

XuanTie-Openc906-UserManual

Jun 05, 2022

Copyright 2022 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	openc906 first version	2021.10.19

1	Overview	1
1.1	Introduction	1
1.2	Features	1
1.3	Configuration options	2
1.4	Naming conventions	2
1.4.1	Symbols	2
1.4.2	Terms	2
1.5	Release notes	2
2	C906 overview	5
2.1	Structure	5
2.2	Unit introduction	5
2.2.1	IFU	5
2.2.2	IDU	6
2.2.3	Execution units	6
2.2.4	LSU	7
2.2.5	MMU	7
2.2.6	PMP unit	7
2.2.7	Master device interface unit	7
2.2.8	PLIC	7
2.2.9	Timer	8
3	Programming models	9
3.1	Working mode and register view	9
3.2	General-purpose registers	11
3.3	Floating-point registers	11
3.3.1	Transmit data between floating-point and general-purpose registers	12

3.3.2	Maintain consistency of register precision	12
3.4	Vector registers	13
3.4.1	Transmit data between floating-point and general-purpose registers	13
3.4.2	Transmit data between floating-point and vector registers	13
3.5	System control registers	13
3.5.1	M-mode control registers	13
3.5.2	S-mode control registers	17
3.5.3	U-mode control registers	18
3.6	Exception handling	19
3.7	Data formats	22
3.7.1	Floating-point data format	23
3.7.2	Integer data format	23
3.7.3	Big-endian and little-endian	23
3.8	Memory model	24
4	Instruction sets	26
4.1	RV64GC instructions	26
4.1.1	RV64I	26
4.1.2	RV64M	30
4.1.3	RV64A	32
4.1.4	RV64F	33
4.1.5	RV64D	35
4.1.6	RVC	38
4.2	T-Head extended instruction sets	40
4.2.1	Cache instruction subset	41
4.2.2	Synchronization instruction subset	42
4.2.3	Arithmetic operation instructions	42
4.2.4	Bit operation instruction subset	43
4.2.5	Store instruction subset	43
4.2.6	Half-precision floating-point instruction subset	47
5	MMU	50
5.1	MMU overview	50
5.2	Programming models and address translation	50
5.2.1	MMU control register	50
5.2.1.1	MMU address translation register (SATP)	51
5.2.2	Address translation process	51
5.2.2.1	Page table structure	52
5.2.2.2	Address translation process	54
5.3	TLB	55
6	PMP	56
6.1	PMP overview	56

6.2	PMP control registers	56
6.2.1	Physical memory protection configuration (PMPCFG) register	57
6.2.2	Physical memory protection address (PMPADDR) register	61
7	Memory subsystem	62
7.1	I-Cache subsystem	62
7.1.1	Instruction prefetch	62
7.1.2	Branch history table	63
7.1.3	Branch and jump target predictor	63
7.1.4	Return address predictor	63
7.2	D-Cache subsystem	64
7.2.1	Data prefetch	64
7.2.2	Adaptive write allocation mechanism	65
7.2.3	Exclusive access	65
7.3	L1 cache operations	66
7.3.1	Extended registers of the L1 cache	66
7.3.2	Extended instructions of L1 cache	66
8	Bus interface protocol	68
8.1	Master device interface unit	68
8.1.1	Features	68
8.1.2	Protocol content	68
8.1.2.1	Supported transmission types	69
8.1.2.2	Supported response types	69
8.1.3	CPU behavior in different bus responses	69
9	CLINT	70
9.1	Register address mapping	70
9.2	Software interrupts	71
9.3	Timer interrupts	72
10	PLIC	74
10.1	Interrupt handling mechanism	74
10.1.1	Interrupt arbitration	74
10.1.2	Interrupt request and response	75
10.1.3	Interrupt completion	76
10.2	PLIC register address mapping	76
10.3	PLIC_PRIO register	78
10.4	PLIC_IP register	78
10.5	PLIC_IE register	79
10.6	PLIC_CTRL register	79
10.7	PLIC_TH register	80
10.8	PLIC_CLAIM register	80

11 Debug	82
11.1 Overview	82
11.2 DM registers	83
11.3 Resource configuration	85
12 HPM	87
12.1 HPM control registers	87
12.1.1 MCOUNTEREN register	87
12.1.2 SCOUNTEREN register	88
12.1.3 MCOUNTINHIBIT register	89
12.1.4 SCOUNTINHIBIT register	90
12.1.5 MCOUNTERWEN register	91
12.1.6 MHPMCR register	91
12.1.7 SHPMCR register	93
12.1.8 HPMSP register	94
12.2 Performance monitoring event select register	95
12.3 Event counters	97
12.4 HPM event overflow interrupt	98
13 Appendix A Standard Instructions	99
13.1 Appendix A-1 I instructions	99
13.1.1 ADD: a signed add instruction	99
13.1.2 ADDIW: a signed add immediate instruction that operates on the lower 32 bits . . .	100
13.1.3 ADDW: a signed add instruction that operates on the lower 32 bits	101
13.1.4 AND: a bitwise AND instruction	101
13.1.5 AUIPC: an instruction that adds the immediate in the upper bits to the PC	102
13.1.6 BEQ: a branch-if-equal instruction	102
13.1.7 BGE: a signed branch-if-greater-than-or-equal instruction	103
13.1.8 BGEU: an unsigned branch-if-greater-than-or-equal instruction	104
13.1.9 BLT: a signed branch-if-less-than instruction	104
13.1.10 BLTU: an unsigned branch-if-less-than instruction	105
13.1.11 BNE: a branch-if-not-equal instruction	106
13.1.12 CSRRC: a move instruction that clears control registers	106
13.1.13 CSRRCI: a move instruction that clears immediates in control registers	107
13.1.14 CSRRS: a move instruction for setting control registers	107
13.1.15 CSRRSI: a move instruction for setting immediates in control registers	108
13.1.16 CSRRW: a move instruction that reads/writes control registers	109
13.1.17 CSRRWI: a move instruction that reads/writes immediates in control registers . . .	109
13.1.18 EBREAK: a breakpoint instruction	110
13.1.19 ECALL: an environment call instruction	110
13.1.20 FENCE: a memory synchronization instruction	111
13.1.21 FENCE.I: an instruction stream synchronization instruction	111

13.1.22	JAL: an instruction for directly jumping to a subroutine	112
13.1.23	JALR: an instruction for jumping to a subroutine by using an address in a register	112
13.1.24	LB: a sign-extended byte load instruction	113
13.1.25	LBU: an unsign-extended byte load instruction	113
13.1.26	LD: a doubleword load instruction	114
13.1.27	LH: a sign-extended halfword load instruction	114
13.1.28	LHU: an unsign-extended halfword load instruction	115
13.1.29	LUI: an instruction for loading the immediate in the upper bits	115
13.1.30	LW: a sign-extended word load instruction	116
13.1.31	LWU: an unsign-extended word load instruction	116
13.1.32	MRET: an instruction for returning from exceptions in M-mode	116
13.1.33	OR: a bitwise OR instruction	117
13.1.34	ORI: an immediate bitwise OR instruction	117
13.1.35	SB: a byte store instruction	118
13.1.36	SD: a doubleword store instruction	118
13.1.37	SFENCE.VMA: a virtual memory synchronization instruction	119
13.1.38	SH: a halfword store instruction	119
13.1.39	SLL: a logical left shift instruction	120
13.1.40	SLLI: an immediate logical left shift instruction	121
13.1.41	SLLIW: an immediate logical left shift instruction that operates on the lower 32 bits	121
13.1.42	SLLW: a logical left shift instruction that operates on the lower 32 bits	121
13.1.43	SLT: a signed set-if-less-than instruction	122
13.1.44	SLTI: a signed set-if-less-than-immediate instruction	122
13.1.45	SLTIU: an unsigned set-if-less-than-immediate instruction	123
13.1.46	SLTU: an unsigned set-if-less-than instruction	123
13.1.47	SRA: an arithmetic right shift instruction	124
13.1.48	SRAI: an immediate arithmetic right shift instruction	124
13.1.49	SLLIW: an immediate arithmetic right shift instruction that operates on the lower 32 bits	125
13.1.50	SRAW: an arithmetic right shift instruction that operates on the lower 32 bits	125
13.1.51	SRET: an instruction for returning from exceptions in S-mode	126
13.1.52	SRL: a logical right shift instruction	126
13.1.53	SRLI: an immediate logical right shift instruction	127
13.1.54	SRLIW: an immediate logical right shift instruction that operates on the lower 32 bits	127
13.1.55	SRLW: a logical right shift instruction that operates on the lower 32 bits	128
13.1.56	SUB: a signed subtract instruction	128
13.1.57	SUBW: a signed subtract instruction that operates on the lower 32 bits	128
13.1.58	SW: a word store instruction	129
13.1.59	WFI: an instruction for entering the low power mode	129
13.1.60	XOR: a bitwise XOR instruction	130
13.1.61	XORI: an immediate bitwise XOR instruction	130
13.2	Appendix A-2 M instructions	131

13.2.1	DIV: a signed divide instruction	131
13.2.2	DIVU: an unsigned divide instruction	131
13.2.3	DIVUW: an unsigned divide instruction that operates on the lower 32 bits	132
13.2.4	DIVW: a signed divide instruction that operates on the lower 32 bits	132
13.2.5	MUL: a signed multiply instruction	133
13.2.6	MULH: a signed multiply instruction that extracts the upper bits	133
13.2.7	MULHSU: a signed-unsigned multiply instruction that extracts the upper bits	134
13.2.8	MULHU: an unsigned multiply instruction that extracts the upper bits	134
13.2.9	MULW: a signed multiply instruction that operates on the lower 32 bits	135
13.2.10	REM: a signed remainder instruction	135
13.2.11	REMU: an unsigned remainder instruction	136
13.2.12	REMUW: an unsigned remainder instruction that operates on the lower 32 bits	136
13.2.13	REMW: a signed remainder instruction that operates on the lower 32 bits	137
13.3	Appendix A-3 A instructions	137
13.3.1	AMOADD.D: an atomic add instruction	137
13.3.2	AMOADD.W: an atomic add instruction that operates on the lower 32 bits	138
13.3.3	AMOAND.D: an atomic bitwise AND instruction	139
13.3.4	AMOAND.W: an atomic bitwise AND instruction that operates on the lower 32 bits	140
13.3.5	AMOMAX.D: an atomic signed MAX instruction	141
13.3.6	AMOMAX.W: an atomic signed MAX instruction that operates on the lower 32 bits	142
13.3.7	MOMAXU.DA: an atomic unsigned MAX instruction	143
13.3.8	AMOMAXU.W: an atomic unsigned MAX instruction that operates on the lower 32 bits.	144
13.3.9	AMOMIN.D: an atomic signed MIN instruction	145
13.3.10	AMOMIN.W: an atomic signed MIN instruction that operates on the lower 32 bits	146
13.3.11	AMOMINU.D: an atomic unsigned MIN instruction	147
13.3.12	AMOMINU.W: an atomic unsigned MIN instruction that operates on the lower 32 bits	148
13.3.13	AMOOR.D: an atomic bitwise OR instruction.	149
13.3.14	AMOSWAP.D: an atomic swap instruction	150
13.3.15	AMOSWAP.W: an atomic swap instruction that operates on the lower 32 bits	151
13.3.16	AMOXOR.D: an atomic bitwise XOR instruction	152
13.3.17	AMOXOR.W: an atomic bitwise XOR instruction that operates on the lower 32 bits	153
13.3.18	LR.D: a doubleword load-reserved instruction	154
13.3.19	LR.W: a word load-reserved instruction	155
13.3.20	SC.D: a doubleword store-conditional instruction	156
13.3.21	SC.W: a word store-conditional instruction	157
13.4	Appendix A-4 F instructions	158
13.4.1	FADD.S: a single-precision floating-point add instruction	158
13.4.2	FCLASS.S: a single-precision floating-point classify instruction	159
13.4.3	FCVT.L.S: an instruction that converts a single-precision floating-point number into a signed long integer	160

13.4.4	FCVT.LU.S: an instruction that converts a single-precision floating-point number into an unsigned long integer	161
13.4.5	FCVT.S.L: an instruction that converts a signed long integer into a single-precision floating-point number	162
13.4.6	FCVT.S.LU: an instruction that converts an unsigned long integer into a single-precision floating-point number	163
13.4.7	FCVT.S.W: an instruction that converts a signed integer into a single-precision floating-point number	164
13.4.8	FCVT.S.WU: an instruction that converts an unsigned integer into a single-precision floating-point number	165
13.4.9	FCVT.W.S: an instruction that converts a single-precision floating-point number into a signed integer	166
13.4.10	FCVT.WU.S: an instruction that converts a single-precision floating-point number into an unsigned integer	167
13.4.11	FDIV.S: a single-precision floating-point divide instruction	168
13.4.12	FEQ.S: a single-precision floating-point compare equal instruction	169
13.4.13	FLE.S: a single-precision floating-point compare less than or equal to instruction	169
13.4.14	FLT.S: a single-precision floating-point compare less than instruction	170
13.4.15	FLW: a single-precision floating-point load instruction	171
13.4.16	FMADD.S: a single-precision floating-point multiply-add instruction	171
13.4.17	FMAX.S: a single-precision floating-point MAX instruction	172
13.4.18	FMIN.S: a single-precision floating-point MIN instruction	173
13.4.19	FMSUB.S: a single-precision floating-point multiply-subtract instruction	173
13.4.20	FMUL.S: a single-precision floating-point multiply instruction	174
13.4.21	FMV.W.X: a single-precision floating-point write move instruction	175
13.4.22	FMV.X.H: a single-precision floating-point read move instruction	176
13.4.23	FNMADD.S: a single-precision floating-point negate-(multiply-add) instruction	176
13.4.24	FNMSUB.S: a single-precision floating-point negate-(multiply-subtract) instruction	177
13.4.25	FSGNJ.S: a single-precision floating-point sign-injection instruction	178
13.4.26	FSGNJS: a single-precision floating-point negate sign-injection instruction	179
13.4.27	FSGNJX.S: a single-precision floating-point XOR sign-injection instruction	179
13.4.28	FSQRT.S: a single-precision floating-point square-root instruction	180
13.4.29	FSUB.S: a single-precision floating-point subtract instruction	181
13.4.30	FSW: a single-precision floating-point store instruction	182
13.5	Appendix A-5 D instructions	182
13.5.1	FADD.D: a double-precision floating-point add instruction	182
13.5.2	FCLASS.D: a double-precision floating-point classify instruction	183
13.5.3	FCVT.D.L: an instruction that converts a signed long integer into a double-precision floating-point number	184
13.5.4	FCVT.D.LU: an instruction that converts an unsigned long integer into a double-precision floating-point number	185

13.5.5	FCVT.D.S: an instruction that converts a single-precision floating-point number into a double-precision floating-point number	186
13.5.6	FCVT.D.W: an instruction that converts a signed integer into a double-precision floating-point number	187
13.5.7	FCVT.D.WU: an instruction that converts an unsigned integer into a double-precision floating-point number	187
13.5.8	FCVT.L.D: an instruction that converts a double-precision floating-point number into a signed long integer	188
13.5.9	FCVT.LU.D: an instruction that converts a double-precision floating-point number into an unsigned long integer	189
13.5.10	FCVT.S.D: an instruction that converts a double-precision floating-point number into a single-precision floating-point number	190
13.5.11	FCVT.W.D: an instruction that converts a double-precision floating-point number into a signed integer	191
13.5.12	FCVT.WU.D: an instruction that converts a double-precision floating-point number into an unsigned integer	192
13.5.13	FDIV.D: a double-precision floating-point divide instruction	193
13.5.14	FEQ.D: a double-precision floating-point compare equal instruction	194
13.5.15	FLD: a double-precision floating-point load instruction	194
13.5.16	FLE.D: a double-precision floating-point compare less than or equal to instruction	195
13.5.17	FLT.D: a double-precision floating-point compare less than instruction	195
13.5.18	FMAX.D: a double-precision floating-point MAX instruction	197
13.5.19	FMIN.D: a double-precision floating-point MIN instruction	197
13.5.20	FMSUB.D: a double-precision floating-point multiply-subtract instruction	198
13.5.21	FMUL.D: a double-precision floating-point multiply instruction	199
13.5.22	FMV.D.X: a double-precision floating-point write move instruction	200
13.5.23	FMV.X.D: a double-precision floating-point read move instruction	201
13.5.24	FNMADD.D: a double-precision floating-point negate-(multiply-add) instruction	201
13.5.25	FNMSUB.D: a double-precision floating-point negate-(multiply-subtract) instruction	202
13.5.26	FSD: a double-precision floating-point store instruction	203
13.5.27	FSGNJ.D: a double-precision floating-point sign-injection instruction	204
13.5.28	FSGNJN.D: a double-precision floating-point negate sign-injection instruction	204
13.5.29	FSGNJX.D: a double-precision floating-point XOR sign-injection instruction	205
13.5.30	FSQRT.D: a double-precision floating-point square-root instruction	205
13.5.31	FSUB.D: a double-precision floating-point subtract instruction	206
13.6	Appendix A-6 C Instructions	207
13.6.1	C.ADD: a signed add instruction	207
13.6.2	C.ADDI: a signed add immediate instruction	208
13.6.3	C.ADDIW: an add immediate instruction that operates on the lower 32 bits	208
13.6.4	C.ADDI4SPN: an instruction that adds an immediate scaled by 4 to the stack pointer	209
13.6.5	C.ADDI16SP: an instruction that adds an immediate scaled by 16 to the stack pointer	210
13.6.6	C.ADDW: a signed add instruction that operates on the lower 32 bits	210

13.6.7	C.AND: a bitwise AND instruction	211
13.6.8	C.ANDI: an immediate bitwise AND instruction	212
13.6.9	C.BEQZ: a branch-if-equal-to-zero instruction	213
13.6.10	C.BNEZ: a branch-if-not-equal-to-zero instruction	214
13.6.11	C.EBREAK: a break instruction	215
13.6.12	C.FLD: a floating-point load doubleword instruction	215
13.6.13	C.FLDSP: a floating-point doubleword load stack instruction	216
13.6.14	C.FSD: a floating-point store doubleword instruction	217
13.6.15	C.FSDSP: a floating-point store doubleword stack pointer instruction	218
13.6.16	C.J: a unconditional jump instruction	218
13.6.17	C.JALR: a jump and link register instruction	219
13.6.18	C.JR: a jump register instruction	219
13.6.19	C.LD: a load doubleword instruction	220
13.6.20	C.LDSP: a load doubleword instruction	221
13.6.21	C.LI: a load immediate instruction	221
13.6.22	C.LUI: a load upper immediate instruction	222
13.6.23	C.LW: a load word instruction	223
13.6.24	C.LWSP: a load word stack pointer instruction	223
13.6.25	C.MV: an instruction that copies the value in rs to rd	224
13.6.26	C.NOP: a no-operation instruction	225
13.6.27	C.OR: a bitwise OR instruction	225
13.6.28	C.SD: a store doubleword instruction	226
13.6.29	C.SDSP: a store doubleword stack pointer instruction	227
13.6.30	C.SLLI: an immediate logical left shift instruction	227
13.6.31	C.SRAI: a right shift arithmetic immediate instruction	228
13.6.32	C.SRLI: an immediate right shift instruction	229
13.6.33	C.SW: a store word instruction	229
13.6.34	C.SWSP: a store word stack pointer instruction	230
13.6.35	C.SUB: a signed subtract instruction	231
13.6.36	C.SUBW: a signed subtract instruction that operates on the lower 32 bits	232
13.6.37	C.XOR: a bitwise XOR instruction	232
13.7	Appendix A-7 V instructions	233
13.7.1	VAADD.VI: a vector-immediate instruction that averages integer adds	234
13.7.2	VAADD.VV: a vector instruction that averages integer adds	235
13.7.3	VAADD.VX: a vector-scalar instruction that averages integer adds	235
13.7.4	VADC.VIM: an integer vector-immediate add-with-carry instruction	236
13.7.5	VADC.VVM: an integer vector add-with-carry instruction	237
13.7.6	VADC.VXM: a vector-scalar integer add-with-carry instruction	237
13.7.7	VADD.VI: an integer vector-immediate add instruction	238
13.7.8	VADD.VV: an integer vector add instruction	238
13.7.9	VADD.VX: a vector-scalar integer add instruction	239
13.7.10	VAMOADD.V: a vector atomic doubleword add instruction	240

13.7.11	VAMOADDW.V: a vector atomic word add instruction	240
13.7.12	VAMOANDD.V: a vector atomic doubleword bitwise AND instruction	241
13.7.13	VAMOANDW.V: a vector atomic word bitwise AND instruction	242
13.7.14	VAMOMAXD.V: a vector atomic doubleword signed MAX instruction	243
13.7.15	VAMOMAXW.V: a vector atomic word signed MAX instruction	244
13.7.16	VAMOMAXUD.V: a vector atomic doubleword unsigned MAX instruction	245
13.7.17	VAMOMAXUW.V: a vector atomic word unsigned MAX instruction	246
13.7.18	VAMOMIND.V: a vector atomic doubleword signed MIN instruction	247
13.7.19	VAMOMINW.V: a vector atomic word signed MIN instruction	248
13.7.20	VAMOMINUD.V: a vector atomic doubleword unsigned MIN instruction	248
13.7.21	VAMOMINUW.V: a vector atomic word unsigned MIN instruction	249
13.7.22	VAMOORD.V: a vector atomic doubleword bitwise OR instruction	250
13.7.23	VAMOORW.V: a vector atomic word bitwise OR instruction	251
13.7.24	VAMOSWAPD.V: a vector atomic doubleword swap instruction	252
13.7.25	VAMOSWAPW.V: a vector atomic word swap instruction	253
13.7.26	VAMOXORD.V: a vector atomic doubleword bitwise XOR instruction	254
13.7.27	VAMOXORW.V: a vector atomic doubleword bitwise XOR instruction	255
13.7.28	VAND.VI: a vector-immediate bitwise AND instruction	256
13.7.29	VAND.VV: a vector bitwise AND instruction	256
13.7.30	VAND.VX: a vector-scalar bitwise AND instruction	257
13.7.31	VASUB.VV: a vector integer subtract-average instruction	257
13.7.32	VASUB.VX: a vector-scalar integer subtract-average instruction	258
13.7.33	VCOMPRESS.VM: a vector integer element compress instruction	259
13.7.34	VDIV.VV: an integer vector signed divide instruction	260
13.7.35	VDIV.VX: a vector-scalar integer signed divide instruction	260
13.7.36	VDIVU.VV: an integer vector unsigned divide instruction	261
13.7.37	VDIVU.VX: a vector-scalar integer unsigned divide instruction	262
13.7.38	VEXT.X.V: an integer vector get element instruction	262
13.7.39	VFADD.VF: a vector-scalar floating-point add instruction	263
13.7.40	VFADD.VV: a vector floating-point add instruction	264
13.7.41	VFCLASS.V: a vector floating-point classify instruction	265
13.7.42	VFCVT.F.X.V: a single-width floating-point/integer type-convert instruction that converts signed integers to floating-point values	266
13.7.43	VFCVT.F.XU.V: a single-width floating-point/integer type-convert instruction that converts unsigned vector integers to floating-point values	267
13.7.44	VFCVT.X.F.V: a single-width floating-point/integer type-convert instruction that converts vector floating-point values to signed integers	268
13.7.45	VFCVT.XU.F.V: a single-width floating-point/integer type-convert instruction that converts vector floating-point values to unsigned integers	269
13.7.46	VFDIV.VF: a vector-scalar floating-point divide instruction	270
13.7.47	VFDIV.VV: a vector floating-point divide instruction	271
13.7.48	VFMACC.VF: an FP multiply-accumulate instruction that overwrites addends . . .	272

13.7.49	VFMACC.VV: an FP multiply-accumulate instruction that overwrites addends	273
13.7.50	VFMADD.VF: an FP multiply-add instruction that overwrites multiplicands	274
13.7.51	VFMADD.VV: an FP multiply-add instruction that overwrites multiplicands	275
13.7.52	VFMAX.VF: a vector-scalar floating-point MAX instruction	276
13.7.53	VFMAX.VV: a vector floating-point MAX instruction	277
13.7.54	VFMERGE.VFM: a vector floating-point element select instruction	278
13.7.55	VFMIN.VF: a vector-scalar floating-point MIN instruction	278
13.7.56	VFMIN.VV: a vector floating-point MIN instruction	279
13.7.57	VFMSAC.VF: a vector-scalar floating-point multiply-sub instruction that overwrites subtrahends	280
13.7.58	VFMSAC.VV: a vector floating-point multiply-sub instruction that overwrites sub- trahends	281
13.7.59	VFMSUB.VF: a vector-scalar floating-point multiply-sub instruction that overwrites multiplicands	282
13.7.60	VFMSUB.VV: a vector floating-point multiply-sub instruction that overwrites mul- tiplicands	283
13.7.61	VFMUL.VF: a vector-scalar floating-point multiply instruction	284
13.7.62	VFMUL.VV: a vector floating-point multiply instruction	285
13.7.63	VFMV.F.S: an instruction that moves element 0 of a vector to a floating-point scalar	286
13.7.64	VFMV.S.F: an instruction that moves a floating-point scalar to element 0 of a vector	287
13.7.65	VFMV.V.F: an instruction that moves a floating-point scalar to a vector	287
13.7.66	VFNCVT.F.F.V: a vector floating-point reduction instruction	288
13.7.67	VFNCVT.F.XU.V: a vector reduction-type instruction that converts unsigned inte- gers to floating-point values	289
13.7.68	VFNCVT.F.X.V: a vector reduction-type instruction that converts signed integers to floating-point values	290
13.7.69	VFNCVT.X.F.V: a vector reduction-type instruction that converts floating-point val- ues to signed integers	291
13.7.70	VFNCVT.XU.F.V: a vector reduction-type instruction that converts floating-point values to unsigned integers	292
13.7.71	VFNMACC.VF: a vector-scalar floating-point negate-(multiply-add) instruction that overwrites subtrahends	293
13.7.72	VFNMACC.VV: a vector floating-point negate-(multiply-add) instruction that over- writes subtrahends	294
13.7.73	VFNMADD.VF: a vector-scalar floating-point negate-(multiply-add) instruction that overwrites multiplicands	295
13.7.74	VFNMADD.VV: a vector floating-point negate-(multiply-add) instruction that over- writes multiplicands	296
13.7.75	VFNMSAC.VF: a vector-scalar floating-point negate-(multiply-sub) instruction that overwrites minuends	297
13.7.76	VFNMSAC.VV: a vector floating-point negate-(multiply-sub) instruction that over- writes minuends	298

13.7.77	VFNMSUB.VF: a vector-scalar floating-point negate-(multiply-sub) instruction that overwrites multiplicands	299
13.7.78	VFNMSUB.VV: a vector floating-point negate-(multiply-sub) instruction that overwrites multiplicands	300
13.7.79	VFRDIV.VF: a scalar-vector floating-point divide instruction	301
13.7.80	VFREDMAX.VS: a vector floating-point reduction instruction that obtains the MAX value	302
13.7.81	VFREDMIN.VS: a vector floating-point reduction instruction that obtains the MIN value	303
13.7.82	VFREDOSUM.VS: a vector floating-point reduction instruction that sums values in element order	304
13.7.83	VFREDSUM.VS: a vector floating-point reduction instruction that sums values in any order	305
13.7.84	VFRSUB.VF: a vector-scalar floating-point subtract instruction	305
13.7.85	VFSGNJ.VF: a vector-scalar floating-point sign-injection instruction	306
13.7.86	VFSGNJ.VV: a vector floating-point sign-injection instruction	307
13.7.87	VFSGNJN.VF: a vector-scalar floating-point NOT-sign-injection instruction	307
13.7.88	VFSGNJN.VV: a vector floating-point NOT-sign-injection instruction	308
13.7.89	VFSGNJX.VF: a vector-scalar floating-point XOR-sign-injection instruction	309
13.7.90	VFSGNJX.VV: a vector floating-point XOR-sign-injection instruction	309
13.7.91	VFSQRT.V: a vector floating-point square-root instruction	310
13.7.92	VFSUB.VF: a vector-scalar floating-point subtract instruction	311
13.7.93	VFSUB.VV: a vector floating-point subtract instruction	312
13.7.94	VFWADD.VF: a vector-scalar floating-point widening add instruction	313
13.7.95	VFWADD.VV: a vector floating-point widening add instruction	314
13.7.96	VFWADD.WF: a widening vector-scalar floating-point widening add instruction	315
13.7.97	VFWADD.WF: a widening vector floating-point widening add instruction	316
13.7.98	VFWCVT.F.F.V: a vector floating-point widening type-convert instruction	317
13.7.99	VFWCVT.F.X.V: a vector widening type-convert instruction that converts signed integers to floating-point values	318
13.7.100	VFWCVT.F.XU.V: a vector widening type-convert instruction that converts unsigned integers to floating-point values	319
13.7.101	VFWCVT.X.F.V: a vector widening type-convert instruction that converts floating-point values to signed integers	320
13.7.102	VFWCVT.XU.F.V: a vector widening type-convert instruction that converts floating-point values to unsigned integers	321
13.7.103	VFWMACC.VF: a vector-scalar floating-point widening multiply-add instruction that overwrites addends	322
13.7.104	VFWMACC.VV: a vector floating-point widening multiply-add instruction that overwrites addends	323
13.7.105	VFWMSAC.VF: a vector floating-point widening multiply-sub instruction that overwrites addends	324

13.7.106VFWMSAC.VV: a vector floating-point widening multiply-sub instruction that overwrites addends	325
13.7.107VFWMUL.VF: a vector-scalar widening floating-point multiply instruction	326
13.7.108VFWMUL.VV: a vector widening floating-point multiply instruction	327
13.7.109VFWNMACC.VF: a vector-scalar floating-point widening negate-(multiply-add) instruction that overwrites addends	328
13.7.110VFWNMACC.VV: a vector floating-point widening negate-(multiply-add) instruction that overwrites addends	330
13.7.111VFWNMSAC.VF: a vector-scalar floating-point widening negate-(multiply-sub) instruction that overwrites addends	331
13.7.112VFWNMSAC.VV: a vector floating-point widening negate-(multiply-sub) instruction that overwrites addends	332
13.7.113VFWREDOSUM.VS: a vector widening floating-point reduction instruction that sums values in element order	333
13.7.114VFWREDSUM.VS: a vector widening floating-point reduction instruction that sums values in any order	334
13.7.115VFWSUB.VF: a vector-scalar widening floating-point subtract instruction	334
13.7.116VFWSUB.VV: a vector widening floating-point subtract instruction	335
13.7.117VFWSUB.WF: a widening vector-scalar floating-point widening subtract instruction	337
13.7.118VFWSUB.WV: a widening vector floating-point widening subtract instruction	338
13.7.119VID.V: a vector element index instruction that writes each element's index to the destination	339
13.7.120VIOTA.M: a vector instruction that gets destination offsets of active elements	339
13.7.121VLB.V: a vector signed byte load instruction	341
13.7.122VLBFF.V: a vector fault-only-first (FOF) signed byte load instruction:	341
13.7.123VLBU.V: a vector unsigned byte load instruction	342
13.7.124VLBUFF.V: a vector FOF unsigned byte load instruction	343
13.7.125VLE.V: a vector element load instruction	343
13.7.126VLEFF.V: a vector FOF unsigned element load instruction	344
13.7.127VLH.V: a vector signed halfword load instruction	345
13.7.128VLHFF.V: a vector FOF signed halfword load instruction	346
13.7.129VLHU.V: a vector unsigned halfword load instruction	346
13.7.130VLHUFF.V: a vector FOF unsigned halfword load instruction	347
13.7.131VLSB.V: a vector strided signed byte load instruction	348
13.7.132VLSBU.V: a vector strided unsigned byte load instruction	349
13.7.133VLSE.V: a vector strided element load instruction	349
13.7.134VLSEG<NF>B.V: a vector SEGMENT signed byte load instruction	350
13.7.135VLSEG<NF>BFF.V: a vector FOF SEGMENT signed byte load instruction	351
13.7.136VLSEG<NF>BU.V: a vector SEGMENT unsigned byte load instruction	352
13.7.137VLSEG<NF>BUFF.V: a vector FOF SEGMENT unsigned byte load instruction	353
13.7.138VLSEG<NF>E.V: a vector SEGMENT element load instruction	354
13.7.139VLSEG<NF>EFF.V: a vector FOF SEGMENT element load instruction	355

