, fs2

Operation:

if(fs1 < fs2)

rd ← 1

else

rd ← 0

Permission:

M mode/S mode/U mode

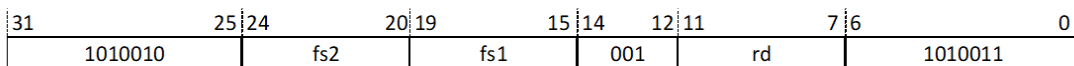
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



17.6.20 FMADD.H: a half-precision floating-point multiply-add instruction

Syntax:

fmadd.h fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1*fs2 + fs3

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

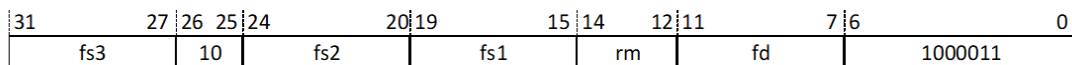
Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmadd.h fd, fs1, fs2, fs3`.

Instruction format:**17.6.21 FMAX.H: a half-precision floating-point maximum instruction****Syntax:**

```
fmmax.h fd, fs1, fs2
```

Operation:

```
if(fs1 >= fs2)
```

```
    fd ← fs1
```

```
else
```

```
    fd ← fs2
```

Permission:

```
M mode/S mode/U mode
```

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0010110	fs2	fs1	001	fd	1010011	

17.6.22 FMIN.H: a half-precision floating-point minimum instruction

Syntax:

fmin.h fd, fs1, fs2

Operation:

if(fs1 >= fs2)

fd ← fs2

else

fd ← fs1

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0010110	fs2	fs1	000	fd	1010011	

17.6.23 FMSUB.H: a half-precision floating-point multiply-subtract instruction

Syntax:

fmsub.h fd, fs1, fs2, fs3, rm

Operation:

fd ← fs1 * fs2 - fs3

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmsub.h fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			10		fs2		fs1		rm		fd		1000111

17.6.24 FMUL.H: a half-precision floating-point multiply instruction**Syntax:**

`fmul.h fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 * fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmul.h fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmul.h fd, fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmul.h fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmul.h fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmul.h fd, fs1, fs2, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmul.h fs1, fs2`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0001010	fs2	fs1	rm	fd	1010011	

17.6.25 FMV.H.X: a half-precision floating-point write transmit instruction**Syntax:**

```
fmv.h.x fd, rs1
```

Operation:

$$fd[15:0] \leftarrow rs1[15:0]$$

$$fd[63:16] \leftarrow 48' \text{ hfffffffffff}$$
Permission:

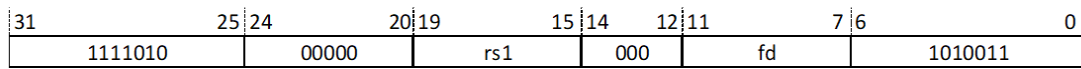
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

17.6.26 FMV.X.H: a transmission instruction that reads half-precision floating-point registers

Syntax:

fmv.x.h rd, fs1

Operation:

tmp[15:0] ← fs1[15:0]

rd ← sign_extend(tmp[15:0])

Permission:

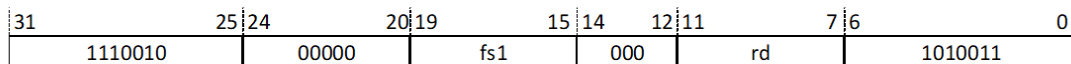
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

17.6.27 FNMADD.H: a half-precision floating-point negate-(multiply-add) instruction

Syntax:

fnmadd.h fd, fs1, fs2, fs3, rm

Operation:

fd ← -(fs1*fs2 + fs3)

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fmadd.h fd,fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3		10		fs2		fs1		rm		fd		1001111	

17.6.28 FNMSUB.H: a half-precision floating-point negate-(multiply-subtract) instruction

Syntax:

`fnmsub.h fd, fs1, fs2, fs3, rm`

Operation:

$fd \leftarrow -(fs1 * fs2 - fs3)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3, rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fnmsub.h fd,fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			10		fs2		fs1		rm		fd		1001011

17.6.29 FSGNJ.H: a half-precision floating-point sign-injection instruction

Syntax:

`fsgnj.h fd, fs1, fs2`

Operation:

$fd[14:0] \leftarrow fs1[14:0]$

$fd[15] \leftarrow fs2[15]$

$fd[63:16] \leftarrow 48' \text{ hfffffffffff}$

Permission:

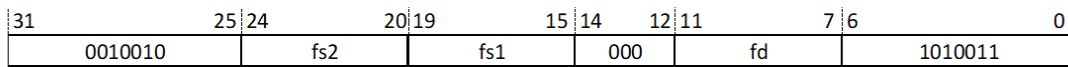
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

17.6.30 FSGNHN.H: a half-precision floating-point sign-injection negate instruction

Syntax:

fsgnfn.h fd, fs1, fs2

Operation: $fd[14:0] \leftarrow fs1[14:0]$ $fd[15] \leftarrow ! fs2[15]$ $fd[63:16] \leftarrow 48' hfffffff$ **Permission:**

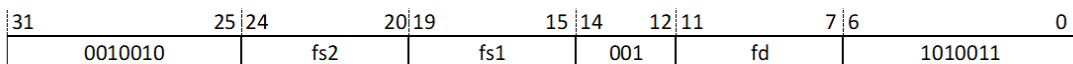
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

17.6.31 FSGNHN.H: a half-precision floating-point sign-injection XOR instruction

Syntax:

fsgnfn.h fd, fs1, fs2

Operation: $fd[14:0] \leftarrow fs1[14:0]$ $fd[15] \leftarrow fs1[15] \wedge fs2[15]$ $fd[63:16] \leftarrow 48' hfffffff$ **Permission:**

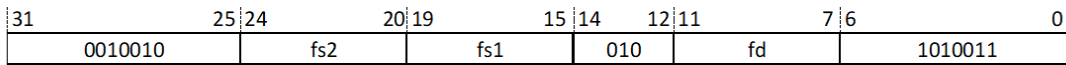
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:**17.6.32 FSH: an instruction that stores half-precision floating point numbers****Syntax:**

fsh fs2, imm12(fs1)

Operation:

address ← fs1 + sign_extend(imm12)

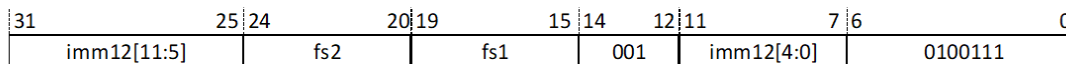
mem[(address+1):address] ← fs2[15:0]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:**17.6.33 FSQRT.H: an instruction that calculates the square root of the half-precision floating-point number****Syntax:**

fsqrt.h fd, fs1, rm

Operation:

fd ← sqrt(fs1)

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fsqrt.h fd, fs1,rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fsqrt.h fd, fs1,rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fsqrt.h fd, fs1,rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fsqrt.h fd, fs1,rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fsqrt.h fd, fs1,rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fsqrt.h fd, fs1`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0101110	00000	fs1	rm	fd	1010011	

17.6.34 FSUB.H: a half-precision floating-point subtract instruction**Syntax:**

`fsub.h fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 - fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fsub.h fd, fs1,fs2,rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fsub.h fd, fs1,fs2,rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fsub.h fd, fs1,fs2,rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fsub.h fd, fs1,fs2,rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fsub.h fd, fs1,fs2,rmm`.
- 3' b101: This code is reserved and not used.
- 3' b110: This code is reserved and not used.
- 3' b111: Dynamically rounds off based on the `rm` bit of the `fcsr` register. The corresponding assembler instruction is `fsub.h fd, fs1,fs2`.

Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
0000110	fs2	fs1	rm	fd	1010011	

Appendix C Control Registers

This section describes the M-mode control registers, S-mode control registers, and U-mode control registers.

18.1 Appendix C-1 M-mode control registers

M-mode control registers are classified by function into M-mode information registers, M-mode exception configuration registers, M-mode exception handling registers, M-mode memory protection registers, M-mode counter registers, and M-mode counter configuration registers.

18.1.1 M-mode information register group

18.1.1.1 Machine vendor ID register (mvendorid)

The mvendorid register stores the vendor IDs of T-Head Semiconductor Co., Ltd. It is not defined and the values are all zero.

This register is 64 bits wide and is read-only in M-mode. Accesses in non-M-mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.2 Machine architecture ID register (marchid)

The marchid register stores the architecture IDs of CPU cores. It stores internal IDs of T-Head Semiconductor Co., Ltd. and its reset value is subject to the product.

This register is 64 bits wide and is read-only in M-mode. Accesses in non-M-mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.3 Machine implementation ID register (mimpid)

The mimpid register stores hardware implementation IDs of CPU cores. This register is not implemented by C908, and its read access value is 0.

This register is 64 bits wide and is read-only in M-mode. Accesses in non-M-mode and writes in M-mode will cause an illegal instruction exception.

18.1.1.4 Machine hart ID register (mhartid)

The mhartid register stores hart IDs of CPU cores.

This register is 64 bits wide and is read-only in M-mode. Accesses in non-M-mode and writes in M-mode will cause an illegal instruction exception.

18.1.2 M-mode exception configuration register group

18.1.2.1 Machine status register (mstatus)

The mstatus register stores status and control information of the CPU in M-mode, including the global interrupt enable bit, exception preserve interrupt enable bit, and exception preserve privilege mode bit.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

	63	62																																36	35	34	33	32
SD	0																																		SXL	UXL		
Reset	0	0																																		2	2	

	31																								25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0																							VS	TSR	TW				XS	FS	MPP	0	SPP		0		0	MIE	0	SIE	0									
Reset	0																							0	0	0	0	0	0	0	0	0	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	

Machine status register (mstatus)

SIE: supervisor interrupt enable bit

- When SIE is 0, S-mode interrupts are invalid.
- When SIE is 1, S-mode interrupts are valid.

This bit is reset to 0 when the CPU is downgraded to the S-mode response interrupt, and is set to the value of SPIE when the CPU exits the interrupt service program.

MIE: machine interrupt enable bit

- When MIE is 0, M-mode interrupts are invalid.
- When MIE is 1, M-mode interrupts are valid.

This bit is reset to 0 when the response is interrupted in M-mode on the CPU, and is set to the value of MPIE when the CPU exits the interrupt service program.

SPIE: supervisor preserve interrupt enable bit

This bit stores the value of the SIE bit before the response is interrupted in S-mode on the CPU.

This bit will be reset to 0, and set to 1 when the CPU exits the interrupt service program.

MPIE: machine preserve interrupt enable bit

This bit stores the value of the MIE bit before the response is interrupted in M-mode on the CPU.

This bit will be reset to 0, and set to 1 when the CPU exits the interrupt exception service program.

SPP: supervisor preserve privilege bit

This bit stores the privilege status before the CPU accesses the exception service program in S-mode.

- When SPP is 2' b00, the CPU is in U-mode before accessing the exception service program.
- When SPP is 2' b01, the CPU is in S-mode before accessing the exception service program.

This bit will be reset to 2' b01.

MPP: machine preserve privilege bit

This bit stores the privilege status before the CPU accesses the exception service program in M-mode.

- When MPP is 2' b00, the CPU is in U-mode before entering the exception service program.
- When MPP is 2' b01, the CPU is in S-mode before accessing the exception service program.
- When MPP is 2' b11, the CPU is in M-mode before entering the exception service program.

This bit will be reset to 2' b11.

FS: floating-point status bit

This bit determines whether to store floating-point registers during context switching.

- When FS is 2' b00, the floating-point unit is in the Off state and exceptions will occur for accesses to related floating-point registers.
- When FS is 2' b01, the floating-point unit is in the Initial state.

- When FS is 2' b10, the floating-point unit is in the Clean state.
- When FS is 2' b11, the floating-point unit is in the Dirty state, which means the floating-point and control registers have been modified.

XS: extension status bit

C908 has no extension units, and therefore this bit is fixed to 0.

MPRV: modify privilege mode

- When MPRV is 1, load and store requests are executed based on the privilege mode in MPP.
- When MPRV is 0, load and store requests are executed based on the current privilege mode of the CPU.

SUM: allow S-mode accesses to U-mode virtual memory spaces

- When SUM is 1, load, storage, and value-taking requests can be initiated in S-mode to access U-mode virtual memory spaces.
- When SUM is 0, load, storage, and value-taking requests cannot be initiated in S-mode to access virtual memory spaces marked as U-mode.

MXR: allow accesses of load requests to memory spaces marked as executable

- When MXR is 1, accesses of load requests are allowed to virtual memory spaces marked as executable or readable.
- When MXR is 0, accesses of load requests are allowed only to virtual memory spaces marked as readable.

TVM: trap virtual memory

- When TVM is 1, an illegal instruction exception occurs for reads and writes to the satp control register and for the execution of the sfence instruction in S-mode.
- When TVM is 0, reads and writes to the satp control register and the execution of the sfence instruction are allowed in S-mode.

TW: timeout wait

- When TW is 1, an illegal instruction exception occurs if the WFI instruction is executed in S-mode.
- When TW is 0, the WFI instruction can be executed in S-mode.

TSR: trap sret

- When TSR is 1, an illegal instruction exception occurs if the sret instruction is executed in S-mode.

When TSR is 0, the sret instruction can be executed in S-mode.

VS: vector status bit

This bit determines whether to store vector registers during context switching.

- When VS is 2' b00, the vector unit is in the Off state and exceptions will occur for accesses to related vector registers.
- When VS is 2' b01, the vector unit is in the Initial state.
- When VS is 2' b10, the vector unit is in the Clean state.
- When VS is 2' b11, the vector unit is in the Dirty state, which means the vector registers and vector control registers have been modified.

The VS bit is valid only when the vector execution unit is configured, and is always 0 if the vector execution unit is not configured.

UXL: register width

This bit is read-only and always 2, which means the register is 64 bits wide in U-mode.

SXL: register width

This bit is read-only and always 2, which means the register is 64 bits wide in S-mode.

SD: dirty state sum bit of the floating-point, vector, and extension units

- When SD is 1, the floating-point unit, vector unit, or extension unit is in the Dirty state.
- When SD is 0, none of the floating-point, vector, and extension units is in the Dirty state.

18.1.2.2 M-mode instruction set architecture register (misa)

The misa register stores the features of the instruction set architecture supported by the CPU.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

C908 supports the RV64GC instruction set architecture, and the reset value of the MISA register is 0x80000000094112d. For more information about the assignment rules, see the official document of RISC-V riscv-privileged.

C908 does not support the dynamic configuration of the MISA register. Writes to this register do not take effect.

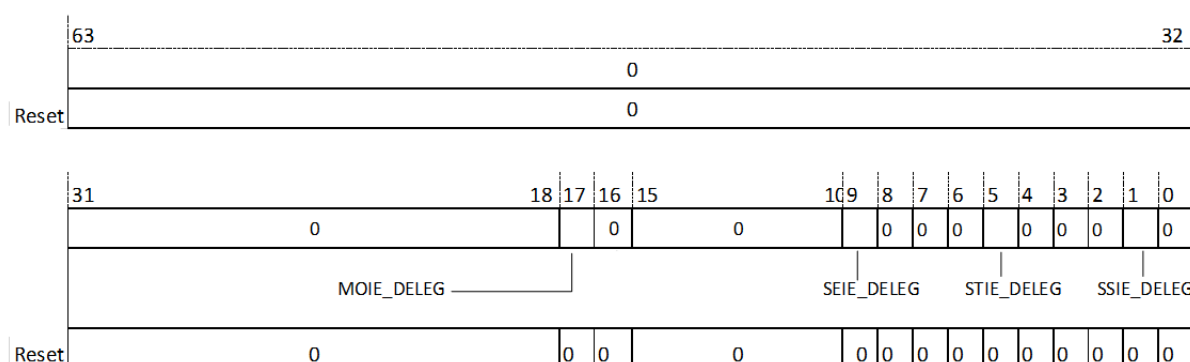
18.1.2.3 M-mode exception downgrade control register (medeleg)

The medeleg register can downgrade exceptions that occur in S-mode and U-mode to S-mode responses. The lower 16 bits of the medeleg register are in one-to-one correspondence to exception vector tables. Exceptions to be downgraded to S-mode responses can be selected.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

18.1.2.4 M-mode interrupt downgrade control register (mideleg)

The mideleg register can downgrade S-mode interrupts to S-mode responses.