13.7.140VLSEGE<NF>H.V: a vector SEGMENT signed halfword load instruction	356
13.7.141VLSEGE<NF>HFF.V: a vector FOF SEGMENT signed halfword load instruction	357
13.7.142VLSEGE<NF>HU.V: a vector SEGMENT unsigned halfword load instruction	358
13.7.143VLSEGE<NF>HUFF.V: a vector FOF SEGMENT unsigned halfword load instruction	359
13.7.144VLSEGE<NF>W.V: a vector SEGMENT signed word load instruction	360
13.7.145VLSEGE<NF>WFF.V: a vector FOF SEGMENT signed word load instruction	361
13.7.146VLSEGE<NF>WU.V: a vector SEGMENT unsigned word load instruction	362
13.7.147VLSEGE<NF>WUFF.V: a vector FOF SEGMENT unsigned word load instruction	363
13.7.148VLSSEGE<NF>B.V: a vector strided SEGMENT signed byte load instruction	364
13.7.149VLSSEGE<NF>BU.V: a vector strided SEGMENT unsigned byte load instruction	365
13.7.150VLSSEGE<NF>E.V: a vector strided SEGMENT element load instruction	366
13.7.151VLSSEGE<NF>H.V: a vector strided SEGMENT signed halfword load instruction	367
13.7.152VLSSEGE<NF>HU.V: a vector strided SEGMENT unsigned halfword load instruction	368
13.7.153VLSSEGE<NF>W.V: a vector strided SEGMENT signed word load instruction	369
13.7.154VLSSEGE<NF>WU.V: a vector strided SEGMENT unsigned word load instruction	370
13.7.155VLSH.V: a vector strided signed halfword load instruction	371
13.7.156VLSHU.V: a vector strided unsigned halfword load instruction	372
13.7.157VLSW: a vector strided signed word load instruction	373
13.7.158VLSWU.V: a vector strided unsigned word load instruction	373
13.7.159VLW.V: a vector signed word load instruction	374
13.7.160VLWFF.V: a vector FOF signed word load instruction	375
13.7.161VLWU.V: a vector unsigned word load instruction	376
13.7.162VLWUFF.V: a vector FOF unsigned word load instruction	376
13.7.163VLXB.V: a vector indexed signed byte load instruction	377
13.7.164VLXBU.V: a vector indexed unsigned byte load instruction	378
13.7.165VLXE.V: a vector indexed element load instruction	378
13.7.166VLXH.V: a vector indexed signed halfword load instruction	379
13.7.167VLXHU.V: a vector indexed unsigned halfword load instruction	380
13.7.168VLXSEGE<NF>B.V: a vector indexed SEGMENT signed byte load instruction	381
13.7.169VLXSEGE<NF>BU.V: a vector indexed SEGMENT unsigned byte load instruction	382
13.7.170VLXSEGE<NF>E.V: a vector indexed SEGMENT element load instruction	383
13.7.171VLXSEGE<NF>H.V: a vector indexed SEGMENT signed halfword load instruction	384
13.7.172VLXSEGE<NF>HU.V: a vector indexed SEGMENT unsigned halfword load instruction	385
13.7.173VLXSEGE<NF>W.V: a vector indexed SEGMENT signed word load instruction	386
13.7.174VLXSEGE<NF>WU.V: a vector indexed SEGMENT unsigned word load instruction	387
13.7.175VLXW.V: a vector indexed signed word load instruction	388
13.7.176VLXWU.V: a vector indexed unsigned word load instruction	388
13.7.177VMACC.VV: a vector lower-bit multiply-add instruction that overwrites addends	389
13.7.178VMACC.VX: a vector-scalar lower-bit multiply-add instruction that overwrites addends	390
13.7.179VMADC.VVM: a vector integer add-with-carry instruction that produces the carry out	390

13.7.180VMADC.VXM: a vector-scalar integer add-with-carry instruction that produces the carry out	391
13.7.181VMADC.VIM: a vector-immediate integer add-with-carry instruction that produces the carry out	391
13.7.182VMADD.VV: a vector lower-bit multiply-add instruction that overwrites multiplicands	392
13.7.183VMADD.VX: a vector-scalar lower-bit multiply-add instruction that overwrites multiplicands	393
13.7.184VMAND.MM: a vector mask AND instruction	393
13.7.185VMANDNOT.MM: a vector mask AND NOT instruction	394
13.7.186VMAX.VV: a vector signed integer MAX instruction	394
13.7.187VMAX.VX: a vector-scalar signed integer MAX instruction	395
13.7.188VMAXU.VV: a vector unsigned integer MAX instruction	396
13.7.189VMAXU.VX: a vector-scalar unsigned integer MAX instruction	396
13.7.190VMERGE.VVM: a vector element select instruction	397
13.7.191VMERGE.VXM: a vector-scalar element select instruction	397
13.7.192VMERGE.VIM: a vector-immediate element select instruction	398
13.7.193VMFEQ.VF: a vector-scalar floating-point compare equal instruction	398
13.7.194VMFEQ.VV: a vector floating-point compare equal instruction	399
13.7.195VMFGE.VF: a vector-scalar floating-point compare greater than or equal to instruction	400
13.7.196VMFGT.VF: a vector-scalar floating-point compare greater than instruction	401
13.7.197VMFIRST: a vector mask find-first-set instruction	401
13.7.198VMFLE.VF: a vector-scalar floating-point compare less than or equal to instruction	402
13.7.199VMFLE.VV: a vector floating-point compare less than or equal to instruction	403
13.7.200VMFLT.VF: a vector-scalar floating-point compare less than instruction	404
13.7.201VMFLT.VV: a vector floating-point compare less than instruction	404
13.7.202VMFNE.VF: a vector-scalar floating-point compare not equal instruction	405
13.7.203VMFNE.VV: a vector floating-point compare not equal instruction	406
13.7.204VMFORD.VF: a vector-scalar floating-point Not a Number (NaN) check instruction	407
13.7.205VMFORD.VV: a vector floating-point NaN check instruction	407
13.7.206VMIN.VV: a vector signed integer MIN instruction	408
13.7.207VMIN.VX: a vector-scalar signed integer MIN instruction	409
13.7.208VMINU.VV: a vector unsigned integer MIN instruction	409
13.7.209VMINU.VX: a vector-scalar unsigned integer MIN instruction	410
13.7.210VMNAND.MM: a vector mask NOT AND instruction	411
13.7.211VMNOR.MM: a vector mask NOT OR instruction	411
13.7.212VMOR.MM: a vector mask OR instruction	412
13.7.213VMORNOR.MM: a vector mask OR NOT instruction	412
13.7.214VMPOPC.M: a vector mask population count instruction	413
13.7.215VMSBC.VVM: a vector integer subtract-with-borrow instruction that produces the borrow out	414
13.7.216VMSBC.VXM: a vector-scalar integer subtract-with-borrow instruction that produces the borrow out	414

13.7.217VMSBF.M: a vector mask set-before-first instruction	415
13.7.218VMSIF.M: a vector mask set-including-first instruction	415
13.7.219VMSOF.M: a vector mask set-only-first instruction	416
13.7.220VMSEQ.VV: a vector integer compare equal instruction	417
13.7.221VMSEQ.VX: a vector-scalar integer compare equal instruction	418
13.7.222VMSEQ.VI: a vector-immediate integer compare equal instruction	418
13.7.223VMSGT.VX: a vector-scalar signed integer compare greater than instruction	419
13.7.224VMSGT.VI: a vector-immediate signed integer compare greater than instruction	420
13.7.225VMSGTU.VX: a vector-scalar unsigned integer compare greater than instruction	421
13.7.226VMSGTU.VI: a vector-immediate unsigned integer compare greater than instruction	421
13.7.227VMSLE.VV: a vector signed integer compare less than or equal to instruction	422
13.7.228VMSLE.VX: a vector-scalar signed integer compare less than or equal to instruction	423
13.7.229VMSLE.VI: a vector-immediate signed integer compare less than or equal to instruction	424
13.7.230VMSLEU.VV: a vector unsigned integer compare less than or equal to instruction	424
13.7.231VMSLEU.VX: a vector-scalar unsigned integer compare less than or equal to instruction	425
13.7.232VMSLEU.VI: a vector-immediate unsigned integer compare less than or equal to instruction	426
13.7.233VMSLT.VV: a vector signed integer compare less than instruction	427
13.7.234VMSLT.VX: a vector-scalar signed integer compare less than instruction	427
13.7.235VMSLTU.VV: a vector unsigned integer compare less than instruction	428
13.7.236VMSLTU.VX: a vector-scalar unsigned integer compare less than instruction	429
13.7.237VMSNE.VV: a vector integer compare not equal instruction	430
13.7.238VMSNE.VX: a vector-scalar integer compare not equal instruction	430
13.7.239VMSNE.VI: a vector-immediate integer compare not equal instruction	431
13.7.240VMUL.VV: a vector integer multiply instruction that returns lower bits	432
13.7.241VMUL.VX: a vector-scalar integer multiply instruction that returns lower bits	432
13.7.242VMULH.VV: a vector signed integer multiply instruction that returns upper bits	433
13.7.243VMULH.VX: a vector-scalar signed integer multiply instruction that returns upper bits	434
13.7.244VMULHU.VV: a vector unsigned integer multiply instruction that returns upper bits	434
13.7.245VMULHU.VX: a vector-scalar unsigned integer multiply instruction that returns upper bits	435
13.7.246VMULHSU.VV: a vector signed-unsigned integer multiply instruction that returns upper bits	436
13.7.247VMULHSU.VX: a vector-scalar signed-unsigned integer multiply instruction that returns upper bits	436
13.7.248VMV.V.V: a vector element move instruction	437
13.7.249VMV.V.X: an instruction that moves an integer scalar to a vector	438
13.7.250VMV.V.I: an instruction that moves an immediate to a vector	438
13.7.251VMV.S.X: an instruction that moves an integer scalar to element 0 of a vector	439
13.7.252VMXOR.MM: a vector mask XOR instruction	439
13.7.253VMXNOR.MM: a vector mask XNOR instruction	440

13.7.254VNCLIP.VV: a vector narrowing signed arithmetic right shift instruction with result saturated when necessary	440
13.7.255VNCLIP.VX: a vector-scalar narrowing signed arithmetic right shift instruction with result saturated when necessary	441
13.7.256VNCLIP.VI: a vector-immediate narrowing signed arithmetic right shift instruction with result saturated when necessary	442
13.7.257VNCLIPU.VV: a vector narrowing unsigned arithmetic right shift instruction with result saturated when necessary	443
13.7.258VNCLIPU.VX: a vector narrowing unsigned arithmetic right shift instruction with result saturated when necessary	444
13.7.259VNCLIPU.VI: a vector-immediate narrowing unsigned arithmetic right shift instruction with result saturated when necessary	444
13.7.260VNMSAC.VV: a vector lower-bit multiply-sub instruction that overwrites minuends	445
13.7.261VNMSAC.VX: a vector-scalar lower-bit multiply-sub instruction that overwrites minuends	446
13.7.262VNMSUB.VV: a vector lower-bit negate-(multiply-sub) instruction that overwrites multiplicands	447
13.7.263VNMSUB.VX: a vector-scalar lower-bit negate-(multiply-sub) instruction that overwrites multiplicands	447
13.7.264VNSRA.VV: a vector narrowing arithmetic right shift instruction	448
13.7.265VNSRA.VX: a vector-scalar narrowing arithmetic right shift instruction	449
13.7.266VNSRA.VI: a vector-immediate narrowing arithmetic right shift instruction	449
13.7.267VNSRL.VV: a vector narrowing logical right shift instruction	450
13.7.268VNSRL.VX: a vector-scalar narrowing logical right shift instruction	450
13.7.269VNSRL.VI: a vector-immediate narrowing logical right shift instruction	451
13.7.270VOR.VV: a vector bitwise OR instruction	452
13.7.271VOR.VX: a vector-scalar bitwise OR instruction	452
13.7.272VOR.VI: a vector-immediate bitwise OR instruction	453
13.7.273VREDAND.VS: a vector reduction bitwise AND instruction	454
13.7.274VREDMAX.VS: a vector reduction signed MAX instruction	454
13.7.275VREDMAXU.VS: a vector reduction unsigned MAX instruction	455
13.7.276VREDMIN.VS: a vector reduction signed MIN instruction	456
13.7.277VREDMINU.VS: a vector reduction unsigned MIN instruction	457
13.7.278VREDOR.VS: a vector reduction bitwise OR instruction	458
13.7.279VREDXOR.VS: a vector reduction bitwise XOR instruction	459
13.7.280VREDSUM.VS: a vector reduction sum instruction	459
13.7.281VREM.VV: a vector signed integer remainder instruction	460
13.7.282VREM.VX: a vector-scalar signed integer remainder instruction	461
13.7.283VREMU.VV: a vector unsigned integer remainder instruction	461
13.7.284VREMU.VX: a vector-scalar unsigned integer remainder instruction	462
13.7.285VRGATHER.VV: a vector index-based gather instruction for integer elements	463
13.7.286VRGATHER.VX: a vector-scalar index-based gather instruction for integer elements	463

13.7.287VRGATHER.VI: a vector-immediate index-based gather instruction for integer elements	464
13.7.288VRSUB.VX: a vector-scalar integer subtract instruction	465
13.7.289VRSUB.VI: an immediate-vector integer subtract instruction	465
13.7.290VSADD.VV: a vector saturating signed integer add instruction	466
13.7.291VSADD.VX: a vector-scalar saturating signed integer add instruction	467
13.7.292VSADD.VI: a vector-immediate saturating signed integer add instruction	467
13.7.293VSADDU.VV: a vector saturating unsigned integer add instruction	468
13.7.294VSADDU.VX: a vector-scalar saturating unsigned integer add instruction	468
13.7.295VSADDU.VI: a vector-immediate saturating unsigned integer add instruction	469
13.7.296VSB.V: a vector byte store instruction	470
13.7.297VSBC.VVM: a vector integer subtract-with-borrow instruction	470
13.7.298VSBC.VXM: a vector-scalar integer subtract-with-borrow instruction	471
13.7.299VSE.V: a vector element store instruction	471
13.7.300VSETVL: an instruction that sets vtype and vl CSRs	472
13.7.301VSETVLI: an instruction that sets vl and vtype with immediate values	473
13.7.302VSH.V: a vector halfword store instruction	473
13.7.303VSLIDEDOWN.VX: a vector slide instruction that moves elements down	474
13.7.304VSLIDEDOWN.VI: a vector-immediate slide instruction that moves elements down	475
13.7.305VSLIDE1DOWN.VX: a vector slide instruction that moves elements down by 1 index	475
13.7.306VSLIDEUP.VX: a vector slide instruction that moves elements up	476
13.7.307VSLIDEUP.VI: a vector-immediate slide instruction that moves elements up	477
13.7.308VSLIDE1UP.VX: a vector slide instruction that moves elements up by 1 index	478
13.7.309VSLL.VV: a vector logical left shift instruction	478
13.7.310VSLL.VX: a vector-scalar logical left shift instruction	479
13.7.311VSLL.VI: a vector-immediate logical left shift instruction	480
13.7.312VSMUL.VV: a vector saturating multiply instruction	480
13.7.313VSMUL.VX: a vector-scalar saturating multiply instruction	481
13.7.314VSRA.VV: a vector arithmetic right shift instruction	482
13.7.315VSRA.VX: a vector-scalar arithmetic right shift instruction	482
13.7.316VSRA.VI: a vector-immediate arithmetic right shift instruction	483
13.7.317VSRL.VV: a vector logical right shift instruction	484
13.7.318VSRL.VX: a vector-scalar logical right shift instruction	484
13.7.319VSRL.VI: a vector-immediate logical right shift instruction	485
13.7.320VSSB.V: a vector strided byte store instruction	486
13.7.321VSSE.V: a vector strided element store instruction	486
13.7.322VSSEG<NF>B.V: a vector SEGMENT byte store instruction	487
13.7.323VSSEG<NF>E.V: a vector SEGMENT element store instruction	488
13.7.324VSSEG<NF>H.V: a vector SEGMENT halfword store instruction	489
13.7.325VSSEG<NF>W.V: a vector SEGMENT word store instruction	490
13.7.326VSSSEG<NF>B.V: a vector strided SEGMENT byte store instruction	491
13.7.327VSSSEG<NF>E.V: a vector strided SEGMENT element store instruction	492

13.7.328VSSSEG<NF>H.V: a vector strided SEGMENT halfword store instruction	493
13.7.329VSSSEG<NF>W.V: a vector strided SEGMENT word store instruction	494
13.7.330VSXSEG<NF>B.V: a vector indexed SEGMENT byte store instruction	495
13.7.331VSXSEG<NF>E.V: a vector indexed SEGMENT element store instruction	496
13.7.332VSXSEG<NF>H.V: a vector indexed SEGMENT halfword store instruction	497
13.7.333VSXSEG<NF>W.V: a vector indexed SEGMENT word store instruction	498
13.7.334VSSH.V: a vector strided halfword store instruction	499
13.7.335VSSRA.VV: a vector scaling arithmetic right shift instruction	499
13.7.336VSSRA.VX: a vector-scalar scaling arithmetic right shift instruction	500
13.7.337VSSRA.VI: a vector-immediate scaling arithmetic right shift instruction	501
13.7.338VSSRL.VV: a vector scaling logical right shift instruction	502
13.7.339VSSRL.VX: a vector-scalar scaling logical right shift instruction	502
13.7.340VSSRL.VI: a vector-immediate scaling logical right shift instruction	503
13.7.341VSSUB.VV: a vector saturating subtract instruction for signed integers	504
13.7.342VSSUB.VX: a vector-scalar saturating subtract instruction for signed integers	505
13.7.343VSSUBU.VV: a vector saturating unsigned-integer subtract instruction	505
13.7.344VSSUBU.VX: a vector-scalar saturating unsigned-integer subtract instruction	506
13.7.345VSSW.V: a vector strided word store instruction	506
13.7.346VSUB.VV: a vector integer subtract instruction	507
13.7.347VSUB.VX: a vector-scalar integer subtract instruction	508
13.7.348VSUXB.V: a vector unordered-indexed byte store instruction	508
13.7.349VSUXE.V: a vector unordered-indexed element store instruction	509
13.7.350VSUXH.V: a vector unordered-indexed halfword store instruction	510
13.7.351VSUXW.V: a vector unordered-indexed word store instruction	511
13.7.352VSW.V: a vector word store instruction	511
13.7.353VSXB.V: a vector ordered-indexed byte store instruction	512
13.7.354VSXE.V: a vector ordered-indexed element store instruction	513
13.7.355VSXH.V: a vector ordered-indexed halfword store instruction	513
13.7.356VSXW.V: a vector ordered-indexed word store instruction	514
13.7.357VWADD.VV: a vector widening signed-integer add instruction	515
13.7.358VWADD.VX: a vector-scalar widening signed-integer add instruction	515
13.7.359VWADDU.VV: a vector widening unsigned-integer add instruction	516
13.7.360VWADDU.VX: a vector-scalar widening unsigned-integer add instruction	517
13.7.361VWADD.WV: a widening vector widening signed-integer add instruction	517
13.7.362VWADD.WX: a widening vector-scalar widening signed-integer add instruction	518
13.7.363VWADDU.WV: a widening vector widening unsigned-integer add instruction	519
13.7.364VWADDU.WX: a widening vector-scalar widening unsigned-integer add instruction	520
13.7.365VWMAcc.VV: a vector widening signed-integer multiply-add instruction that over- writes addends	520
13.7.366VWMAcc.VX: a vector-scalar widening signed-integer multiply-add instruction that overwrites addends	521

13.7.367VWMACCSU.VV: a vector widening signed-unsigned-integer multiply-add instruction that overwrites addends	522
13.7.368VWMACCSU.VX: a vector-scalar widening signed-unsigned-integer multiply-add instruction that overwrites addends	522
13.7.369VWMACCU.VV: a vector widening unsigned-integer multiply-add instruction that overwrites addends	523
13.7.370VWMACCU.VX: a vector-scalar widening unsigned-integer multiply-add instruction that overwrites addends	524
13.7.371VWMACCUS.VX: a vector-scalar widening unsigned-signed-integer multiply-add instruction that overwrites addends	525
13.7.372VWMUL.VV: a vector widening signed-integer multiply instruction	525
13.7.373VWMUL.VX: a vector-scalar widening signed-integer multiply instruction	526
13.7.374VWMULSU.VV: a vector widening signed-unsigned integer multiply instruction	527
13.7.375VWMULSU.VX: a vector-scalar widening signed-unsigned integer multiply instruction	527
13.7.376VWMULU.VV: a vector widening unsigned integer multiply instruction	528
13.7.377VWMULU.VX: a vector-scalar widening unsigned integer multiply instruction	529
13.7.378VWREDSUM.VS: a vector widening reduction instruction that sign-extends vector elements before summing them	529
13.7.379VWREDSUMU.VS: a vector widening reduction instruction that unsign-extends vector elements before summing them	530
13.7.380VWSMACC.VV: a vector widening signed-integer saturating scaled multiply-add instruction	531
13.7.381VWSMACC.VX: a vector-scalar widening signed-integer saturating scaled multiply-add instruction	532
13.7.382VWSMACCSU.VV: a vector widening signed-unsigned-integer saturating scaled negate-(multiply-sub) instruction	533
13.7.383VWSMACCSU.VX: a vector-scalar widening signed-unsigned-integer saturating scaled negate-(multiply-sub) instruction	534
13.7.384VWSMACCU.VV: a vector widening unsigned-integer saturating scaled multiply-add instruction	535
13.7.385VWSMACCU.VX: a vector-scalar widening unsigned-integer saturating scaled multiply-add instruction	536
13.7.386VWSMACCUS.VX: a vector-scalar widening unsigned-signed-integer saturating scaled multiply-sub instruction	536
13.7.387VWSUB.VV: a vector widening signed-integer subtract instruction	537
13.7.388VWSUB.VX: a vector-scalar widening signed-integer subtract instruction	538
13.7.389VWSUB.WV: a widening vector widening signed-integer subtract instruction	539
13.7.390VWSUB.WX: a widening vector-scalar widening signed-integer subtract instruction	539
13.7.391VWSUBU.VV: a vector widening unsigned-integer subtract instruction	540
13.7.392VWSUBU.VX: a vector-scalar widening unsigned-integer subtract instruction	541
13.7.393VWSUBU.WV: a widening vector widening unsigned-integer subtract instruction	541
13.7.394VWSUBU.WX: a widening vector-scalar widening unsigned-integer subtract instruction	542

13.7.395VXOR.VV: a vector bitwise XOR instruction	543
13.7.396VXOR.VX: a vector-scalar bitwise XOR instruction	543
13.7.397VXOR.VI: a vector-immediate bitwise XOR instruction	544
13.8 Appendix A-8 Pseudo instructions	545
14 Appendix B T-Head extended instructions	548
14.1 Appendix B-1 Cache instructions	548
14.1.1 DCACHE.CALL: an instruction that clears all dirty page table entries in the D-Cache.	548
14.1.2 DCACHE.CVA: an instruction that clears dirty page table entries in the D-Cache that match the specified virtual address.	549
14.1.3 DCACHE.CPA: an instruction that clears dirty page table entries in the D-Cache that match the specified physical address.	550
14.1.4 DCACHE.CSW: an instruction that clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache.	550
14.1.5 DCACHE.IALL: an instruction that invalidates all page table entries in the D-Cache.	551
14.1.6 DCACHE.IVA: an instruction that invalidates page table entries in the D-Cache that match the specified virtual address.	552
14.1.7 DCACHE.IPA: an instruction that invalidates page table entries in the D-Cache that match the specified physical addresses.	552
14.1.8 DCACHE.ISW: an instruction that invalidates page table entries in the D-Cache based on the specified way and set.	553
14.1.9 DCACHE.CIALL: an instruction that clears all dirty page table entries in the D-Cache and invalidates the D-Cache.	553
14.1.10 DCACHE.CIVA: an instruction that clears dirty page table entries in the D-Cache that match the specified virtual address and invalidates the D-Cache.	554
14.1.11 DCACHE.CIPA: an instruction that clears dirty page table entries in the D-Cache that match the specified physical address and invalidates the D-Cache.	555
14.1.12 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.	555
14.1.13 ICACHE.IALL: an instruction that invalidates all page table entries in the I-Cache. .	556
14.1.14 ICACHE.IALLS: an instruction that invalidates all page table entries in the I-Cache through broadcasting.	557
14.1.15 ICACHE.IVA: an instruction that invalidates page table entries in the I-Cache that match the specified virtual address.	557
14.1.16 ICACHE.IPA: an instruction that invalidates page table entries in the I-Cache that match the specified physical address.	558
14.2 Appendix B-2 Multi-core synchronization instructions	559
14.2.1 SYNC: an instruction that performs the synchronization operation	559
14.2.2 SYNC.I: an instruction that synchronizes the clearing operation.	559
14.3 Appendix B-3 Arithmetic operation instructions	560
14.3.1 ADDSL: an add register instruction that shifts registers	560
14.3.2 MULA: a multiply-add instruction	560

14.3.3	MULAH: a multiply-add instruction that operates on the lower 16 bits	561
14.3.4	MULAW: a multiply-add instruction that operates on the lower 32 bits	561
14.3.5	MULS: a multiply-subtract instruction	562
14.3.6	MULSH: a multiply-subtract instruction that operates on the lower 16 bits	562
14.3.7	MULSW: a multiply-subtract instruction that operates on the lower 32 bits	563
14.3.8	MVEQZ: an instruction that sends a message when the register is 0	563
14.3.9	MVNEZ: an instruction that sends a message when the register is not 0	564
14.3.10	SRRI: an instruction that implements a cyclic right shift operation on a linked list .	564
14.3.11	SRRIW: an instruction that implements a cyclic right shift operation on a linked list of low 32 bits of registers.	565
14.4	Appendix B-4 Bitwise operation instructions	565
14.4.1	EXT: a signed extension instruction that extracts consecutive bits of a register . . .	565
14.4.2	EXTU: a zero extension instruction that extracts consecutive bits of a register . . .	566
14.4.3	FF0: an instruction that finds the first bit with the value of 0 in a register	566
14.4.4	FF1: an instruction that finds the bit with the value of 1	567
14.4.5	REV: an instruction that reverses the byte order in a word stored in the register . .	567
14.4.6	RE VW: an instruction that reverses the byte order in a low 32-bit word	568
14.4.7	TST: an instruction that tests bits with the value of 0	568
14.4.8	TSTNBZ: an instruction that tests bytes with the value of 0	569
14.5	Appendix B-5 Storage instructions	570
14.5.1	FLRD: a load doubleword instruction that shifts floating-point registers	570
14.5.2	FLRW: a load word instruction that shifts floating-point registers	570
14.5.3	FLURD: a load doubleword instruction that shifts low 32 bits of floating-point registers	571
14.5.4	FLURW: a load word instruction that shifts low 32 bits of floating-point registers . .	571
14.5.5	FSRD: a store doubleword instruction that shifts floating-point registers	572
14.5.6	FSRW: a store word instruction that shifts floating-point registers.	572
14.5.7	FSURD: a store doubleword instruction that shifts low 32 bits of floating-point registers	573
14.5.8	FSURW: a store word instruction that shifts low 32 bits of floating-point registers . .	573
14.5.9	LBIA: a base-address auto-increment instruction that extends signed bits and loads bytes	574
14.5.10	LBIB: a load byte instruction that auto-increments the base address and extends signed bits	575
14.5.11	LBUIA: a base-address auto-increment instruction that extends zero bits and loads bytes	575
14.5.12	LBUIB: a load byte instruction that auto-increments the base address and extends zero bits	576
14.5.13	LDD: an instruction that loads double registers	576
14.5.14	LDIA: a base-address auto-increment instruction that loads doublewords and extends signed bits	577
14.5.15	LDIB: a load doubleword instruction that auto-increments the base address and ex- tends the signed bits	577

14.5.16	LHIA: a base-address auto-increment instruction that loads halfwords and extends signed bits	578
14.5.17	LHIB: a load halfword instruction that auto-increments the base address and extends signed bits	579
14.5.18	LHUIA: a base-address auto-increment instruction that extends zero bits and loads halfwords	579
14.5.19	LHUIB: a load halfword instruction that auto-increments the base address and extends zero bits	580
14.5.20	LRB: a load byte instruction that shifts registers and extends signed bits	580
14.5.21	LRBU: a load byte instruction that shifts registers and extends zero bits	581
14.5.22	LRD: a load doubleword instruction that shifts registers	581
14.5.23	LRH: a load halfword instruction that shifts registers and extends signed bits	581
14.5.24	LRHU: a load halfword instruction that shifts registers and extends zero bits	582
14.5.25	LRW: a load word instruction that shifts registers and extends signed bits	582
14.5.26	LRWU: a load word instruction that shifts registers and extends zero bits	583
14.5.27	LURB: a load byte instruction that shifts low 32 bits of registers and extends signed bits	583
14.5.28	LURBU: a load byte instruction that shifts low 32 bits of registers and extends zero bits	584
14.5.29	LURD: a load doubleword instruction that shifts low 32 bits of registers	584
14.5.30	LURH: a load halfword instruction that shifts low 32 bits of registers and extends signed bits	585
14.5.31	LURHU: a load halfword instruction that shifts low 32 bits of registers and extends zero bits	585
14.5.32	LURW: a load word instruction that shifts low 32 bits of registers and extends signed bits	586
14.5.33	LURWU: a load word instruction that shifts 32 bits of registers and extends zero bits	586
14.5.34	LWD: a load word instruction that loads double registers and extends signed bits	587
14.5.35	LWIA: a base-address auto-increment instruction that extends signed bits and loads words	588
14.5.36	LWIB: a load word instruction that auto-increments the base address and extends signed bits	588
14.5.37	LWUD: a load word instruction that loads double registers and extends zero bits	589
14.5.38	LWUIA: a base-address auto-increment instruction that extends zero bits and loads words	589
14.5.39	LWUIB: a load word instruction that auto-increments the base address and extends zero bits	590
14.5.40	SBIA: a base-address auto-increment instruction that stores bytes	590
14.5.41	SBIB: a store byte instruction that auto-increments the base address	591
14.5.42	SDD: an instruction that stores double registers	591
14.5.43	SDIA: a base-address auto-increment instruction that stores doublewords	592
14.5.44	SDIB: a store doubleword instruction that auto-increments the base address	592