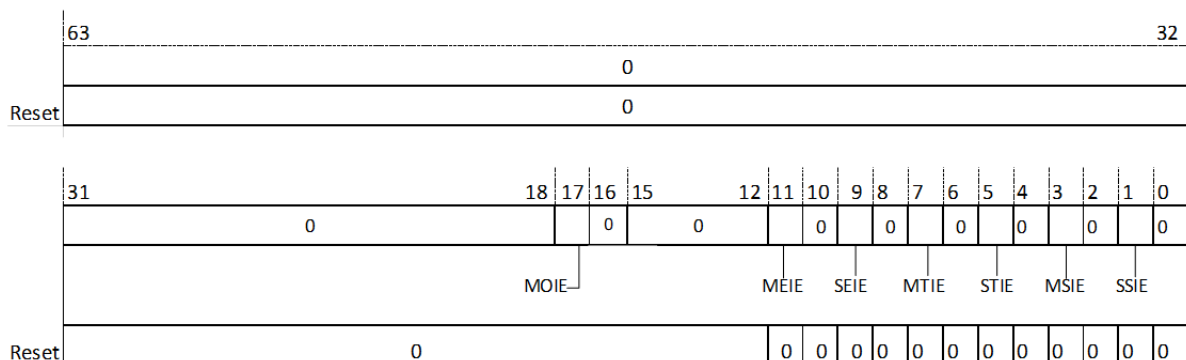


M-mode interrupt downgrade control register (mideleg)

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

18.1.2.5 M-mode interrupt-enable register (mie)

The mie register enables and masks different types of interrupts. This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode interrupt-enable register (mie)

SSIE: S-mode software interrupt enable bit

- When SSIE is 0, S-mode software external interrupts are invalid.
- When SSIE is 1, S-mode software external interrupts are valid.

MSIE: M-mode software interrupt enable bit

Chapter 18. Appendix C Control Registers

- When MSIE is 0, M-mode software interrupts are invalid.
- When MSIE is 1, M-mode software interrupts are valid.

STIE: S-mode timer interrupt enable bit

- When STIE is 0, S-mode timer interrupts are invalid.
- When STIE is 1, S-mode timer interrupts are valid.

MTIE: M-mode timer interrupt enable bit

- When MTIE is 0, M-mode timer interrupts are invalid.
- When MTIE is 1, M-mode timer interrupts are valid.

SEIE: S-mode external interrupt enable bit

- When SEIE is 0, S-mode external interrupts are invalid.
- When SEIE is 1, S-mode external interrupts are valid.

MEIE: M-mode external interrupt enable bit

- When MEIE is 0, M-mode external interrupts are invalid.
- When MEIE is 1, M-mode external interrupts are valid.

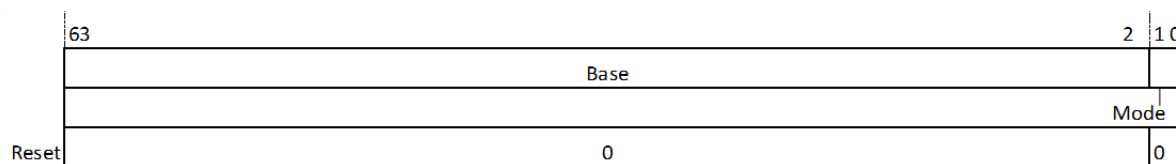
MOIE: M-mode overflow interrupt enable bit

- When MOIE is 0, M-mode counter overflow interrupts are invalid.
- When MOIE is 1, M-mode counter overflow interrupts are valid.

18.1.2.6 M-mode trap vector base address register (mtvec)

The mtvec register stores the entry address of the exception service program.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode trap vector base address register (mtvec)

BASE: vector base address bit

The BASE bit indicates the upper 62 bits of the entry address of the exception service program. Combining this base address with 2^7 b00 obtains the entry address of the exception service program.

This bit will be reset to 0.

MODE: vector entry mode bit

- When MODE[1:0] is 2' b00, the base address is used as the entry address for both exceptions and interrupts.
- When MODE[1:0] is 2' b01, the base address is used as the entry address for exceptions, and BASE + 4*cause is used as the entry address for interrupts.

18.1.2.7 M-mode counter access enable register (mcounteren)

The mcounteren register determines whether U-mode counters can be accessed in S-mode.

For more information, see ref:*performance_test*.

18.1.3 M-mode exception handling register group

18.1.3.1 M-mode scratch register (mscratch)

The mscratch register is used by the CPU to back up temporary data in the exception service program. It is usually used to store the entry pointer to the local context space in M-mode.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

18.1.3.2 M-mode exception program counter register (mepc)

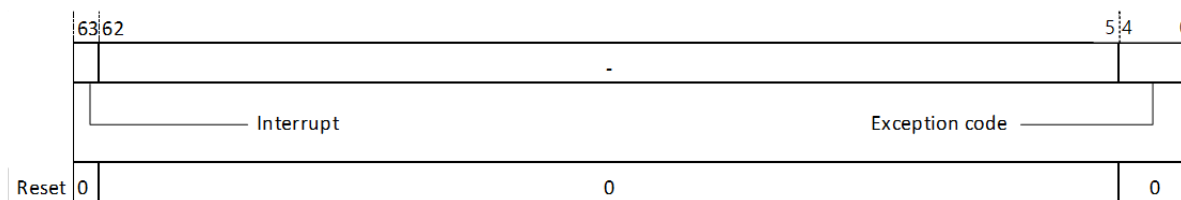
The mepc register stores the program counter value (PC value) when the CPU exits from the exception service program. C908 supports 16 bits wide instructions. The MEPC value is aligned with 16 bits and the lowest bit is 0.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

18.1.3.3 M-mode cause register (mcause)

The mcause register stores the vector numbers of events that trigger exceptions. The vector numbers are used to handle corresponding events in the exception service program.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode cause register (mcause)

Interrupt: interrupt bit

- When the Interrupt bit is 0, the corresponding exception is not triggered by an interrupt. The exception code is parsed as an exception.
- When the Interrupt bit is 1, the corresponding exception is triggered by an interrupt. The exception code is parsed as an interrupt.

Exception Code: exception vector number

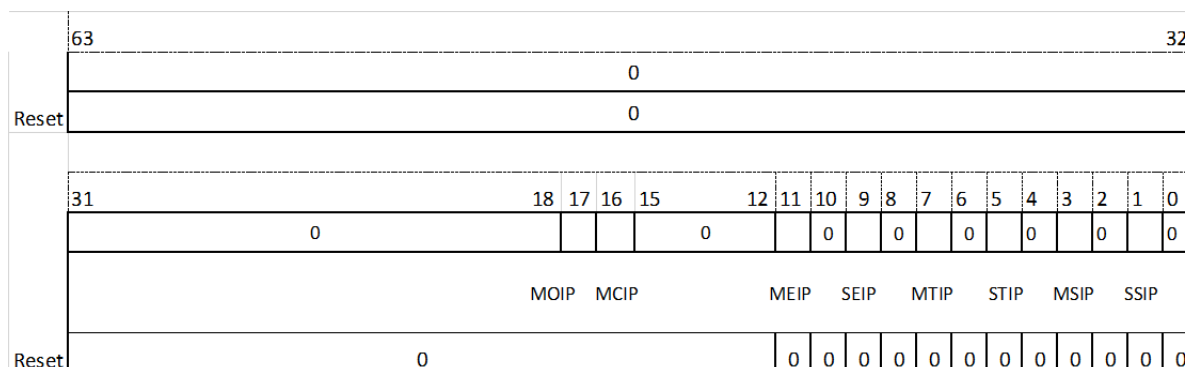
When the CPU encounters an exception, the Exception Code bit will be updated to the value of the exception source.

18.1.3.4 M-mode interrupt-pending register (mip)

The mip register stores information about pending interrupts. When the CPU cannot immediately respond to an interrupt, the corresponding bit in the mip register will be set.

Writing the msip and ssip registers in the CLINT interrupt controller can trigger corresponding interrupts. After the interrupts become valid, the msip bit and ssip bit can be queried based on the corresponding bits in the mip register.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode interrupt-pending register (mip)

SSIP: supervisor software interrupt pending bit

Chapter 18. Appendix C Control Registers

- When SSIP is 0, there is no pending S-mode software interrupt on the CPU.
- When SSIP is 1, there are pending S-mode software interrupts on the CPU.

The SSIP bit is readable and writable in M-mode. After it is delegated to S-mode, it is readable and writable in S-mode. Otherwise, it is read-only in S-mode.

MSIP: M-mode software interrupt pending bit

- When MSIP is 0, there is no pending M-mode software interrupt on the CPU.
- When MSIP is 1, there are pending M-mode software interrupts on the CPU.

This bit is read-only.

STIP: S-mode timer interrupt pending bit

- When STIP is 0, there is no pending S-mode timer interrupt on the CPU.
- When STIP is 1, there are pending S-mode timer interrupts on the CPU.

MTIP: M-mode timer interrupt pending bit

- When MTIP is 0, there is no pending M-mode timer interrupt on the CPU.
- When MTIP is 1, there are pending M-mode timer interrupts on the CPU.

SEIP: S-mode external interrupt pending bit

- When SEIP is 0, there is no pending S-mode external interrupt on the CPU.
- When SEIP is 1, there are pending S-mode external interrupts on the CPU.

MEIP: machine external interrupt pending bit

- When MEIP is 0, there is no pending M-mode external interrupt on the CPU.
- When MEIP is 1, there are pending M-mode external interrupts on the CPU.

MOIP: M-mode overflow interrupt pending bit

- When MOIP is 0, there is no pending M-mode counter overflow interrupt on the CPU.
- When MOIP is 1, there are pending M-mode counter overflow interrupts on the CPU.

18.1.4 M-mode memory protection registers

M-mode memory protection registers are related to the settings of the memory protection unit.

18.1.4.1 Physical memory protection configuration register (pmpcfg)

The pmpcfg register configures access permissions and address matching mode for the physical memory.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

Chapter 18. Appendix C Control Registers

For more information, see [ref:physical_mem_pmpcfg](#).

18.1.4.2 Physical memory address register (pmpaddr)

The pmpaddr register configures the address range of each entry of the physical memory.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

For more information, see [ref:physical_mem_pmpaddr](#).

18.1.5 M-mode counter registers

M-mode counter registers belong to the PMU and collect software and hardware information during a program operation for software development personnel to optimize programs.

18.1.5.1 M-mode cycle counter (mcycle)

The mcycle counter stores the cycles executed by the CPU. When the CPU is in the execution state (non-low power state), the mcycle register increases the count upon each execution cycle.

The mcycle counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.1.5.2 M-mode instructions-retired counter (minstret)

The minstret counter stores the number of retired instructions of the CPU. The minstret register increases the count when each instruction retires.

The minstret counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.1.5.3 M-mode event counter (mhpmcountern)

The mhpmcountern counter counts events.

The mhpmcountern counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.1.6 M-mode counter configuration registers

The M-mode counter configuration register (mhpmeventn) selects events for M-mode event counters.

18.1.6.1 M-mode event selector (mhpmeventn)

The events mhpmevent3-31 selected by the mhpmeventn register for M-mode event counters mhpcounter3-31 are in one-to-one correspondence. In C908, event counters can count only specified events. Therefore, only specified values can be written to mhpmevent3-31.

The mhpmeventn counter is 64 bits wide and will be reset to 0.

For more information, see *ref:performance_test_mhpmevent*.

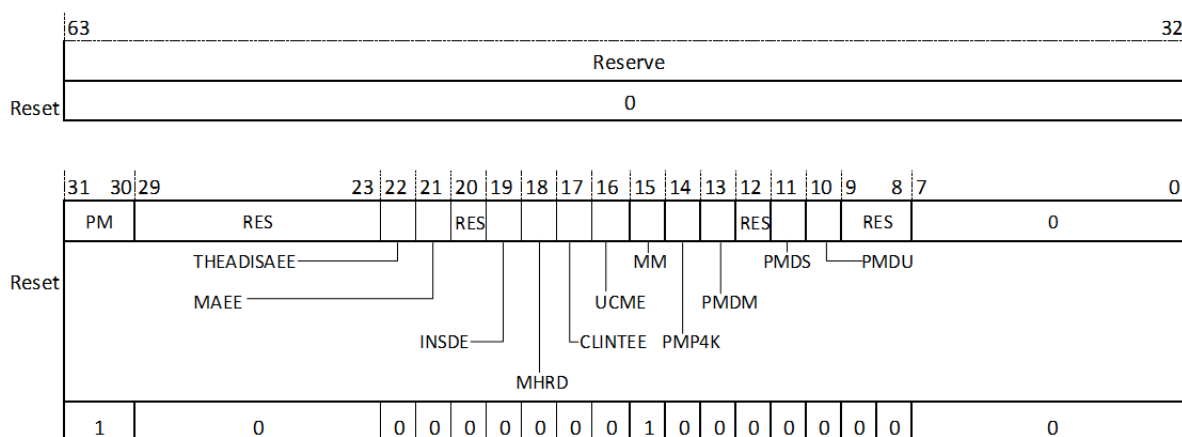
18.1.7 M-mode CPU control and status extension registers

C908 extends some registers for the CPU and status, including the M-mode extension status register (mxstatus) and M-mode hardware control register (mhcr), M-mode hardware operation register (mcor), M-mode L2 Cache control register (mccr2), M-mode implicit operation register (mhint), M-mode reset vector base address register (mrvbr), S-mode counter write enable register (mcounterwen), M-mode event interrupt enable register (mcounterinten), and M-mode event overflow mark register (mcounterof).

18.1.7.1 M-mode extension status register (mxstatus)

The mxstatus register stores the current privilege mode of the CPU and the enable bit of the extension functions of C908.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode extension status register (mxstatus)

PMDU: U-mode performance monitoring count enable bit

When PMDU is 0, performance counters are allowed to count in U-mode.

When PMDU is 1, performance counters are not allowed to count in U-mode.

PMDS: S-mode performance monitoring count enable bit

When PMDS is 0, performance counters are allowed to count in S-mode.

When PMDS is 1, performance counters are not allowed to count in S-mode.

PMDM: M-mode performance monitoring count enable bit

When PMDM is 0, performance counters are allowed to count in M-mode.

When PMDM is 1, performance counters are not allowed to count in M-mode.

PMP4K: PMP minimum granularity control bit

The minimum PMP granularity supported by C908 is 4 KB, which is not affected by this bit.

MM: misaligned access enable bit

When MM is 0, misaligned accesses are not supported and cause misaligned exceptions.

When MM is 1, misaligned accesses are supported and processed by hardware. (The default value of this bit is 1 in C908.)

UCME: execute extended cache instructions in U-mode

When UCME is 0, extended cache instructions cannot be executed in U-mode. Otherwise, instruction exceptions may occur.

When UCME is 1, extended cache instructions can be executed in U-mode.

CLINTEE: Clint timer/software interrupt supervisor extension enable bit

When CLINTEE is 0, supervisor software interrupts and timer interrupts initiated by Clint are not responded to.

When CLINTEE is 1, supervisor software interrupts and timer interrupts initiated by Clint can be responded to.

MHRD: disable hardware writeback

When MHRD is 0, hardware writeback is performed if the TLB is missing.

When MHRD is 1, hardware writeback is not performed after the TLB is missing.

INSDE: disable Icache snoop D-Cache

When INSDE is 0, D-Cache is snooped after I-Cache is missing.

When INSDE is 1, D-Cache is not snooped after I-Cache is missing.

MAEE: extend MMU address attribute

When MAEE is 0, the MMU address attribute is not extended.

When MAEE is 1, the address attribute is extended in the PTE of the MMU. Users can configure the address attribute of pages.

THEADISAEAE: enables extended instruction sets

When THEADISAEAE is 0, using C908 extended instruction sets causes instruction exceptions.

When THEADISAEAE is 1, C908 extended instruction sets can be used.

PM: privilege mode

When PM is 2' b00, the CPU is running in U-mode.

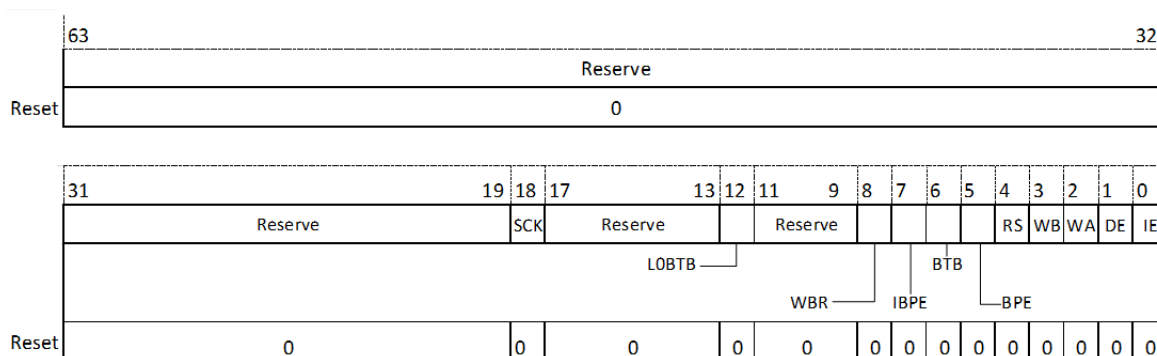
When PM is 2' b01, the CPU is running in S-mode.

When PM is 2' b11, the CPU is running in M-mode. (The PM bit will be reset to M-mode.)

18.1.7.2 M-mode hardware configuration register (mhcr)

The mhcr register configures the performance and functions of the CPU.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode hardware configuration register (mhcr)

IE: I-Cache enable bit

When IE is 0, I-Cache is disabled.

When IE is 1, I-Cache is enabled.

DE: D-Cache enable bit

When DE is 0, D-Cache is disabled.

When DE is 1, D-Cache is enabled.

WA: cache write allocate set bit

When WA is 0, the data cache is in write non-allocate mode.

When WA is 1, the data cache is in write allocate mode.

WB: cache writeback set bit

Chapter 18. Appendix C Control Registers

When WB is 0, the data cache is in write-through mode.

When WB is 1, the data cache is in writeback mode.

C908 supports only the writeback mode. Therefore, the WB bit is fixed to 1.

RS: address return stack set bit

When RS is 0, the return stack is disabled.

When RS is 1, the return stack is enabled.

BPE: branch prediction enable bit

When BPE is 0, branch prediction is disabled.

When BPE is 1, branch prediction is enabled.

BTB: branch target prediction enable bit

When BTB is 0, branch target prediction is disabled.

When BTB is 1, branch target prediction is enabled.

IBPE: indirect branch prediction enable bit

When IBPE is 0, indirect branch prediction is disabled.

When IBPE is 1, indirect branch prediction is enabled.

WBR: write burst transmission enable bit

When WBR is 0, write burst transmission is not supported.

When WBR is 1, write burst transmission is supported.

The WBR bit is fixed to 1 by default in C908, and cannot be modified.

L0BTB: level-1 branch target prediction enable bit

When L0BTB is 0, level-1 branch target prediction is disabled.

When L0BTB is 1, level-1 branch target prediction is enabled.

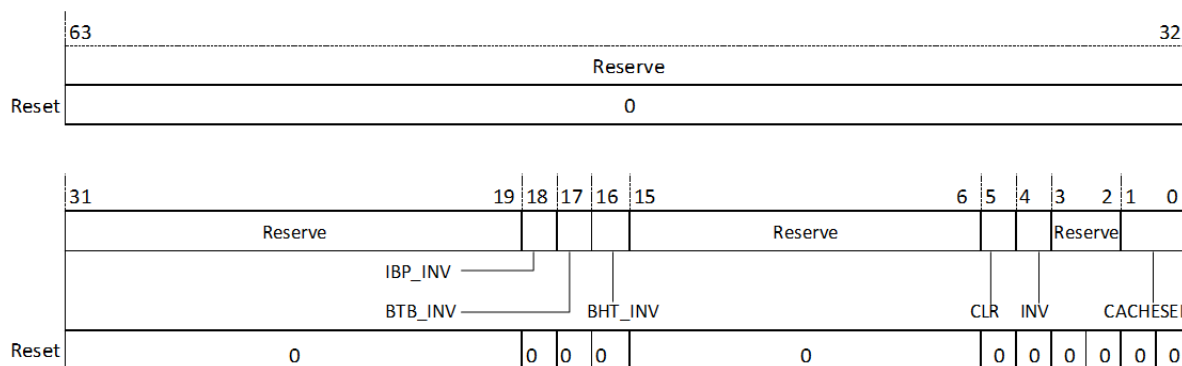
SCK: ratio of system clock to CPU clock

This bit indicates the ratio of the system clock to the CPU clock. The calculation format is SCK+1. There are corresponding pins on the CPU. The SCK bit is configured during a reset and cannot be modified later.

18.1.7.3 M-mode hardware operation register (mcor)

The mcor register performs related operations on caches and branch predictors.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode hardware operation register (mcor)

CACHE_SEL: cache select bit

When CACHE_SEL is 2' b01, the instruction cache is selected.

When CACHE_SEL is 2' b10, the data cache is selected.

When CACHE_SEL is 2' b11, the instruction and data caches are selected.

INV: cache invalidate bit

When INV is 0, caches are not invalidated.

When INV is 1, caches are invalidated.

CLR: dirty entry clear bit

When CLR is 0, dirty entries in caches are not written out of the chip.

When CLR is 1, dirty entries in caches are written out of the chip.

BHT_INV: BHT invalidate bit

When BHT_INV is 0, data in branch history tables (BHTs) is not invalidated.

When BHT_INV is 1, data in BHTs is invalidated.

BTB_INV: BTB invalidate bit

When BTB_INV is 0, data in branch target buffers (BTBs) is not invalidated.

When BTB_INV is 1, data in BTBs is invalidated.

IBP_INV: IBP invalidate bit

When IBP_INV is 0, indirect branch prediction (IBP) data is not invalidated.

When IBP_INV is 1, IBP data is invalidated.

All the preceding invalidate and dirty entry clear bits are set to 1 when corresponding operations are in progress and reset to 0 when the operations are completed.

Chapter 18. Appendix C Control Registers

When DSETUP is 1, the data RAM requires an additional setup cycle.

TLTNCY: tag RAM access cycle configure bit for the L2 Cache

When TLTNCY is 0, the tag RAM access cycle is 1.

When TLTNCY is 1, the tag RAM access cycle is 2.

When TLTNCY is 2, the tag RAM access cycle is 3.

When TLTNCY is 3, the tag RAM access cycle is 4.

When TLTNCY is 4, the tag RAM access cycle is 5.

TSETUP: tag RAM setup configure bit for the L2 Cache

When TSETUP is 0, the tag RAM does not require an additional setup cycle.

When TSETUP is 1, the tag RAM requires an additional setup cycle.

IPRF: instruction prefetch capability of the L2 Cache

This bit indicates the number of prefetched cache lines when desired data of a value-taking request is missing in the L2 Cache.

When IPRF is 0, instruction prefetch is disabled for the L2 Cache.

When IPRF is 1, one cache line is prefetched.

When IPRF is 2, two cache lines are prefetched.

When IPRF is 3, three cache lines are prefetched.

TPRF: TLB prefetch enable bit for the L2 Cache

When TPRF is 0, TLB prefetch is disabled for the L2 Cache.

When TPRF is 1, TLB prefetch is enabled for the L2 Cache.

18.1.7.5 M-mode implicit operation register (mhint)

The mhint register controls the enable/disable of multiple functions of caches.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

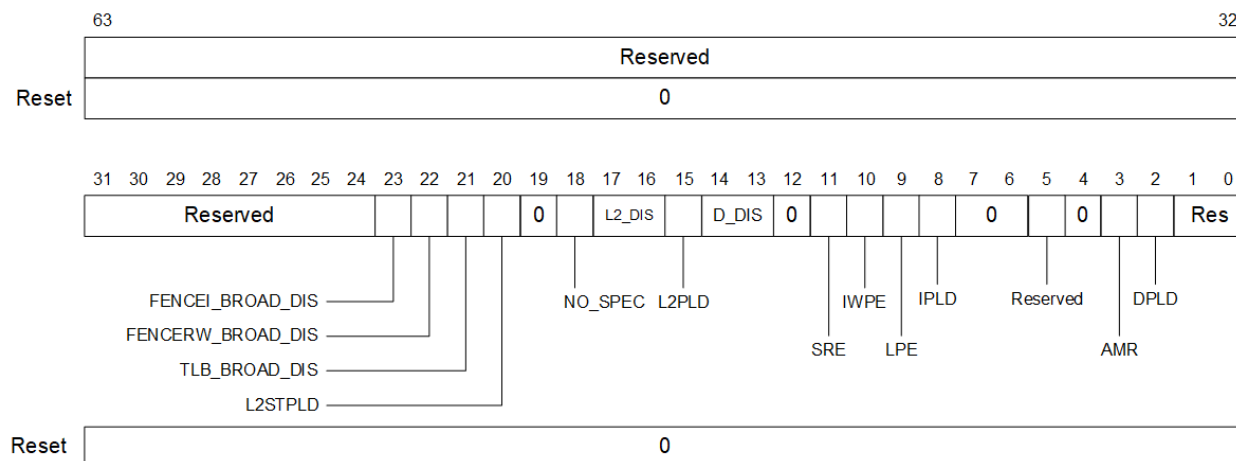
M-mode implicit operation register (mhint)

DPLD: D-Cache prefetch enable bit

When DPLD is 0, D-Cache prefetch is disabled.

When DPLD is 1, D-Cache prefetch is enabled.

AMR: write allocate policy automatic adjustment enable bit for the L1 Cache



When AMR is 0, the write allocate policy is subject to the page attribute WA of the access address.

When AMR is 1, if multiple cache lines are stored continuously, desired data of subsequent storage operations of continuous addresses is no longer written to the L1 Cache.

IPLD: I-Cache prefetch enable bit

When IPLD is 0, I-Cache prefetch is disabled.

When IPLD is 1, I-Cache prefetch is enabled.

LPE: loop acceleration enable bit

When LPE is 0, loop acceleration is disabled.

When LPE is 1, loop acceleration is enabled.

IWPE: I-Cache way prediction enable bit

When IWPE is 0, way prediction is disabled for I-Cache.

When IWPE is 1, way prediction is enabled for I-Cache.

SRE: single retirement enable bit

When SRE is 0, single retirement mode is disabled.

When SRE is 1, single retirement mode is enabled.

D_DIS: number of prefetched cache lines in D-Cache

When D_DIS is 0, two cache lines are prefetched.

When D_DIS is 1, four cache lines are prefetched.

When D_DIS is 2, eight cache lines are prefetched.

When D_DIS is 3, 16 cache lines are prefetched.

The default value is 0.

Chapter 18. Appendix C Control Registers

L2PLD: the L2 Cache prefetch enable bit

When L2PLD is 0, L2 Cache prefetch is disabled.

When L2PLD is 1, L2 Cache prefetch is enabled.

L2_DIS: number of prefetched cache lines in the L2 Cache

When L2_DIS is 0, eight cache lines are prefetched.

When L2_DIS is 1, 16 cache lines are prefetched.

When L2_DIS is 2, 32 cache lines are prefetched.

When L2_DIS is 3, 64 cache lines are prefetched.

The L2 Cache prefetch is based on the L1 Cache prefetch.

NO_SPEC: spec fail prediction enable bit

When NO_SPEC is 0, spec fail prediction is disabled.

When NO_SPEC is 1, spec fail prediction is enabled.

L2STPLD: store prefetch enable bit for the L2 Cache

When L2STPLD is 0, store prefetch is disabled for the L2 Cache.

When L2STPLD is 1, store prefetch is enabled for the L2 Cache.

TLB_BROAD_DIS: the TLB fence operation broadcast disable bit

When TLB_BROAD_DIS is 0, sfence.vma instruction operations are broadcast to other cores.

When TLB_BROAD_DIS is 1, sfence.vma instruction operations are not broadcast.

This bit is invalid when there is only one core.

FENCERW_BROAD_DIS: fence operation broadcast disable bit

When FENCERW_BROAD_DIS is 0, fence instruction operations are broadcast to other cores.

When FENCERW_BROAD_DIS is 1, fence instruction operations are not broadcast.

This bit is invalid when there is only one core.

FENCEI_BROAD_DIS: fence.i operation broadcast disable bit

When FENCEI_BROAD_DIS is 0, fence.i instruction operations are broadcast to other cores.

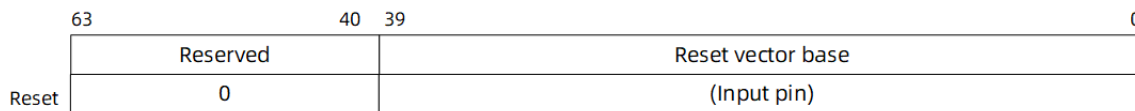
When FENCEI_BROAD_DIS is 1, fence.i instruction operations are not broadcast.

This bit is invalid when there is only one core.

18.1.7.6 M-mode reset vector base address register (mrvbr)

The mrvbr register stores base addresses of reset exception vectors. Each C908 core has an independent mrvbr register.

This register is 64 bits wide and is read-only in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode reset vector base address register (mrvbr)

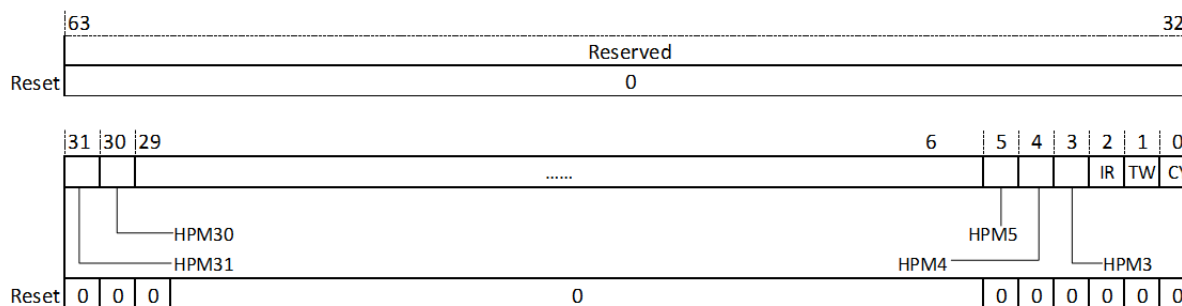
Reset vector base: reset base address

It controls the reset base address of a core.