14.5.45	SHIA: a base-address auto-increment instruction that stores halfwords	593
14.5.46	SHIB: a store halfword instruction that auto-increments the base address	593
14.5.47	SRB: a store byte instruction that shifts registers	594
14.5.48	SRD: a store doubleword instruction that shifts registers	594
14.5.49	SRH: a store halfword instruction that shifts registers	594
14.5.50	SRW: a store word instruction that shifts registers	595
14.5.51	SURB: a store byte instruction that shifts low 32 bits of registers	595
14.5.52	SURD: a store doubleword instruction that shifts low 32 bits of registers	596
14.5.53	SURH: a store halfword instruction that shifts low 32 bits of registers	596
14.5.54	SURW: a store word instruction that shifts low 32 bits of registers	597
14.5.55	SWIA: a base-address auto-increment instruction that stores words	597
14.5.56	SWIB: a store word instruction that auto-increments the base address	598
14.5.57	SWD: an instruction that stores the low 32 bits of double registers	598
14.6	Appendix B-6 Half-precision floating-point instructions	599
14.6.1	FADD.H: a half-precision floating-point add instruction	599
14.6.2	FCLASS.H: a half-precision floating-point classification instruction	600
14.6.3	FCVT.D.H: an instruction that converts half-precision floating-point data to double-precision floating-point data	601
14.6.4	FCVT.H.D: an instruction that converts double-precision floating-point data to half-precision floating-point data	601
14.6.5	FCVT.H.L: an instruction that converts a signed long integer into a half-precision floating-point number	602
14.6.6	FCVT.H.LU: an instruction that converts an unsigned long integer into a half-precision floating-point number	603
14.6.7	FCVT.H.S: an instruction that converts single precision floating-point data to half-precision floating-point data	604
14.6.8	FCVT.H.W: an instruction that converts a signed integer into a half-precision floating-point number	605
14.6.9	FCVT.H.WU: an instruction that converts an unsigned integer into a half-precision floating-point number	606
14.6.10	FCVT.L.H: an instruction that converts a half-precision floating-point number to a signed long integer	607
14.6.11	FCVT.LU.H: an instruction that converts a half-precision floating-point number to an unsigned long integer	608
14.6.12	FCVT.S.H: an instruction that converts half-precision floating-point data to single precision floating-point data	609
14.6.13	FCVT.W.H: an instruction that converts a half-precision floating-point number to a signed integer	610
14.6.14	FCVT.WU.H: an instruction that converts a half-precision floating-point number to an unsigned integer	611
14.6.15	FDIV.H: a half-precision floating-point division instruction	612
14.6.16	FEQ.H: an equal instruction that compares two half-precision numbers	613

14.6.17	FLE.H: a less than or equal to instruction that compares two half-precision floating-point numbers	613
14.6.18	FLH: an instruction that loads half-precision floating-point data	614
14.6.19	FLT.H: a less than instruction that compares two half-precision floating-point numbers	615
14.6.20	FMADD.H: a half-precision floating-point multiply-add instruction	615
14.6.21	FMAX.H: a half-precision floating-point maximum instruction	616
14.6.22	FMIN.H: a half-precision floating-point minimum instruction	617
14.6.23	FMSUB.H: a half-precision floating-point multiply-subtract instruction	617
14.6.24	FMUL.H: a half-precision floating-point multiply instruction	618
14.6.25	FMV.H.X: a half-precision floating-point write transmit instruction	619
14.6.26	FMV.X.H: a transmission instruction that reads half-precision floating-point registers	620
14.6.27	FNMADD.H: a half-precision floating-point negate-(multiply-add) instruction	620
14.6.28	FNMSUB.H: a half-precision floating-point negate-(multiply-subtract) instruction . .	621
14.6.29	FSGNJ.H: a half-precision floating-point sign-injection instruction	622
14.6.30	FSGNJN.H: a half-precision floating-point sign-injection negate instruction	623
14.6.31	FSGNJX.H: a half-precision floating-point sign-injection XOR instruction	623
14.6.32	FSH: an instruction that stores half-precision floating point numbers	624
14.6.33	FSQRT.H: an instruction that calculates the square root of the half-precision floating-point number	624
14.6.34	FSUB.H: a half-precision floating-point subtract instruction	625
15	Appendix C Control registers	627
15.1	Appendix C-1 M-mode control registers	627
15.1.1	M-mode information register group	627
15.1.1.1	Machine vendor ID register (MVENDORID)	627
15.1.1.2	Machine architecture ID register (MARCHID)	627
15.1.1.3	Machine implementation ID register (MIMPID)	628
15.1.1.4	Machine hart ID register (MHARTID)	628
15.1.2	M-mode exception configuration register group	628
15.1.2.1	Machine status register (MSTATUS)	628
15.1.2.2	M-mode instruction set architecture register (MISA)	631
15.1.2.3	M-mode exception downgrade control register (MEDELEG)	632
15.1.2.4	M-mode interrupt downgrade control register (MIDELEG)	632
15.1.2.5	M-mode interrupt-enable register (MIE)	633
15.1.2.6	M-mode vector base address register (MTVEC)	634
15.1.2.7	MCOUNTEREN register	635
15.1.3	M-mode exception handling register group	635
15.1.3.1	M-mode scratch register (MSCRATCH)	635
15.1.3.2	M-mode exception program counter register (MEPC)	635
15.1.3.3	M-mode cause register (MCAUSE)	636
15.1.3.4	M-mode interrupt pending register (MIP)	636
15.1.4	M-mode memory protection register group	638

15.1.4.1	Physical memory protection configuration register (PMPCFG)	638
15.1.4.2	Physical memory address register (PMPADDR)	638
15.1.5	M-mode counter register group	638
15.1.5.1	M-mode cycle counter (MCYCLE)	638
15.1.5.2	M-mode instructions-retired counter (MINSTRET)	639
15.1.5.3	M-mode event counter (MHPMCOUNTERn)	639
15.1.6	M-mode counter configuration register group	639
15.1.6.1	M-mode event selector (MHPMEVENTn)	639
15.1.7	M-mode CPU control and status extension register group	639
15.1.7.1	M-mode extension status register (MXSTATUS)	639
15.1.7.2	M-mode hardware configuration register (MHCR)	642
15.1.7.3	M-mode hardware operation register (MCOR)	643
15.1.7.4	M-mode implicit operation register (MHINT)	644
15.1.7.5	M-mode reset vector base address register (MRVBR)	646
15.1.7.6	MCOUNTERWEN register	646
15.1.7.7	M-mode event interrupt enable register (MCOUNTERINTEN)	647
15.1.7.8	M-mode event counter overflow mark register (MCOUNTEROF)	647
15.1.7.9	M-mode device address upper bit register (MAPBADDR)	647
15.1.8	M-mode cache access extension register group	647
15.1.8.1	M-mode cache instruction register (MCINS)	647
15.1.8.2	M-mode cache access index register (MCINDEX)	648
15.1.8.3	M-mode cache data register (MCDATA0/1)	649
15.1.9	M-mode CPU model register group	649
15.1.9.1	M-mode CPU model register (MCPUID)	650
15.2	Appendix C-2 S-mode control registers	650
15.2.1	S-mode exception configuration registers	651
15.2.1.1	S-mode status register (SSTATUS)	651
15.2.1.2	S-mode interrupt enable register (SIE)	651
15.2.1.3	S-mode vector base address register (STVEC)	651
15.2.1.4	S-mode counter access enable register (SCOUNTEREN)	652
15.2.2	S-mode exception handling registers	652
15.2.2.1	S-mode scratch register (SSCRATCH)	652
15.2.2.2	S-mode exception program counter register (SEPC)	653
15.2.2.3	S-mode cause register (SCAUSE)	653
15.2.2.4	S-mode interrupt-pending register (SIP)	653
15.2.2.5	S-mode event cause register (STVAL)	653
15.2.3	S-mode address translation registers	654
15.2.3.1	S-mode address translation register (SATP)	654
15.2.4	Extended S-mode CPU control and status registers	654
15.2.4.1	Extended S-mode status register (SXSTATUS)	654
15.2.4.2	S-mode hardware control register (SHCR)	654
15.2.4.3	S-mode event overflow interrupt enable register (SCOUNTERINTEN)	654

15.2.4.4	S-mode event overflow mark register (SCOUNTEROF)	655
15.2.4.5	S-mode cycle counter (SCYCLE)	655
15.2.4.6	S-mode instructions-retired counter (SINSTRET)	655
15.2.4.7	S-mode event counter (SHPMCOUNTERn)	655
15.2.5	Extended S-mode MMU registers	655
15.2.5.1	S-mode MMU control register (SMCIR)	656
15.2.5.2	S-mode MMU control register (SMIR)	656
15.2.5.3	S-mode MMU control register (SMEH)	656
15.2.5.4	S-mode MMU control register (SMEL)	656
15.3	Appendix C-3 U-mode control registers	656
15.3.1	U-mode floating-point control registers	656
15.3.1.1	Floating-point accrued exceptions register (FFLAGS)	656
15.3.1.2	Floating-point dynamic rounding mode register (FRM)	657
15.3.1.3	Floating-point control and status register (FCSR)	657
15.3.2	U-mode event counter registers	658
15.3.2.1	User-mode cycle register (CYCLE)	658
15.3.2.2	U-mode timer register (TIME)	659
15.3.2.3	U-mode instructions-retired counter (INSTRET)	659
15.3.2.4	U-mode event counter (HPMCOUNTERn)	659
15.3.3	Extended U-mode floating-point control registers	659
15.3.3.1	Extended U-mode floating-point control register (FXCR)	659
15.3.4	Extended vector registers	660
15.3.4.1	Vector start position register (VSTART)	660
15.3.4.2	Fixed-point overflow flag bit register (VXSAT)	661
15.3.4.3	Fixed-point rounding mode register (VXRM)	661
15.3.4.4	Vector length register (VL)	661
15.3.4.5	Vector data type register (VTYPE)	661
15.3.4.6	Vector register width (unit: byte) register (VLENB)	663
16	Program examples	664
16.1	MMU setting example	664
16.2	PMP setting example	668
16.3	Cache setting example	669
16.3.1	Cache enabling example	669
16.3.2	Example of synchronization between the instruction and data caches	670
16.3.3	Example of synchronization between the TLB and the data cache	670
16.4	PLIC setting example	670
16.5	HPM setting example	671
16.6	CPU power-off software process setting example	672

1.1 Introduction

C906 is an ultra high-performance 64-bit CPU built on the RISC-V instruction set architecture, mainly designed for security surveillance, smart speakers, and payment using QR code scanning or face swiping.

1.2 Features

Architectural features of C906

- RV64IMAFDC instruction set architecture,
- 5-stage single-issue in-order execution pipeline,
- L1 instruction cache (I-Cache) and data cache (D-Cache) running on the Harvard architecture with a size of 32 KB and cache line size of 64 bytes,
- Sv39 memory management unit for virtual/physical address translation and memory management,
- AXI4.0 128-bit master interface supported,
- Core local interrupt (CLINT) controller and platform-level interrupt controller (PLIC) supported,
- RISC-V debug standard supported.

1.3 Configuration options

Table 1.1 describes configuration options of OpenC906.

Table 1.1: OpenC906 configuration options

Configurable unit	Configuration option	Description
Floating-point unit (FPU)	Half-, single-, and double-precision floating point	Half/Single/Double-precision FPUs can be configured as a whole.
L1 I-Cache	32 KB	OpenC906 provides a 32 KB I-Cache.
L1 D-Cache	32 KB	OpenC906 provides a 32 KB D-Cache.
MMU entries	128	OpenC906 provides a jTLB of 128 entries.
PMP entries	8	OpenC906 provides a physical memory protection (PMP) unit of eight entries.
BHT	16 KB	The branch history table (BHT) of OpenC906 is 16 KB.
Interrupts	240	OpenC906 supports up to 240 PLIC interrupts.
Debugging resources Configuration	Maximum	OpenC906 provides abundant debugging resources, including four Program buffers and debugging buses.

1.4 Naming conventions

1.4.1 Symbols

Fig. 1.1 shows symbols and operators used in this document.

1.4.2 Terms

Fig. 1.2 shows terms used in this document.

1.5 Release notes

C906 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 1.10.*
- The MCOUNTINHIBIT register in *RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 20190125-Public-Review-draft* is added to the hardware performance monitor (HPM).

符号	功能
+	加
-	减
*	乘
/	除
>	大于
<	小于
=	等于
≥	大于或等于
≤	小于或等于
≠	不等于
&	与
	或
⊕	异或
NOT	取反
:	连接
⇒	传输
↔	交换
±	误差
0b0011	二进制数
0x0F	十六进制数
rd	整型目的寄存器
rs1	整型源寄存器1
rs2	整型源寄存器2
rs3	整型源寄存器3
fd	浮点目的寄存器
fs1	浮点源寄存器1
fs2	浮点源寄存器2
fs3	浮点源寄存器3
vd	矢量目的寄存器
vs1	矢量源寄存器1
vs2	矢量源寄存器2
vs3	矢量源寄存器3

Fig. 1.1: Symbols

术语	描述
逻辑1	指对应于布尔逻辑真的电平值。
逻辑0	指对应于布尔逻辑伪的电平值。
置位	指使得某个或某几个位达到逻辑1 对应的电平值。
清除	指使得某个或某几个位达到逻辑0 对应的电平值。
保留位	为功能的扩展而预留的，没有特殊说明时其值为0。
信号	指通过它的状态或状态间的转换来传递信息的电气值。
引脚	表示一种外部电气物理连接，同一个引脚可以连接多个信号。
使能	指使某个离散信号处在有效的状态： - 低电平有效信号从高电平切换到低电平； - 高电平有效信号从低电平切换到高电平。
禁止	指使某个处在使能状态的信号状态改变： - 低电平有效信号从低电平切换到高电平； - 高电平有效信号从高电平切换到低电平。
标识符	以addr这一标识符为例，后面接不同的数字表明不同含义： addr[15:0]表示一组信号，位宽为16位； addr表示单独的一个信号； addr15表示一组信号中的第16个。
LSB	最低有效位。
MSB	最高有效位。
RAW	写后读。
WAW	写后写。

Fig. 1.2: Terms

- *RISC-V External Debug Support, Version 0.13.2.*

2.1 Structure

Fig. 2.1 shows the C906 structure.

2.2 Unit introduction

C906 consists of the following in its core subsystems: instruction fetch unit (IFU), instruction decoding unit (IDU), integer unit (IU), floating-point unit (FPU), load/store unit (LSU), retirement unit (RTU), memory management unit (MMU), physical memory protection (PMP) unit, and master device interface unit (AXI Master IF).

2.2.1 IFU

The IFU can fetch and parallel process up to two instructions at a time. It is equipped with a cache. When the cache is unavailable, key instructions are preferentially fetched. It is also equipped with a instruction register for holding pre-fetched instructions. It adopts a low-cost Gshared branch instruction jump predictor with high prediction accuracy. The IFU features low power consumption, high branch prediction accuracy, and high prefetch efficiency.

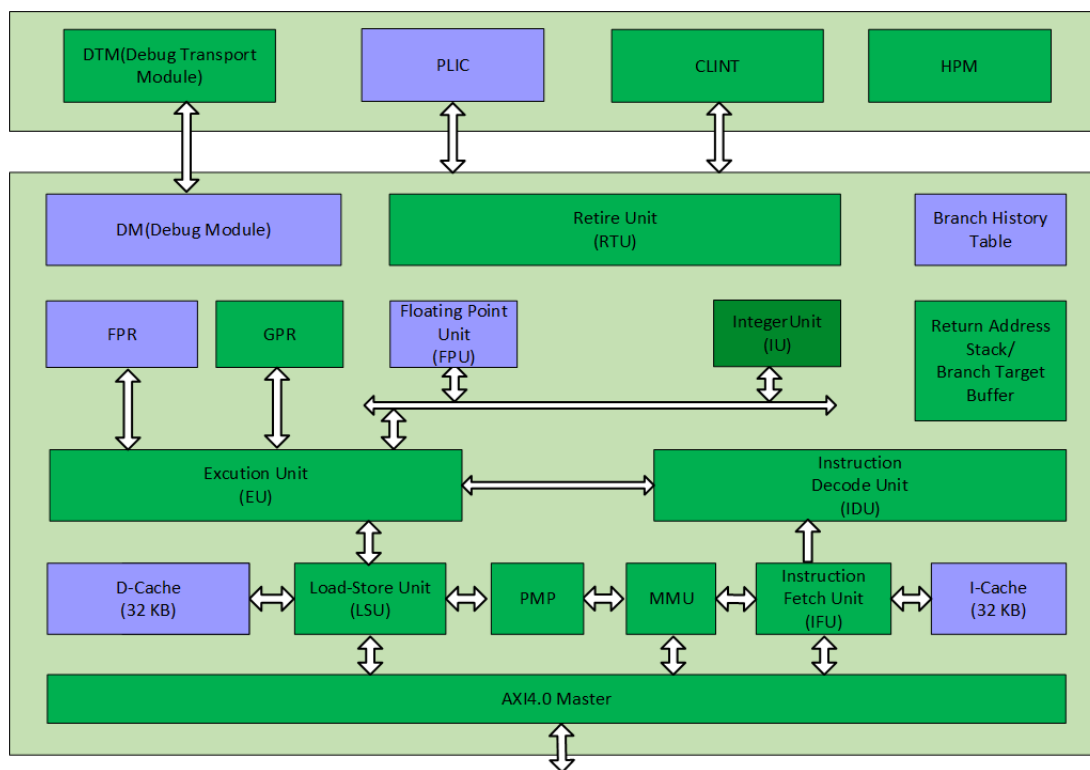


Fig. 2.1: C906 structure

2.2.2 IDU

The IDU decodes one instruction at a time and detects data correlation. Based on next-level pipeline execution, the IDU updates and solves data correlation between instructions and sends instructions to the next-level pipeline for execution.

2.2.3 Execution units

Execution units include IUs, configurable FPUs, and configurable VPU. When VPUs are configured, FPUs are extended into vector floating-point units (VFPUs) to support vector floating-point calculation in addition to original scalar floating-point calculation. In addition, a vector integer unit is added to support vector integer calculation.

IUs include the arithmetic logic unit (ALU), multiplication (MULT) unit, division (DIV) unit, and branch/jump unit (BJU). The ALU performs standard 64-bit integer operations. Computing results of most common instructions are generated in a single cycle, such as addition, subtraction, shift, and logical operation instructions. The ALU reduces the true correlation of data through operand feedforward, and no delay is caused by the true correlation of data in a single-cycle ALU instruction. The MULT unit supports 16×16 , 32×32 , and 64×64 integer multiplication. The division execution cycles are 2 to 36. The BJU completes branch prediction error handling in a single cycle to accelerate the CPU.

FPU includes the floating-point arithmetic logic unit (FALU), floating-point multiply-add unit (FMAU), and floating-point divide and square root unit (FDSU). FPUs support half-precision, single-precision, and double-precision operations. The FALU performs addition, subtraction, comparison, conversion, register data transmission, sign-injection, and classification operations. The FMAU performs common multiply and fused multiply-add operations. The FDSU performs floating-point divide and square root operations. The FDSU uses the radix-4 SRT algorithm. Its execution cycles are 4 to 17.

2.2.4 LSU

The LSU supports execution of scalar and vector store/load instructions and non-blocking access to caches. It supports byte, halfword, word, doubleword, and quadword store/load instructions, and supports sign/zero extension for byte, halfword, and word load instructions. It has an internal feedforward mechanism to eliminate the correlation between write-back data of store instructions. Store/load instructions can be pipelined to achieve a data throughput of accessing one piece of data in one cycle. The LSU supports multi-channel hardware prefetch. It can pre-fetch data from the memory to the L1 D-Cache in advance.

2.2.5 MMU

The MMU translates 39-bit virtual addresses to 40-bit physical addresses in compliance with the RISC-Sv39 standard. The MMU provides extended software writeback methods and address attributes based on the hardware writeback criteria defined in Sv39.

For more information, see [MMU](#).

2.2.6 PMP unit

The PMP unit complies with the RISC-V standard, and can configure 8 or 16 entries, but does not support the NA4 mode. The minimum granularity supported by the PMP unit is 4 KB.

For more information, see [PMP](#).

2.2.7 Master device interface unit

The master device interface unit supports the AXI4.0 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.

2.2.8 PLIC

The platform-level interrupt controller (PLIC) controls sampling and distribution of up to 1023 external interrupt sources. It supports level and pulse interrupts. You can set 32 interrupt priorities.

For more information, see *PLIC*.

2.2.9 Timer

RISC-V defines a 64-bit system timer shared by SoC systems. C906 has a private timer comparison value register. The CPU collects and compares values of the system timer and private timer comparison value register to generate timer interrupt signals.

For more information, see `time_interrupts`.

3.1 Working mode and register view

C906 supports three RISC-V privilege modes: machine mode (M-mode), supervisor mode (S-mode), and user mode (U-mode). C906 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. General user programs are allowed to access only the registers specific to the U-mode. This prevents general user programs from accessing privileged information. When general user programs access registers specific to the U-mode, the operating system manages and serves general user programs by coordinating their access behaviors. The PMP unit protects access of user programs to the memory.

The S-mode provides higher privileges than the U-mode but lower privileges than the M-mode. Programs running in S-mode can access registers specific to U-mode and S-mode but are not allowed to access control registers specific to the M-mode. The PMP unit protects access to the memory of programs running in 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. The CPU switches to



Fig. 3.1: Programming models

the M-mode to respond to exceptions and interrupts that occur in any mode by default 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 the execution permission of the instruction, see *Appendix A Standard Instructions* and *Appendix B T-Head extended instructions*.

The working mode of the CPU changes when it responds to an exception. The processor switches to a higher privilege mode to respond to the exception, and switches back to the lower privilege mode after the exception is handled.

3.2 General-purpose registers

C906 provides thirty-two 64-bit general-purpose registers that have the same features as those defined in RISC-V, as described in [Table 3.1](#).

Table 3.1: General-purpose registers

Register	ABI name	Description
x0	zero	A 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 store instruction operands, instruction execution results, and address information.

3.3 Floating-point registers

In addition to standard RV64FD instruction sets, C906 supports half-precision floating-point computing and provides 32 independent 64-bit floating-point registers. These registers are accessible in U-mode, S-mode, and M-mode.

Table 3.2: 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 general-purpose register x0, floating-point register f0 is not a zero register. 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.

3.3.1 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.
- FMV.X.D/FMV.D.X: A double-precision data move instruction for floating-point registers.

When half-precision, single-precision, or double-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 data formats.

For specific instruction descriptions and definitions, see [Appendix A-4 F instructions](#).

3.3.2 Maintain consistency of register precision

Floating-point registers can store half-precision, single-precision, double-precision, and integer data. For example, the type of data stored in floating-point register 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.

3.4 Vector registers

When VPU are configured, C906 owns 32 independent 128-bit vector registers. 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.

3.4.1 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.

3.4.2 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.

3.5 System control registers

This section describes control registers in M-mode, S-mode, and U-mode.

3.5.1 M-mode control registers

Table 3.3 describes the RISC-V standard M-mode control registers implemented in C906.

Table 3.3: 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 architecture 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.
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.
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	PMP configuration register 0.
PMPCFG2	Read/Write in M-mode	0x3A2	PMP configuration register 2.

Continued on next page

Table 3.3 – continued from previous page

Register	Read/Write permission	ID	Description
PMPADDR0	Read/Write in M-mode	0x3B0	PMP base address register 0.
....
PMPADDR15	Read/Write in M-mode	0x3BF	PMP base address register 15.
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	M-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.

Table 3.4 describes extended M-mode control registers of C906.

Table 3.4: Extended M-mode control registers of C906

Register	Read/Write permission	ID	Description
Extended M-mode CPU control and status registers			
MCOUNTINHIBIT	Read/Write in M-mode	0x320	An M-mode count inhibit register.
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.
MHINT	Read/Write in M-mode	0x7C5	An M-mode implicit operation register.
MRVBR	Read/Write in M-mode	0x7C7	An M-mode reset vector base address register.
MCER	Read/Write in M-mode	0x7C8	An M-mode L1 Cache ECC register.
MOUNTERWRITEEN	Read/Write in M-mode	0x7C9	An S-mode counter write enable register.
MOUNTERINTEN	Read/Write in M-mode	0x7CA	An M-mode event interrupt enable register.
MOUNTEROF	Read/Write in M-mode	0x7CB	An M-mode event overflow flag register.
MHPMCR	Read/Write in M-mode	0x7F0	An M-mode event monitoring control register.
MHPMSP	Read/Write in M-mode	0x7F1	An M-mode event trigger start address register.
MHPMEP	Read/Write in M-mode	0x7F2	An M-mode event trigger end address register.
MAPBADDR	Read-only in M-mode	0xFC1	An M-mode device address upper bits 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	M-mode cache data register 0.
MCDATA1	Read/Write in M-mode	0x7D5	M-mode cache data register 1.
Extended M-mode CPU model registers			
MCPUID	Read-only in M-mode	0xFC0	An M-mode CPU model register.

For specific register definitions and functions, see *Appendix C-1 M-mode control registers*.

3.5.2 S-mode control registers

Table 3.5 describes the RISC-V standard S-mode control registers implemented in C906.

Table 3.5: RISC-V standard S-mode control registers

Register	Read/Write permission	per-	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.

Table 3.6 describes the extended S-mode control registers of C906.

Table 3.6: Extended S-mode control registers of C906

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.
SCOUNTINHIBIT	Read/Write in S-mode	0x5C8	An S-mode count inhibit register.
SHPMCR	Read/Write in S-mode	0x5C9	An S-mode event monitoring control register.
SHPMSP	Read/Write in S-mode	0x5CA	An S-mode event trigger start address register.
SHPMEP	Read/Write in S-mode	0x5CB	An S-mode event trigger end address register.
SCYCLE	Read/Write in S-mode	0x5E0	An S-mode cycle counter.
SINSTRET	Read/Write in S-mode	0x5E2	An S-mode retired instruction counter.
SHPM-COUNTER3	Read/Write in S-mode	0x5E3	S-mode counter 3.
...
SHPM-COUNTER31	Read/Write in S-mode	0x5FF	S-mode counter 31.

For specific register definitions and functions, see *Appendix C-2 S-mode control registers*.

3.5.3 U-mode control registers

RISC-V standard U-mode control registers describes the RISC-V standard U-mode control registers implemented in C906.

Table 3.7: 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-only in U-mode	0xC00	A U-mode cycle counter.
TIME	Read-only in U-mode	0xC01	A U-mode timer.
INSTRET	Read-only in U-mode	0xC02	A U-mode retired instruction counter.
HPM-COUNTER3	Read-only in U-mode	0xC03	A U-mode counter 3.
.....
HPM-COUNTER17	Read-only in U-mode	0xC1F	A U-mode counter 17.

Table 3.8 describes the extended U-mode control registers of C906.

Table 3.8: Extended U-mode control registers of C906

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 specific register definitions and functions, see *Appendix C-3 U-mode control registers*.

3.6 Exception handling

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

services.

The key of exception handling is to save the current operating status of the CPU when an exception occurs and resume the status when the CPU exits exception handling. Exceptions can be identified at all stages of the pipeline. The hardware ensures that subsequent instructions which throw exceptions 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 handles 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 exception in the following procedure:

Step 1: Save the exception instruction PC to the MEPC register.

Step 2: Set MCAUSE based on the exception type and update MTVAL to the instruction fetch address, store/load address, or instruction code upon exceptions.

Step 3: Save the machine interrupt-enable (MIE) bit field 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 if the exception is not downgraded.

Step 5: Obtain the entry address of the exception program based on the base address and mode in the MTVEC register. The CPU start with the first instruction of the exception program to handle exceptions.

C906 conforms to the RISC-V standard exception vector table, as described in [Table 3.9](#).

Table 3.9: 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 monitoring overflow interrupt if the performance monitoring unit (PMU) is configured.

Continued on next page

Table 3.9 – continued from previous page

Interrupt flag	Exception vector ID	Description
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	A fetch instruction 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.

When multiple interrupt requests occur simultaneously, the interrupt priorities are as follows: M-mode external interrupts > M-mode software interrupts > M-mode timer interrupts > S-mode external interrupts > S-mode software interrupts > S-mode external interrupts > Performance monitoring overflow interrupt. The platform-level interrupt controller (PLIC) transmits M-mode external interrupts and S-mode external interrupts to the in-core subsystems, and the PLIC control register determines the priorities of external interrupts in the PLIC.

When an exception or interrupt occurs and the CPU responds to it in M-mode, the CPU updates the PC to the MEPC register and updates MTVAL based on the exception type. When the CPU responds to an interrupt, MEPC is updated to the PC of the next instruction, and MTVAL is updated to 0. When the CPU responds to an exception, MEPC is updated to the instruction PC that triggers the exception and MTVAL is updated based on the exception type. [Table 3.10](#) describes the update values of MTVAL when the CPU responds to exceptions in M-mode.

Table 3.10: 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	See the note below.
6	Store/Atomic instruction unaligned access exception	Virtual address accessed by the store/atomic instruction
7	Store/Atomic instruction access error exception	See the note below.
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

Note: When the access error exception is caused by a PMP authentication failure, MTVAL saves the virtual address accessed by the memory. When the access error exception occurs because the hardware bus returns an error response, MTVAL saves the physical address accessed by the memory.

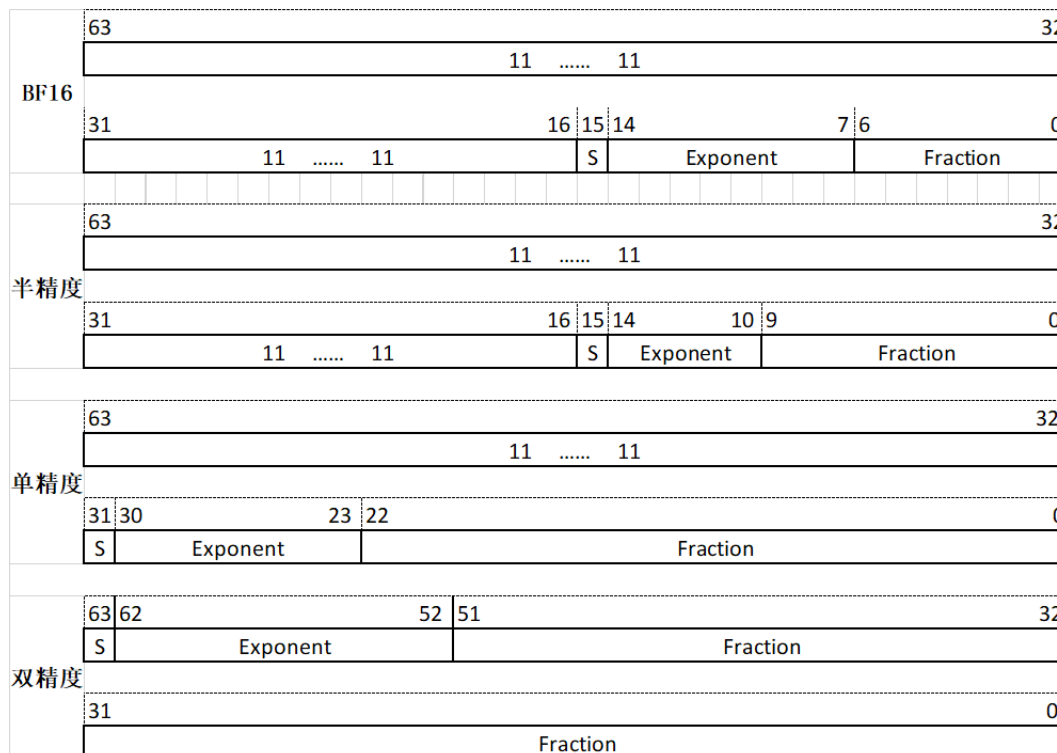
C906 supports exception and interrupt delegation (Delegation). When an exception or interrupt occurs in S-mode, the CPU needs to switch to the M-mode for handling. This causes performance loss of the CPU. Delegation enables the CPU to respond to exceptions and interrupts in S-mode. Exceptions that occur in M-mode are not subject to Delegation, but responded to only in M-mode. Interrupts can be delegated to the S-mode for handling, except the external interrupts, software interrupts, and timer interrupts 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 handles undelegated exceptions and interrupts in M-mode, and updates the M-mode exception handling registers. The CPU responds to delegated exceptions and interrupts in S-mode, and updates the S-mode exception handling registers.

3.7 Data formats

3.7.1 Floating-point data format

FPU of C906 are complied with the RISC-V standard and compatible with ANSI/IEEE 754-2008 floating-point standard, and support half-precision, single-precision, and double-precision floating-point calculation. Fig. ?? shows the floating-point data format. 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.



Floating-point data format in floating-point registers

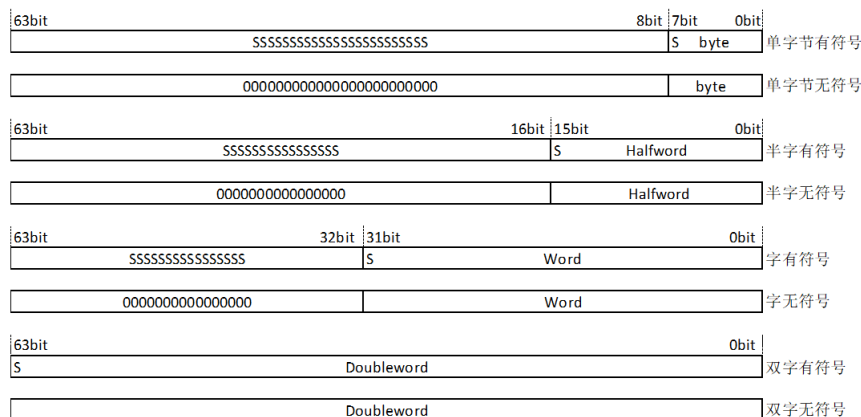
3.7.2 Integer data format

Values in an integer register are not distinguished by big-endian or little-endian type, but by signed or unsigned type. Values 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. ??.

Integer data format in integer registers

3.7.3 Big-endian and little-endian

The concepts of big-endian and little-endian are proposed with respect to the data storage formats of memory. 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. Fig. ?? shows the data format in little-endian scheme.

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

Data structure in memory

C906 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 implicitly indicated in instruction operations (index operation and byte extraction). Usually, an instruction receives a 64-bit operand and generates a 64-bit result.

3.8 Memory model

C906 supports two memory types: memory and device, which are distinguished by the SO bit. Depending on cacheability (Cacheable, C), memory is further classified into cacheable memory and non-cacheable memory. The device type does not allow speculative execution and therefore is non-cacheable. Depending on bufferability (Bufferable, B), device type is further classified into bufferable device and non-bufferable device. To be bufferable indicates that a response to a write request can be quickly returned on an intermediate node. To be non-bufferable indicates that a response to a write request is returned only after the device completes writing.

Table 3.11 describes the page attributes corresponding to each memory type. Page attributes can be configured using the following methods:

1. If virtual addresses are not translated into physical addresses: the machine mode (M-mode) or memory management unit (MMU) is disabled, and page attributes of addresses are determined by macro definition in the sysmap.h file. sysmap.h is an extended address attribute configuration file of C906 that is open to users. You can define page attributes for up to eight address ranges as required.

- If virtual addresses are translated into physical addresses: page attributes of addresses are configured from the `sysmap.h` file or extended page attributes of C906 in page table entries (PTEs) when the CPU is not in M-mode and the MMU is enabled. Which configuration method is used depends on the value of the MAEE field in C906 extended register MXSTATUS. If the MAEE field is 1, page attributes of addresses are determined by extended page attributes in corresponding PTEs. If the MAEE field is 0, page attributes of addresses are determined by the `sysmap.h` file.

Table 3.11: Memory types

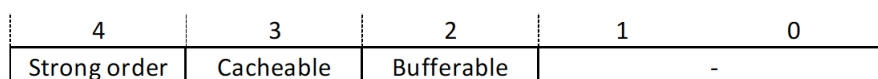
Memory type	SO	C	B
Cacheable memory	0	1	1
Non-cacheable memory	0	0	1
Bufferable device	1	0	1
Non-bufferable device	1	0	0

`sysmap.h` allows you to set attributes for up to 8 memory 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, `SYSMAP_BASE_ADDR(i - 1) <= Address of address space i < SYSMAP_BASE_ADDRi`.

The smallest address of address space 0 is 0x0. Address attributes of eight address ranges where memory addresses not defined in the `sysmap.h` file are Strong order/ Non-cacheable/Non-bufferable 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. Fig. 3.2 shows the attribute layout.

Fig. 3.2: Address attributes in the `sysmap.h` file

During configuration of `sysmap` and page table, it's important to set the CLINT and PLIC address spaces to non-bufferable device.

C906 provides some extended custom instructions based on the RV64GC instruction sets. The half-precision floating-point instructions of C906 extended instruction sets can be directly used. All the other C906 extended instruction sets cannot be used until THEADISAEE bit is enabled in the MXSTATUS register, otherwise, illegal instruction exceptions will be triggered upon execution.

4.1 RV64GC instructions

This section describes RV64GC instruction sets implemented in C906.

RV64GC (RV64IMAFDC) instruction sets include standard integer instruction sets (RV64I), multiply and divide instruction sets (RV64M), atomic instruction sets (RV64A), single-precision floating-point instruction sets (RV64F), double-precision floating-point instruction sets (RV64D), and compression instruction sets (RVC).

4.1.1 RV64I

RV64I is a 64-bit base integer instruction. The RV64I base 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 4.1: RVI 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

Continued on next page

Table 4.1 – continued from previous page

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
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

Continued on next page

Table 4.1 – continued from previous page

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.	Cacheable LOAD: ≥ 2 Cacheable STORE: 1 Non-Cacheable 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.	Aperiodic
SD	A doubleword store instruction.	Same as above

Continued on next page

Table 4.1 – continued from previous page

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 registers.	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).	Blocked Aperiodic
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 Aperiodic
SFENCE.VMA	A virtual memory synchronization instruction.	Same as above
ECALL	An environment call instruction.	1
EBREAK	A breakpoint instruction.	1

For specific instruction descriptions and definitions, see [Appendix A-1 I instructions](#).

4.1.2 RV64M

Table 4.2 describes RV64M instructions.

Table 4.2: RVM instructions

Instruction	Description	Execution delay
MUL	A signed multiply instruction.	3/6
MULW	A signed multiply instruction operating on the lower 32 bits	3
MULH	A signed multiply instruction that extracts the upper bits	3/6
MULHSU	A signed-unsigned multiply instruction that extracts the upper bits	3/6
MULHU	An unsigned multiply instruction that extracts the upper bits	3/6
DIV	A signed divide instruction	5-36
DIVW	A signed divide instruction operating on the lower 32 bits.	5-20
DIVU	An unsigned divide instruction.	5-36
DIVUW	An unsigned divide instruction operating on the lower 32 bits.	5-20
REM	A signed remainder instruction.	5-36
REMW	A signed remainder instruction operating on the lower 32 bits.	5-20
REMU	An unsigned remainder instruction.	5-36
REMUW	An unsigned remainder instruction operating on the lower 32 bits.	5-20

For specific instruction descriptions and definitions, see *Appendix A-2 M instructions*.

4.1.3 RV64A

Table 4.3: RVA instructions

Instruction	Description	Execution delay
LR.W	A word load-reserved instruction.	This instruction is split into multiple atomic instructions for execution.
LR.D	A doubleword load-reserved instruction.	
SC.W	A word store-conditional instruction.	This instruction can be split into atomic instructions for blocked execution, but delay is not allowed.
SC.D	A doubleword store-conditional instruction.	
AMOSWAP.W	An atomic swap instruction that operates on the lower 32 bits.	Expected
AMOSWAP.D	An atomic swap instruction.	
AMOADD.W	An atomic add instruction operating on the lower 32 bits.	
AMOADD.D	An atomic add instruction.	
AMOXOR.W	An atomic bitwise XOR instruction operating on the lower 32 bits.	
AMOXOR.D	An atomic bitwise XOR instruction.	
AMOAND.W	An atomic bitwise AND instruction operating on the lower 32 bits.	
AMOAND.D	An atomic bitwise AND instruction.	
AMOOR.W	An atomic bitwise OR instruction operating 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.	
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 specific instruction descriptions and definitions, see [Appendix A-3 A instructions](#).

4.1.4 RV64F

This section describes single-precision floating-point instructions.

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 4.4: RVF instructions

Instruction	Description	Execution latency
Operation instructions		
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.	3
FMADD.S	A single-precision floating-point multiply-add instruction.	4
FMSUB.S	A single-precision floating-point multiply-subtract instruction.	4
FNMADD.S	A single-precision floating-point negate-(multiply-add) instruction.	4
FNMSUB.S	A single-precision floating-point negate-(multiply-subtract) instruction.	4
FDIV.S	A single-precision floating-point divide instruction.	4 to 17
FSQRT.S	A single-precision floating-point square-root instruction.	4 to 17
Sign injection instructions		

Continued on next page

Table 4.4 – continued from previous page

FSGNJS	A single-precision floating-point sign-injection instruction.	3
FSGNJS	A single-precision floating-point negate sign-injection instruction.	3
FSGNJS	A single-precision floating-point sign-injection XOR instruction.	3
Data transmission instructions		
FMV.X.D	A single-precision floating-point read move instruction.	3
FMV.D.X	A single-precision floating-point write move instruction.	3
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
FLT.S	A single-precision floating-point compare less than instruction.	3
FLE.S	A single-precision floating-point compare less than or equal to instruction.	3
Data type conversion instructions		
FCVT.W.S	An instruction that converts a single-precision floating-point number into a signed integer.	3
FCVT.W.U.S	An instruction that converts a single-precision floating-point number into an unsigned integer.	3
FCVT.S.W	An instruction that converts a signed integer into a single-precision floating-point number.	3
FCVT.S.W.U	An instruction that converts an unsigned integer into a single-precision floating-point number.	3

Continued on next page

Table 4.4 – continued from previous page

FCVT.L.S	An instruction that converts a single-precision floating-point number into a signed long integer.	3
FCVT.LU.S	An instruction that converts a single-precision floating-point number into an unsigned long integer.	3
FCVT.S.L	An instruction that converts a signed long integer into a single-precision floating-point number.	3
FCVT.S.LU	An instruction that converts an unsigned long integer into a single-precision floating-point number.	3
Memory store instructions		
FLW	A single-precision floating-point load instruction.	Cacheable LOAD: ≥ 2 Cacheable STORE:1 Non-Cacheable 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.	3

For specific instruction descriptions and definitions, see *Appendix A-4 F instructions*.

4.1.5 RV64D

This section describes double-precision floating-point instructions.

A double-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

Table 4.5: RVD instructions

Instruction	Description	Execution latency
Operation instructions		
FADD.D	A double-precision floating-point add instruction.	4
FSUB.D	A double-precision floating-point subtract instruction.	4
FMUL.D	A double-precision floating-point multiply instruction.	4
FMADD.D	A double-precision floating-point multiply-add instruction.	5
FMSUB.D	A double-precision floating-point multiply-subtract instruction.	5
FNMSUB.D	A double-precision floating-point negate-(multiply-add) instruction.	5
FNMADD.D	A double-precision floating-point negate-(multiply-subtract) instruction.	5
FDIV.D	A double-precision floating-point divide instruction.	4 to 31
FSQRT.D	A double-precision floating-point square-root instruction.	4 to 31
Sign injection instructions		
FSGNJ.D	A double-precision floating-point sign-injection instruction.	3
FSGNJN.D	A double-precision floating-point negate sign-injection instruction.	3
FSGNJX.D	A double-precision floating-point sign-injection XOR instruction.	3
Data transmission instructions		
FMV.X.D	A double-precision floating-point read move instruction.	3
FMV.D.X	A double-precision floating-point write move instruction.	3
Compare instructions		

Continued on next page

Table 4.5 – continued from previous page

FMIN.D	A double-precision floating-point instruction for extracting the minimum value.	3
FMAX.D	A double-precision floating-point instruction for extracting the maximum value.	3
FEQ.D	A double-precision floating-point compare equal instruction.	3
FLT.D	A double-precision floating-point compare less than instruction.	3
FLE.D	A double-precision floating-point compare less than or equal to instruction.	3
Data type conversion instructions		
FCVT.S.D	An instruction that converts a double-precision floating-point number into a single-precision floating-point number.	3
FCVT.D.S	An instruction that converts a single-precision floating-point number into a double-precision floating-point number.	3
FCVT.W.D	An instruction that converts a double-precision floating-point number into a signed integer.	3
FCVT.WU.D	An instruction that converts a double-precision floating-point number into an unsigned integer.	3
FCVT.D.W	An instruction that converts a signed integer into a double-precision floating-point number.	3
FCVT.D.WU	An instruction that converts an unsigned integer into a double-precision floating-point number.	3
FCVT.L.D	An instruction that converts a double-precision floating-point number into a signed long integer.	3

Continued on next page

Table 4.5 – continued from previous page

FCVT.LU.D	An instruction that converts a double-precision floating-point number into an unsigned long integer.	3
FCVT.D.L	An instruction that converts a signed long integer into a double-precision floating-point number.	3
FCVT.D.LU	An instruction that converts an unsigned long integer into a double-precision floating-point number.	3
Memory store instructions		
FLD	A double-precision floating-point load instruction.	Cacheable LOAD: ≥ 2 Cacheable STORE: 1 Non-Cacheable Aperiodic
FSD	A double-precision floating-point store instruction.	Same as above
Floating-point classify instructions		
FCLASS.D	A double-precision floating-point classify instruction.	3

For specific instruction descriptions and definitions, see *Appendix A-5 D instructions*.

4.1.6 RVC

This section describes 16-bit compression instructions.

The compress 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
- Other special instructions

Table 4.6: RVC instructions

Instruction	Description	Execution delay
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		
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		

Continued on next page

Table 4.6 – continued from previous page

Instruction	Description	Execution delay
C.LW	A word load instruction.	Cacheable LOAD: ≥ 2 Cacheable STORE: 1 Non-Cacheable 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
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 specific instruction descriptions and definitions, see *Appendix A-6 C Instructions*.

4.2 T-Head extended instruction sets

4.2.1 Cache instruction subset

Table 4.7: Cache instructions

Cache instruction subset	Description	Execution delay
DCACHE.CALL	An instruction that clears all dirty page table entries in the D-Cache.	Aperiodic
DCACHE.CVA	An instruction that clears dirty page table entries in the D-Cache that match the specified virtual address.	Aperiodic
DCACHE.CPA	An instruction that clears dirty page table entries in the D-Cache that match the specified physical address.	Aperiodic
DCACHE.CSW	An instruction that clears dirty page table entries the D-Cache based on the specified way and set.	Aperiodic
DCACHE.IALL	An instruction that invalidates all page table entries in the D-Cache.	Aperiodic
DCACHE.IVA	An instruction that invalidates page table entries in the D-Cache that match the specified virtual address.	Aperiodic
DCACHE.IPA	An instruction that invalidates page table entries in the D-Cache that match the specified physical address.	Aperiodic
DCACHE.ISW	An instruction that invalidates page table entries in the D-Cache based on the specified way and set.	Aperiodic
DCACHE.CIALL	An instruction that clears all dirty page table entries in the D-Cache and invalidates the D-Cache.	Aperiodic
DCACHE.CIVA	An instruction that clears dirty page table entries in the D-Cache that match the specified virtual address and invalidates the D-Cache.	Aperiodic
DCACHE.CIPA	An instruction that clears dirty page table entries in the D-Cache that match the specified physical address and invalidates the D-Cache.	Aperiodic
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.	Aperiodic
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.	Aperiodic
ICACHE.IVA	An instruction that invalidates page table entries in the I-Cache that match the specified virtual address.	Aperiodic
ICACHE.IPA	An instruction that invalidates page table entries in the I-Cache that match the specified physical address.	Aperiodic

For specific instruction descriptions and definitions, see *Appendix B-1 Cache instructions*.

4.2.2 Synchronization instruction subset

Table 4.8: Synchronization instructions

Cache instruction subset	Description	Execution delay
SYNC	A synchronization instruction.	Aperiodic
SYNC.I	A synchronization and clear instruction.	Aperiodic

For specific instruction descriptions and definitions, see *Appendix B-2 Multi-core synchronization instructions*.

4.2.3 Arithmetic operation instructions

Table 4.9: Arithmetic operation instructions

Instruction	Description	Execution delay
Add/Subtract instructions		
ADD _{SL}	An add instruction that shifts registers.	1
MUL _A	A multiply-add instruction.	3/6
MUL _S	A multiply-subtract instruction.	3/6
MUL _{AW}	A multiply-add instruction operating on the lower 32 bits.	3
MUL _{SW}	A multiply-subtract instruction operating on the lower 32 bits.	3
MUL _{AH}	A multiply-add instruction operating on the lower 16 bits.	3
MUL _{SH}	A multiply-subtract instruction operating on the lower 16 bits.	3
Shift instructions		
SRRI	A cyclic right shift instruction.	1
SRRIW	A cyclic right shift instruction operating 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 specific instruction descriptions and definitions, see *Appendix B-3 Arithmetic operation instructions*.

4.2.4 Bit operation instruction subset

Table 4.10: Bit operation instructions

Instruction	Description	Execution delay
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
REVW	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 specific instruction descriptions and definitions, see [Appendix B-4 Bitwise operation instructions](#).

4.2.5 Store instruction subset

Table 4.11: Store instructions

Instruction	Description	Execution delay
FLRD	A doubleword load instruction for shifting floating-point registers.	Cacheable LOAD ≥ 2 Aperiodic
FLRW	A word load instruction for shifting floating-point registers.	
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 word load instruction for shifting registers and extending signed bits.	
LRD	A doubleword load instruction for shifting registers.	

Continued on next page

Table 4.11 – continued from previous page

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 the lower 32 bits in registers and extending signed bits.
LURH	A halfword load instruction for shifting registers and extending signed bits.
LURW	A word load instruction for shifting registers and extending signed bits.
LURD	A doubleword load instruction for shifting the lower 32 bits in 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.
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 4.11 – continued from previous page

LWIB	A 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 halfwords 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.
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.	Cacheable1 Noncacheable
FSRW	A word store instruction for shifting floating-point registers.	Aperiodic
FSURD	A doubleword store instruction for shifting the lower 32 bits in floating-point registers.	

Continued on next page

Table 4.11 – continued from previous page

FSURW	A word store instruction for shifting the lower 32 bits in floating-point registers.		
SRB	A byte store instruction for shifting registers.		
SRH	A halfword 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.		
SURD	A doubleword store instruction for shifting the lower 32 bits in registers.		
SBIA	A base-address auto-increment instruction for storing bytes.		
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 specific instruction descriptions and definitions, see *Appendix B-5 Storage instructions*.

4.2.6 Half-precision floating-point instruction subset

Table 4.12: 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 to 11
FSQRT.H	A half-precision floating-point square-root instruction.	4 to 11
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.	3
FMV.H.X	A half-precision floating-point write move instruction.	3
Compare instructions		
FMIN.H	A half-precision floating-point MIN instruction.	3

Continued on next page

Table 4.12 – continued from previous page

FMAX.H	A half-precision floating-point MAX instruction.	3
FEQ.H	A half-precision floating-point compare equal instruction.	3
FLT.H	A half-precision floating-point compare less than instruction.	3
FLE.H	A half-precision floating-point compare less than or equal to instruction.	3
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.D.H	An instruction that converts a half-precision floating-point number into a double-precision floating-point number.	3
FCVT.H.D	An instruction that converts a double-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
FCVT.WU.H	An instruction that converts a half-precision floating-point number into an unsigned integer.	3
FCVT.H.W	An instruction that converts a signed integer into a half-precision floating-point number.	3
FCVT.H.WU	The instruction that converts an unsigned integer into a half-precision floating-point number.	3
FCVT.L.H	An instruction that converts a half-precision floating-point number into a signed long integer.	3
FCVT.LU.H	An instruction that converts a half-precision floating-point number into an unsigned long integer.	3

Continued on next page

Table 4.12 – continued from previous page

FCVT.H.L	An instruction that converts a signed long integer into a half-precision floating-point number.	3
FCVT.H.LU	An instruction that converts an unsigned long integer into a half-precision floating-point number.	3
Memory store instructions		
FLH	A half-precision floating-point load instruction.	Cacheable LOAD: ≥ 2 Cacheable STORE: 1 Non-Cacheable Aperiodic
FSH	A half-precision floating-point store instruction.	Same as above
Floating-point classify instructions		
FCLASS.H	A single-precision floating-point classify instruction.	3

For specific instruction descriptions and definitions, see *Appendix B-6 Half-precision floating-point instructions*.

5.1 MMU overview

The memory management unit (MMU) of C906 is compatible with the RISC-V Sv39 memory system. It provides the following features:

- **Address translation:** translates the 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.

The MMU uses Translation Look-aside Buffers (TLBs) to implement its features. The TLB stores virtual addresses as an input that are used when the CPU accesses the memory. It checks the page attributes in the TLB before translation and outputs a physical address corresponding to the virtual address. The MMU 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. The uTLB provides 20 fully associative entries (10 I-uTLB entries and 10 D-uTLB entries). The jTLB is 2-way set-associative RAM. Each way contains at least 256 entries.

5.2 Programming models and address translation

5.2.1 MMU control register

The read/write permission of MMU control registers includes the S-mode and M-mode.

5.2.1.1 MMU address translation register (SATP)

The SATP is an MMU control register defined in the Sv39 standard.

63	60	59	44	43	32
Mode		ASID			-
31	28	27			
-			PPN		

Descriptions

Mode: MMU address translation mode

Table 5.1: MMU address translation mode

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

When Mode is 0, the MMU is disabled. C906 supports only the MMU disabled and Sv39 modes.

ASID: the current address space identifier (ASID)

Indicates the ASID of the current program.

PPN: root PPN for hardware writeback

Indicates the PPN used for L1 hardware writeback.

5.2.2 Address translation process

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 address translation page tables. C906 implements address translation through indexing at most three levels of page tables: C906 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. It is possible to obtain the final physical address, a leaf page table, through each level access. The virtual page number (VPN) consists of 27 bits and is equally divided into three 9-bit VPN[i] ($\{VPN[2], VPN[1], VPN[0]\}$). A part of the VPN is used for indexing in

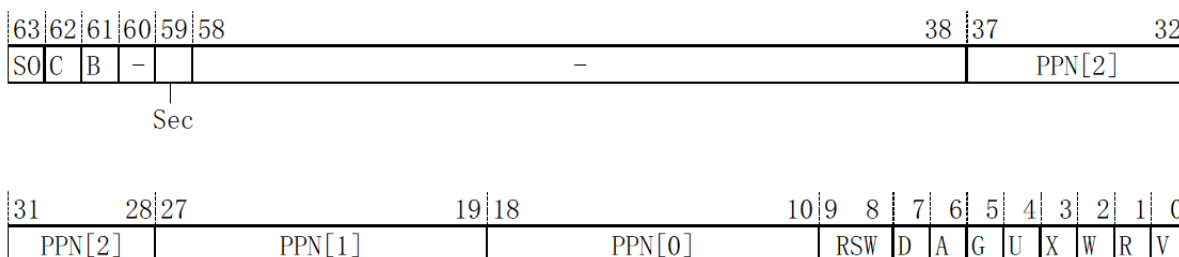
each access in the following sequence: $VPN[2] > VPN[1] > VPN[0]$. When the page table size is set to 4 KB, 2 MB, or 1 GB, the page table is indexed by 3, 2, or 1 times, respectively.

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. The TLB includes two levels: the uTLB at level 1 and the jTLB at level 2. The uTLB includes instruction and data with 10 entries for each of them to further accelerate TLB access. The jTLB is 2-way set-associative and shared by data and instructions.

If the uTLB is mismatched, the MMU accesses the jTLB. If the jTLB is mismatched, the MMU enables a hardware page table walk to access the memory to obtain the final address translation result.

5.2.2.1 Page table structure

A page table stores entry addresses of next-level page tables or physical information of the final page table. Fig. ?? shows the page table structure.



Page table structure

PPN: physical page number

PPN[i] indicates the PPN corresponding to each level of page table.

RSW – Reserved for Software

A bit reserved for software to implement custom page table features. The default value is 2' b0.

D – Dirty

When the D bit is 1, 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.

Hardware implementation of this bit in C906 is similar to the W attribute. When the D bit is 0, a write operation to the page will trigger a page fault (store) exception. You can maintain the definition of whether the page has been/can be written to by configuring the D bit in the exception service program. This bit is reset to 0.

A – Accessed

When the A bit is 1, it indicates that the page is accessed. When the A bit is 0, it indicates that the page is not accessed. Access to the page will trigger a page fault (corresponding access type) exception and this field is set to 1. This bit is reset to 0.

G – Global

The global page ID, which indicates whether the page can be shared by multiple processes. This bit is reset to 0.

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

It is accessible in U-mode. This bit is reset to 0.

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: executable, W: writable, R: readable

Table 5.2: XRW 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

A page fault exception is triggered when XWR permissions are violated.

V – Valid

Indicates whether the physical page has been allocated in memory. If the V bit of a page is 0, access to the page will cause a page fault exception. This bit is reset to 0.

1' b0: indicates that the page has not been allocated.

1' b1: indicates that the page has been allocated.

C906 extended page attributes

SO– Strong order

Indicates the access order required by memory.

1' b0: no strong order (Normal-memory),

1' b1: strong order (Device)。

The default value is no strong order.

C – Cacheable

1' b0: Non-cacheable,

1' b1: Cacheable。

The default value is Non-cacheable.

B – Buffer

1' b0: Non-bufferable,

1' b1: Bufferable。

The default value is Non-bufferable.

Sec (T – Trustable)

Indicates whether the page belongs to the trustable or non-trustable world. This bit makes sense only when the TEE extension is in place. This bit is not defined in C906.

1' b0: Non-trustable,

1' b1: Trustable,

The default value is Trustable.

C906 extended page attributes exist only when the MAEE bit in the MXSTATUS register is 1.

5.2.2.2 Address translation process

If the TLB is hit, then the physical address and the corresponding attributes is directly obtained from the TLB. If the TLB is missed, the address is translated through the following steps:

1. Obtain the memory access address $\{\text{SATP.PPN}, \text{VPN}[2], 3' \text{ b0}\}$ of the L1 page table based on SATP.PPN and VPN[2], 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 Table 5.2. If yes, it's indicated that the final physical address has been found. Then go to step 3. If no, go back to step 1, use PTE.PPN to concatenate the next-level VPN[] and 3' b0 to get memory access address of next-level page table to continue access to the D-Cache/memory.

3. After the leaf page table is found, combine the X/W/R/L bit in the PMP register with the X/W/R bit in the PTE to obtain the minimum permissions for permission check, 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 occurs 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 table size. If no, generate a page fault exception.

When a request misses the uTLB, it takes at least four CPU clock cycles to obtain page tables from the next-level cache. When an in-core instruction fetch or other request to the MMU misses the jTLB, it takes at least 16 CPU clock cycles to obtain page tables from the next-level cache.

5.3 TLB

The MMU of C906 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 10 fully associative entries for storing pages of 4 KB, 2 MB, and/or 1 GB size in each. 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 10 fully associative entries for storing pages of 4 KB, 2 MB, and/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 2-way set-associative and shared by instructions and data. It provides 128, 256, or 512 entries for storing pages of 4 KB, 2 MB, and/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.

6.1 PMP overview

The physical memory protection (PMP) unit of C906 complies with the RISC-V standard. In a protected system, access to two types of resources needs to be monitored: memory and device. The PMP unit checks the validity of access to the memory (including the memory and device). It determines whether the CPU has the read/write/execution permissions to a memory address in the current mode.

The PMP unit of C906 provides the following features:

- Supports eight PMP entries, which are indexed by 0 to 7.
- 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.

6.2 PMP control registers

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

6.2.1 Physical memory protection configuration (PMPCFG) register

Each 64-bit PMPCFG register supports permission configuration for 8 entries. Fig. 6.1 shows the layout of the PMPCFG register.

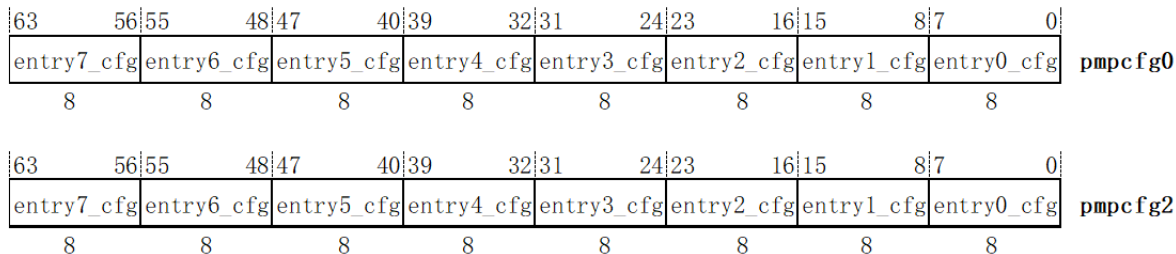


Fig. 6.1: Layout of the PMPCFG register

Fig. 6.2 shows the layout of a 8-bit PMPCFG register of an entry.

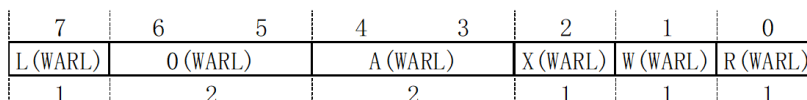


Fig. 6.2: PMPCFG register

For more information about the PMPCFG register, see Table 6.1.

Table 6.1: Descriptions of the PMPCFG register

Bit	Name	Description
0	R	0 : indicates that the address matching the entry is unreadable. 1 : indicates that the address matching the entry is readable.
1	W	0 : indicates that the address matching the entry is unwritable. 1 : indicates that the address matching the entry is writable.
2	X	0 : indicates that the address matching the entry is inexecutable. 1 : indicates that the address matching the entry is executable.
4:3	A	The address matching mode of the entry. 00 : indicates OFF, an invalid entry. 01 : TOR (Top of range), the address of the adjacent entry is used as the mode of matching range. 10 : NA4 (Naturally aligned four-byte region), the matching range is 4 bytes. This mode is not supported. 11 : NAPOT (Naturally aligned power-of-2 regions), 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, the size of the range controlled by entry i is $[\{PMPADDR_{i-1}[37:10], 12' b0\}, \{PMPADDR_i[37:10], 12' b0\})$, values of $PMPADDR_i[9:0]$ do not participate in logical operations of address matching. Entry 0 uses $0x0$ as the address space lower boundary, that is, $[0, \{PMPADDR_0[37:10], 12' b0\})$. If the lower boundary of this range is greater than or equal to the upper boundary, this entry is in OFF mode. That is, this entry is disabled, and all access requests will not hit this entry.

Table 6.2 describes the relationship between addresses and corresponding protection region sizes in NAPOT mode.

In NAPOT mode, the region size and base address of entry i are determined by $PMPADDR_i$. If the first bit value of 0 occurs in the n th bit of $PMPADDR_i$, that is, $PMPADDR_i[n]$ from bit[0] to bit[37] in $PMPADDR_i$, the base address of this entry is $\{PMPADDR_i[37:n+1], (n+3)\{1' b0\}\}$, and its space size is $2^{(n+3)}$ bytes. Use $PMPADDR_i=38' byy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yy01_1111_1111$ as an example. Its base address is $40' byy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yy00_0000_0000_00$, and its space size is 4 KB.