18.1.7.7 S-mode counter write enable register (mcounterwen)

The mcounterwen register determines whether S-mode event counters can be written in S-mode.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



S-mode counter write enable register (mcounterwen)

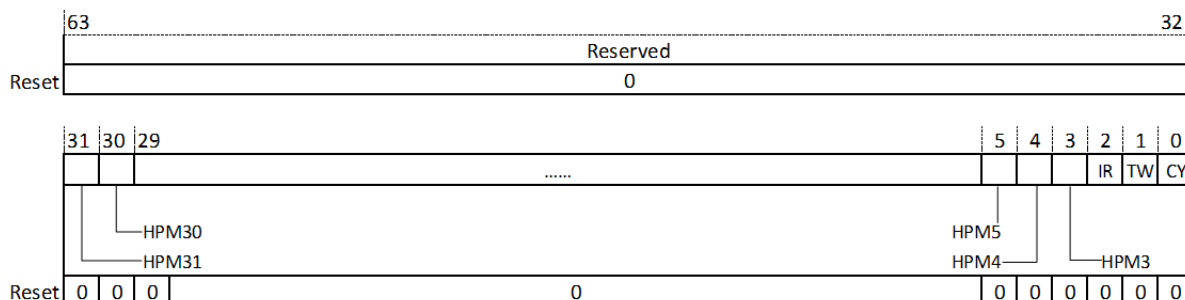
When mcounterwen.bit[n] is 1, writes to the corresponding shpmcounter are allowed in S-mode.

When mcounterwen.bit[n] is 0, writes to the corresponding shpmcounter are not allowed in S-mode, and cause instruction exceptions.

18.1.7.8 M-mode event interrupt enable register (mcounterinten)

The mcounterinten register enables the triggering of interrupts when event counters overflow.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode event interrupt enable register (mcounterinten)

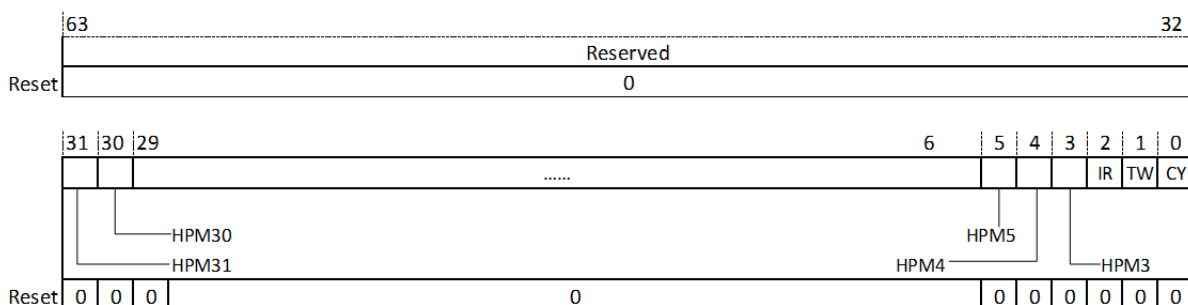
When mcounterinten.bit[n] is 1, an interrupt is triggered when the corresponding mhpmpcounter overflows.

When mcounterinten.bit[n] is 0, an interrupt is not triggered when the corresponding mhpmpcounter overflows.

18.1.7.9 M-mode event overflow mark register (mcounteren)

The mcounteren register marks whether event counters overflow.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode event overflow mark register (mcounteren)

When mcounterof.bit[n] is 1, the corresponding mhpmpcounter overflows.

When mcounterof.bit[n] is 0, the corresponding mhpmpcounter does not overflow.

18.1.8 M-mode cache access extension registers

M-mode cache access extension registers directly read content in the L1 Cache and the L2 Cache for cache debugging.

18.1.8.1 M-mode cache instruction register (mcins)

The mcins register initiates read requests to the L1 or L2 Cache.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

	63		1	0
	-			R
Reset	0			0

M-mode cache instruction register (mcins)

R: cache read access

- When R is 0, cache read requests are not initiated.
- When R is 1, cache read requests are initiated.

18.1.8.2 M-mode cache access index register (mcindex)

The mcindex register configures the location of a cache accessed by read requests.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

	63		32	
	-			
Reset	0			

	31	28	27	25	24	21	20		0
	RID		-		WAY		INDEX		
Reset	0		0		0		0		

M-mode cache access index register (mcindex)

RID: RAM flag bit

This bit specifies the accessed RAM.

- When RID is 0, I-Cache tag RAM is accessed.
- When RID is 1, I-Cache data RAM is accessed.
- When RID is 2, D-Cache tag RAM is accessed.
- When RID is 3, D-Cache data RAM is accessed.
- When RID is 4, L2 Cache tag RAM is accessed.

- When RID is 5, L2 Cache data RAM is accessed.
- When RID is 12, D-Cache LD tag RAM is accessed.

WAY: cache way information

This bit specifies the RAM access way.

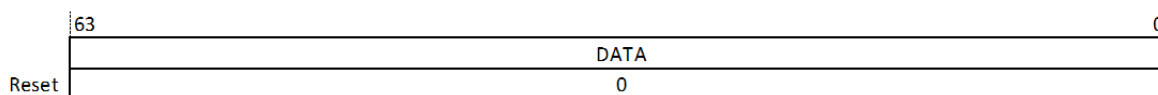
INDEX: cache index

This bit specifies the RAM access index.

18.1.8.3 M-mode cache data register (mcddata0/1)

The mcddata0/1 register records data read from the L1 or L2 Cache.

This register is 64 bits wide and is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.



M-mode cache access data register (mcddata)

Table 18.1: Correspondence between cache data content and RAM types

RAM type	CDATA content
ICACHE TAG	CDATA0[39:12]: TAG CDATA0[0]: VALID
ICACHE DATA	CDATA0–CDATA1: 128bit DATA
DCACHE TAG	CDATA0[39:12]: TAG CDATA0[2]: DIRTY CDATA0[1]: SHARED CDATA0[0]: VALID
DCACHE DATA	CDATA0–CDATA1: 128bit DATA
L2CACHE TAG	CDATA0[39:12]: TAG CDATA[1]: DIRTY CDATA0[0]: VALID
L2CACHE DATA	CDATA0–CDATA1: 128bit DATA

18.1.9 M-mode CPU model registers

18.1.9.1 M-mode CPU model register (mcpuid)

The mcpuid registers store CPU models. The reset value is subject to the product.

18.1.9.2 On-chip bus base address register (mapbaddr)

The mapbaddr register stores the base addresses of on-chip registers (CLINT and PLIC) of the CPU. The value of this register is subject to hardware integration.

18.1.10 Multi-core extension registers

18.1.10.1 Snoop listening enable register (msmpr)

The msmpr register controls whether cores can process listening requests. The listening request processing capability is configured for each core separately. The consistency bus of the L2 subsystem controls the sending of listening requests based on the listening status of each core. This register is readable and writable in M-mode.

The msmpr register is 64 bits wide. Only bit 0 is defined and the other bits are reserved.

Bit 0: SMPEN: core listening enable bit

- When SMPEN is 1' b0, the corresponding core cannot process listening requests, and the L2 subsystem masks the listening requests bound for the core. (This is the reset value.)
- When SMPEN is 1' b1, the corresponding core can process listening requests, and the L2 subsystem sends the listening requests bound for the core.

Before a CPU core is powered off, its SMPEN bit must be set to 0 to disable listening. After a core is powered on, its SMPEN bit must be set to 1 before D-Cache and MMU are enabled. The SMPEN bit must be set to 1 when a core runs properly (including WFI mode). Otherwise, unexpected results may be caused.

18.2 Appendix C-2 S-mode control registers

S-mode control registers are classified by function into S-mode exception configuration registers, S-mode exception handling registers, and S-mode address translation registers.

18.2.1 S-mode exception configuration registers

When exceptions and interrupts are downgraded to S-mode responses, exceptions must be configured through the S-mode exception configuration registers, like in M-mode.

18.2.1.1 S-mode status register (sstatus)

The sstatus register stores status and control information of the CPU in S-mode, including the global interrupt enable bit, exception preserve interrupt enable bit, and exception preserve privilege mode bit. The sstatus register is a partial mapping of the mstatus register.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

	63	62																															36	35	34	33	32
	SD		0																														0		UXL		
Reset	0		0																														2		2		

	31																			25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0		VS		0	0	0			0	XS	FS	0	0	SPP	0	0		0	0	0	0	SIE	0																					
																			MXR			SUM												SPIE											
Reset	0		0		0	0	0	0	0	0	0	0	0	0	3	0	1	0	0	0	0	0	0	0	0																				

S-mode status register (sstatus)

For more information, see [ref:appendix_c12_mstatus](#).

18.2.1.2 S-mode interrupt-enable register (sie)

The sie register controls the enable and mask of different types of interrupts, and is a partial mapping of the mie register. This register is 64 bits wide and readable in S-mode. The write permission in S-mode is determined by the mideleg register of the corresponding bit. Accesses in U-mode will cause an illegal instruction exception.

	63																																																											32		
Reset	0																																																												0	

	31															18	17	16	15					10	9	8	7	6	5	4	3	2	1	0
	0																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
															MOIE			SEIE				STIE			SSIE									
Reset	0																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			

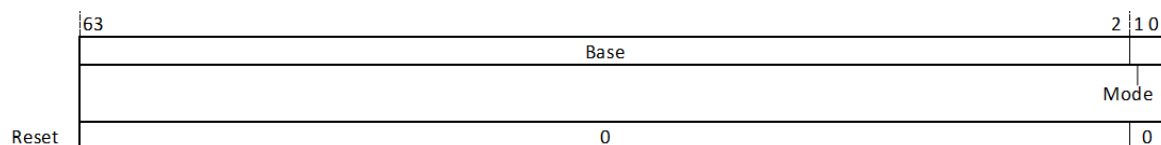
S-mode interrupt-enable register (sie)

For more information, see [ref:appendix_c12_mie](#).

18.2.1.3 S-mode trap vector base address register (stvec)

The stvec register stores the entry address of the exception service program.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.



S-mode trap vector base address register (stvec)

For more information, see [ref:appendix_c12_mtvec](#).

18.2.1.4 S-mode counter access enable register (scounteren)

The scounteren register determines whether U-mode counters can be accessed in U-mode.

For more information, see [ref:performance_test_scounteren](#).

18.2.2 S-mode exception handling registers

18.2.2.1 S-mode scratch register (sscratch)

The sscratch register is used by the CPU to back up temporary data in the exception service program. It is usually used to store the entry pointer to the local context space in S-mode.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.2.2 S-mode exception program counter register (sepc)

The sepc register stores the program counter value (PC value) when the CPU exits from the exception service program. C908 supports 16 bits wide instructions. The values of sepc are aligned with 16 bits and the lowest bit is 0.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.2.3 S-mode cause register (scause)

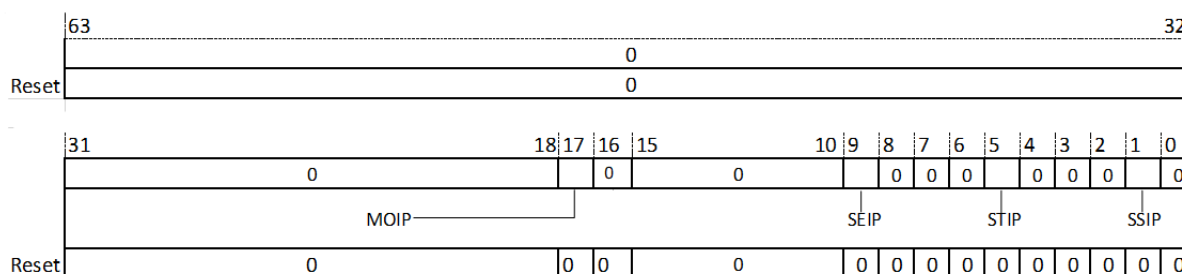
The scause register stores the vector numbers of events that trigger exceptions. The vector numbers are used to handle corresponding events in the exception service program.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.2.4 S-mode interrupt-pending register (sip)

The sip register stores information about pending interrupts. When the CPU cannot immediately respond to an interrupt, the corresponding bit in the sip register will be set.

This register is 64 bits wide and readable in S-mode. The write permission is determined by the mideleg register of the corresponding bit. Accesses in U-mode will cause an illegal instruction exception.



S-mode interrupt-pending register (sip)

18.2.3 S-mode address translation registers

Virtual memory spaces need to be accessed in S-mode. The S-mode address translation register (satp) controls MMU mode switching, hardware writeback base address, and process ID.

18.2.3.1 S-mode address translation register (satp)

The S-mode address translation register (satp) controls MMU mode switching, hardware writeback base address, and process ID.

This register is 64 bits wide and is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see ref:*virtual_mem_manage_satp*.

18.2.4 S-mode CPU control and status extension registers

18.2.4.1 S-mode extension status register (sxstatus)

The `sxstatus` register is the mapping of the `mxstatus` register. For more information, see [ref:appendix_c17_mxstatus](#).

This register is 64 bits wide and is readable in S-mode. Only the MM bit is writable. Accesses in U-mode will cause an illegal instruction exception.

18.2.4.2 S-mode hardware control register (shcr)

The `shcr` register is the mapping of the `mhcr` register. For more information, see [ref:appendix_c17_mhcr](#).

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

18.2.4.3 S-mode event overflow interrupt enable register (scounterinten)

The `scounterinten` register is the mapping of the `mcounterinten` register. For more information, see [ref:appendix_c17_mcounterinten](#).

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

When `mcounterwen.bit[n]` is 1, `scounterinten.bit[n]` determines whether to generate an interrupt when the corresponding `shpmcounter` overflows.

18.2.4.4 S-mode event overflow mark register (scounterof)

The `scounterof` register is the mapping of the `mcounterof` register. For more information, see [ref:appendix_c17_mcounterof](#).

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

When `mcounterwen.bit[n]` is 1, `scounterof.bit[n]` indicates whether the corresponding `shpmcounter` overflows.

18.2.4.5 S-mode cycle counter (scycle)

The `scycle` counter stores the cycles executed by the CPU. When the CPU is in the execution state (non-low power state), the `scycle` register increases the count upon each execution cycle.

The `mcycle` counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.2.4.6 S-mode instructions-retired counter (sinstret)

The sinstret counter stores the number of retired instructions of the CPU. The sinstret register increases the count when each instruction retires.

The sinstret counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.2.4.7 S-mode event counter (shpmcountern)

The shpmcountern counter is the mapping of the mhpmcountern counter.

For more information, see [ref:performance_test_cont](#).

18.2.5 S-mode MMU extension register

C908 extends MMU related registers to implement software writeback. Software can directly write and read the TLB.

18.2.5.1 S-mode MMU control register (smcir)

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see [ref:virtual_mem_manage_smcir](#).

18.2.5.2 S-mode MMU control register (smir)

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see [ref:virtual_mem_manage_smir](#).

18.2.5.3 S-mode MMU control register (smeh)

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see [ref:virtual_mem_manage_sme](#).

18.2.5.4 S-mode MMU control register (smel)

This register is 64 bits wide and is readable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see [ref:virtual_mem_manage_smel](#).

18.3 Appendix C-3 U-mode control registers

U-mode control registers are classified by function into floating-point registers, counter registers, and vector control registers.

18.3.1 U-mode floating-point control registers

18.3.1.1 Floating-point accrued exceptions register (fflags)

The fflags register is the mapping of accrued exceptions of the fcsr register. For more information, see [ref:appendix_c31_fcsr](#).