Table 6.2: Protection region code

PMPADDR	PMPCFG.A	Protection region size	Remarks
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy	NA4	4B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy	NA8POT	8B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy	NA16POT	16B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy	NA32POT	32B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy	NA64POT	64B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy0	NA128POT	128B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yy01	NA256POT	256B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_y011	NA512POT	512B	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_0111	NA1KB	1KB	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yyy0_1111	NA2KB	2KB	This mode is not supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_yy01_1111	NA4KB	4KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_y011_1111	NA8KB	8KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyyy_0111_1111	NA16KB	16KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yyy0_1111_1111	NA32KB	32KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_yy01_1111_1111	NA64KB	64KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_y011_1111_1111	NA128KB	128KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyyy_0111_1111_1111	NA256KB	256KB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_yyy0_1111_1111_1111	NA512KB	512KB	This mode is supported.

Continued on next page

Table 6.2 – continued from previous page

PMPADDR	PMPCFG.A	Protection region size	Remarks
yy_yyyy_yyyy_yyyy_yyyy_yy01_1111_1111_1111_1111	NAPOT	1MB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_y011_1111_1111_1111_1111	NAPOT	2MB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyyy_0111_1111_1111_1111_1111	NAPOT	4MB	This mode is supported.
yy_yyyy_yyyy_yyyy_yyy0_1111_1111_1111_1111_1111	NAPOT	8MB	This mode is supported.
yy_yyyy_yyyy_yyyy_yy01_1111_1111_1111_1111_1111	NAPOT	16MB	This mode is supported.
yy_yyyy_yyyy_yyyy_y011_1111_1111_1111_1111_1111	NAPOT	32MB	This mode is supported.
yy_yyyy_yyyy_yyyy_0111_1111_1111_1111_1111_1111	NAPOT	64MB	This mode is supported.
yy_yyyy_yyyy_yyy0_1111_1111_1111_1111_1111_1111	NAPOT	128MB	This mode is supported.
yy_yyyy_yyyy_yy01_1111_1111_1111_1111_1111_1111	NAPOT	256MB	This mode is supported.
yy_yyyy_yyyy_y011_1111_1111_1111_1111_1111_1111	NAPOT	512MB	This mode is supported.
yy_yyyy_yyyy_0111_1111_1111_1111_1111_1111_1111	NAPOT	1GB	This mode is supported.
yy_yyyy_yyy0_1111_1111_1111_1111_1111_1111_1111	NAPOT	2GB	This mode is supported.
yy_yyyy_yy01_1111_1111_1111_1111_1111_1111_1111	NAPOT	4GB	This mode is supported.
yy_yyyy_y011_1111_1111_1111_1111_1111_1111_1111	NAPOT	8GB	This mode is supported.
yy_yyyy_0111_1111_1111_1111_1111_1111_1111_1111	NAPOT	16GB	This mode is supported.
yy_yyy0_1111_1111_1111_1111_1111_1111_1111_1111	NAPOT	32GB	This mode is supported.
yy_yy01_1111_1111_1111_1111_1111_1111_1111_1111	NAPOT	64GB	This mode is supported.
yy_y011_1111_1111_1111_1111_1111_1111_1111_1111	NAPOT	128GB	This mode is supported.

Continued on next page

Table 6.2 – continued from previous page

PMPADDR	PMPCFG.A	Protection region size	Remarks
yy_0111_1111_1111_1111_1111_1111_1111_1111_1111	1NAPOT	256GB	This mode is supported.
y0_1111_1111_1111_1111_1111_1111_1111_1111_1111	1NAPOT	512GB	This mode is supported.
01_1111_1111_1111_1111_1111_1111_1111_1111_1111	1NAPOT	1TB	This mode is supported.
11_1111_1111_1111_1111_1111_1111_1111_1111_1111	1NAPOT	2TB	This mode is supported.

The PMP unit of C906 supports the minimum address matching granularity of 4 KB.

6.2.2 Physical memory protection address (PMPADDR) register

The PMP unit provides PMPADDR0 to PMPADDR7/15 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 C906 supports the minimum granularity of 4 KB. Therefore, bit [8:0] is not used for address authentication logic.

	63	38	37	9	8	0
	0		Physical address [39:11] (WARL)		0 (WARL)	
Reset	0		0		0	

Fig. 6.3: PMPADDR registers

C906 adopts the L1 cache running on the Harvard architecture, including the independent instruction cache (I-Cache) and data cache (D-Cache).

7.1 I-Cache subsystem

The L1 instruction cache (I-Cache) provides the following features:

- Configurable size: 8 KB/16 KB/32 KB/64 KB;
- 2-way set-associative, with a cache line size of 64 bytes;
- Virtually indexed, physically tagged (VIPT);
- Data width per read access: 32 bits; Data width per writeback access: 128 bits;
- First-in, first-out (FIFO);
- Invalidation by I-Cache or cache line supported;
- Instruction prefetch supported;

7.1.1 Instruction prefetch

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 I-Cache.

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

7.1.2 Branch history table

C906 uses the branch history table (BHT) to predict jump directions of conditional branch instructions. The BHT can be 8 KB or 16 KB in size. The GSHARE branch predictor predicts one branch result per cycle.

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

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

7.1.3 Branch and jump target predictor

The C906 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 16 fully associative entries.
- Supports indexing by using a part of the PC of the current branch instruction.
- Initiates jump when the BHT is hit without extra jump loss.

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

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

7.1.4 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 4 nested function calls. If more than 4 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 D-Cache subsystem

The L1 data cache (D-Cache) provides the following features:

- Configurable size: 8 KB/16 KB/32 KB/64 KB;
- 4-way set-associative, with a cache line size of 64 bytes;
- Virtually indexed, physically tagged (VIPT);
- Maximum data width per read access: 64 bits, supporting byte, halfword, word, and doubleword access;
- Maximum data width per write access: 128 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;
- Data prefetch supported.

7.2.1 Data prefetch

C906 supports data prefetch to reduce the access delay of large-sized memory such as DDR SDRAMs. The LSU 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. C906 supports two prefetch methods: consecutive prefetch and stride-based prefetch (stride ≤ 16 cache lines). C906 also implements forward prefetch and backward prefetch (the stride is negative) under stride-based prefetch mode 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, LD;
- FLH, FLW, FLD;
- LRB, LRH, LRW, LRD, LRBU, LRHU, LRWU, LURB, LURH, LURW, LURD, LURBU, LURHU, LURWU, LBI, LHI, LWI, LDI, LBUI, LHUI, LWUI, LDD, LWD, LWUD;

7.2.2 Adaptive write allocation mechanism

C906 supports adaptive write allocation. When the CPU detects consecutive memory write operations, the write allocation attribute of pages is automatically disabled. You can configure the AMRL1 field in the HINT register to determine the number of consecutive cache lines to be stored before disabling the write allocation policy of the L1 D-Cache.

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, SD;
- FSH, FSW, FSD;
- SRB, SRH, SRW, SRD, SURB, SURH, SURW, SURD, SBI, SHI, SWI, SDI, SDD, SWD;

7.2.3 Exclusive access

C906 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. The LR instruction tags the address to be exclusively accessed. The SC instruction determines whether the tagged address is preempted by other processes. C906 sets a local monitor in the L1 D-Cache. The monitor consists of a state machine and an address buffer. The state machine has two states: IDLE and EXCLUSIVE.

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. In addition, the local monitor must be cleared when a process is switched.

7.3 L1 cache operations

After the CPU is reset, the I-Cache and D-Cache are automatically invalidated and disabled by default. C906 provides extended control registers and instructions related to cache operations and supports cache enabling and disabling, dirty table entry clearing, invalidation, and read.

7.3.1 Extended registers of the L1 cache

Extended registers of the C906 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 M-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 register group*.

7.3.2 Extended instructions of L1 cache

C906 provides extended instructions for cache-related operations, including invalidation by address, invalidating all, clearing dirty table entries by address, clearing all dirty table entries, clearing and invalidating dirty table entries by address, and clearing and invalidating all dirty table entries, as described in [Table 7.2](#).

Table 7.2: L1_cache_extension_instruction

DCACHE.CALL	Clears all dirty page table entries in the D-Cache.
DCACHE.CVA	Clears dirty page table entries in the D-Cache that match the specified virtual address.
DCACHE.CPA	Clears dirty page table entries in the D-Cache that match the specified physical address.
DCACHE.CSW	Clears dirty page table entries in the D-Cache based on the specified way and set.
DCACHE.IALL	Invalidates all page table entries in the D-Cache.
DCACHE.IVA	Invalidates page table entries in the D-Cache that match the specified virtual address.
DCACHE.IPA	Invalidates page table entries in the D-Cache that match the specified physical address.
DCACHE.ISW	Invalidates page table entries in the D-Cache based on the specified way and set.
DCACHE.CIALL	Clears all dirty page table entries in the D-Cache and invalidates the D-Cache.
DCACHE.CIVALL	Clears dirty page table entries that match the specified virtual addresses in the D-Cache and invalidates the D-Cache.
DCACHE.CIPALL	Clears dirty page table entries in the D-Cache that match the specified physical address and invalidates the D-Cache.
DCACHE.CISW	Clears dirty page table entries in the D-Cache based on the specified way and set and invalidates the D-Cache.
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.IVA	Invalidates page table entries in the I-Cache that match the specified virtual address.
ICACHE.IPA	Invalidates page table entries in the I-Cache that match the specified physical address.

This section describes the bus interface protocol of C906. The C906 bus can transfer data between the CPU and other devices, including data store and fetch, address control, and bus control.

8.1 Master device interface unit

The main device interface of C906 supports the AMBA 4.0 AXI protocol. (For details, see AMBA specifications - AMBA AXI and ACE Protocol Specification)

8.1.1 Features

The master device interface controls address accesses and data transmission between C906 and the AXI bus. It provides the following features:

- Complies with the AMBA 4.0 AXI protocol.
- Supports a bus width of 128 bits.
- Supports dynamic frequency conversion.
- Delays all bus input and output signals at the register to ensure good timing.

8.1.2 Protocol content

8.1.2.1 Supported transmission types

The master device interface supports the following transmission types:

- Burst types: INCR1, INCR4, and WRAP4;
- Transmission lengths: 1 and 4;
- Exclusive access;
- Transmission sizes: quadword, doubleword, word, halfword, and byte;
- Read and write.
- Outstanding capability of the read channel: 10, outstanding capability of the write channel: 16.

8.1.2.2 Supported response types

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

- OKAY
- EXOKAY
- SLVERR
- DECERR

8.1.3 CPU behavior in different bus responses

numref:*trunk_exception_handling* describes CPU behavior in different bus responses.

Table 8.1: 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.

C906 implements the core local interrupt (CLINT) controller. It is a memory address mapping module that handles software and timer interrupts.

9.1 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 described in [Table 9.1](#). All registers support only access to word-aligned addresses.

Table 9.1: Memory-mapped addresses in CLINT registers

Address	Name	Attribute	Initial Value	Description
0x4000000	MSIP0	Read/Write	0x00000000	M-mode software interrupt pending register. The upper bits are tied to 0, and bit[0] is valid.
Reserved	-	-	-	-
0x4004000	MTIMECMPLO	Read/Write	0xFFFFFFFF	M-mode system timer compare value register (lower 32 bits).
0x4004004	MTIMECMPH0	Read/Write	0xFFFFFFFF	M-mode system timer compare value register (upper 32 bits).
Reserved	-	-	-	-
0x400C000	SSIP0	Read/Write	0x00000000	S-mode software interrupt pending register. The upper bits are tied to 0, and bit[0] is valid.
Reserved	-	-	-	-
0x400D000	STIMECMPLO	Read/Write	0xFFFFFFFF	S-mode system timer compare value register (lower 32 bits).
0x400D004	STIMECMPH0	Read/Write	0xFFFFFFFF	S-mode system timer compare value register (upper 32 bits).
Reserved	-	-	-	-

9.2 Software interrupts

The CLINT controller can be used to set 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 register (MSIPR). S-mode software interrupts are controlled by the supervisor software interrupt pending register (SSIPR).

You can set the MSIPR/SSIPR to 1 to generate software interrupts or reset it to 0 to clear software interrupts. CLINT S-mode software interrupt requests are valid only when the CLINTEE bit in the MXSTATUS register is 1.

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 SSIPR. In U-mode, the CPU has no access to software interrupt registers.

The two groups of registers have the same structure. Fig. 9.1 shows the bit layout and definition of the registers.

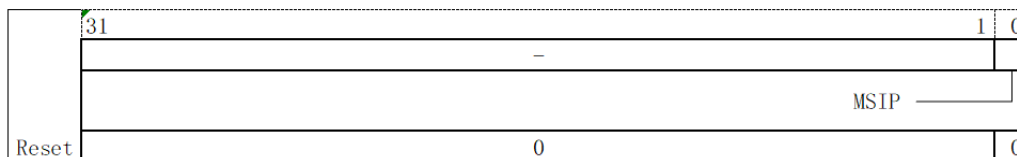


Fig. 9.1: MSIPR

- **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.

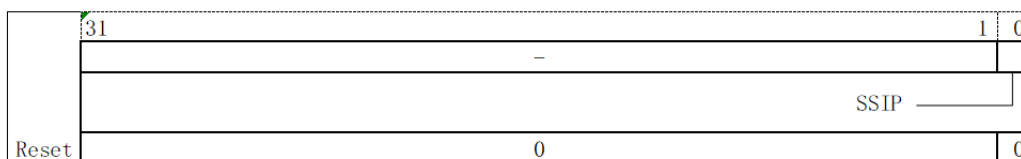


Fig. 9.2: SSIPR

- **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.

9.3 Timer interrupts

According to RISC-V definition, a 64-bit system timer `MTIME` needs to be implemented in the system and run in the always-on clock domain. In low-power mode, the CPU determines whether to disable the working clock based on the SoC. The counter register of this timer needs to be implemented in the SoC always-on clock domain and can allocate address space to the `MTIME` register in the SoC. When the load or store instruction is used to read data from or write data to the register, the register can be reset to 0. The current value of the counter register needs to be transferred to the CPU. The CPU can use the CSR instruction to read the mirrored CSR of the `MTIME` register. The CPU cannot use the CSR instruction to modify the `MTIME` register value.

This system timer defines a group of 64-bit *M-mode timer compare value registers* (`MTIMECMPH` and `MTIMECMPL`) and *a group of 64-bit S-mode timer compare value registers* (`STIMECMPH`, `STIMECMPL`). You can access and modify these registers through word-aligned address access. These two groups of registers are implemented in the CLINT module of C906. Table 9.1 shows the address mapping.

The CLINT controller compares the current value of the system timer (MTIME) with the current value of compare value register $\{M/STIMECMPH[31:0], M/STIMECMPL[31:0]\}$ to determine whether to generate a timer interrupt. When the value of the system timer is less than or equal to the value of $\{M/STIMECMPH[31:0], M/STIMECMPL[31:0]\}$, the CLINT controller does not generate an interrupt. When the value of the system timer is greater than the value of $\{M/STIMECMPH[31:0], M/STIMECMPL[31:0]\}$, the CLINT controller generates the 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 in the MXSTATUS register is 1.

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 extended S-mode timer compare value 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. 9.3.

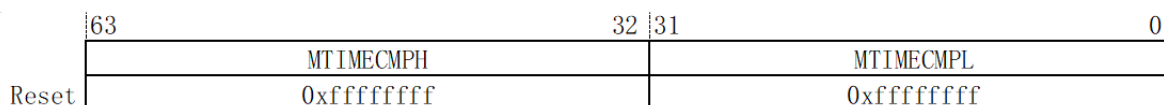


Fig. 9.3: MTIMECMPH/MTIMECMPL registers

- **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.

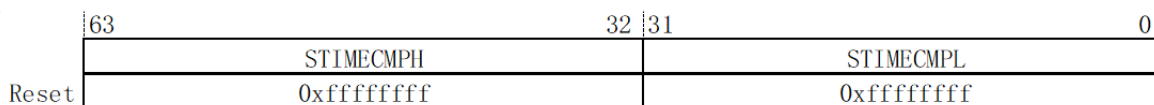


Fig. 9.4: STIMECMPH/STIMECMPL registers

- **STIMECMPH/STIMECMPL: the S-mode clock timer compare value registers for the upper bits and the lower bits**

These registers store timer compare values, which are T-Head extended registers.

- STIMECMPH: stores the upper 32 bits of timer compare values.
- STIMECMPL: stores the lower 32 bits of timer compare values.

The platform-level interrupt controller (PLIC) controls sampling, priority arbitration, and distribution of external interrupt sources.

The PLIC of C906 provides the following features:

- Configurable M-mode interrupts and S-mode interrupts;
- Sampling of up to 1023 interrupt sources, supporting level and pulse interrupts;
- 32 interrupt priorities;
- Independent enable of interrupts in each interrupt mode (M-mode/S-mode);
- Independent threshold for interrupts in each interrupt mode (M-mode/S-mode);
- Configurable access permissions on PLIC registers.

10.1 Interrupt handling mechanism

10.1.1 Interrupt arbitration

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

- The interrupt source is in the pending state ($IP = 1$).
- The interrupt priority is greater than 0. (Interrupts whose priority is 0 are invalid.)

- The enable bit (PLIC_H0_MIEn/SIEn register) for the interrupt target mode is enabled.

Note: The interrupt target mode indicates whether an interrupt is an S-mode interrupt or M-mode interrupt. For more information about responses to different interrupt target modes, see Table 10.1.

When multiple interrupts of an interrupt target mode are in the pending state, the PLIC selects the interrupt with the highest priority through arbitration. A larger value of the priority configuration register indicates a higher priority. In the PLIC of C906, M-mode interrupts have higher priorities than S-mode interrupts. The CPU core preferentially responds to valid M-mode interrupts arbitrated by the PLIC. 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.

Table 10.1 describes the interrupt claim conditions and interrupt complete modes based on the interrupt target mode and CPU working mode arbitrated by the PLIC. “MCAUSE” listed in the “Update register” column indicates that the CPU switches to the M-mode to respond to interrupts. “SCAUSE” listed in the “Update register” column indicates that the CPU switches to the S-mode to respond to interrupts. The “Mie/SiE register” column indicates whether the values of the MIE and SIE bits in the MSTATUS register are 1. The “Delegation register” column corresponds to the bit of the interrupt in the MIDELEG register.

Table 10.1: Interrupt claim and complete in different PLIC interrupt target modes and CPU working modes

CPU mode	Interrupt mode	Mie/Sie register	Delegation register	Claim interrupt	Update register
M-mode	M-mode	Mie = 1	Unconfigurable	Yes	MCAUSE
	S-mode	Mie = 1	0	Yes	MCAUSE
	S-mode	Don't care	1	No	No
S-mode	M-mode	Don't care	Unconfigurable	Yes	MCAUSE
	S-mode	Don't care	0	Yes	MCAUSE
		Sie=1	1	Yes	SCAUSE
U-mode	M-mode	Don't care	Unconfigurable	Yes	MCAUSE
	S-mode	Don't care	0	Yes	MCAUSE
		Don't care	1	Yes	SCAUSE

10.1.2 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.

10.1.3 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, valid interrupts of the external interrupt source must be cleared before the write operation is initiated.
- After receiving the interrupt completion message, the PLIC updates the interrupt claim/complete register and unblocks sampling on the interrupt source corresponding to the interrupt ID to end the interrupt handling process.

10.2 PLIC register address mapping

C906 assigns 64 MB memory space to the PLIC. Addresses in the upper 13 bits depend on the configuration of SoC integration in use of C906. Address mapping in the lower 27 bits is described in [Table 10.2](#). All registers support only access to word-aligned addresses. (*PLIC registers are accessible through the load word instruction. Access results are placed in the lower 32 bits of 64-bit GPRs.*)

Table 10.2: PLIC register address space mapping

Address	Name	Type	Initial value	Description
0x0000004	PLIC_PRI01	R/W	0x0	The priority configuration register for interrupts 1 to 1023.
0x0000008	PLIC_PRI02	R/W	0x0	
0x000000C	PLIC_PRI03	R/W	0x0	
...	
0x0000FFC	PLIC_PRI01023	R/W	0x0	
0x0001000	PLIC_IP0	R/W	0x0	The interrupt pending register for interrupts 1 to 31.
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.
0x0002004	PLIC_H0_MIE1	R/W	0x0	The M-mode interrupt enable register for interrupts 32 to 63.
...
0x000207C	PLIC_H0_MIE31	R/W	0x0	The M-mode interrupt enable registers for interrupts 992 to 1023.
0x0002080	PLIC_H0_SIE0	R/W	0x0	The S-mode interrupt enable register for interrupts 1 to 31.
0x0002084	PLIC_H0_SIE1	R/W	0x0	The S-mode interrupt enable register for interrupts 32-63.
...
0x00020FC	PLIC_H0_SIE31	R/W	0x0	The S-mode interrupt enable register for interrupts 992 to 1023.
Reserved	-	-	-	-
0x01FFFFC	PLIC_PER	R/W	0x0	The PLIC permission control register.
0x0200000	PLIC_H0_MTH	R/W	0x0	The M-mode interrupt threshold register.
0x0200004	PLIC_H0_MCLAIM	R/W	0x0	The M-mode interrupt claim/complete register.
Reserved	-	-	-	-
0x0201000	PLIC_H0_STH	R/W	0x0	The S-mode interrupt threshold register.
0x0201004	PLIC_H0_SCLAIM	R/W	0x0	The S-mode interrupt claim/complete register.
Reserved	-	-	-	-

10.3 PLIC_PRIO register

You can use the 32-bit PLIC_PRIO register with mapping addresses to set the priority for each interrupt source. The current register value is the priority of the interrupt source. For more information about the read and write permissions on the register, see the descriptions of the PLIC_CTRL register. Fig. 10.1 shows the bit layout and definition of the register.

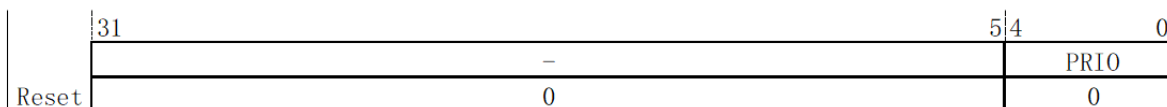


Fig. 10.1: PLIC_PRIO register

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 interrupt target mode, the priority 1 is the lowest priority, and the priority 31 is the highest priority. When multiple interrupts with the same priority are waiting for arbitration, IDs of these interrupts are further compared. An interrupt with a smaller ID has a higher priority.

10.4 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. For more information about the read and write permissions on the register, see the descriptions of the PLIC_CTRL register. Fig. 10.2 shows the bit layout and definition of the register.

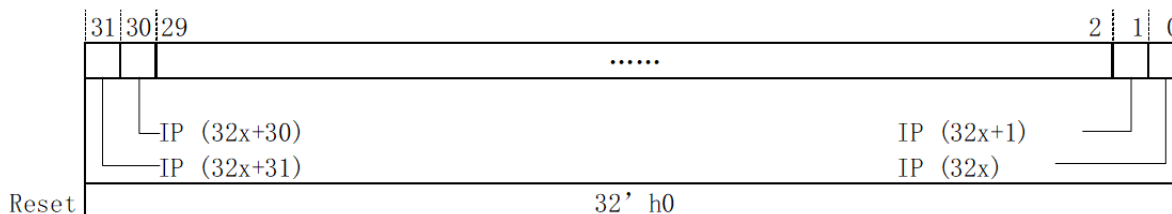


Fig. 10.2: PLIC_IPx register

IP: Interrupt pending state

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 (SW 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 (SW instruction) to set this bit to 0. After an interrupt is handled, the PLIC clears the IP bit of the corresponding interrupt.

10.5 PLIC_IE register

Each interrupt target has an interrupt enable bit (M-mode interrupt enable and S-mode interrupt enable) for each interrupt source, to enable the corresponding interrupts. The M-mode interrupt enable register is used to enable M-mode interrupts. The S-mode interrupt enable register is used to enable S-mode 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 ID 0 is set to 0. For more information about the read and write permissions on the register, see the descriptions of the PLIC_CTRL register. Fig. 10.3 shows the bit layout and definition of the register.

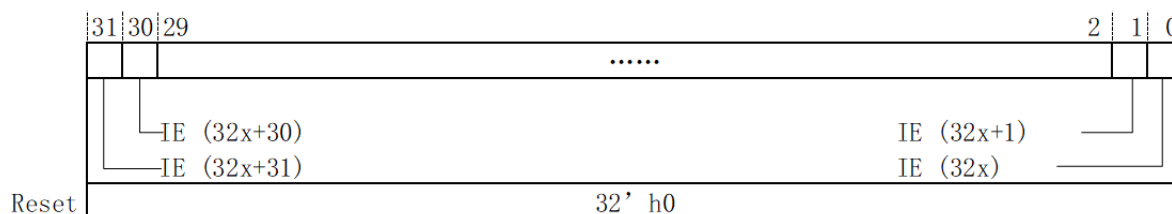


Fig. 10.3: PLIC_IEx register

IE interrupt enable:

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.

10.6 PLIC_CTRL register

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

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 no access to the PLIC_CTRL, PLIC_PRIO, PLIC_IP, and PLIC_IE registers

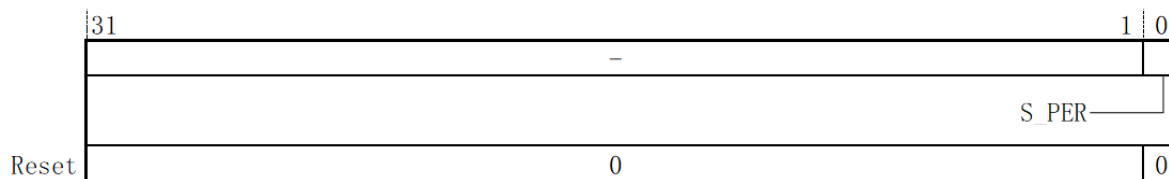


Fig. 10.4: PLIC_CTRL register

and can access only the interrupt threshold register and interrupt claim/complete 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. In S-mode, the CPU has access to all PLIC registers except the PLIC_CTRL register. In U-mode, the CPU has no access to any PLIC registers.

10.7 PLIC_TH register

Each interrupt mode has a corresponding PLIC_TH register. The PLIC initiates an interrupt request to the core only when the interrupt request is valid and the interrupt priority is higher than the interrupt threshold. For more information about the read and write permissions on the register, see the descriptions of the PLIC_CTRL register. Fig. 10.5 shows the bit layout and definition of the register.

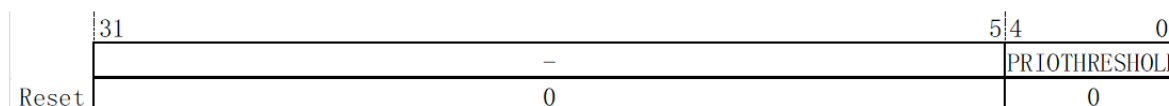


Fig. 10.5: PLIC_TH register

PRIOTHRESHOLD: the priority threshold

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

10.8 PLIC_CLAIM register

Each interrupt mode 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. 10.6.

CLAIM_ID: the interrupt request ID

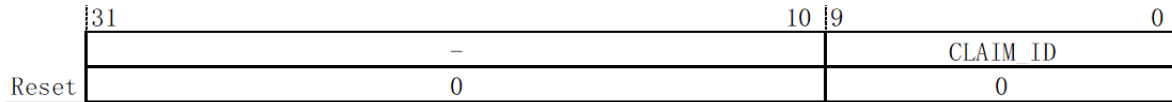


Fig. 10.6: PLIC_CLAIM register

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. For more information about how the PLIC starts the interrupt claim process, see *Interrupt request and response*.

A write operation to the register indicates that the interrupt corresponding to the ID has been handled. The write operation does not update the PLIC_CLAIM register. For more information about how the PLIC starts the interrupt complete process, see *Interrupt completion*.

11.1 Overview

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.

C906 uses a standard 5-wire JTAG debug interface, which is compatible with the RISC-V debug V0.13.2 standard.

The debug interface provides the following features:

- Supports synchronous and asynchronous debug, enabling the CPU to enter the debug mode in extreme conditions.
- Supports software breakpoints.
- Supports multiple hardware 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 in U-mode.
- Enables the CPU to directly access the memory in full-speed running mode.

Debug of C906 is jointly completed by the debug software, debug proxy, debugger, and debug interface. Fig. 11.1 shows the location of the debug interface in the CPU debug environment. 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.

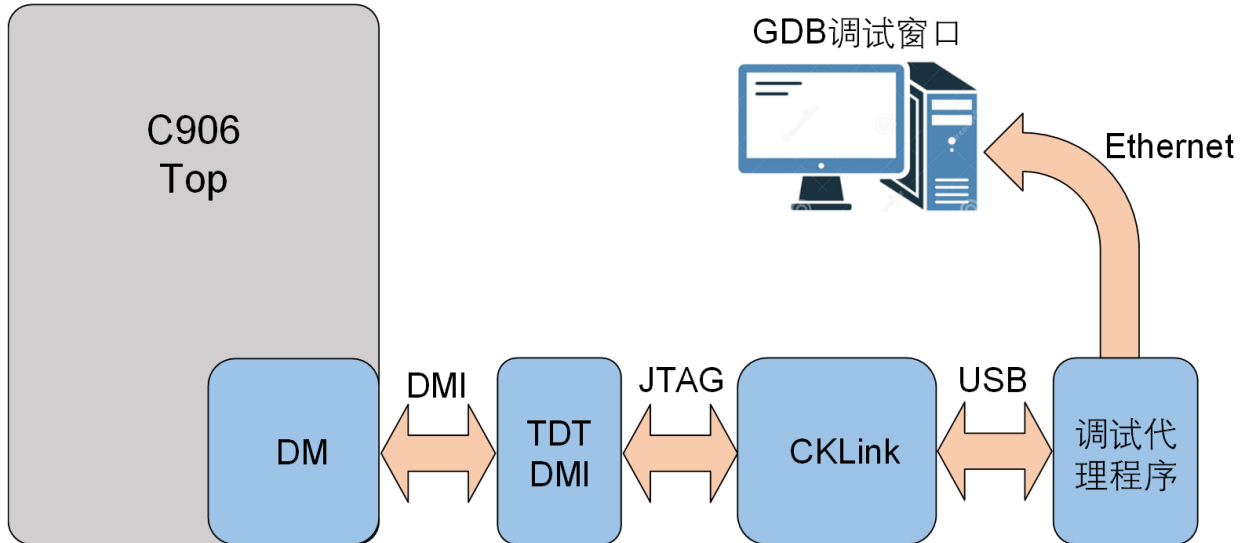


Fig. 11.1: Location of the debug interface in the CPU debug environment

11.2 DM registers

The following table describes registers implemented by the DM module of C906. In addition to registers defined in the standard, C906 provides some extended DM registers in the DMI address encoding domain.