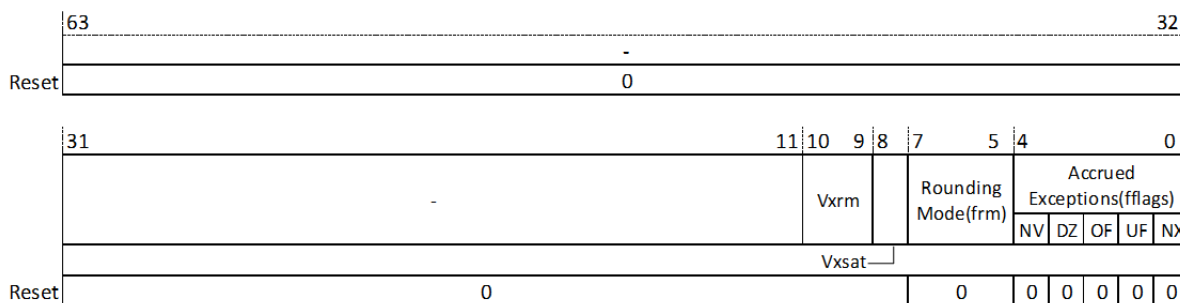
18.3.1.2 Floating-point dynamic rounding mode register (frm)

The frm register is the mapping of the rounding mode of the fcsr register. For more information, see [ref:appendix_c31_fcsr](#).

18.3.1.3 Floating-point control and status register (fcsr)

The fcsr register records floating-point accrued exceptions and the rounding mode.

This register is 64 bits wide and readable and writable in any mode.



Floating-point control and status register (fcsr)

NX: imprecise exception

Chapter 18. Appendix C Control Registers

- When NX is 0, no imprecise exception occurs.

When NX is 1, imprecise exceptions occur.

UF: underflow exception

- When UF is 0, no underflow exception occurs.
- When UF is 1, underflow exceptions occur.

OF: overflow exception

- When OF is 0, no overflow exception occurs.
- When OF is 1, overflow exceptions occur.

DZ: division by zero exception

- When DZ is 0, no division by zero exception occurs.
- When DZ is 1, division by zero exceptions occur.

NV: illegal instruction operand exception

- When NV is 0, no exception of illegal instruction operands occurs.
- When NV is 1, exceptions of illegal instruction operands occur.

RM: rounding mode

- When RM is 0, the RNE rounding mode takes effect, and values are rounded off to the nearest even number.
- When RM is 1, the RTZ rounding mode takes effect, and values are rounded off to zero.
- When RM is 2, the RDN rounding mode takes effect, and values are rounded off to negative infinity.
- When RM is 3, the RUP rounding mode takes effect, and values are rounded off to positive infinity.
- When RM is 4, the RMM rounding mode takes effect, and values are rounded off to the nearest number.

VXSAT: vector overflow flag bit

This register is the mapping of the corresponding bit.

VXRM: vector rounding mode bit

This register is the mapping of the corresponding bit.

18.3.2 U-mode counter/timer registers

18.3.2.1 User cycle register (cycle)

The cycle register stores the cycles executed by the CPU. When the CPU is in the execution state (non-low power state), the cycle register increases the count upon each execution cycle.

Chapter 18. Appendix C Control Registers

The mcycle counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.3.2.2 U-mode timer register (time)

The time register is the read-only mapping of the mtime register.

For more information, see [ref:performance_test_cont](#).

18.3.2.3 User instructions-retired counter (instret)

The instret counter stores the number of retired instructions of the CPU. The instret register increases the count when each instruction retires.

The sinstret counter is 64 bits wide and will be reset to 0.

For more information, see [ref:performance_test_cont](#).

18.3.2.4 User event counter (hpmcountern)

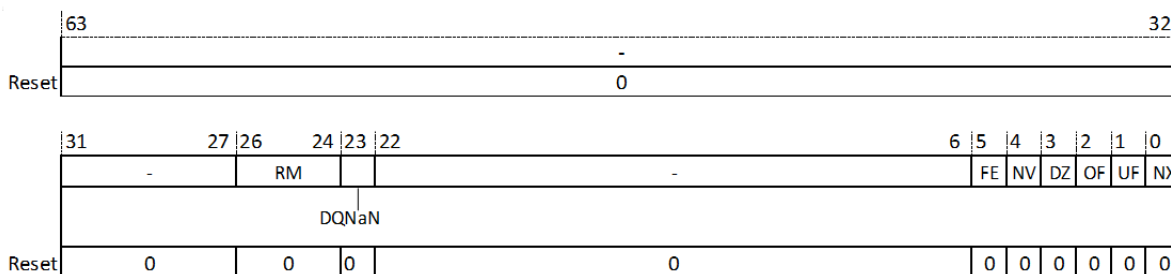
The hpmcountern counter is the mapping of the mhpmcountern counter.

For more information, see [ref:performance_test_cont](#).

18.3.3 U-mode floating-point extension control registers

18.3.3.1 U-mode floating-point extension control register (fxcr)

The fxcr register controls the floating-point extension function and floating-point exception accrue bit.



Floating-point extension control register (fxcr)

NX: imprecise exception

It is the mapping of the corresponding bit of the fcsr register.

UF: underflow exception

Chapter 18. Appendix C Control Registers

It is the mapping of the corresponding bit of the fcsr register.

OF: overflow exception

It is the mapping of the corresponding bit of the fcsr register.

DZ: division by zero exception

It is the mapping of the corresponding bit of the fcsr register.

NV: illegal instruction operand exception

It is the mapping of the corresponding bit of the fcsr register.

FE: floating-point exception accrue bit

This bit is set to 1 when any floating-point exception occurs.

DQNaN: output QNaN mode bit

When DQNaN is 0, the output QNaN value is the default value.

When DQNaN is 1, the output QNaN value is consistent with the IEEE754 standard.

RM: rounding mode

It is the mapping of the corresponding bit of the fcsr register.

18.3.4 Vector extension registers

18.3.4.1 Vector start position register (vstart)

The vstart register specifies the position of the start element when a vector instruction is executed. The vstart bit will be reset to 0 after each vector instruction is executed.

18.3.4.2 Fixed-point overflow flag bit register (vxsat)

The vxsat register specifies whether any fixed-point instruction overflows.

18.3.4.3 Fixed-point rounding mode register (vxrm)

The vxrm register specifies the rounding mode used by fixed-point instructions.

	63		2	1	0
	Base				RM
Reset	0				0

Fixed-point rounding mode register (vxrm)

RM: fixed-point rounding mode

- When RM is 0, the RNU rounding mode takes effect, and values are rounded off to a large number.
- When RM is 1, the RNE rounding mode takes effect, and values are rounded off to an even number.
- When RM is 2, the RDN rounding mode takes effect, and values are rounded off to zero.
- When RM is 3, the ROD rounding mode takes effect, and values are rounded off to an odd number.

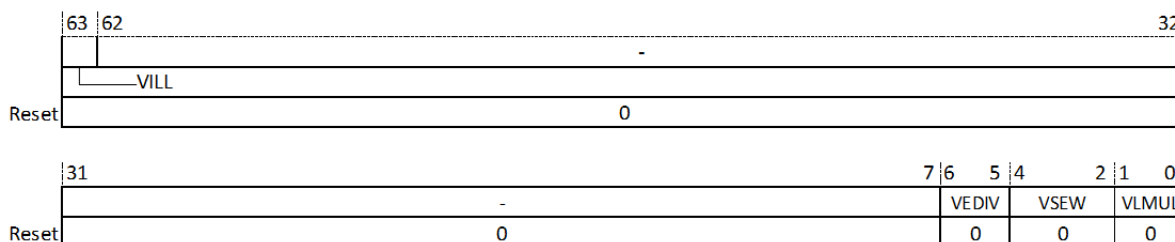
18.3.4.4 Vector length register (vl)

The vl register specifies the range of the destination register to be updated by a vector instruction. The vector instruction updates the elements with a sequence number smaller than the vl register value in the destination register, and clears those with a sequence number greater than the vl register value. Particularly, when $vstart \geq vl$ or vl is 0, all elements in the destination register are not updated.

This register is read-only in any mode, but its value can be updated by using the vsetvli, vsetvl, and fault-only-first instructions.

18.3.4.5 Vector data type register (vtype)

The vtype register specifies the data type and elements of the vector registers.



Fixed-point type register (vtype)

This register is read-only in any mode, but its value can be updated by using the vsetvli and vsetvl instructions.

VLL: valid operation flag bit

This bit is set only when the vsetvli/vsetvl instruction updates the vtype register with a value not supported by C908. Otherwise, it is 0. When this bit is set, the execution of vector instructions will cause an illegal instruction exception.

VEDIV: EDIV extension enable bit

C908 does not support EDIV extensions. Therefore, the EDIV bit is 0.

VSEW: vector element width set bit

The VSEW bit determines the standard vector element width (SEW). The vector element widths supported by C908 are shown in : numref: *Vector_element_bit_width*.

Table 18.2: Vector element widths

VSEW[2:0]			Element width
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64

When VSEW takes other values, executing C908 vector instructions causes instruction exceptions.

VLMUL: vector register group bit

Multiple vector registers can form a vector register group. Vector instructions act on all vector registers in the register group. VLMUL specifies the number of vector registers in the vector register group, as shown in : numref: *num_vector_register_group*.

Table 18.3: Number of registers in the vector register group

VLMUL[1:0]		LMUL
0	0	1
0	1	2
1	0	4
1	1	8

18.3.4.6 Vector width (unit: byte) register (vlenb)

The vlenb register specifies the vector width of the CPU, in bytes.

The vector width of C908 is 128 bits (VLEN=128). Therefore, VLENB = 128/8 = 16.

18.4 Appendix C-4 Additional Register description

Machine Trap Value Register (MTVAL)

The mtval register is an 64-bit read-write register formatted as shown in Figure Fig. 18.1, when a trap is taken into M-mode, mtval is either set to zero or written with exception-specific information to assist software in handling the trap.

This register is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

Supervisor Trap Value Register (STVAL)

The stval register is an 64-bit read-write register formatted as shown in Figure Fig. 18.2, when a trap is taken into S-mode, stval is written with exception-specific information to assist software in handling the trap.

This register is readable and writable in S-mode. Accesses in U-mode will cause an illegal instruction exception.

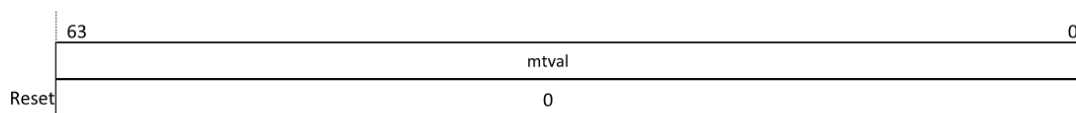


Fig. 18.1: mtval

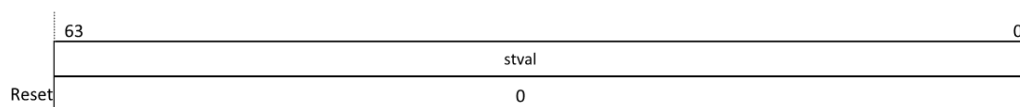


Fig. 18.2: stval

Vector extension register file

Vector Start Index(vstart)

The vstart read-write CSR specifies the index of the first element to be executed by a vector instruction.

Normally, vstart is only written by hardware on a trap on a vector instruction, with the vstart value representing the element on which the trap was taken (either a synchronous exception or an asynchronous interrupt), and at which execution should resume after a resumable trap is handled.

All vector instructions are defined to begin execution with the element number given in the vstart CSR, leaving earlier elements in the destination vector undisturbed, and to reset the vstart CSR to zero at the end of execution.

Vector Fixed-Point Saturation Flag (vxsat)

The vxsat CSR holds a single read-write bit that indicates if a fixed-point instruction has had to saturate an output value to fit into a destination format.

The vxsat bit is mirrored in the upper bits of fcsr.

Vector Fixed-Point Rounding Mode Register vxrm

The vector fixed-point rounding-mode register holds a two-bit read-write rounding-mode field. The vector fixed-point rounding-mode is given a separate CSR address to allow independent access.

RM:rounding mode

	63	2	1	0
	Reserved			RM
Reset	0			0

Fig. 18.3: VXRm

- when RM is 0, the RNU rounding mode takes effect, and values are rounded to nearest up.
- when RM is 1, the RNE rounding mode takes effect, and values are rounded to nearest even.
- when RM is 2, the RDN rounding mode takes effect, and values are rounded down.
- when RM is 3, the ROD rounding mode takes effect, and values are rounded to odd.

Vector Length Register vl

The 64-bit-wide read-only vl CSR can only be updated by the vsetvli and vsetvl instructions, and the fault-only-first vector load instruction variants.

The vl register holds an unsigned integer specifying the number of elements to be updated by a vector instruction. Elements in any destination vector register group with indices vl are unmodified during execution of a vector instruction. When vstart vl, no elements are updated in any destination vector register group.

Vector type register (vtype)

The read-only 64bit-wide vector type CSR, vtype provides the default type used to interpret the contents of the vector register file, and can only be updated by vsetvl{i} instructions. The vector type also determines the organization of elements in each vector register, and how multiple vector registers are grouped.

In the base vector extension, the vtype register has three `_elds`, `vill`, `vsew[2:0]`, and `vlmul[1:0]`.

Table. vtype register layout

Bits	Name	Description
63	vill	Illegal value if set
62:7		Reserved (write 0)
6:5	vediv[1:0]	Used by EDIV extension
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

Vill: Vector Type Illegal

The vill bit is used to encode that a previous vsetvl{i} instruction attempted to write an unsupported value to vtype. If the vill bit is set, then any attempt to execute a vector instruction (other than a vector configuration instruction) will raise an illegal instruction exception. When the vill bit is set, the other 63 bits in vtype shall be zero.

VEDIV: EDIV extension enable

The `vediv[1:0]` field is used by the EDIV extension described below. C910 does not support EDIV extension, this field is fix to 0.

Vsew: Vector standard element width

The value in `vsew` sets the dynamic standard element width (SEW). By default, a vector register is viewed as being divided into $VLEN / SEW$ standard-width elements. In the base vector extension, only SEW up to $\max(XLEN, FLEN)$ are required to be supported.

VSEW[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64

VLMUL: Vector Register Grouping

Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. Vector register groups allow double-width or larger elements to be operated on with the same vector length as standard-width elements. Vector register groups also provide greater execution efficiency for longer application vectors.

The term vector register group is used herein to refer to one or more vector registers used as a single operand to a vector instruction. The number of vector registers in a group, LMUL, is an integer power of two set by the `vlmul` field in `vtype` ($LMUL = 2^{vlmul[1:0]}$).

The derived value $VLMAX = LMUL * VLEN / SEW$ represents the maximum number of elements that can be operated on with a single vector instruction given the current SEW and LMUL settings.

vlmul[1:0]		LMUL
0	0	1
0	1	2
1	0	4
1	1	8

Vector Byte Length `vlenb`

The 64-bit-wide read-only CSR `vlenb` holds the value $VLEN/8$, i.e., the vector register length in bytes.

This chapter describes various program examples, including the MMU setting example, PMP setting example, cache setting example, multi-core startup example, synchronization primitive example, PLIC setting example, and PMU setting example.

19.1 Optimal CPU performance configuration

The optimal performance of C908 can be achieved by using the following configurations:

- MHCR = 0x11ff
- MHINT = 0x6e30c
- MCCR2 = 0xe0000009 (Note: MCCR2 contains RAM delay settings. In this example, all delays are 0. Customers need to set a proper RAM delay based on the actual situation.)
- MXSTATUS = 0x638000
- MSMPR = 0x1

```
# mhcr
li x3, 0x11ff
csrs mhcr,x3

#mhint
li x3, 0x6e30c
```

(continues on next page)

(continued from previous page)

```

csrs mhint,x3

# mxstatus
li x3, 0x638000
csrs mxstatus,x3

# msmpr
csrsi msmpr,0x1

# mCCR2
li x3, 0xe0000009
csrs mCCR2,x3

```

19.2 MMU setting example

```

/*****

* Function: An example of setting C908MP MMU.
* Memory space: Virtual address <-> physical address.
*
* Pagesize 4K: vpn: {vpn2,vpn1,vpn0} <-> ppn: {ppn2,ppn1,ppn0}
* Pagesize 2M: vpn: {vpn2,vpn1} <-> ppn:{ppn2,ppn1}
* Pagesize 1G: vpn: {vnp2} <-> ppn: {ppn2}
*
*****/

/*C908 will invalidate all MMU TLB entries automatically when reset*/
/*You can use sfence.vma to invalid all MMU TLB entries if necessary*/
sfence.vma x0, x0

/* Pagesize 4K: vpn: {vpn2, vpn1, vpn0} <-> ppn: {ppn2, ppn1, ppn0}*/
/* First-level page addr base: PPN (defined in satp)*/
/* Second-level page addr base: BASE2 (self define)*/
/* Third-level page addr base: BASE3 (self define)*/
/* 1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff

```

(continues on next page)

(continued from previous page)

```

and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{ 44' b BASE2, 10' b1}
→ */
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44' b BASE2, 10' b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3' b0}, second-level page pte:{ 44' b BASE3, 10'
b1} */
/*Get second-level page addr*/
/* VPN1*/
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*
li x6, {44' b BASE3, 10' b1}
sd x6, 0(x5)

/*4. Config third-level page*/
/*Third-level page addr: {BASE3, vpn0, 3' b0}, third-level page pte:{
theadflag, ppn2, ppn1, ppn0, 9' b flags,1' b1} */
/*Get second-level page addr*/
/* VPN0*/
li x4, VPN
li x5, 0x1ff

```

(continues on next page)

(continued from previous page)

```

and x4, x4, x5
srli x4, x4, 3
/*BASE3*/
li x5, BASE3
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, {theadflag, ppn2, ppn1, ppn0, 9' b flags, 1' b1}
sd x6, 0(x5)

/* Pagesize 2M: vpn: {vpn2, vpn1} <-> ppn: {ppn2, ppn1}*/
/*First-level page addr base: PPN (defined in satp)*/
/*Second-level page addr base: BASE2 (self define)*/

/*1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff
and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{ 44' b
BASE2, 10' b1}*/
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, {44' b BASE2, 10' b1}
sd x6, 0(x5)

/*3. Config second-level page*/
/*Second-level page addr: {BASE2, vpn1, 3' b0}, second-level page pte:{
theadflag, ppn2, ppn1, 9' b0, 9' b flags, 1' b1} */
/*Get second-level page addr*/

```

(continues on next page)

(continued from previous page)

```

/*VPN1*/
li x4, VPN
li x5, 0x3fe00
and x4, x4, x5
srli x4, x4, 9
/*BASE2*/
li x5, BASE2
srli x5, x5, 12
and x5, x5, x4
/*Store pte at second-level page addr*/
li x6, { theadflag, ppn2, ppn1, 9' b0, 9' b flags,1' b1}
sd x6, 0(x5)

/* Pagesize 1G: vpn: {vpn2} <-> ppn: {ppn2}*/
/*First-level page addr base: PPN (defined in satp)*/
/*1. Get first-level page addr base: PPN and vpn*/
/* Get PPN*/
csrr x3, satp
li x4, 0xffffffff
and x3, x3, x4

/*2. Config first-level page*/
/*First-level page addr: {PPN, vpn2, 3' b0}, first-level page pte:{
theadflag, ppn2, 9' b0, 9' b0, 9' b flags,1' b1}*/
/*Get first-level page addr*/
slli x3, x3, 12
/*Get vpn2*/
li x4, VPN
li x5, 0x7fc0000
and x4, x4, x5
srli x4, x4, 15
and x5, x3, x4
/*Store pte at first-level page addr*/
li x6, { theadflag, ppn2, 9' b0, 9' b0, 9' b flags,1' b1}
sd x6, 0(x5)

```

19.3 PMP setting example

```


/*****
* Function: An example of setting C908MP PMP.
* 0x0 ~ 0xf0000000, TOR mode, RWX
* 0xf0000000 ~ 0xf8000000, NAPOT mode, RW
* 0xffff73000 ~ 0xffff74000, NAPOT mode, RW
* 0xfffc0000 ~ 0xfffc2000, NAPOT mode, RW
*Different execution permissions are configured for the preceding four regions. PMP must
↳be configured to prevent the CPU from executing instructions to unsupported address
↳regions in different modes, especially in M-mode where the CPU has full execution
↳permissions by default. Specifically, after you configure address regions that require
↳execution permissions, no permission should be configured for the rest address regions.
↳ For more information, see the following example.
*****/

# pmpaddr0,0x0 0xf0000000, TOR mode, read and write permissions
li x3, (0xf0000000 >> 2)
csrw pmpaddr0, x3
# pmpaddr1,0xf0000000 0xf8000000, NAPOT mode, read and write permissions
li x3, ( 0xf0000000 >> 2 | (0x8000000-1) >> 3))
csrw pmpaddr1, x3
# pmpaddr2,0xffff73000 0xffff74000, NAPOT mode, read and write permissions
li x3, ( 0xffff73000 >> 2 | (0x1000-1) >> 3))
csrw pmpaddr2, x3
# pmpaddr3,0xfffc0000 0xfffc2000, NAPOT mode, read and write permissions
li x3, ( 0xfffc0000 >> 2 | (0x2000-1) >> 3))
csrw pmpaddr3, x3
# pmpaddr4,0xf0000000 0x100000000, NAPOT mode, no permissions
li x3, ( 0xf0000000 >> 2 | (0x10000000-1) >> 3))
csrw pmpaddr4, x3
# pmpaddr5,0x100000000 0xffffffff, TOR mode, no permissions
li x3, (0xffffffff >> 2)
csrw pmpaddr5, x3
# PMPCFG0 configures the execution permission, mode, and lock bit of entries.
When lock is 1, it is valid only in M-mode.
li x3,0x88989b9b9b8f
csrw pmpcfg0, x3
# pmpaddr5,0x100000000 0xffffffff: In TOR mode, when 0x100000000 <= addr <
0xffffffff, pmpaddr5 will be hit. However, pmpaddr5 cannot be hit in the address
↳range 0xffffffff000 ~

```

(continues on next page)

(continued from previous page)

0xffffffff (the minimum PMP granularity is 4 KB in C908). An NAPOT entry must be  configured to mask the last 4 KB space of a 1 TB space.

19.4 Cache examples

19.4.1 Cache enabling example

```

/*C908 will invalidate all I-cache automatically when reset*/
/*You can invalidate I-cache by yourself if necessary*/
/*Invalidate I-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x11
csrs mcor, x3
// You can also use icache instrucitons to replace the invalidate sequence
// if theadisae is enabled.
//icache.iall
//sync.is

/*Enable I-cache*/
li x3, 0x1
csrs mhcr, x3

/*C908 will invalidate all D-cache automatically when reset*/
/*You can invalidate D-cache by yourself if necessary*/
/*Invalidate D-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x12
csrs mcor, x3

// You can also use dcache instrucitons to replace the invalidate sequence
// if theadisae is enabled.
// dcache.iall
// sync.is

/*Enable D-cache*/
li x3, 0x2

```

(continues on next page)

(continued from previous page)

```

csrs mhcr, x3

/*C908 will invalidate all L2 cache automatically when reset*/
/*You can invalidate L2 by yourself if necessary*/
/*Invalidate L2-cache if theadisae is enabled*/
l2cache.iall
sync.is

/*Enable L2-cache*/
li x3, 8
csrs mCCR2, x3

```

19.4.2 Example of synchronization between the instruction and data caches

CPU0

```

sd x3,0(x4) // a new instruction defined in x3
             // is stored to program memory address defined in x4.
dcache.cval1 r0 // clean the new instruction to the shared L2 cache.
sync.s        // ensure completion of clean operation.
             // the dcache clean is not necessarily if INSD is not enabled.
icache.iva r0 // invalid icache according to shareable configuration.
sync.s/fence.i // ensure completion in all CPUs.
sd x5,0(x6)   // set flag to signal operation completion.
sync.is
jr x4 // jmp to new code

```

CPU1 CPU3

```

WAIT_FINISH:
ld x7,0(x6)
bne x7,x5, WAIT_FINISH // wait CPU0 modification finish.
sync.is
jr x4                  // jmp to new code

```

19.4.3 Example of synchronization between the TLB and the data cache

CPU0

```
sd x4,0(x3) // update a new translation table entry
sync.is/fence.i // ensure completion of update operation.
sfence.vma x5,x0 // invalid the TLB by va
sync.is/fence.i // ensure completion of TLB invalidation and
                // synchronises context
```

19.5 Multi-core startup example

Note: The content of this section is outdated. V1.4.x: mrrmr has been deleted and mrvbr is private to cores, with the MRO privilege.

CPU0

```
.....          // CPU1~CPU3 are in reset mode and
                // CPU0 executes system initialize operation.
li x3, RVBA
csrw mrvbr, x3 // Set reset vector base address
li x3, 0x2
csrs mrrmr, x3 // Release CPU1' s reset signal.
li x3, 0x4
csrs mrrmr, x3 // Release CPU2' s reset signal.
li x3, 0x8
csrs mrrmr, x3 // Release CPU3' s reset signal.
                // CPU1~CPU3 start to execute reset exception routine.
```

19.6 Synchronization primitive examples

CPU0

```
li x1, 0x1
li x6, 0x0

ACQUIRE_LOCK:      // (x3) is the lock address. 0: Free; 1: Busy.
lr x4, 0(x3)        // Read lock
bnez x4, ACQUIRE_LOCK // Try again if the lock is in use
sc x5, x1, 0(x3)    // Attempt to store new value
bne x6, x5, ACQUIRE_LOCK // Try again if fail
sync.s
```

(continues on next page)

(continued from previous page)

```
...                // Critical section code
```

CPU1

```
sync.s/fence.i      // Ensure all operations are observed before clearing the lock.
sd x0, 0(x3)        // Clear the lock.
```

19.7 PLIC setting example

```
//Init id 1 machine mode int for hart 0
/*1. Set hart threshold if needed*/
li x3, (plic_base_addr + 0x200000) // h0 mthreshold addr
li x4, 0xa //threshold value
sw x4,0x0(x3) // set hart0 threshold as 0xa

/*2. Set priority for int id 1*/
li x3, (plic_base_addr + 0x0) // int id 1 prio addr
li x4, 0x1f // prio value
sw x4,0x4(x3) // init id1 priority as 0x1f

/*3. Enable m-mode int id1 to hart*/
li x3, (plic_base_addr + 0x2000) // h0 mie0 addr
li x4, 0x2
sw x4,0x0(x3) // enable int id1 to hart0

/*4. Set ip or wait external int*/
/*following code set ip*/
li x3, (plic_base_addr + 0x1000) // h0 mthreshold addr
li x4, 0x2 // id 1 pending
sw x4, 0x0(x3) // set int id1 pending

/*5. Core enters interrupt handler, read PLIC_CLAIM and get ID*/

/*6. Core takes interrupt*/

/*7. Core needs to clear external interrupt source if LEVEL(not PULSE)
configured, then core writes ID to PLIC_CLAIM and exits interrupt*/
```

19.8 PMU setting example

```
/*1. Inhibit counters counting*/
li x3, 0xffffffff
csrw mcountinhibit, x3

/*2. C908 will initial all pmu counters when reset*/
/*you can initial pmu counters manually if necessarily*/
csrw mcycle, x0
csrw minstret, x0
csrw mhpmcounter3, x0
.....
csrw mhpmcounter31, x0

/*3. Configure mhpmevent*/
li x3, 0x1
csrw mhpmevent3, x3 // mhpmcounter3 count event: L1 ICache Access Counter
li x3, 0x2
csrw mhpmevent4, x3 // mhpmcounter4 count event: L1 ICache Miss Counter
.....
li x3, 0x13
csrw mhpmevent21, x3 // mhpmcounter21 count event: L2 Cache write miss Counter

/*4. Configure mcounteren and scounteren*/
li x3, 0xffffffff
csrw mcounteren, x3 // enable super mode to read hpmcounter
li x3, 0xffffffff
csrw scounteren, x3 // enable user mode to read hpmcounter

/*5. Enable counters to count when you want*/
csrw mcountinhibit, x0
```

Appendix D XuanTie C900 Multi-core Synchronization Related Instructions and Program Implementations

20.1 Overview

The multi-core synchronization of XuanTie C900 is based on the RISC-V architecture, and complies with the definitions about instruction synchronization (fence.i), TLB maintenance (sfence.vma), and atomic instruction set extension in RISC-V privileged spec.

To improve maintenance efficiency in the scenario of multi-core and non-uniform bus, XuanTie C900 further enhances instruction synchronization, TLB maintenance, and DMA synchronization to meet different market requirements.

20.2 RISC-V instructions

20.2.1 fence instruction

The base RISC-V instruction set includes the fence instruction, which explicitly ensures the order of program instructions.

FENCE IORW, IORW

The fence instruction distinguishes the IO address space and memory address space. IO represents input/output, and RW represents read/write.

FENCE RW ensures that preorder read/write instructions are not executed later than the local fence instruction. FENCE RW ensures that postorder read/write instructions are not executed before the local fence instruction.

The following instruction can be independently formed: FENCE R, RW / FENCE R, R / FENCE R / FENCE RW / FENCE RW, W ...IO equals RW. Therefore, the following instruction can be formed: FENCE I, IO / FENCE I, I / FENCE I / FENCE IO / FENCE IO, O ...IO and RW can even be mixed in an instruction, for example, FENCE RI, IORW / FENCE IORW, IORW ...

By defining the eight preorder and postorder R, W, I, and O bits, FENCE enables programmers to clearly understand the requirements of load/store on the sequence of memory or IO operations.

20.2.2 fence.i instruction

This instruction clears I-Cache to ensure that the data access results before this instruction can be accessed by the fetch operations after the instruction.

20.2.3 sfence.vma instruction

sfence.vma rs1,rs2 is used for invalidation and synchronization in virtual memories. rs1 indicates the virtual address and rs2 indicates the ASID.

- When rs1 and rs2 are both x0, all TLB entries are invalidated.
- When rs1! and rs2 are both x0, all TLB entries that hit the virtual address specified by rs1 are invalidated.
- When rs1 and rs2! are both x0, all TLB entries that hit the process ID specified by rs2 are invalidated.
- When rs1! and rs2! are both x0, all TLB entries that hit the virtual address specified by rs1 and the process ID specified by rs2 are invalidated.

20.2.4 AMO instruction

An atomic operation indicates the exclusive consecutive read, modify, and writeback operations on a shared memory address by multiple threads.

In a single-core system, exclusive operations are applicable if not being interrupted by interrupts/exceptions. In a single-core system, the memory model is simple and conforms to the instincts of programmers, namely, the return value of a read operation is always from the previous write operation of the same address. However the LSU of a single-core CPU is designed, programmers' expectations for memory access can be met.

In a multi-core system, the memory model is complex. What is the last written value? Are the results of a read sequence sequential? What is the next write operation? These questions that do not need to be concerned in single-core systems may become extremely complex in multi-core systems.

Currently, multiple memory ordering models are defined by different hardware implementations:

- Sequential consistency
- Processor consistency
- Weak consistency
- Release consistency (RISC-V)

As a result, the definition of atomic operation varies according to the architecture.

In the RISC-V architecture, the AMO instruction covers abundant ALU operations (such as add, AND/OR/XOR, and MIN/MAX) and meet the requirements of Linux for atomic operation primitives. However, the data types supported by the AMO instruction are limited. RV32 supports only word instructions, and RV64 supports word and doubleword instructions but not halfword instructions. However, the halfword XCHG operation is strongly required in qspinlock. As a result, RISC-V cannot support qspinlock.