Table 11.1: Mapping and description of DM registers

Address	Register	Description
0x4	DATA0	Abstract DATA 0 register.
0x5	DATA1	Abstract DATA 1 register.
0x10	DMCONTROL	DM control register.
0x11	DMSTATUS	DM status register.
0x15	HAWINDOW	Hart array window register.
0x16	ABSTRACTCS	Abstract control and status register.
0x17	COMMAND	Abstract command register.
0x18	ABSTRACTAUTO	Abstract command auto-execution register.
0x1D	NEXTDM	Next DM base address register.
0x20-0x2F	PROGBUF0-1F	A register with 0-15 programmable buffers.
0x32	DMCS2	DM control and status register 2.
0x38	SBCS	SBA control and status register.
0x39	SBADDRESS0	Bit[31:0] of the SBA address register.
0x3C	SBDATA0	Bit[31:0] of the SBA data register.
0x40	HALTSUM0	HALT overview register 0.
XuanTie extended registers		
0x1F	ITR	Instruction transfer register.
0x70	CUSTOMCS	Custom control and status register.
0x71	CUSTOMCMD	Custom command register.
0x72-0x79	CUSTOMBUF0-7	Custom register with 0 to 7 buffers.
0x7F	COMPID	Component ID register.

ITR Unlike the program buffer, instructions written to the instruction transfer register (ITR) are sent to the LSU for execution. Address fetch and other operations are not required, ensuring robustness.

CUSTOMCS register The CUSTOMCS register describes implementation and extension of XuanTie C906 extended abstract commands. The following table describes the register.

Table 11.2: DM CUSTOMCS register

Bit field	Name	Description
31:29	custcmderr	Indicates the error status when the CUSTOMCMD register is used to run commands. 0: No error occurs. 1: The command is not supported.
28:25	cusbufcnt	Indicates the number of buffers implemented.
24:28	-	-
17	cuscmdbusy	Indicates the status when the CUSTOMCMD register is used to run commands. 0: No command runs. 1: A command is running.
16	-	-
15:0	coredbginfo	Indicates in-core debug resources supported by C906.

CUSTOMCMD register The CUSTOMCMD register is used to run extended commands. If the input command is not supported, CUSTOMCS.custcmderr is set to 1.

Table 11.3: DM CUSTOMCMD register

Bit field	Name	Description
31:24	type	Indicates the custom command type. Values: 0: A command used to initiate an asynchronous debug request to the selected core. 1: A command used for instruction move in the register. 2: A command used for PC sampling. After the command runs, the next PC value of the latest jump instruction executed by the CPU is copied to the DATA register. Others: Reserved.
23: 0	-	-

COMPID register The COMPID register specifies the content and version information implemented by the DM module.

11.3 Resource configuration

C906 provides three debug resource configuration combinations for you to choose.

- Minimum configuration (1) One program buffer with implicit EBREAK implemented; (2) One hardware breakpoint;

- Typical configuration (1) Two program buffers with implicit EBREAK implemented; (2) Three hardware breakpoints; (3) Eight PCFIFO entries for recording historical PC jump streams;
- Maximum configuration (1) Two program buffers with implicit EBREAK implemented; (2) Eight hardware breakpoints that can form a trigger chain; (3) Sixteen PCFIFO entries for recording PC jump streams; (4) Independent debug AXI bus interface for independent memory space access.

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

The hardware performance monitor (HPM) of C906 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 HPM includes the following:

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

12.1 HPM control registers

HPM control registers include the M-mode counter access enable register (MCOUNTEREN), S-mode counter access enable register (SCOUNTEREN), M-mode count inhibit register (MCOUNTINHIBIT), S-mode count inhibit register (SCOUNTINHIBIT), M-mode counter write enable register (MCOUNTERWEN), M-mode performance monitoring control register (MHPMCR), S-mode performance monitoring control register (SH-PMCR), and trigger register.

12.1.1 MCOUNTEREN register

The MCOUNTEREN register determines whether the CPU can access the image registers of the M-mode counters in S-mode.

Table 12.1 describes the MCOUNTEREN register.

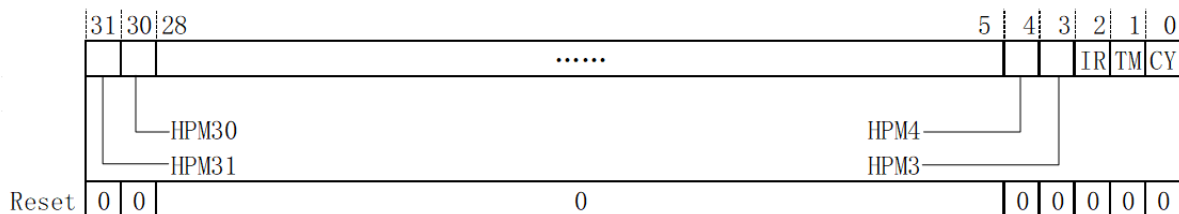


Fig. 12.1: MCOUNTEREN register

Table 12.1: Description of the MCOUNTEREN register

Bit	Read/Write	Name	Description
31:18	Read/Write	N/A	Reserved.
17:3	Read/Write	S-mode access bit of the HPM n	SHPMCOUNTER n register. 0: When the CPU accesses the SHPMCOUNTER n register in S-mode, an illegal instruction exception occurs. 1: The CPU can access the SHPMCOUNTER register in S-mode. n S-mode access bit of the SINSTRET register.
2	Read/Write	IR	0: When the CPU accesses the SINSTRET register in S-mode, an illegal instruction exception occurs. 1: The CPU can access the SINSTRET register in S-mode. S-mode access bit of the TIME register.
1	Read/Write	TM	0: When the CPU accesses the TIME register in S-mode, an illegal instruction exception occurs. 1: The CPU can access the TIME register in S-mode. S-mode access bit of the SCYCLE register.
0	Read/Write	CY	0: When the CPU accesses the SCYCLE register in S-mode, an illegal instruction exception occurs. 1: The CPU can access the SCYCLE register in S-mode.

12.1.2 SCOUNTEREN register

The SCOUNTEREN register determines whether the CPU can access the image registers of the M-mode counters in U-mode.

Table 12.2 describes the SCOUNTEREN register.

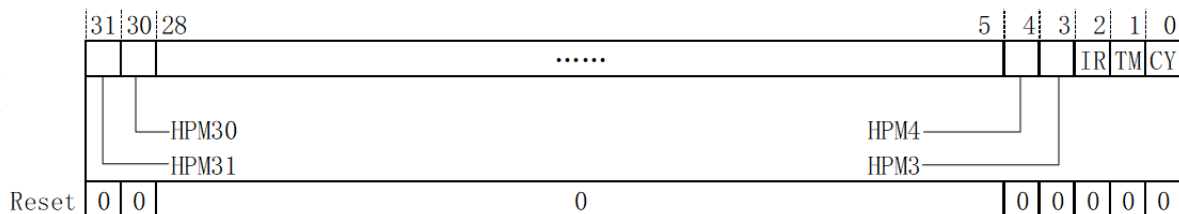


Fig. 12.2: SCOUNTEREN register

Table 12.2: Description of the SCOUNTEREN register

Bit	Read/Write	Description
31:18	Read/Write	Reserved.
17:3	Read/Write	HPMCOUNTER n register. 0: When the CPU accesses the HPMCOUNTER n register in U-mode, an illegal instruction exception occurs. 1: The CPU can access the HPMCOUNTER n register in U-mode.
2	Read/Write	U-mode access bit of the INSTRET register. 0: When the CPU accesses the INSTRET register in U-mode, an illegal instruction exception occurs. 1: The CPU can access the INSTRET register in U-mode.
1	Read/Write	U-mode access bit of the TIME register. 0: When the CPU accesses the TIME register in U-mode, an illegal instruction exception occurs. 1: The CPU can access the TIME register in U-mode.
0	Read/Write	U-mode access bit of the CYCLE register. 0: When the CPU accesses the CYCLE register in U-mode, an illegal instruction exception occurs. 1: The CPU can access the CYCLE register in U-mode.

12.1.3 MCOUNTINHIBIT register

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.

Table 12.3 describes the MCOUNTINHIBIT register. According to the RISC-V standard, MTIME is a timer implemented in the system, and C906 cannot inhibit counting of this timer. Therefore, MCOUNTINHIBIT[1] is not implemented.

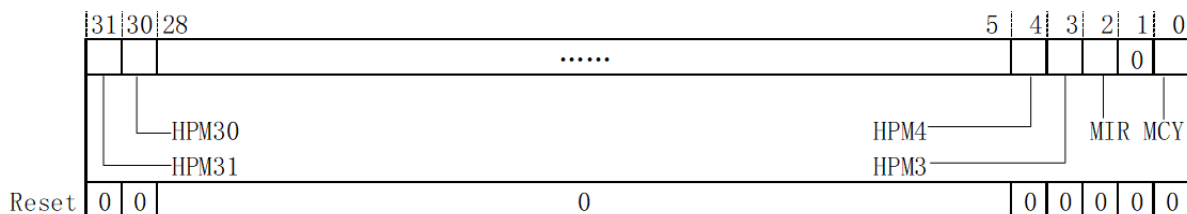


Fig. 12.3: MCOUNTINHIBIT register

Table 12.3: Description of the MCOUNTINHIBIT register

Bit	Read/Write	Name	Description
31:3	Read/Write	Count inhibit bit of the MHPM n	MHPMCOUNTER n 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.

12.1.4 SCOUNTINHIBIT register

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. The SCOUNTINHIBIT register is a C906 extended register.

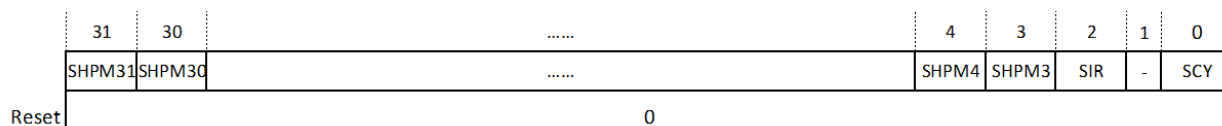


Fig. 12.4: SCOUNTINHIBIT register

Table 12.4 describes the SCOUNTINHIBIT register.

Table 12.4: Description of the SCOUNTINHIBIT register

Bit	Read/Write	Name	Description
31:3	Read/Write	Count inhibit bit of the SHPM _n	SHPMCOUNTER _n 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.

12.1.5 MCOUNTERWEN register

The MCOUNTERWEN register determines whether the CPU can write data to M-mode image event counters in S-mode. This register is a C906 extended register in M-mode.

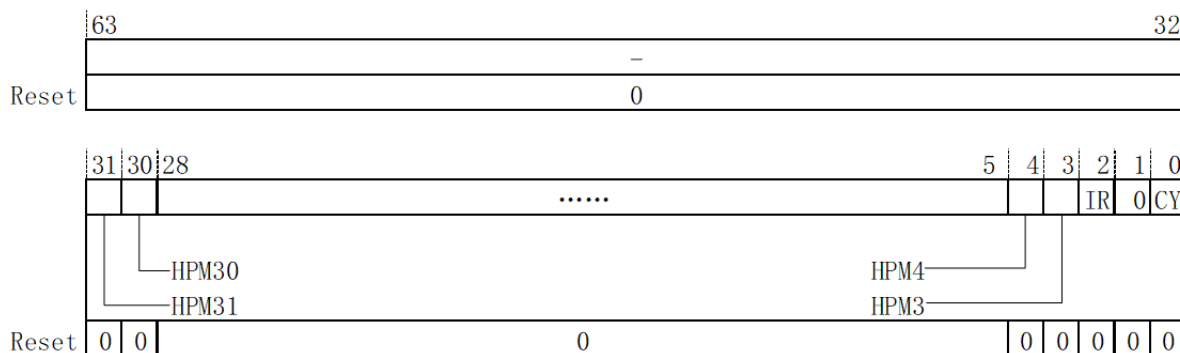


Fig. 12.5: MCOUNTERWEN register

When MCOUNTERWEN.bit[n] is 1, the CPU is allowed to write data to the SHPMCOUNTER_n, SINSTERT, and SCYCLE registers in S-mode. Write operations to these registers will modify the values of the MHPMCOUNTER_n, MINSTRET, and MCYCLE registers.

When MCOUNTERWEN.bit[n] is 0, the CPU is not allowed to write data to the SHPMCOUNTER_n, SINSTERT, or SCYCLE register in S-mode. Otherwise, an illegal instruction exception occurs.

12.1.6 MHPMCR register

The MHPMCR register is used to set the performance monitoring mode and trigger enable. This register is a C906 extended M-mode register.

	63	62	61								32
	TS	SCE		-							
Reset	0	0		0							

	31			14	13	12	11	10	9	2	1	0
				PMDM	-	PMDS	PMDU	-		TME		
Reset	0			0	0	0	0	0	0	0		

Fig. 12.6: MHPMCR register

Table 12.5: Description of the MHPMCR register

Bit	Read/Write	Name	Description
63	Read/Write	TS	Event trigger status bit, which determines whether a performance monitoring event is triggered. We recommend that this bit is not modified by software. 1' b0: No event is triggered. 1' b1: Any performance monitoring event is triggered.
62	Read/Write	SCE	Control enable bit in S-mode. 1' b0: When the CPU accesses the SHPMCR or trigger register in S-mode, an illegal instruction exception occurs. 1' b1: The CPU can access the SHPMCR or trigger register in S-mode.
61:14	-	-	Reserved.
13	Read/Write	PMDM	Count control bit in M-mode. This bit maps to a bit in the MXSTATUS register. 1' b0: The CPU can count in M-mode. 1' b1: The CPU cannot count in M-mode.
11	Read/Write	PMDS	Count control bit in S-mode. This bit maps to a bit in the MXSTATUS register. 1' b0: The CPU can count in S-mode. 1' b1: The CPU cannot count in S-mode.
10	Read/Write	PMDS	Count control bit in U-mode. This bit maps to a bit in the MXSTATUS register. 1' b0: The CPU can count in U-mode. 1' b1: The CPU cannot count in U-mode.
9:2	-	-	Reserved.
1:0	Read/Write	TME	Trigger mode enable bit. 2' b00: No trigger mode. After performance monitoring is enabled, the CPU starts to count. 2' b01: Trigger/Stop trigger mode. In this mode, when the program address is the same as the value of the start trigger register, counting starts. When the program address is the same as the value of the end trigger register, counting stops. 2' b10:
www.t-head.cn			Start/End trigger mode. In this mode, when the program address is within the value range of the start trigger and end trigger registers, the event counters count normally. Otherwise, the event counters do not count. 2' b11: Reserved.

12.1.7 SHPMCR register

The SHPMCR register is used to set the performance monitoring mode and trigger enable in S-mode. This register is a C906 extended register in S-mode.

	63	62															32						
	TS		-																				
Reset	0		0																				
														14	13	12	11	10	9	2		1	0
														PMDM	-	PMDS	PMDU	-	TME				
Reset	0													0	0	0	0	0	0		0		

Fig. 12.7: SHPMCR register

Table 12.6: Description of the SHPMCR register

Bit	Read/Write	Name	Description
63	Read/Write	TS	Event trigger status bit, which determines whether a performance monitoring event is triggered. We recommend that this bit is not modified by software. 1' b0: No event is triggered. 1' b1: Any performance monitoring event is triggered.
61:14	-	-	Reserved.
13	Read-only	PMDM	Count control bit in M-mode. This bit maps to a bit in the SXSTATUS register. 1' b0: The CPU can count in M-mode. 1' b1: The CPU cannot count in M-mode.
11	Read/Write	PMDS	Count control bit in S-mode. This bit maps to a bit in the SXS-TATUS register. 1' b0: The CPU can count in S-mode. 1' b1: The CPU cannot count in S-mode.
10	Read/Write	PMDS	Count control bit in U-mode. This bit maps to a bit in the SXSTATUS register. 1' b0: The CPU can count in U-mode. 1' b1: The CPU cannot count in U-mode.
9:2	-	-	Reserved.
1:0	Read/Write	TME	Trigger mode enable bit. 2' b00: No trigger mode. After performance monitoring is enabled, the CPU starts to count. 2' b01: Trigger/Stop trigger mode. In this mode, when the program address is the same as the value of the start trigger register, counting starts. When the program address is the same as the value of the end trigger register, counting stops. 2' b10: Start/End trigger mode. In this mode, when the program address is within the value range of the start trigger and end trigger registers, the event counters count normally. Otherwise, the event counters do not count. 2' b11: Reserved.

12.1.8 HPMSP register

The HPMSP register is used to set the start and end addresses for event triggering. To facilitate HPMSP register setting in S-mode, each start and end HPMSP register has two register indexes. These registers are C906 extended registers, as described in the following table.

Table 12.7: HPMSP registers

Name	Index	Read/Write	Register	Description
MH-PMSP	0x7F1	MRW	M-mode start trigger register.	Sets the start trigger program address.
MHP-MEP	0x7F2	MRW	M-mode end trigger register.	Sets the end trigger program address.
SHPMSP	0x5CA	SRW	S-mode start trigger register.	Sets the start trigger program address.
SHPMEP	0x5CB	SRW	S-mode end trigger register.	Sets the end trigger program address.

The start and end trigger program addresses are used to fix the address range for event triggering. When the CPU sets this group of registers in M-mode, the start and end trigger program addresses are physical addresses. When the CPU sets this group of registers in S-mode, the start and end trigger program addresses are virtual addresses. MHPMSP and SHPMSP share the same physical register, and MHPMEP and SHPMEP share the same physical register. When the CPU configures MHPMSP/MHPMEP in M-mode SHPMSP/SHPMEP is also updated. When the CPU configures SHPMSP/SHPMEP in S-mode, MHPMSP/MHPMEP is also updated.

12.2 Performance monitoring event select register

Performance monitoring event select register (MHPMEVENT registers 3 to 17) are used to select count events and map to event counter registers. After event index m is written into $HPMEVENTn$ and related control registers are configured, you can read the $HPMCOUNTERn$ register to obtain the count value.

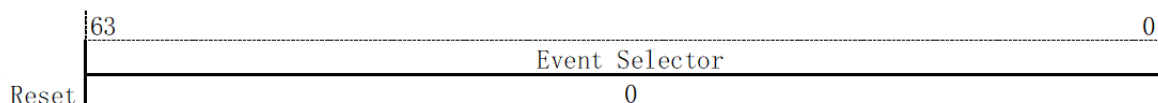


Fig. 12.8: MHPMEVENT registers

Table 12.8 describes the MHPMEVENT registers

Table 12.8: Description of the MHPMEVENT registers

Bit	Read/Write	Name	Description
63:0	Read/Write	Event index.	Indexes performance monitoring events. 0: no event. 0x1 to 0x2A: hardware-implemented performance monitoring events. For more information, see Table 12.9.

Table 12.9 describes the event indexes and events.

Table 12.9: Counter events

Index	Event	Remarks
0x1	L1 ICache Access Counter	
0x2	L1 ICache Miss Counter	
0x3	I-uTLB Miss Counter	
0x4	D-uTLB Miss Counter	
0x5	jTLB Miss Counter	
0x6	Conditional Branch Mispredict Counter	
0x7	Conditional Branch Instruction Counter	
0x8-0xA	Not defined or implemented.	
0xB	Store Instruction Counter	
0xC	L1 DCache Read Access Counter	
0xD	L1 DCache Read Miss Counter	
0xE	L1 DCache Write Access Counter	
0xF	L1 DCache Write Miss Counter	
0x10-0x1C	Not defined or implemented.	
0x1D	ALU Instruction Counter	
0x1E	LOAD & Store Instruction Counter	
0x1F	Vector Instruction Counter	
0x20	CSR Access Instruction Counter	
0x21	Sync Instruction Counter	AMO/LR/SC instruction
0x22	Load & Store Unaligned Access Instruction Counter	
0x23	Interrupt Number Counter	Number of responded interrupts
0x24	Interrupt Off Cycle Counter	When the CPU is in M-mode and MIE is 0, the PLIC arbitrates the time when the interrupt is not responded. When the CPU is in S-mode and Delegation and SIE are 0, the PLIC arbitrates the time when the interrupt is not responded.
0x25	Environment Call Instruction Counter	
0x26	Long Jump Instruction Counter	Number of instructions whose jump distance exceeds 8 MB.
0x27	Frontend Stalled Cycle Counter	Number of stall cycles of the instruction fetch unit (IFU).
0x28	Backend Stalled Cycle Counter	Number of stall cycles of the instruction decoding unit (IDU) and next-level pipeline unit.
0x29	Sync Stalled Cycle Counter	FENCE/FENCE.i/SYNC /SFENCE
0x2A www.t-head.cn	Float Point Instruction Counter 96	Only floating-point operation instructions are included.

12.3 Event counters

Event counters are divided into three groups: M-mode event counters, U-mode event counters, and S-mode event counters (extended in C906). For more information, see [Table 12.10](#).

Table 12.10: M-mode event counter list

Name	Index	Read/Write	Initial value	Description
MCYCLE	0xB00	MRW	0x0	cycle counter
MINSTRET	0xB02	MRW	0x0	instructions-retired counter
MHPMCOUNTER3	0xB03	MRW	0x0	performance-monitoring counter3
MHPMCOUNTER4	0xB04	MRW	0x0	performance-monitoring counter4
...
MHPMCOUNTER31	0xB1F	MRW	0x0	performance-monitoring counter31

[Table 12.11](#) lists the U-mode event counters.

Table 12.11: U-mode event counters

Name	Index	Read/Write	Initial value	Description
CYCLE	0xC00	URO	0x0	cycle counter
TIME	0xC01	URO	0x0	timer
INSTRET	0xC02	URO	0x0	instructions-retired counter
HPMCOUNTER3	0xC03	URO	0x0	performance-monitoring counter3
HPMCOUNTER4	0xC04	URO	0x0	performance-monitoring counter4
...
HPMCOUNTER31	0xC1F	URO	0x0	performance-monitoring counter31

[Table 12.12](#) lists the S-mode event counters.

Table 12.12: S-mode event counters

Name	Index	Read/Write	Initial value	Description
SCYCLE	0x5E0	SRW	0x0	cycle counter
SINSTRET	0x5E2	SRW	0x0	instructions-retired counter
SHPMCOUNTER3	0x5E3	SRW	0x0	performance-monitoring counter3
SHPMCOUNTER4	0x5E4	SRW	0x0	performance-monitoring counter4
...
SHPMCOUNTER31	0x5FF	SRW	0x0	performance-monitoring counter31

The U-mode CYCLE, INSTRET, and HPMCOUNTER_n counters are read-only mappings of corresponding M-mode event counters, and the TIMER counter 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.

12.4 HPM event overflow interrupt

C906 implements the HPM M-mode event counter overflow mark register (MCOUNTEROF) and M-mode event counter overflow interrupt enable register (MCOUNTERINTEN). These registers are readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

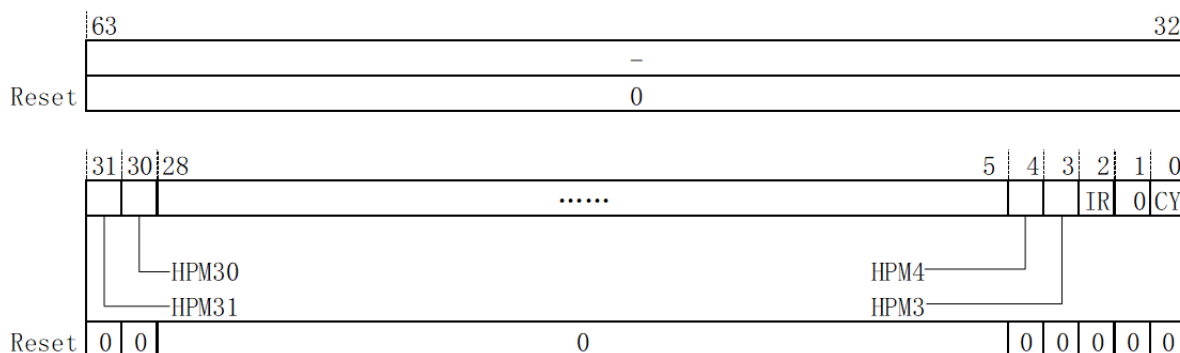


Fig. 12.9: MCOUNTEROF register

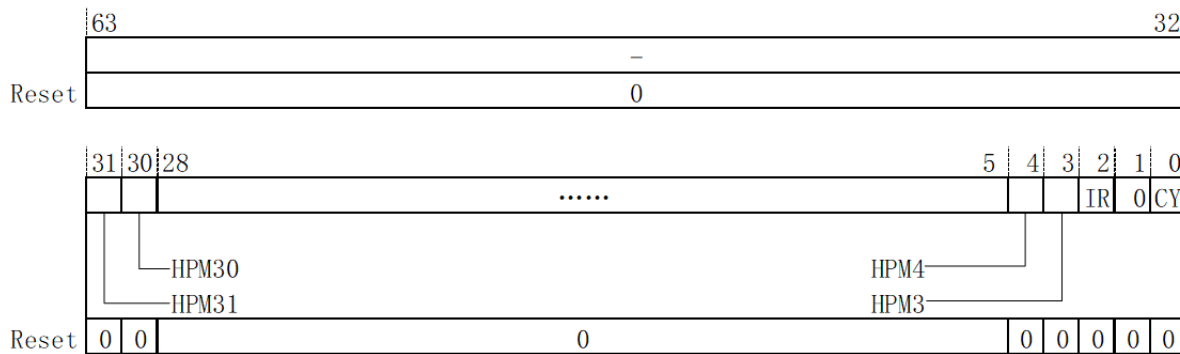


Fig. 12.10: MCOUNTERINTEN register

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 HPM is 17. The interrupts are S-mode interrupts, can be delegated to and handled in the S-mode. For more information, see *Exception handling*.

C906 implements the RV64IMAFDC instruction set package. The following chapters describe each instruction in detail according to different instruction sets.

13.1 Appendix A-1 I instructions

The following describes the RISC-V I instructions implemented by C906. 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*.

13.1.1 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		000		rd		0110011

Syntax:

addi rd, rs1, imm12

Operation: $rd \leftarrow rs1 + \text{sign_extend}(imm12)$ **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		000		rd		0010011

13.1.2 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:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1		000		rd		0011011

13.1.3 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		000		rd		0111011

13.1.4 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		111		rd		0110011

Syntax:

andi rd, rs1, imm12

Operation:

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

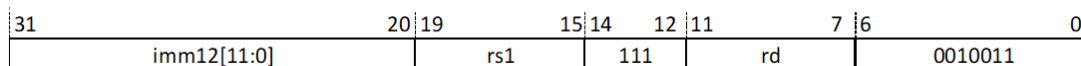
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



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

Syntax:

`auipc rd, imm20`

Operation:

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

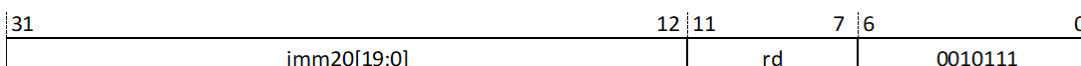
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.6 BEQ: a branch-if-equal instruction

Syntax:

`beq rs1, rs2, label`

Operation:

if ($rs1 == rs2$)

$\text{next pc} = \text{current pc} + \text{sign_extend}(\text{imm12} \ll 1)$

else

$\text{next pc} = \text{current pc} + 4$

Permission:

M mode/S mode/U mode

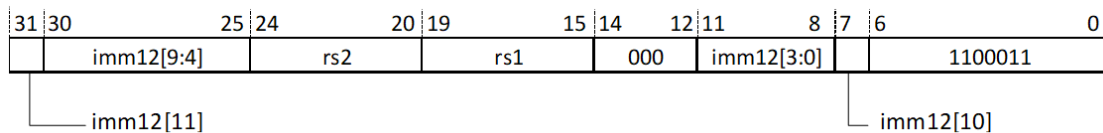
Exception:

None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:

13.1.7 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:

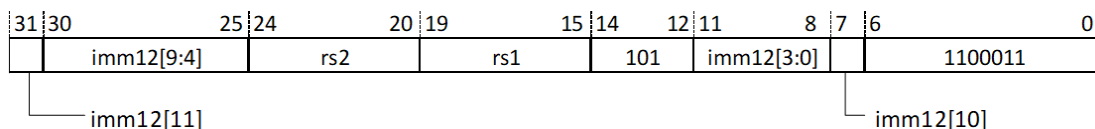
None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:



13.1.8 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:

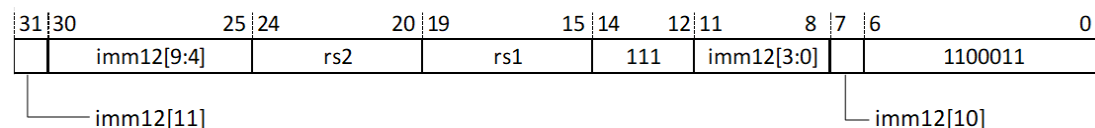
None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:



13.1.9 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:

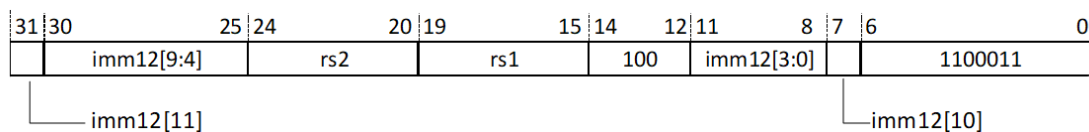
None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:



13.1.10 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:

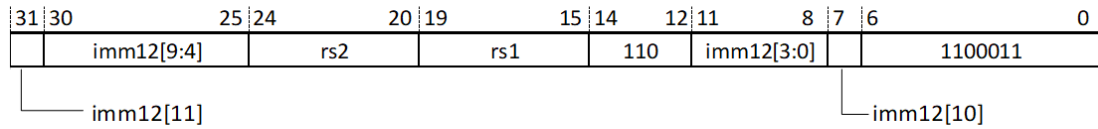
None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:



13.1.11 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:

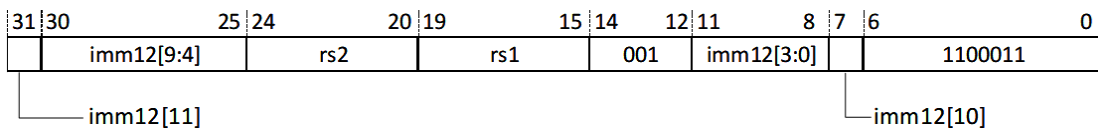
None

Notes:

The compiler calculates immediate 12 based on the label.

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

Instruction format:



13.1.12 CSRRC: a move instruction that clears control registers

Syntax:

csrcc 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	

13.1.13 CSRRCI: a move instruction that clears immediates in control registers**Syntax:**

csrrci rd, csr, imm5

Operation: $rd \leftarrow csr$ $csr \leftarrow csr \& \sim zero_extend(imm5)$ **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.

Instruction format:

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

13.1.14 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

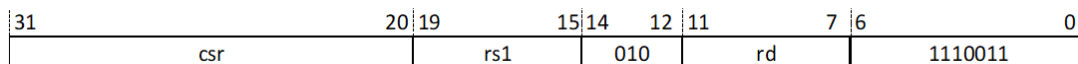
Exception:

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:

13.1.15 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.

Instruction format:

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

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

Syntax:

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.

Instruction format:

31	20	19	15	14	12	11	7	6	0	
csr			rs1		001		rd		1110011	

13.1.17 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.

Instruction format:

31	20	19	15	14	12	11	7	6	0	
csr			imm5		101		rd		1110011	

13.1.18 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	

13.1.19 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	

13.1.20 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	000	00000	00000	00000	0001111	0

13.1.21 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	0000	00000	001	00000	00000	001	00000	00000	00000	0001111	0

13.1.22 JAL: an instruction for directly jumping to a subroutine

Syntax:

```
jal rd, label
```

Operation:

$$\text{next pc} \leftarrow \text{current pc} + \text{sign_extend}(\text{imm20} \ll 1)$$

$$\text{rd} \leftarrow \text{current pc} + 4$$

Permission:

M mode/S mode/U mode

Exception:

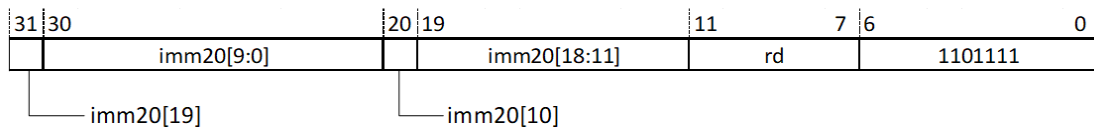
None

Notes:

The compiler calculates immediate 20 based on the label.