20.2.5 Load-Reserved/Store-Conditional instruction

The Load-Reserved/Store-Conditional (LR/SC) instruction is widely applied in the ARM architecture. The compare-and-swap (CAS) instruction in the x86 architecture is equivalent to the LR/SC instruction.

Introduction to LR/SC instruction in RISC-V:

LR is similar to load. It obtains data from a specified memory and monitors subsequent write operations of this address. After performing ALU calculation for the obtained data, the CPU uses the SC instruction to write a new value into the memory address of the previous LR operation. If no CPU write operation is performed on this memory address, the SC instruction writes the new value into the memory and sets rd to 0 (indicating success), like a common store instruction. Otherwise, the SC instruction does not write the new value into the memory, and sets rd to a non-zero value (indicating failure).

RISC-V lists the following advantages of LR/SC against CAS:

1. CAS suffers from the ABA problem, namely, CAS cares for only the results rather than the process. If the previous load value is consistent with the value obtained by CAS, a new value can be successfully written, even if someone has written this address. This does not meet the expectations of programmers and damages atomicity. However, LR/SC monitors all accesses to the address, and the SC instruction is damaged even the same value is written.
2. CAS requires three source registers and one destination register (result), which would complicate hardware implementations.
3. To avoid the ABA problem, other systems provide a double-wide CAS (DW-CAS), which requires five source registers and two destination registers, further complicating implementations.
4. LR/SC provides a more efficient implementation of many primitives as it only requires one load as opposed to two with CAS (load + CAS).

The preceding are reasons provided by RISC-V for choosing LR/SC over CAS. The fact is that software APIs are not perfectly compatible with LR/SC. For example, Linux uses only the `cmpxchg` primitive for CAS, without the `load_reserved/store_conditional` primitive.

LR/SC is used to implement `cmpxchg`.

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0 otherwise
cas:
lr.w t0, (a0) # Load original value.
bne t0, a1, fail # Doesn' t match, so fail.
sc.w t0, a2, (a0) # Try to update.
bnez t0, cas # Retry if store-conditional failed.
li a0, 0 # Set return to success.
jr ra # Return.
fail:
li a0, 1 # Set return to failure.
jr ra # Return.
```

Based on the loop structure of `cmpxchg`, double-loop implementation is formed.

```
c = v->counter;
while ((old = cmpxchg(&v->counter, c, c c_op i)) != c)
    c = old;
```

If this is the case, the first, third, and fourth advantages no longer exist. Therefore, abandoning CAS produces negative impact on software compatibility. This is why ARM64 supports CAS.

The livelock problem of LR/SC is more complex. More problems may exist for Non Uniform Memory Access (NUMA) systems with more than 128 harts. (This is not described in this document.)

Compared with ARM64, the RISC-V does not have the `load_reserved/wait for event (LR/WFE)` instructions. As a result, the `load_cond` primitive instruction cannot be implemented. When a single core has multiple threads, the `load_cond` primitive instruction is required to stop occupying the pipeline.

20.3 T-Head enhancement instruction

20.3.1 sync.is

This instruction ensures that all preorder instructions retire earlier than this instruction and all postorder instructions retire later than this instruction. When this instruction retires, the pipeline is cleared and the

request is broadcast to other cores. This instruction can be used as the sync.s instruction (only for flush).

20.3.2 dcache.cipa rs1

This instruction writes the D-Cache/L2 Cache entry that hits the physical address specified by rs1 back to the lower-level storage and invalidates this entry. This instruction can also be used as the dcache.cpa (only for flush) or dcache.ipa (only for invalidation) instruction.

20.3.3 icache.iva rs1

This instruction invalidates the I-Cache entries that hit the virtual address specified by rs1.

20.4 Software example

The following lists the software examples of MMU and cache maintenance in the Linux RISC-V architecture.

20.4.1 TLB maintenance

20.4.1.1 TLB flush

```
static inline void local_flush_tlb(unsigned long asid)
{
    __asm__ __volatile__ ("sfence.vma" : : : "memory");
}
```

20.4.1.2 Flush process related TLB entries based on ASID

```
static inline void local_flush_tlb(unsigned long asid)
{
    __asm__ __volatile__ ("sfence.vma , %0" : : "r" (asid) : "memory");
}
```

20.4.1.3 Flush TLB entries based on VA

```
static inline void local_flush_tlb_range(unsigned long start, unsigned long size)
{
    unsigned long page_add = PAGE_DOWN(start);
```

(continues on next page)

(continued from previous page)

```
unsigned long page_end = PAGE_UP(start + size);

while(page_add < page_end) {
    __asm__ __volatile__ ("sfence.vma %0, zero"
        :
        : "r" (page_add), "r" (asid)
        : "memory");
    page_add += PAGE_SIZE;
}
}
```

20.4.1.4 Flush TLB entries based on VA and ASID

```
static inline void local_flush_tlb_range_asid(unsigned long start, unsigned long size, unsigned long asid)
{
    unsigned long page_add = PAGE_DOWN(start);
    unsigned long page_end = PAGE_UP(start + size);

    while(page_add < page_end) {
        __asm__ __volatile__ ("sfence.vma %0, %1"
            :
            : "r" (page_add), "r" (asid)
            : "memory");
        page_add += PAGE_SIZE;
    }
}
```

20.4.2 Instruction space synchronization

20.4.2.1 Intra-core global instruction space synchronization

```
static inline void local_flush_icache_all(void)
{
    asm volatile ("fence.i" ::: "memory");
}
```


20.4.2.2 Inter-core global instruction space synchronization

```
static void ipi_remote_fence_i(void *info)
{
    asm volatile ("fence.i" ::: "memory");
}

void flush_icache_all(void)
{
    on_each_cpu(ipi_remote_fence_i, NULL, 1);
}
```

20.4.2.3 T-Head inter-core precise instruction space synchronization

```
static inline void flush_icache_range(unsigned long va_start, unsigned long size)
{
    register unsigned long i asm("a0") = va_start & ~(L1_CACHE_BYTES - 1);

    for (; i < (start + size); i += L1_CACHE_BYTES)
        __asm__ __volatile__ ("icache.iva" : : "r" (asid) : "memory");

    __asm__ __volatile__ ("sync.is");
}
```

20.4.3 DMA synchronization

20.4.3.1 T-Head inter-core precise DMA synchronization, including three directions

```
void dma_sync_from_cpu_to_dev(unsigned long pa_start, unsigned long size)
{
    register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

    for (; i < (start + size); i += L1_CACHE_BYTES)
        __asm__ __volatile__ ("dcache.cpa" : : "r" (asid) : "memory");

    __asm__ __volatile__ ("sync.s");
}

void dma_sync_from_dev_to_cpu(unsigned long pa_start, unsigned long size)
```

(continues on next page)

(continued from previous page)

```

{
    register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

    for (; i < (start + size); i += L1_CACHE_BYTES)
        __asm__ __volatile__ ("dcache.ipa" : : "r" (asid) : "memory");

    __asm__ __volatile__ ("sync.s");
}

void dma_sync_all(unsigned long pa_start, unsigned long size)
{
    register unsigned long i asm("a0") = pa_start & ~(L1_CACHE_BYTES - 1);

    for (; i < (start + size); i += L1_CACHE_BYTES)
        __asm__ __volatile__ ("dcache.cipa" : : "r" (asid) : "memory");

    __asm__ __volatile__ ("sync.s");
}

```

20.4.4 AMO implementations for reference

The following are riscv arch_atomic and cmpxchg implementations in Linux.

```

/*
 * First, the atomic ops that have no ordering constraints and therefor don't
 * have the AQ or RL bits set.  These don't return anything, so there's only
 * one version to worry about.
 */
#define ATOMIC_OP(op, asm_op, I, asm_type, c_type, prefix)          \
static __always_inline                                             \
void atomic##prefix##_##op(c_type i, atomic##prefix##_t *v)      \
{                                                                    \
    __asm__ __volatile__ (                                          \
        "        amo" #asm_op "." #asm_type " zero, %1, %0"      \
        : "+A" (v->counter)                                       \
        : "r" (I)                                                 \
        : "memory");                                              \
}

```

(continues on next page)

(continued from previous page)

```

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS(op, asm_op, I) \
    ATOMIC_OP (op, asm_op, I, w, int,  )
#else
#define ATOMIC_OPS(op, asm_op, I) \
    ATOMIC_OP (op, asm_op, I, w, int,  ) \
    ATOMIC_OP (op, asm_op, I, d, s64, 64)
#endif

ATOMIC_OPS(add, add,  i)
ATOMIC_OPS(sub, add, -i)
ATOMIC_OPS(and, and,  i)
ATOMIC_OPS( or, or,   i)
ATOMIC_OPS(xor, xor,  i)

#undef ATOMIC_OP
#undef ATOMIC_OPS

/*
 * Atomic ops that have ordered, relaxed, acquire, and release variants.
 * There's two flavors of these: the arithmetic ops have both fetch and return
 * versions, while the logical ops only have fetch versions.
 */
#define ATOMIC_FETCH_OP(op, asm_op, I, asm_type, c_type, prefix) \
static __always_inline \
c_type atomic##prefix##_fetch_##op##_relaxed(c_type i, \
                                               atomic##prefix##_t *v) \
{ \
    register c_type ret; \
    __asm__ __volatile__ ( \
        "        amo" #asm_op "." #asm_type " %1, %2, %0" \
        : "+A" (v->counter), "=r" (ret) \
        : "r" (I) \
        : "memory"); \
    return ret; \
}
static __always_inline \
c_type atomic##prefix##_fetch_##op(c_type i, atomic##prefix##_t *v) \
{ \
    register c_type ret; \

```

(continues on next page)

(continued from previous page)

```

__asm__ __volatile__ (
    "      amo" #asm_op "." #asm_type ".aqrl %1, %2, %0" \
    : "+A" (v->counter), "=r" (ret) \
    : "r" (I) \
    : "memory"); \
return ret; \
}

#define ATOMIC_OP_RETURN(op, asm_op, c_op, I, asm_type, c_type, prefix) \
static __always_inline \
c_type atomic##prefix##_##op##_return_relaxed(c_type i, \
                                               atomic##prefix##_t *v) \
{ \
    return atomic##prefix##_fetch_##op##_relaxed(i, v) c_op I; \
}

static __always_inline \
c_type atomic##prefix##_##op##_return(c_type i, atomic##prefix##_t *v) \
{ \
    return atomic##prefix##_fetch_##op(i, v) c_op I; \
}

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS(op, asm_op, c_op, I) \
    ATOMIC_FETCH_OP( op, asm_op,      I, w, int,  ) \
    ATOMIC_OP_RETURN(op, asm_op, c_op, I, w, int,  )
#else
#define ATOMIC_OPS(op, asm_op, c_op, I) \
    ATOMIC_FETCH_OP( op, asm_op,      I, w, int,  ) \
    ATOMIC_OP_RETURN(op, asm_op, c_op, I, w, int,  ) \
    ATOMIC_FETCH_OP( op, asm_op,      I, d, s64, 64) \
    ATOMIC_OP_RETURN(op, asm_op, c_op, I, d, s64, 64)
#endif

ATOMIC_OPS(add, add, +,  i)
ATOMIC_OPS(sub, add, +, -i)

#define atomic_add_return_relaxed      atomic_add_return_relaxed
#define atomic_sub_return_relaxed      atomic_sub_return_relaxed
#define atomic_add_return              atomic_add_return
#define atomic_sub_return              atomic_sub_return

```

(continues on next page)

(continued from previous page)

```

#define atomic_fetch_add_relaxed      atomic_fetch_add_relaxed
#define atomic_fetch_sub_relaxed      atomic_fetch_sub_relaxed
#define atomic_fetch_add              atomic_fetch_add
#define atomic_fetch_sub              atomic_fetch_sub

#ifndef CONFIG_GENERIC_ATOMIC64
#define atomic64_add_return_relaxed   atomic64_add_return_relaxed
#define atomic64_sub_return_relaxed   atomic64_sub_return_relaxed
#define atomic64_add_return           atomic64_add_return
#define atomic64_sub_return           atomic64_sub_return

#define atomic64_fetch_add_relaxed    atomic64_fetch_add_relaxed
#define atomic64_fetch_sub_relaxed    atomic64_fetch_sub_relaxed
#define atomic64_fetch_add            atomic64_fetch_add
#define atomic64_fetch_sub            atomic64_fetch_sub
#endif

#undef ATOMIC_OPS

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS(op, asm_op, I)      \
    ATOMIC_FETCH_OP(op, asm_op, I, w, int,  )
#else
#define ATOMIC_OPS(op, asm_op, I)      \
    ATOMIC_FETCH_OP(op, asm_op, I, w, int,  ) \
    ATOMIC_FETCH_OP(op, asm_op, I, d, s64, 64)
#endif

ATOMIC_OPS(and, and, i)
ATOMIC_OPS( or, or, i)
ATOMIC_OPS(xor, xor, i)

#define atomic_fetch_and_relaxed      atomic_fetch_and_relaxed
#define atomic_fetch_or_relaxed       atomic_fetch_or_relaxed
#define atomic_fetch_xor_relaxed      atomic_fetch_xor_relaxed
#define atomic_fetch_and              atomic_fetch_and
#define atomic_fetch_or                atomic_fetch_or
#define atomic_fetch_xor              atomic_fetch_xor

```

(continues on next page)

(continued from previous page)

```
#ifndef CONFIG_GENERIC_ATOMIC64
#define atomic64_fetch_and_relaxed    atomic64_fetch_and_relaxed
#define atomic64_fetch_or_relaxed    atomic64_fetch_or_relaxed
#define atomic64_fetch_xor_relaxed   atomic64_fetch_xor_relaxed
#define atomic64_fetch_and           atomic64_fetch_and
#define atomic64_fetch_or             atomic64_fetch_or
#define atomic64_fetch_xor           atomic64_fetch_xor
#endif

#undef ATOMIC_OPS

#undef ATOMIC_FETCH_OP
#undef ATOMIC_OP_RETURN

/* This is required to provide a full barrier on success. */
static __always_inline int atomic_fetch_add_unless(atomic_t *v, int a, int u)
{
    int prev, rc;

    __asm__ __volatile__ (
        "0:    lr.w    %[p],  %[c]\n"
        "      beq     %[p],  %[u], 1f\n"
        "      add     %[rc],  %[p], %[a]\n"
        "      sc.w.rl %[rc],  %[rc], %[c]\n"
        "      bnez   %[rc],  0b\n"
        "      fence   rw, rw\n"
        "1:\n"
        : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"A" (v->counter)
        : [a]"r" (a), [u]"r" (u)
        : "memory");
    return prev;
}

#define atomic_fetch_add_unless atomic_fetch_add_unless

#endif CONFIG_GENERIC_ATOMIC64
static __always_inline s64 atomic64_fetch_add_unless(atomic64_t *v, s64 a, s64 u)
{
    s64 prev;
    long rc;
```

(continues on next page)

(continued from previous page)

```
__asm__ __volatile__ (  
    "0:    lr.d    %[p], %[c]\n"  
    "     beq     %[p], %[u], 1f\n"  
    "     add     %[rc], %[p], %[a]\n"  
    "     sc.d.rl  %[rc], %[rc], %[c]\n"  
    "     bnez    %[rc], 0b\n"  
    "     fence   rw, rw\n"  
    "1:\n"  
    : [p]"=&r" (prev), [rc]"=&r" (rc), [c]" +A" (v->counter)  
    : [a]"r" (a), [u]"r" (u)  
    : "memory");  
    return prev;  
}  
#define atomic64_fetch_add_unless atomic64_fetch_add_unless  
#endif  
  
/*  
 * atomic_{cmp,}xchg is required to have exactly the same ordering semantics as  
 * {cmp,}xchg and the operations that return, so they need a full barrier.  
 */  
#define ATOMIC_OP(c_t, prefix, size) \br/>static __always_inline \br/>c_t atomic##prefix##_xchg_relaxed(atomic##prefix##_t *v, c_t n) \br/>{ \br/>    return __xchg_relaxed(&(v->counter), n, size); \br/>}  
static __always_inline \br/>c_t atomic##prefix##_xchg_acquire(atomic##prefix##_t *v, c_t n) \br/>{ \br/>    return __xchg_acquire(&(v->counter), n, size); \br/>}  
static __always_inline \br/>c_t atomic##prefix##_xchg_release(atomic##prefix##_t *v, c_t n) \br/>{ \br/>    return __xchg_release(&(v->counter), n, size); \br/>}  
static __always_inline \br/>c_t atomic##prefix##_xchg(atomic##prefix##_t *v, c_t n) \br/>{ \br/>    return __xchg(&(v->counter), n, size); \br/>}
```

(continues on next page)

(continued from previous page)

```

}
static __always_inline
c_t atomic##prefix##_cmpxchg_relaxed(atomic##prefix##_t *v,
                                     c_t o, c_t n)
{
    return __cmpxchg_relaxed(&(v->counter), o, n, size);
}
static __always_inline
c_t atomic##prefix##_cmpxchg_acquire(atomic##prefix##_t *v,
                                     c_t o, c_t n)
{
    return __cmpxchg_acquire(&(v->counter), o, n, size);
}
static __always_inline
c_t atomic##prefix##_cmpxchg_release(atomic##prefix##_t *v,
                                     c_t o, c_t n)
{
    return __cmpxchg_release(&(v->counter), o, n, size);
}
static __always_inline
c_t atomic##prefix##_cmpxchg(atomic##prefix##_t *v, c_t o, c_t n)
{
    return __cmpxchg(&(v->counter), o, n, size);
}

#ifdef CONFIG_GENERIC_ATOMIC64
#define ATOMIC_OPS()
    ATOMIC_OP(int, , 4)
#else
#define ATOMIC_OPS()
    ATOMIC_OP(int, , 4)
    ATOMIC_OP(s64, 64, 8)
#endif

ATOMIC_OPS()

#define atomic_xchg_relaxed atomic_xchg_relaxed
#define atomic_xchg_acquire atomic_xchg_acquire
#define atomic_xchg_release atomic_xchg_release
#define atomic_xchg atomic_xchg

```

(continues on next page)

(continued from previous page)

```
#define atomic_cmpxchg_relaxed atomic_cmpxchg_relaxed
#define atomic_cmpxchg_acquire atomic_cmpxchg_acquire
#define atomic_cmpxchg_release atomic_cmpxchg_release
#define atomic_cmpxchg atomic_cmpxchg

#undef ATOMIC_OPS
#undef ATOMIC_OP

static __always_inline int atomic_sub_if_positive(atomic_t *v, int offset)
{
    int prev, rc;

    __asm__ __volatile__ (
        "0:    lr.w    %[p],  %[c]\n"
        "      sub     %[rc],  %[p],  %[o]\n"
        "      bltz   %[rc],  1f\n"
        "      sc.w.rl  %[rc],  %[rc],  %[c]\n"
        "      bnez   %[rc],  0b\n"
        "      fence   rw, rw\n"
        "1:\n"
        : [p]"=&r" (prev), [rc]"=&r" (rc), [c]"=A" (v->counter)
        : [o]"r" (offset)
        : "memory");
    return prev - offset;
}

#define atomic_dec_if_positive(v)      atomic_sub_if_positive(v, 1)

#ifndef CONFIG_GENERIC_ATOMIC64
static __always_inline s64 atomic64_sub_if_positive(atomic64_t *v, s64 offset)
{
    s64 prev;
    long rc;

    __asm__ __volatile__ (
        "0:    lr.d    %[p],  %[c]\n"
        "      sub     %[rc],  %[p],  %[o]\n"
        "      bltz   %[rc],  1f\n"
        "      sc.d.rl  %[rc],  %[rc],  %[c]\n"
        "      bnez   %[rc],  0b\n"

```

(continues on next page)

(continued from previous page)

```

        "        fence    rw, rw\n"
        "1:\n"
        : [p]="&r" (prev), [rc]="&r" (rc), [c]+"A" (v->counter)
        : [o]"r" (offset)
        : "memory");
    return prev - offset;
}

#define __xchg_relaxed(ptr, new, size) \
({ \
    __typeof__(ptr) __ptr = (ptr); \
    __typeof__(new) __new = (new); \
    __typeof__(*(ptr)) __ret; \
    switch (size) { \
    case 4: \
        __asm__ __volatile__ ( \
            "        amoswap.w %0, %2, %1\n" \
            : "=r" (__ret), "+A" (*__ptr) \
            : "r" (__new) \
            : "memory"); \
        break; \
    case 8: \
        __asm__ __volatile__ ( \
            "        amoswap.d %0, %2, %1\n" \
            : "=r" (__ret), "+A" (*__ptr) \
            : "r" (__new) \
            : "memory"); \
        break; \
    default: \
        BUILD_BUG(); \
    } \
    __ret; \
})

#define xchg_relaxed(ptr, x) \
({ \
    __typeof__(*(ptr)) _x_ = (x); \
    (__typeof__(*(ptr))) __xchg_relaxed((ptr), \
        _x_, sizeof(*(ptr))); \
})

```

(continues on next page)

(continued from previous page)

```

#define __xchg_acquire(ptr, new, size) \
({ \
    __typeof__(ptr) __ptr = (ptr); \
    __typeof__(new) __new = (new); \
    __typeof__(*(ptr)) __ret; \
    switch (size) { \
    case 4: \
        __asm__ __volatile__ ( \
            "        amoswap.w %0, %2, %1\n" \
            RISCV_ACQUIRE_BARRIER \
            : "=r" (__ret), "+A" (*__ptr) \
            : "r" (__new) \
            : "memory"); \
        break; \
    case 8: \
        __asm__ __volatile__ ( \
            "        amoswap.d %0, %2, %1\n" \
            RISCV_ACQUIRE_BARRIER \
            : "=r" (__ret), "+A" (*__ptr) \
            : "r" (__new) \
            : "memory"); \
        break; \
    default: \
        BUILD_BUG(); \
    } \
    __ret; \
})

#define xchg_acquire(ptr, x) \
({ \
    __typeof__(*(ptr)) _x_ = (x); \
    (__typeof__(*(ptr))) __xchg_acquire((ptr), \
        _x_, sizeof(*(ptr))); \
})

#define __xchg_release(ptr, new, size) \
({ \
    __typeof__(ptr) __ptr = (ptr); \
    __typeof__(new) __new = (new); \

```

(continues on next page)

(continued from previous page)

```

__typeof__(*ptr) __ret;                                \
switch (size) {                                       \
case 4:                                               \
    __asm__ __volatile__ (                            \
        RISCV_RELEASE_BARRIER                        \
        "        amoswap.w %0, %2, %1\n"             \
        : "=r" (__ret), "+A" (*__ptr)                \
        : "r" (__new)                                 \
        : "memory");                                  \
    break;                                           \
case 8:                                               \
    __asm__ __volatile__ (                            \
        RISCV_RELEASE_BARRIER                        \
        "        amoswap.d %0, %2, %1\n"             \
        : "=r" (__ret), "+A" (*__ptr)                \
        : "r" (__new)                                 \
        : "memory");                                  \
    break;                                           \
default:                                             \
    BUILD_BUG();                                     \
}                                                     \
__ret;                                              \
})

#define xchg_release(ptr, x)                          \
({                                                    \
    __typeof__(*ptr) _x_ = (x);                      \
    (__typeof__(*ptr)) __xchg_release((ptr),          \
                                       _x_, sizeof(*ptr)); \
})

#define __xchg(ptr, new, size)                       \
({                                                    \
    __typeof__(ptr) __ptr = (ptr);                  \
    __typeof__(new) __new = (new);                  \
    __typeof__(*ptr) __ret;                          \
    switch (size) {                                  \
case 4:                                             \
    __asm__ __volatile__ (                            \
        "        amoswap.w.aqrl %0, %2, %1\n"        \

```

(continues on next page)

(continued from previous page)

```

        : "=r" (__ret), "+A" (*__ptr)           \
        : "r" (__new)                          \
        : "memory");                           \
    break;                                     \
case 8:                                       \
    __asm__ __volatile__ (                    \
        "      amoswap.d.aqr1 %0, %2, %1\n"    \
        : "=r" (__ret), "+A" (*__ptr)        \
        : "r" (__new)                          \
        : "memory");                           \
    break;                                     \
default:                                     \
    BUILD_BUG();                              \
}                                             \
__ret;                                       \
})

#define xchg(ptr, x)                          \
({                                           \
    __typeof__(*ptr) _x_ = (x);             \
    (__typeof__(*ptr)) __xchg((ptr), _x_, sizeof(*ptr)); \
})

#define xchg32(ptr, x)                        \
({                                           \
    BUILD_BUG_ON(sizeof(*ptr) != 4);        \
    xchg((ptr), (x));                       \
})

#define xchg64(ptr, x)                       \
({                                           \
    BUILD_BUG_ON(sizeof(*ptr) != 8);        \
    xchg((ptr), (x));                       \
})

/*
 * Atomic compare and exchange. Compare OLD with MEM, if identical,
 * store NEW in MEM. Return the initial value in MEM. Success is
 * indicated by comparing RETURN with OLD.
 */

```

(continues on next page)

(continued from previous page)

```

#define __cmpxchg_relaxed(ptr, old, new, size) \
({ \
    __typeof__(ptr) __ptr = (ptr); \
    __typeof__(*(ptr)) __old = (old); \
    __typeof__(*(ptr)) __new = (new); \
    __typeof__(*(ptr)) __ret; \
    register unsigned int __rc; \
    switch (size) { \
    case 4: \
        __asm__ __volatile__ ( \
            "0:    lr.w %0, %2\n" \
            "      bne %0, %z3, 1f\n" \
            "      sc.w %1, %z4, %2\n" \
            "      bnez %1, 0b\n" \
            "1:\n" \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr) \
            : "rJ" ((long)__old), "rJ" (__new) \
            : "memory"); \
        break; \
    case 8: \
        __asm__ __volatile__ ( \
            "0:    lr.d %0, %2\n" \
            "      bne %0, %z3, 1f\n" \
            "      sc.d %1, %z4, %2\n" \
            "      bnez %1, 0b\n" \
            "1:\n" \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr) \
            : "rJ" (__old), "rJ" (__new) \
            : "memory"); \
        break; \
    default: \
        BUILD_BUG(); \
    } \
    __ret; \
})

#define cmpxchg_relaxed(ptr, o, n) \
({ \
    __typeof__(*(ptr)) _o_ = (o); \
    __typeof__(*(ptr)) _n_ = (n); \

```

(continues on next page)

(continued from previous page)

```

        (__typeof__(*(ptr))) __cmpxchg_relaxed((ptr),          \
                                                _o_, _n_, sizeof(*(ptr))); \
    })
#define __cmpxchg_acquire(ptr, old, new, size)                \
({                                                            \
    __typeof__(ptr) __ptr = (ptr);                            \
    __typeof__(*(ptr)) __old = (old);                          \
    __typeof__(*(ptr)) __new = (new);                          \
    __typeof__(*(ptr)) __ret;                                  \
    register unsigned int __rc;                                \
    switch (size) {                                           \
    case 4:                                                    \
        __asm__ __volatile__ (                                \
            "0:    lr.w %0, %2\n"                               \
            "      bne %0, %z3, 1f\n"                           \
            "      sc.w %1, %z4, %2\n"                           \
            "      bnez %1, 0b\n"                               \
            RISCV_ACQUIRE_BARRIER                            \
            "1:\n"                                             \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)      \
            : "rJ" ((long)__old), "rJ" (__new)                  \
            : "memory");                                       \
        break;                                                \
    case 8:                                                    \
        __asm__ __volatile__ (                                \
            "0:    lr.d %0, %2\n"                               \
            "      bne %0, %z3, 1f\n"                           \
            "      sc.d %1, %z4, %2\n"                           \
            "      bnez %1, 0b\n"                               \
            RISCV_ACQUIRE_BARRIER                            \
            "1:\n"                                             \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)      \
            : "rJ" (__old), "rJ" (__new)                        \
            : "memory");                                       \
        break;                                                \
    default:                                                  \
        BUILD_BUG();                                          \
    }                                                         \
    __ret;

```

(continues on next page)

(continued from previous page)

```

})

#define cmpxchg_acquire(ptr, o, n) \
({ \
    __typeof__(*ptr) _o_ = (o); \
    __typeof__(*ptr) _n_ = (n); \
    (__typeof__(*ptr)) __cmpxchg_acquire((ptr), \
        _o_, _n_, sizeof(*ptr)); \
})

#define __cmpxchg_release(ptr, old, new, size) \
({ \
    __typeof__(ptr) __ptr = (ptr); \
    __typeof__(*ptr) __old = (old); \
    __typeof__(*ptr) __new = (new); \
    __typeof__(*ptr) __ret; \
    register unsigned int __rc; \
    switch (size) { \
    case 4: \
        __asm__ __volatile__ ( \
            RISCV_RELEASE_BARRIER \
            "0:    lr.w %0, %2\n" \
            "      bne %0, %z3, 1f\n" \
            "      sc.w %1, %z4, %2\n" \
            "      bnez %1, 0b\n" \
            "1:\n" \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr) \
            : "rJ" ((long)__old), "rJ" (__new) \
            : "memory"); \
        break; \
    case 8: \
        __asm__ __volatile__ ( \
            RISCV_RELEASE_BARRIER \
            "0:    lr.d %0, %2\n" \
            "      bne %0, %z3, 1f\n" \
            "      sc.d %1, %z4, %2\n" \
            "      bnez %1, 0b\n" \
            "1:\n" \
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr) \
            : "rJ" (__old), "rJ" (__new) \
            : "memory"); \
    } \
}

```

(continues on next page)

(continued from previous page)

```

        : "memory");
        break;
    default:
        BUILD_BUG();
    }
    __ret;
})

#define cmpxchg_release(ptr, o, n)
({
    __typeof__(*ptr) _o_ = (o);
    __typeof__(*ptr) _n_ = (n);
    (__typeof__(*ptr)) __cmpxchg_release((ptr),
        _o_, _n_, sizeof(*ptr));
})

#define __cmpxchg(ptr, old, new, size)
({
    __typeof__(ptr) __ptr = (ptr);
    __typeof__(*ptr) __old = (old);
    __typeof__(*ptr) __new = (new);
    __typeof__(*ptr) __ret;
    register unsigned int __rc;
    switch (size) {
    case 4:
        __asm__ __volatile__ (
            "0:    lr.w %0, %2\n"
            "      bne %0, %z3, 1f\n"
            "      sc.w.rl %1, %z4, %2\n"
            "      bnez %1, 0b\n"
            "      fence rw, rw\n"
            "1: \n"
            : "=&r" (__ret), "=&r" (__rc), "+A" (*__ptr)
            : "rJ" ((long)__old), "rJ" (__new)
            : "memory");
        break;
    case 8:
        __asm__ __volatile__ (
            "0:    lr.d %0, %2\n"
            "      bne %0, %z3, 1f\n"

```

(continues on next page)

(continued from previous page)

```

        "      sc.d.rl %1, %z4, %2\n"          \
        "      bnez %1, 0b\n"                \
        "      fence rw, rw\n"              \
        "1:\n"                                \
        : "&r" (__ret), "&r" (__rc), "+A" (*__ptr) \
        : "rJ" (__old), "rJ" (__new)        \
        : "memory");                          \
    break;                                    \
default:                                     \
    BUILD_BUG();                              \
}                                             \
__ret;                                       \
})

```