The jump range of the instruction is ± 1 MB address space.

Instruction format:



13.1.23 JALR: an instruction for jumping to a subroutine by using an address in a register

Syntax:

```
jalr rd, rs1, imm12
```

Operation:

$$\text{next pc} \leftarrow (\text{rs1} + \text{sign_extend}(\text{imm12})) \& 64' \text{ hfffffffffffffe}$$

$$\text{rd} \leftarrow \text{current pc} + 4$$

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:

31	20	19	15	14	12	11	7	6	0
imm12[11:0]			rs1	000		rd	1100111		

13.1.24 LB: a sign-extended byte load instruction

Syntax:

lb rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← sign_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	000		rd	0000011		

13.1.25 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

13.1.26 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

13.1.27 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])

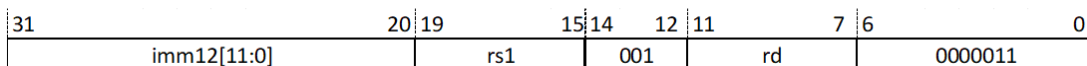
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



13.1.28 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])

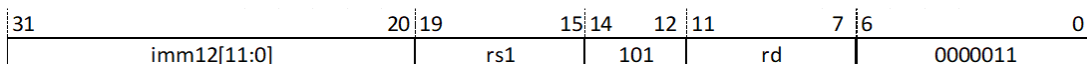
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



13.1.29 LUI: an instruction for loading the immediate in the upper bits

Syntax:

lui rd, imm20

Operation:

rd ← sign_extend(imm20 << 12)

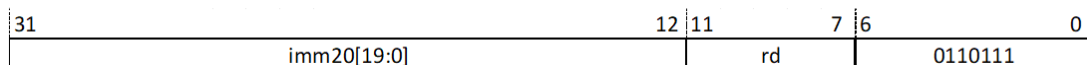
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.30 LW: a sign-extended word load instruction

Syntax:

lw rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← sign_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		010		rd		0000011

13.1.31 LWU: an unsign-extended word load instruction

Syntax:

lwu rd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

rd ← 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		0000011

13.1.32 MRET: an instruction for returning from exceptions in M-mode

Syntax:

mret

Operation:

next pc ← mepc

mstatus.mie ← mstatus.mpie

mstatus.mpie ← 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

13.1.33 OR: a bitwise OR instruction

Syntax:

or rd, rs1, rs2

Operation:

rd ← rs1 | 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		110		rd		0110011

13.1.34 ORI: an immediate bitwise OR instruction

Syntax:

ori rd, rs1, imm12

Operation:

rd ← rs1 | sign_extend(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		110		rd		0010011	

13.1.35 SB: a byte store instruction**Syntax:**

sb rs2, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

mem[:address] ← 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	

13.1.36 SD: a doubleword store instruction**Syntax:**

sd rs2, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

mem[(address+7):address] ← 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

13.1.37 SFENCE.VMA: a virtual memory synchronization instruction**Syntax:**

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

13.1.38 SH: a halfword store instruction**Syntax:**

sh rs2, imm12(rs1)

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{mem}[(\text{address}+1):\text{address}] \leftarrow \text{rs2}[15:0]$$
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	

13.1.39 SLL: a logical left shift instruction**Syntax:**

sll rd, rs1, rs2

Operation:

$$\text{rd} \leftarrow \text{rs1} \ll \text{rs2}[5:0]$$
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	

13.1.40 SLLI: an immediate logical left shift instruction

Syntax:

slli rd, rs1, shamt6

Operation:

$rd \leftarrow rs1 \ll shamt6$

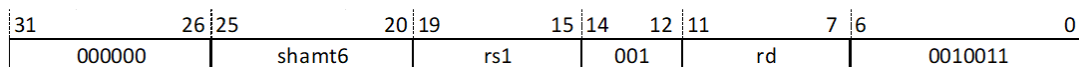
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.41 SLLIW: an immediate logical left shift instruction that operates on the lower 32 bits

Syntax:

slliw rd, rs1, shamt5

Operation:

$tmp[31:0] \leftarrow (rs1[31:0] \ll shamt5)[31:0]$

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

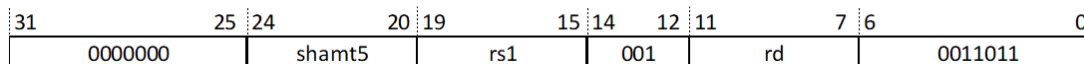
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.42 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		001		rd		0111011

13.1.43 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			rs2		rs1		010		rd		0110011

13.1.44 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	

13.1.45 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	

13.1.46 SLTU: an unsigned set-if-less-than instruction**Syntax:**

```
sltu rd, rs1, rs2
```

Operation:

if ($rs1 < rs2$)

$rd \leftarrow 1$

else

$rd \leftarrow 0$

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		011		rd		0110011

13.1.47 SRA: an arithmetic right shift instruction**Syntax:**

sra rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \gg\gg rs2[5:0]$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0100000			rs2		rs1		101		rd		0110011

13.1.48 SRAI: an immediate arithmetic right shift instruction**Syntax:**

srai rd, rs1, shamt6

Operation:

$rd \leftarrow rs1 \gg\gg shamt6$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:

31	26	25	20	19	15	14	12	11	7	6	0
010000		shamt6			rs1		101		rd		0010011

13.1.49 SLLIW: an immediate arithmetic right shift instruction that operates on the lower 32 bits

Syntax:

sraiw rd, rs1, shamt5

Operation:

$$\text{tmp}[31:0] \leftarrow (\text{rs1}[31:0] \gg \text{shamt5})[31:0]$$

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

M mode/S mode/U mode

Exception:

None

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0100000		shamt5			rs1		101		rd		0011011

13.1.50 SRAW: an arithmetic right shift instruction that operates on the lower 32 bits

Syntax:

sraw rd, rs1, rs2

Operation:

$$\text{tmp} \leftarrow (\text{rs1}[31:0] \gg \text{rs2}[4:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp})$$
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		101		rd		0111011

13.1.51 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:

31	25	24	20	19	15	14	12	11	7	6	0
0001000			00010		00000		000		00000		1110011

13.1.52 SRL: a logical right shift instruction

Syntax:

srl rd, rs1, rs2

Operation:

rd ← rs1 >> rs2[5:0]

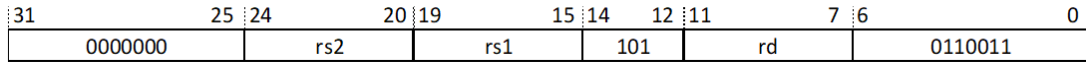
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.53 SRLI: an immediate logical right shift instruction

Syntax:

srl_i rd, rs1, sham_t6

Operation:

rd ← rs1 >> sham_t6

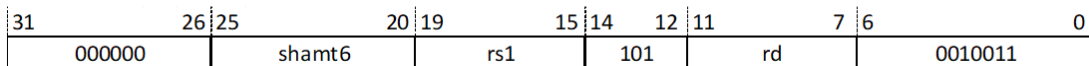
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.54 SRLIW: an immediate logical right shift instruction that operates on the lower 32 bits

Syntax:

srl_iw rd, rs1, sham_t5

Operation:

tmp[31:0] ← (rs1[31:0] >> sham_t5)[31:0]

rd ← sign_extend(tmp[31:0])

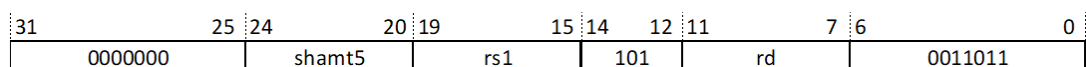
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.1.55 SRLW: a logical right shift instruction that operates on the lower 32 bits

Syntax:

```
srlw rd, rs1, rs2
```

Operation:

$$\text{tmp} \leftarrow (\text{rs1}[31:0] \gg \text{rs2}[4:0])[31:0]$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{tmp})$$

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		101		rd		0111011	

13.1.56 SUB: a signed subtract instruction

Syntax:

```
sub rd, rs1, rs2
```

Operation:

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

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		0110011	

13.1.57 SUBW: a signed subtract instruction that operates on the lower 32 bits

Syntax:

```
subw 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:

31	25	24	20	19	15	14	12	11	7	6	0
0100000			rs2		rs1		000		rd		0111011

13.1.58 SW: a word store instruction**Syntax:**

sw rs2, imm12(rs1)

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{mem}[(\text{address}+3):\text{address}] \leftarrow \text{rs2}[31:0]$$
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

13.1.59 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

13.1.60 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

13.1.61 XORI: an immediate bitwise XOR instruction**Syntax:**

xori rd, rs1, imm12

Operation: $rd \leftarrow rs1 \wedge \text{sign_extend}(\text{imm12})$ **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	

13.2 Appendix A-2 M instructions

The following describes the RISC-V M instructions implemented by C906. The instructions are 32 bits wide and sorted in alphabetic order.

13.2.1 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	

13.2.2 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	

13.2.3 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	

13.2.4 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		rs2	rs1		100		rd		0111011		

13.2.5 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:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		rs2	rs1		000		rd		0110011		

13.2.6 MULH: a signed multiply instruction that extracts the upper bits

Syntax:

mulh rd, rs1, rs2

Operation:

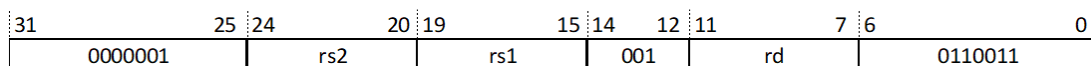
$rd \leftarrow (rs1 * rs2)[127:64]$

Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:**13.2.7 MULHSU: a signed-unsigned multiply instruction that extracts the upper bits****Syntax:**

mulusu rd, rs1, rs2

Operation: $rd \leftarrow (rs1 * rs2)[127:64]$ **Permission:**

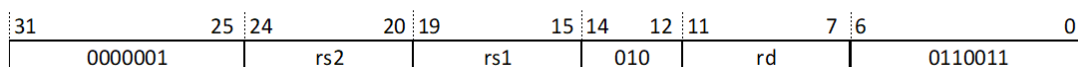
M mode/S mode/U mode

Exception:

None

Notes:

rs1 indicates a signed number, and rs2 indicates an unsigned number.

Instruction format:**13.2.8 MULHU: an unsigned multiply instruction that extracts the upper bits****Syntax:**

mulhu rd, rs1, rs2

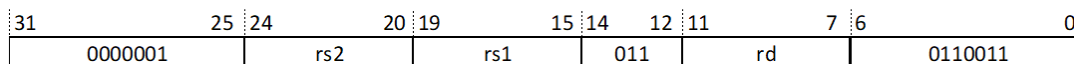
Operation: $rd \leftarrow (rs1 * rs2)[127:64]$ **Permission:**

M mode/S mode/U mode

Exception:

None

Instruction format:



13.2.9 MULW: a signed multiply instruction that operates on the lower 32 bits

Syntax:

mulw rd, rs1, rs2

Operation:

$tmp \leftarrow (rs1[31:0] * rs2[31:0])[31:0]$

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

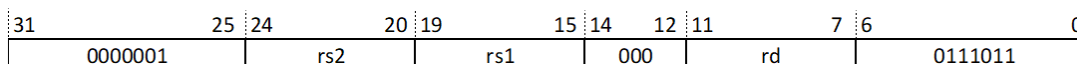
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.2.10 REM: a signed remainder instruction

Syntax:

rem rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \% rs2$

Permission:

M mode/S mode/U mode

Exception:

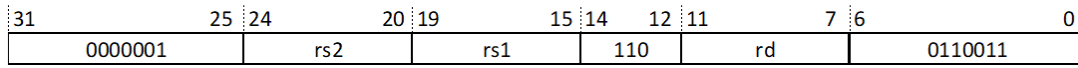
None

Notes:

When the divisor is 0, the remainder operation result is the dividend.

When overflow occurs, the remainder operation result is 0x0.

Instruction format:



13.2.11 REMU: an unsigned remainder instruction

Syntax:

remu rd, rs1, rs2

Operation:

$rd \leftarrow rs1 \% rs2$

Permission:

M mode/S mode/U mode

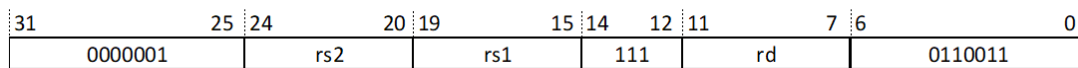
Exception:

None

Notes:

When the divisor is 0, the remainder operation result is the dividend.

Instruction format:



13.2.12 REMUW: an unsigned remainder instruction that operates on the lower 32 bits

Syntax:

remw rd, rs1, rs2

Operation:

$tmp \leftarrow (rs1[31:0] \% rs2[31:0])[31:0]$

$rd \leftarrow sign_extend(tmp)$

Permission:

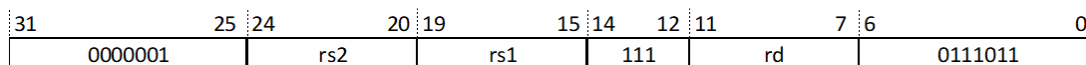
M mode/S mode/U mode

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:**13.2.13 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

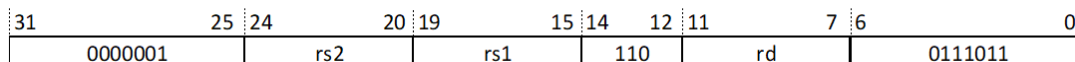
Exception:

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:**13.3 Appendix A-3 A instructions**

The following describes the RISC-V A instructions implemented by C906. The instructions are 32 bits wide and sorted in alphabetic order.

13.3.1 AMOADD.D: an atomic add instruction**Syntax:**

```
amoadd.d.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{mem}[rs1+7:rs1]$$

$$\text{mem}[rs1+7:rs1] \leftarrow \text{mem}[rs1+7:rs1] + rs2$$
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 `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	

13.3.2 AMOADD.W: an atomic add instruction that operates on the lower 32 bits

Syntax:
`amoadd.w.aqrl rd, rs2, (rs1)`
Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+3:rs1])$$

$$\text{mem}[rs1+3:rs1] \leftarrow \text{mem}[rs1+3:rs1] + rs2[31:0]$$

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 `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	

13.3.3 AMOAND.D: an atomic bitwise AND instruction**Syntax:**

$$\text{amoand.d.aqrl rd, rs2, (rs1)}$$
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 `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		

13.3.4 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:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100			aq	rl	rs2		rs1		010		rd	0101111	

13.3.5 AMOMAX.D: an atomic signed MAX instruction**Syntax:**

amomax.d.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}]$

$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \max(\text{mem}[\text{rs1}+7:\text{rs1}], \text{rs2})$

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 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	

13.3.6 AMOMAX.W: an atomic signed MAX instruction that operates on the lower 32 bits

Syntax:

```
amomax.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+3:rs1])$$

$$\text{mem}[rs1+3:rs1] \leftarrow \max(\text{mem}[rs1+3:rs1], rs2[31:0])$$
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, 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		<code>aq</code>	<code>rl</code>	<code>rs2</code>		<code>rs1</code>		010	<code>rd</code>		0101111		

13.3.7 MOMAXU.DA: an atomic unsigned MAX instruction**Syntax:**

```
amomaxu.d.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{mem}[\text{rs1}+7: \text{rs1}]$$

$$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{maxu}(\text{mem}[\text{rs1}+7:\text{rs1}], \text{rs2})$$
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 `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	

13.3.8 AMOMAXU.W: an atomic unsigned MAX instruction that operates on the lower 32 bits.

Syntax:

```
amomaxu.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{maxu}(\text{mem}[\text{rs1}+3:\text{rs1}], \text{rs2}[31:0])$$
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 `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	

13.3.9 AMOMIN.D: an atomic signed MIN instruction**Syntax:**

`amomin.d.aqrl rd, rs2, (rs1)`

Operation:

$rd \leftarrow \text{mem}[rs1+7:rs1]$

$\text{mem}[rs1+7:rs1] \leftarrow \text{minu}(\text{mem}[rs1+7:rs1], rs2)$

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 `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	

13.3.10 AMOMIN.W: an atomic signed MIN instruction that operates on the lower 32 bits

Syntax:

```
amomin.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{minu}(\text{mem}[\text{rs1}+3:\text{rs1}], \text{rs2}[31:0])$$
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 `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.

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
10000			aq	rl	rs2		rs1		010		rd	0101111	

13.3.11 AMOMINU.D: an atomic unsigned MIN instruction

Syntax:

amominu.d.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}]$

$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{minu}(\text{mem}[\text{rs1}+7:\text{rs1}], \text{rs2})$

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 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	

13.3.12 AMOMINU.W: an atomic unsigned MIN instruction that operates on the lower 32 bits

Syntax:

amominu.w.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{minu}(\text{mem}[\text{rs1}+3:\text{rs1}], \text{rs2}[31:0])$

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 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	

13.3.13 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	

Syntax:

```
amoor.w.aqrl rd, rs2, (rs1)
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+3:\text{rs1}] \mid \text{rs2}[31:0]$$
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 `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:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		aq	rl	rs2		rs1		010		rd		0101111	

13.3.14 AMOSWAP.D: an atomic swap instruction**Syntax:**

$$\text{amoswap.d.aqrl rd, rs2, (rs1)}$$
Operation:

$$rd \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}]$$

$$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{rs2}$$
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 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	

13.3.15 AMOSWAP.W: an atomic swap instruction that operates on the lower 32 bits

Syntax:

amoswap.w.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$

$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{rs2}[31:0]$

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	

13.3.16 AMOXOR.D: an atomic bitwise XOR instruction**Syntax:**

amoxor.d.aqrl rd, rs2, (rs1)

Operation:

$rd \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}]$

$\text{mem}[\text{rs1}+7:\text{rs1}] \leftarrow \text{mem}[\text{rs1}+7:\text{rs1}] \hat{=} rs2$

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	

13.3.17 AMOXOR.W: an atomic bitwise XOR instruction that operates on the lower 32 bits

Syntax:

```
amoxor.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 `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	

13.3.18 LR.D: a doubleword load-reserved instruction**Syntax:**

`lr.d.aqrl rd, (rs1)`

Operation:

$rd \leftarrow \text{mem}[\text{rs1}+7: \text{rs1}]$

`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						

13.3.19 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	

13.3.20 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] ← rs2

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.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:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00011		aq	rl	rs2		rs1		011		rd		0101111	

13.3.21 SC.W: a word store-conditional instruction

Syntax:

```
sc.w.aqrl rd, rs2, (rs1)
```

Operation:

if(mem[rs1+3:rs1] is reserved)

$$\text{mem}[\text{rs1}+3:\text{rs1}] \leftarrow \text{rs2}[31:0]$$

$$\text{rd} \leftarrow 0$$

else

$$\text{rd} \leftarrow 1$$

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on atomic instructions

Affected flag bits:

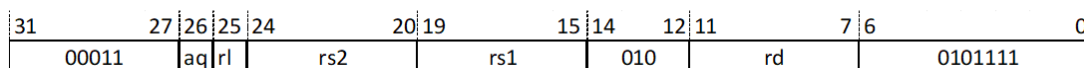
None

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.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:



13.4 Appendix A-4 F instructions

The following describes the RISC-V F instructions implemented by C906. 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.

13.4.1 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), `fcscr`. 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

13.4.2 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

13.4.3 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

13.4.4 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 `fcscr` register. The corresponding assembler instruction is `fcvt.lu.s rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100000			00011		fs1		rm		rd		1010011

13.4.5 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

13.4.6 FCVT.S.LU: an instruction that converts an unsigned long integer into a single-precision floating-point number

Syntax:

```
fcvt.s.lu 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	

13.4.7 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:

$$fd \leftarrow \text{signed_int_convert_to_single}(fs1)$$
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

13.4.8 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

13.4.9 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[31:0] ← single_convert_to_signed_int(fs1)
```

```
rd ← sign_extend(tmp[31:0])
```

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

13.4.10 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

13.4.11 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 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

13.4.12 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

13.4.13 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:

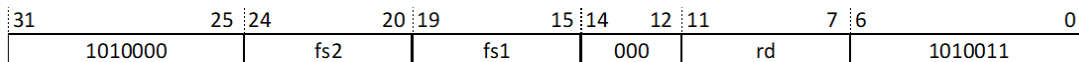
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

13.4.14 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

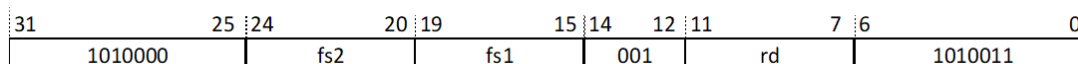
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



13.4.15 FLW: a single-precision floating-point load instruction

Syntax:

flw fd, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

fd[31:0] ← mem[(address+3):address]

fd[63:32] ← 32' hfffffff

Permission:

M mode/S mode/U mode

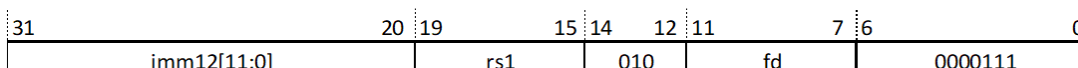
Exception:

Unaligned access, access error, page error, or Illegal instruction.

Affected flag bits:

None

Instruction format:



13.4.16 FMADD.S: a single-precision floating-point multiply-add instruction

Syntax:

fmadd.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 `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

13.4.17 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	

13.4.18 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	

13.4.19 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

13.4.20 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

13.4.21 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

13.4.22 FMV.X.H: a single-precision floating-point read move instruction

Syntax:

fmv.x.w rd, fs1

Operation:

$\text{tmp}[31:0] \leftarrow \text{fs1}[31:0]$

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

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

13.4.23 FNMADD.S: a single-precision floating-point negate-(multiply-add) instruction

Syntax:

fnmadd.s fd, fs1, fs2, fs3, rm

Operation:

$\text{fd} \leftarrow -(\text{fs1} * \text{fs2} + \text{fs3})$

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

13.4.24 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 `fnmsub.s fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fnmsub.s fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fnmsub.s fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fnmsub.s fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fnmsub.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 `fnmsub.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

13.4.25 FSGNJ.S: a single-precision floating-point sign-injection instruction

Syntax:

`fsgnj.s fd, fs1, fs2`

Operation:

`fd[30:0] ← fs1[30:0]`

`fd[31] ← fs2[31]`

`fd[63:32] ← 32' 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
0010000			fs2		fs1		000		fd		1010011

13.4.26 FSGNJN.S: a single-precision floating-point negate sign-injection instruction

Syntax:

fsgnfn.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:

31	25	24	20	19	15	14	12	11	7	6	0
0010000			fs2		fs1		001		fd		1010011

13.4.27 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:

31	25	24	20	19	15	14	12	11	7	6	0
0010000			fs2		fs1		010		fd		1010011

13.4.28 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	

13.4.29 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	

13.4.30 FSW: a single-precision floating-point store instruction

Syntax:

fsw fs2, imm12(rs1)

Operation:

address ← rs1 + sign_extend(imm12)

mem[(address+31):address] ← 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

13.5 Appendix A-5 D instructions

The following describes the RISC-V D instructions implemented by C906. The instructions are 32 bits wide and sorted in alphabetic order.

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.

13.5.1 FADD.D: a double-precision floating-point add instruction

Syntax:

fadd.d fd, fs1, fs2, rm

Operation:

fd ← 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.d fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fadd.d fd, fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fadd.d fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fadd.d fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fadd.d 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.d fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001			fs2		fs1		rm		fd		1010011

13.5.2 FCLASS.D: a double-precision floating-point classify instruction

Syntax:

`fclass.d rd, fs1`

Operation: if (`fs1 = -Inf`)

`rd` ← 64' `h1`

if (`fs1 = -norm`)

`rd` ← 64' `h2`

if (`fs1 = -subnorm`)

```

rd ← 64' h4
if ( fs1 = -zero)
    fd ← 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:

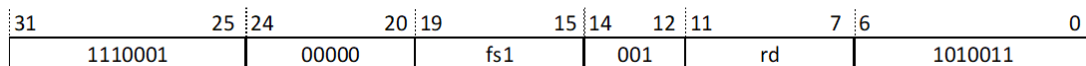
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

13.5.3 FCVT.D.L: an instruction that converts a signed long integer into a double-precision floating-point number

Syntax:

```
fcvt.d.l fd, rs1, rm
```

Operation:

```
fd ← signed_long_convert_to_double(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.d.l fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.d.l fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.d.l fd, rs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.d.l fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.d.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.d.l fd, rs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101001			00010		rs1		rm		fd		1010011

13.5.4 FCVT.D.LU: an instruction that converts an unsigned long integer into a double-precision floating-point number

Syntax:`fcvt.d.lu fd, rs1, rm`**Operation:**`fd ← unsigned_long_convert_to_double(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.d.lu fd, rs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.d.lu fd, rs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.d.lu fd, rs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.d.lu fd, rs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.d.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.d.lu fd, rs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1101001			00011		rs1		rm		fd		1010011

13.5.5 FCVT.D.S: an instruction that converts a single-precision floating-point number into a double-precision floating-point number

Syntax:

`fcvt.d.s fd, fs1`

Operation:

$fd \leftarrow \text{single_convert_to_double}(fs1)$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bits NV

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0		
0100001			00000			fs1		000		fd		1010011	

13.5.6 FCVT.D.W: an instruction that converts a signed integer into a double-precision floating-point number

Syntax:

fcvt.d.w fd, rs1

Operation:fd \leftarrow signed_int_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		
1101001			00000			rs1		000		fd		1010011	

13.5.7 FCVT.D.WU: an instruction that converts an unsigned integer into a double-precision floating-point number

Syntax:

fcvt.d.wu fd, rs1

Operation:fd \leftarrow unsigned_int_convert_to_double(fs1)**Permission:**

M mode/S mode/U mode

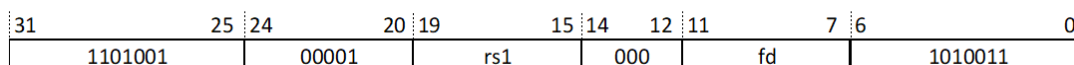
Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:



13.5.8 FCVT.L.D: an instruction that converts a double-precision floating-point number into a signed long integer

Syntax:

`fcvt.l.d rd, fs1, rm`

Operation:

`rd ← double_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.d rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.l.d rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.l.d rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.l.d rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.l.d 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.d rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100001			00010		fs1		rm		rd		1010011

13.5.9 FCVT.LU.D: an instruction that converts a double-precision floating-point number into an unsigned long integer

Syntax:

```
fcvt.lu.d rd, fs1, rm
```

Operation:

```
rd ← double_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.d rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.lu.d rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.lu.d rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.lu.d rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.lu.d 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.d rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100001			00011		fs1		rm		rd		1010011

13.5.10 FCVT.S.D: an instruction that converts a double-precision floating-point number into a single-precision floating-point number

Syntax:

```
fcvt.s.d fd, fs1, rm
```

Operation:

```
fd ← double_convert_to_single(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.s.d fd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.s.d fd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.s.d fd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.s.d fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.s.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.s.d fd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0100000			00001		fs1		rm		fd		1010011

13.5.11 FCVT.W.D: an instruction that converts a double-precision floating-point number into a signed integer

Syntax:

`fcvt.w.d rd, fs1, rm`

Operation:

`tmp ← double_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.d rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.w.d rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.w.d rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.w.d rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.w.d 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.d rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0		
1100001			00000			fs1		rm		rd		1010011	

13.5.12 FCVT.WU.D: an instruction that converts a double-precision floating-point number into an unsigned integer

Syntax:

`fcvt.wu.d rd, fs1, rm`

Operation:

`tmp ← double_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.d rd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fcvt.wu.d rd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fcvt.wu.d rd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fcvt.wu.d rd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fcvt.wu.d 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.d rd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1100001			00001		fs1		rm		rd		1010011

13.5.13 FDIV.D: a double-precision floating-point divide instruction**Syntax:**

`fdiv.d 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.d fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fdiv.d fd fs1, fs2, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fdiv.d fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fdiv.d fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fdiv.d 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.d fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001101			fs2		fs1		rm		fd		1010011

13.5.14 FEQ.D: a double-precision floating-point compare equal instruction**Syntax:**

```
feq.d 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
1010001			fs2		fs1		010		rd		1010011

13.5.15 FLD: a double-precision floating-point load instruction**Syntax:**

```
fld fd, imm12(rs1)
```

Operation:

```
address ← rs1 + sign_extend(imm12)
```

```
fd[63:0] ← mem[(address+7):address]
```

Permission:

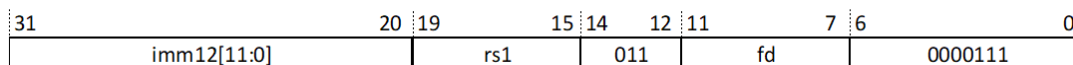
M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Affected flag bits:

None

Instruction format:

13.5.16 FLE.D: a double-precision floating-point compare less than or equal to instruction

Syntax:

fle.d rd, fs1, fs2

Operation:

if(fs1 <= fs2)

rd ← 1

else

rd ← 0

Permission:

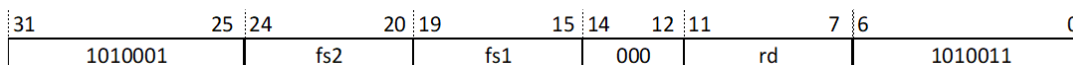
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

13.5.17 FLT.D: a double-precision floating-point compare less than instruction

Syntax:

flt.d rd, fs1, fs2

Operation:

```

if(fs1 < fs2)
    rd ← 1
else
    rd ← 0

```

Permission:

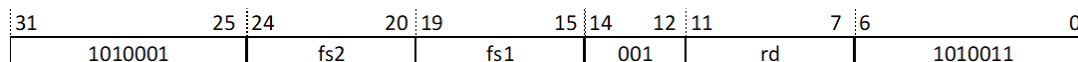
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:**Syntax:**

```
fmadd.d 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 `fmadd.d fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rdn`.

- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rdn`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmadd.d 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.d fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			01	fs2		fs1		rm	fd		1000011		

13.5.18 FMAX.D: a double-precision floating-point MAX instruction**Syntax:**

```
fmax.d 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	
0010101			fs2		fs1		001		fd		1010011	

13.5.19 FMIN.D: a double-precision floating-point MIN instruction**Syntax:**

```
fmin.d fd, fs1, fs2
```

Operation:

```
if(fs1 >= fs2)
```

```
    fd ← fs2
```

```
else
```

```
    fd ← fs1
```

Permission:

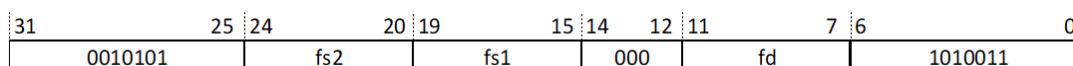
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:

13.5.20 FMSUB.D: a double-precision floating-point multiply-subtract instruction

Syntax:

```
fmsub.d 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.d fd, fs1, fs2, fs3, rne`.

- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmsub.d fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmsub.d fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmsub.d fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmsub.d 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.d fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3		01		fs2		fs1		rm		fd		1000111	

13.5.21 FMUL.D: a double-precision floating-point multiply instruction**Syntax:**

`fmul.d 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.d fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmul.d fd, fs1, fs2, rtz`.

- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmul.d fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmul.d fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmul.d 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. fd, fs1, fs2`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0001001			fs2		fs1		rm		fd		1010011

13.5.22 FMV.D.X: a double-precision floating-point write move instruction**Syntax:**

`fmv.d.x fd, rs1`

Operation:

$fd \leftarrow rs1$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Notes:

This instruction moves data from an integer register to a floating-point register.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
1111001			00000		rs1		000		fd		1010011

13.5.23 FMV.X.D: a double-precision floating-point read move instruction

Syntax:

fmv.x.d rd, fs1

Operation:

rd ← fs1

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Notes:

This instruction moves data from a floating-point register to an integer register.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0		
1110001			00000			fs1		000		rd		1010011	

13.5.24 FNMADD.D: a double-precision floating-point negate-(multiply-add) instruction

Syntax:

fnmadd.d 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.d fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fmadd.d fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fmadd.d 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.d fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			01		fs2		fs1		rm		fd		1001111

13.5.25 FNMSUB.D: a double-precision floating-point negate-(multiply-subtract) instruction

Syntax:

`fnmsub.d 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.d fd, fs1, fs2, fs3, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fnmsub.d fd, fs1, fs2, fs3, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fnmsub.d fd, fs1, fs2, fs3, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fnmsub.d fd, fs1, fs2, fs3, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fnmsub.d 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.d fd, fs1, fs2, fs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
fs3			01		fs2			fs1		rm		fd		1001011	

13.5.26 FSD: a double-precision floating-point store instruction**Syntax:**

```
fsd fs2, imm12(rs1)
```

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm12})$$

$$\text{mem}[(\text{address} + 63) : \text{address}] \leftarrow \text{fs2}[63:0]$$
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		011		imm12[4:0]		0100111	

13.5.27 FSGNJ.D: a double-precision floating-point sign-injection instruction

Syntax:

fsgnj.d fd, fs1, fs2

Operation:

$fd[62:0] \leftarrow fs1[62:0]$

$fd[63] \leftarrow fs2[63]$

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
0010001			fs2		fs1		000		fd		1010011

13.5.28 FSGNJN.D: a double-precision floating-point negate sign-injection instruction

Syntax:

fsgnjn.d fd, fs1, fs2

Operation:

$fd[62:0] \leftarrow fs1[62:0]$

$fd[63] \leftarrow !fs2[63]$

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
0010001			fs2		fs1		001		fd		1010011

13.5.29 FSGNJX.D: a double-precision floating-point XOR sign-injection instruction

Syntax:

fsgnjx.d fd, fs1, fs2

Operation:

$fd[62:0] \leftarrow fs1[62:0]$

$fd[63] \leftarrow fs1[63] \wedge fs2[63]$

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
0010001		fs2		fs1		010		fd		1010011	

13.5.30 FSQRT.D: a double-precision floating-point square-root instruction

Syntax:

fsqrt.d 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.d fd, fs1, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fsqrt.d fd, fs1, rtz`.
- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fsqrt.d fd, fs1, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fsqrt.d fd, fs1, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fsqrt.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 `fsqrt.d fd, fs1`.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0101101			00000		fs1		rm		fd		1010011

13.5.31 FSUB.D: a double-precision floating-point subtract instruction**Syntax:**

`fsub.d fd, fs1, fs2, rm`

Operation:

$fd \leftarrow fs1 - fs2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

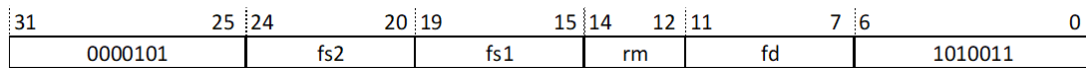
Floating-point status bits NV, OF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fsub.d fd, fs1, fs2, rne`.
- 3' b001: Rounds off to zero. The corresponding assembler instruction is `fsub.d fd, fs1, fs2, rtz`.

- 3' b010: Rounds off to negative infinity. The corresponding assembler instruction is `fsub.d fd, fs1, fs2, rdn`.
- 3' b011: Rounds off to positive infinity. The corresponding assembler instruction is `fsub.d fd, fs1, fs2, rup`.
- 3' b100: Rounds off to the nearest large value. The corresponding assembler instruction is `fsub.d 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.dfd, fs1, fs2`.

Instruction format:

13.6 Appendix A-6 C Instructions

This section describes RISC-V C instructions implemented by C906. The instructions are 16 bits wide and sorted in alphabetic order.

13.6.1 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	

13.6.2 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 != 0

nzimm6 != 0

Instruction format:

15	13	12	11	7	6	2	1	0
000		rs1/rd			nzimm6[4:0]		01	

└─── nzimm6[5]

13.6.3 C.ADDIW: an add immediate instruction that operates on the lower 32 bits

Syntax:

c.addiw rd, imm6

Operation:

$\text{tmp}[31:0] \leftarrow rs1[31:0] + \text{sign_extend}(imm6)$

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

Permission:

M mode/S mode/U mode

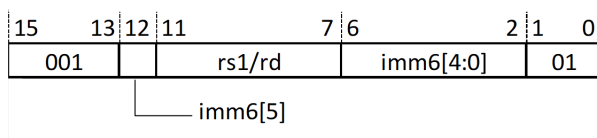
Exception:

None

Notes:

$rs1 = rd \neq 0$

Instruction format:



13.6.4 C.ADDI4SPN: an instruction that adds an immediate scaled by 4 to the stack pointer

Syntax:

`c.addi4spn rd, sp, nzuimm8<<2`

Operation:

$rd \leftarrow sp + \text{zero_extend}(nzuimm8 \ll 2)$

Permission:

M mode/S mode/U mode

Exception:

None

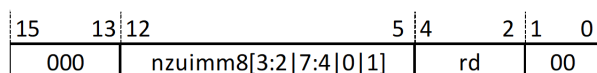
Notes:

$nzuimm8 \neq 0$

Typical rd code registers are:

- 000 x8
- 001 x9
- 010 x10
- 011 x11
- 100 x12
- 101 x13
- 110 x14
- 111 x15

Instruction format:



13.6.5 C.ADDI16SP: an instruction that adds an immediate scaled by 16 to the stack pointer

Syntax:

```
c.addi16sp sp, nzuimm6<<4
```

Operation:

$$sp \leftarrow sp + \text{sign_extend}(nzuimm6 \ll 4)$$

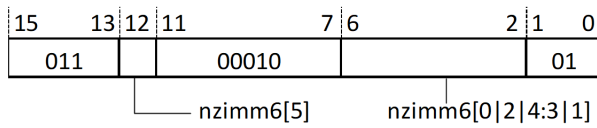
Permission:

M mode/S mode/U mode

Exception:

None

Instruction format:



13.6.6 C.ADDW: a signed add instruction that operates on the lower 32 bits

Syntax:

```
c.addw rd, rs2
```

Operation:

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

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

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						

13.6.7 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						

13.6.8 C.ANDI: an immediate bitwise AND instruction

Syntax:

c.andi rd, imm6

Operation:

$rd \leftarrow rs1 \& \text{sign_extend}(\text{imm6})$

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:

15	13	12	11	10	9	7	6	5	4	3	2	1	0
100		10	rs1/rd	imm6[4:0]				01					

└── imm6[5]

13.6.9 C.BEQZ: a branch-if-equal-to-zero instruction

Syntax:

```
c.beqz rs1, label
```

Operation:

```
if (rs1 == 0)
```

```
    next pc = current pc + sign_ext(imm8<<1);
```

```
else
```

```
    next pc = current pc + 2;
```

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

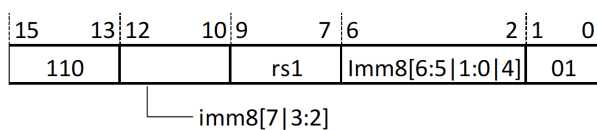
Typical rs1 code registers are:

- 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:



13.6.10 C.BNEZ: a branch-if-not-equal-to-zero instruction

Syntax:

```
c.bnez rs1, label
```

Operation:

```
if (rs1 != 0)
```

```
    next pc = current pc + sign_ext(imm8<<1);
```

```
else
```

```
    next pc = current pc + 2;
```

Permission:

M mode/S mode/U mode

Exception:

None

Notes:

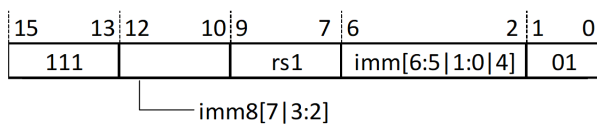
Typical rs1 code registers are:

- 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:



13.6.11 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:

15	13	12	11	7	6	2	1	0
100	1	00000			00000		10	

13.6.12 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:

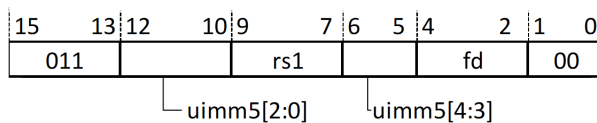
- 000: x8
- 001: x9
- 010: x10
- 011: x11
- 100: x12

- 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:



13.6.13 C.FLDSP: a floating-point doubleword load stack instruction

Syntax:

`c.fldsp fd, uimm6<<3(sp)`

Operation:

$\text{address} \leftarrow \text{sp} + \text{zero_extend}(\text{uimm6} \ll 3)$

$\text{fd} \leftarrow \text{mem}[\text{address}+7:\text{address}]$

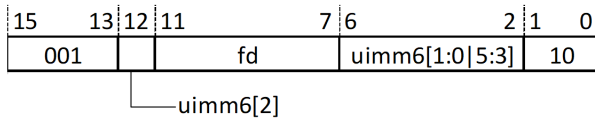
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Instruction format:



13.6.14 C.FSD: a floating-point store doubleword instruction

Syntax:

c.fsd fs2, uimm5<<3(rs1)

Operation:

address \leftarrow rs1 + zero_extend(uimm5<<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

Notes:

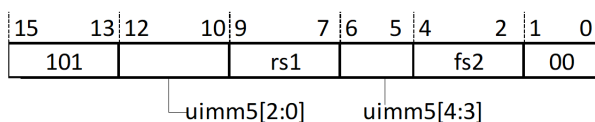
Typical fs1 code registers are:

- 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:**13.6.15 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)

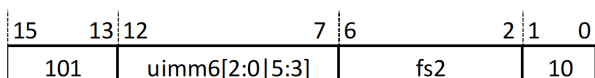
mem[address+7:address] \leftarrow fs2

Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:**13.6.16 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	

13.6.17 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	

13.6.18 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		

13.6.19 C.LD: a load doubleword instruction**Syntax:**

c.ld rd, uimm5<<3(rs1)

Operation:

address \leftarrow rs1 + zero_extend(uimm5<<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

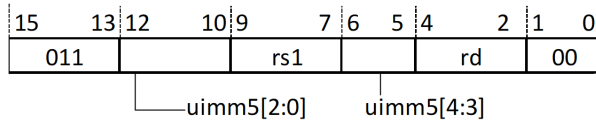
Notes:

Typical rs1/rd code registers are:

- 000: x8
- 001: x9
- 010: x10
- 011: x11

- 100: x12
- 101: x13
- 110: x14
- 111: x15

Instruction format:



13.6.20 C.LDSP: a load doubleword instruction

Syntax:

`c.ldsp rd, uimm6<<3(sp)`

Operation:

$address \leftarrow sp + zero_extend(uimm6 \ll 3)$

$rd \leftarrow mem[address+7:address]$

Permission:

M mode/S mode/U mode

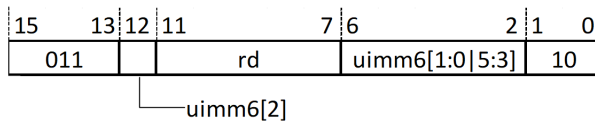
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

$rd \neq 0$

Instruction format:



13.6.21 C.LI: a load immediate instruction

Syntax:

`c.li rd, imm6`

Operation:

$$rd \leftarrow \text{sign_extend}(\text{imm6})$$
Permission:

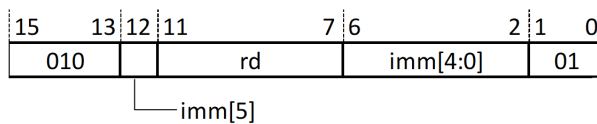
M mode/S mode/U mode

Exception:

None

Notes:

rd != 0

Instruction format:**13.6.22 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

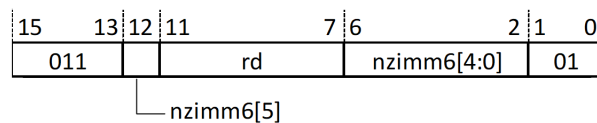
Exception:

None

Notes:

rd != 0.

Nzimm6 != 0.

Instruction format:

13.6.23 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:

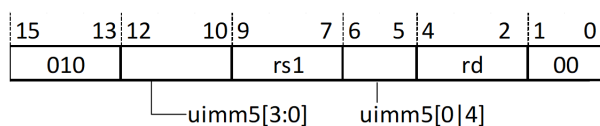
Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

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:



13.6.24 C.LWSP: a load word stack pointer instruction

Syntax:

```
c.lwsp rd, uimm6<<2(sp)
```

Operation:

$$\text{address} \leftarrow \text{sp} + \text{zero_extend}(\text{uimm6} \ll 2)$$

$$\text{tmp}[31:0] \leftarrow \text{mem}[\text{address}+3:\text{address}]$$

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

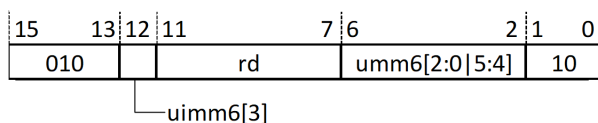
M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on load instructions

Notes:

rd != 0

Instruction format:**13.6.25 C.MV: an instruction that copies the value in rs to rd****Syntax:**

c.mv rd, rs2

Operation:

rd ← rs2;

Permission:

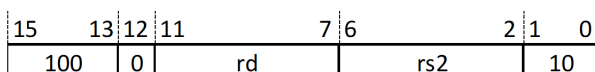
M mode/S mode/U mode

Exception:

None

Notes:

rs2 != 0, rd != 0.

Instruction format:

13.6.26 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	

13.6.27 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		

13.6.28 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:

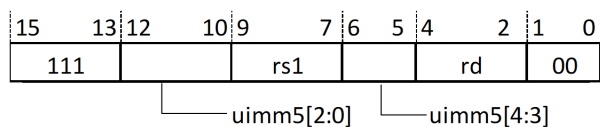
Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

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:



13.6.29 C.SDSP: a store doubleword stack pointer instruction

Syntax:

`c.fsdsp rs2, uimm6<<3(sp)`

Operation:

$\text{address} \leftarrow \text{sp} + \text{zero_extend}(\text{uimm6} \ll 3)$

$\text{mem}[\text{address}+7:\text{address}] \leftarrow \text{rs2}$

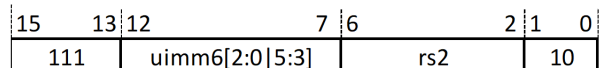
Permission:

M mode/S mode/U mode

Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

Instruction format:



13.6.30 C.SLLI: an immediate logical left shift instruction

Syntax:

`c.slli rd, nzuimm6`

Operation:

$\text{rd} \leftarrow \text{rs1} \ll \text{nzuimm6}$

Permission:

M mode/S mode/U mode

Exception:

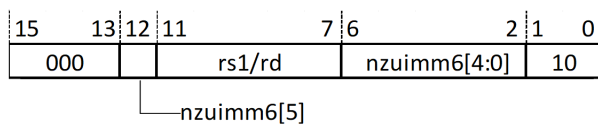
None

Notes:

$\text{rs1} == \text{rd}$

$\text{rd}/\text{rs1} \neq 0, \text{nzuimm6} \neq 0$

Instruction format:



13.6.31 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:

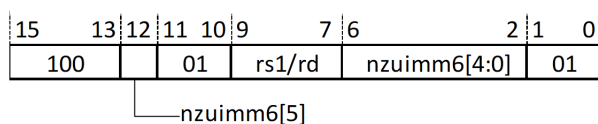
nzuimm6 != 0

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:



13.6.32 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:

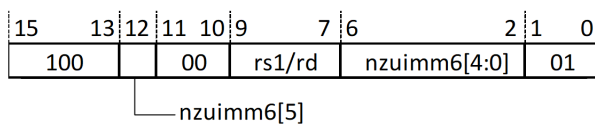
nzuimm6 \neq 0

rs1 \neq 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:



13.6.33 C.SW: a store word instruction

Syntax:

c.sw rs2, uimm5 \ll 2(rs1)

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{uimm5} \ll 2)$$

$$\text{mem}[\text{address}+3:\text{address}] \leftarrow \text{rs2}$$
Permission:

M mode/S mode/U mode

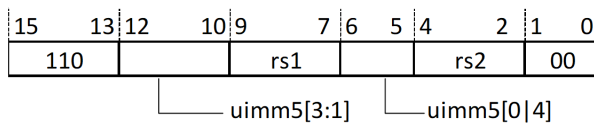
Exception:

Unaligned access exceptions, access error exceptions, and page error exceptions on store instructions

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:**13.6.34 C.SWSP: a store word stack pointer instruction****Syntax:**

$$\text{c.swsp } \text{rs2}, \text{uimm6} \ll 2(\text{sp})$$
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

13.6.35 C.SUB: a signed subtract instruction**Syntax:**

c.sub rd, rs2

Operation:

$rd \leftarrow rs1 - rs2$

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		00	rs2				01		

13.6.36 C.SUBW: a signed subtract instruction that operates on the lower 32 bits

Syntax:

c.subw rd, rs2

Operation:

$tmp[31:0] \leftarrow rs1[31:0] - rs2[31:0]$

$rd \leftarrow sign_extend(tmp)$

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		

13.6.37 C.XOR: a bitwise XOR instruction

Syntax:

c.xor rd, rs2

Operation:

$rd \leftarrow rs1 \hat{=} rs2$

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						

13.7 Appendix A-7 V instructions

All instructions in V instruction sets can be normally executed only after VECTOR_SIMD is configured and mstatus.vs! is set to 2' b00. Otherwise, illegal instruction exceptions will occur. After any vector instruction is executed, mstatus.vs will be reset to 2' b11.

The standard element width (SEW) supported by C906 V instruction sets is 8/16/32. Common integer vector instructions can be executed when the SEW is 8/16/32. However, integer vector extend/shrink instructions can be executed only when the SEW is 8/16. Otherwise, illegal instruction exceptions will occur. Common floating-point vector instructions can be executed only when the SEW is 16/32. Only the floating-point vector extend/shrink swap instructions can be normally executed when SEW is 8. Other floating-point vector extend/shrink instructions can be executed only when SEW is 16. Otherwise, illegal instruction exceptions will occur.

The following describes the RISC-V V instructions implemented by C906. The instructions are 32 bits wide and sorted in alphabetic order.

The abbreviations in this chapter are described as follows:

1. VLEN: vector register width, which is 128 in C906.
2. VLMAX: maximum number of elements in the current vector register group. $VLMAX = VLEN/SEW * LMUL$

13.7.1 VAADD.VI: a vector-immediate instruction that averages integer adds

Syntax:

vaadd.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow (vs2[i] + \text{sign_extend}(imm) + \text{round}) \gg 1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vaadd.vi vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vaadd.vi vd, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100100		vm	vs2	imm		011		vd		1010111		

13.7.2 VAADD.VV: a vector instruction that averages integer adds

Syntax:

```
vaadd.vv vd, vs2, vs1, vm
```

Operation:

$$vd[i] \leftarrow (vs2[i] + vs1[i] + \text{round}) \gg 1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vaadd.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vaadd.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100100	vm	vs2	vs1	000	vd	1010111

13.7.3 VAADD.VX: a vector-scalar instruction that averages integer adds

Syntax:

```
vaadd.vx vd, vs2, rs1, vm
```

Operation:

$$vd[i] \leftarrow (vs2[i] + rs1 + \text{round}) \gg 1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vaadd.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vaadd.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100100			vm	vs2		rs1		100		vd		1010111

13.7.4 VADC.VIM: an integer vector-immediate add-with-carry instruction

Syntax:

`vadc.vim vd, vs2, imm, v0`

Operation:

$vd[i] \leftarrow vs2[i] + \text{sign_extend}(imm) + v0[i].\text{LSB}$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010000		1	vs2	imm		011		vd		1010111		

13.7.5 VADC.VVM: an integer vector add-with-carry instruction

Syntax:

vadc.vvm vd, vs2, vs1, v0

Operation:

$vd[i] \leftarrow vs2[i] + vs1[i] + v0[i].LSB$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010000		1	vs2	vs1		000		vd		1010111		

13.7.6 VADC.VXM: a vector-scalar integer add-with-carry instruction

Syntax:

vadc.vxm vd, vs2, rs1, v0

Operation:

$vd[i] \leftarrow vs2[i] + rs1 + v0[i].LSB$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010000			1	vs2		rs1		100		vd		1010111

13.7.7 VADD.VI: an integer vector-immediate add instruction

Syntax:

vadd.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow vs2[i] + \text{sign_extend}(imm)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vadd.vi vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vadd.vi, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000			vm	vs2		imm		011		vd		1010111

13.7.8 VADD.VV: an integer vector add instruction

Syntax:

vadd.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] + vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vadd.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vadd.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000		vm	vs2	vs1		000		vd		1010111		

13.7.9 VADD.VX: a vector-scalar integer add instruction

Syntax:

`vadd.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] + rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vadd.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vadd.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000		vm	vs2	rs1		100		vd		1010111		

13.7.10 VAMOADD.V: a vector atomic doubleword add instruction

Syntax:

$$\text{vamoadd.v vd, vs2, (rs1), vs3}$$
Operation:

$$\begin{aligned} \text{tmp}[i] &\leftarrow \text{mem}[\text{rs1} + \text{vs2}[i]] \\ \text{mem}[\text{rs1} + \text{vs2}[i]] &\leftarrow \text{tmp}[i] + \text{vs3}[i] \\ \text{if}(\text{vd} \neq \text{x0}) \\ &\quad \text{vd}[i] \leftarrow \text{tmp}[i] \end{aligned}$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamoadd.v x0, vs2, (rs1), vs3, v0.t`.

When *wd* is 0 and *vm* is 1, this instruction is not written back to *vd* or masked. The corresponding assembler instruction is `vamoadd.v x0, vs2, (rs1), vs3`.

When *wd* is 1 and *vm* is 0, this instruction is written back to *vd* and masked. The corresponding assembler instruction is `vamoadd.v vd, vs2, (rs1), vs3, v0.t`.

When *wd* and *vm* are both 1, this instruction is written back to *vd* and is not masked. The corresponding assembler instruction is `vamoadd.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00000		wd	vm	vs2		rs1		111		vs3/vd		0101111	

13.7.11 VAMOADDW.V: a vector atomic word add instruction

Syntax:

$$\text{vamoaddw.v vd, vs2, (rs1), vs3}$$

Operation:

```

tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← tmp[i] + vs3[i]
if(vd != x0)
    vd[i] ← tmp[i]

```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When wd and vm are both 0, this instruction is not written back to vd and is masked. The corresponding assembler instruction is `vamoaddw.v x0, vs2, (rs1), vs3, v0.t`.

When wd is 0 and vm is 1, this instruction is not written back to vd or masked. The corresponding assembler instruction is `vamoaddw.v x0, vs2, (rs1), vs3`.

When wd is 1 and vm is 0, this instruction is written back to vd and masked. The corresponding assembler instruction is `vamoaddw.v vd, vs2, (rs1), vs3, v0.t`.

When wd and vm are both 1, this instruction is written back to vd and is not masked. The corresponding assembler instruction is `vamoaddw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00000		wd	vm	vs2		rs1		110		vs3/vd		0101111	

13.7.12 VAMOANDD.V: a vector atomic doubleword bitwise AND instruction**Syntax:**

```
vamoandd.v vd, vs2, (rs1), vs3
```

Operation:

```

tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← tmp[i] & vs3[i]
if(vd != x0)

```

$$vd[i] \leftarrow tmp[i]$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamoandd.v x0, vs2, (rs1), vs3, v0.t`.

When *wd* is 0 and *vm* is 1, this instruction is not written back to *vd* or masked. The corresponding assembler instruction is `vamoandd.v x0, vs2, (rs1), vs3`.

When *wd* is 1 and *vm* is 0, this instruction is written back to *vd* and is masked. The corresponding assembler instruction is `vamoandd.v vd, vs2, (rs1), vs3, v0.t`.

When *wd* and *vm* are both 1, this instruction is written back to *vd* and is not masked. The corresponding assembler instruction is `vamoandd.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100			<i>wd</i>	<i>vm</i>	<i>vs2</i>		<i>rs1</i>		111		<i>vs3/vd</i>		0101111

13.7.13 VAMOANDW.V: a vector atomic word bitwise AND instruction

Syntax:
`vamoandw.v vd, vs2, (rs1), vs3`
Operation:

$$tmp[i] \leftarrow mem[rs1 + vs2[i]]$$

$$mem[rs1 + vs2[i]] \leftarrow tmp[i] \& vs3[i]$$

$$\text{if}(vd \neq x0)$$

$$vd[i] \leftarrow tmp[i]$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamoandw.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamoandw.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamoandw.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamoandw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01100			<code>wd</code>	<code>vm</code>	<code>vs2</code>		<code>rs1</code>		110		<code>vs3/vd</code>		0101111

13.7.14 VAMOMAXD.V: a vector atomic doubleword signed MAX instruction

Syntax:

`vamomaxd.v vd, vs2, (rs1), vs3`

Operation:

```
tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← max(tmp[i], vs3[i])
if(vd != x0)
    vd[i] ← tmp[i]
```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamomaxd.v x0, vs2, (rs1), vs3, v0.t`.

When *wd* is 0 and *vm* is 1, this instruction is not written back to *vd* or masked. The corresponding assembler instruction is `vamomaxd.v x0, vs2, (rs1), vs3`.

When *wd* is 1 and *vm* is 0, this instruction is written back to *vd* and masked. The corresponding assembler instruction is `vamomaxd.v vd, vs2, (rs1), vs3, v0.t`.

When *wd* and *vm* are both 1, this instruction is written back to *vd* and is not masked. The corresponding assembler instruction is `vamomaxd.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10100		wd	vm	vs2		rs1		111		vs3/vd		0101111	

13.7.15 VAMOMAXW.V: a vector atomic word signed MAX instruction**Syntax:**

`vamomaxw.v vd, vs2, (rs1), vs3`

Operation:

$$\text{tmp}[i] \leftarrow \text{mem}[\text{rs1} + \text{vs2}[i]]$$

$$\text{mem}[\text{rs1} + \text{vs2}[i]] \leftarrow \max(\text{tmp}[i], \text{vs3}[i])$$

if(*vd* != *x0*)

$$\text{vd}[i] \leftarrow \text{tmp}[i]$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamomaxw.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamomaxw.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamomaxw.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamomaxw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10100		wd	vm	vs2		rs1		110		vs3/vd		0101111	

13.7.16 VAMOMAXUD.V: a vector atomic doubleword unsigned MAX instruction

Syntax:

`vamomaxud.v vd, vs2, (rs1), vs3`

Operation:

```
tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← maxu(tmp[i], vs3[i])
if(vd != x0)
    vd[i] ← tmp[i]
```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamomaxud.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamomaxud.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamomaxud.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked.

The corresponding assembler instruction is `vamomaxud.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11100		<code>wd</code>	<code>vm</code>	<code>vs2</code>		<code>rs1</code>		111		<code>vs3/vd</code>		0101111	

13.7.17 VAMOMAXUW.V: a vector atomic word unsigned MAX instruction

Syntax:

`vamomaxuw.v vd, vs2, (rs1), vs3`

Operation:

$\text{tmp}[i] \leftarrow \text{mem}[\text{rs1} + \text{vs2}[i]]$

$\text{mem}[\text{rs1} + \text{vs2}[i]] \leftarrow \text{maxu}(\text{tmp}[i], \text{vs3}[i])$

if(`vd` != `x0`)

$\text{vd}[i] \leftarrow \text{tmp}[i]$

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamomaxuw.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamomaxuw.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamomaxuw.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamomaxuw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11100				wd	vm	vs2		rs1	110		vs3/vd		0101111

13.7.18 VAMOMIND.V: a vector atomic doubleword signed MIN instruction

Syntax:

vamomind.v vd, vs2, (rs1), vs3

Operation:

tmp[i] ← mem[rs1 + vs2[i]]

mem[rs1 + vs2[i]] ← min(tmp[i], vs3[i])

if(vd != x0)

vd[i] ← tmp[i]

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When wd and vm are both 0, this instruction is not written back to vd and is masked. The corresponding assembler instruction is vamomind.v x0, vs2, (rs1), vs3, v0.t.

When wd is 0 and vm is 1, this instruction is not written back to vd or masked. The corresponding assembler instruction is vamomind.v x0, vs2, (rs1), vs3.

When wd is 1 and vm is 0, this instruction is written back to vd and masked. The corresponding assembler instruction is vamomind.v vd, vs2, (rs1), vs3, v0.t.

When wd and vm are both 1, this instruction is written back to vd and is not masked. The corresponding assembler instruction is vamomind.v vd, vs2, (rs1), vs3.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000				wd	vm	vs2		rs1	111		vs3/vd		0101111

13.7.19 VAMOMINW.V: a vector atomic word signed MIN instruction

Syntax:

```
vamominw.v vd, vs2, (rs1), vs3
```

Operation:

```
tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← min(tmp[i], vs3[i])
if(vd != x0)
    vd[i] ← tmp[i]
```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When wd and vm are both 0, this instruction is not written back to vd and is masked. The corresponding assembler instruction is `vamominw.v x0, vs2, (rs1), vs3, v0.t`.

When wd is 0 and vm is 1, this instruction is not written back to vd or masked. The corresponding assembler instruction is `vamominw.v x0, vs2, (rs1), vs3`.

When wd is 1 and vm is 0, this instruction is written back to vd and masked. The corresponding assembler instruction is `vamominw.v vd, vs2, (rs1), vs3, v0.t`.

When wd and vm are both 1, this instruction is written back to vd and is not masked. The corresponding assembler instruction is `vamominw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
10000		wd	vm	vs2		rs1		110		vs3/vd		0101111	

13.7.20 VAMOMINUD.V: a vector atomic doubleword unsigned MIN instruction

Syntax:

```
vamominud.v vd, vs2, (rs1), vs3
```

Operation:

```

tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← minu(tmp[i], vs3[i])
if(vd != x0)
    vd[i] ← tmp[i]

```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When wd and vm are both 0, this instruction is not written back to vd and is masked. The corresponding assembler instruction is `vamominud.v x0, vs2, (rs1), vs3, v0.t`.

When wd is 0 and vm is 1, this instruction is not written back to vd or masked. The corresponding assembler instruction is `vamominud.v x0, vs2, (rs1), vs3`.

When wd is 1 and vm is 0, this instruction is written back to vd and is masked. The corresponding assembler instruction is `vamominud.v vd, vs2, (rs1), vs3, v0.t`.

When wd and vm are both 1, this instruction is written back to vd and is not masked. The corresponding assembler instruction is `vamominud.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
11000		wd	vm	vs2		rs1		111		vs3/vd		0101111	

13.7.21 VAMOMINUW.V: a vector atomic word unsigned MIN instruction**Syntax:**

```
vamominuw.v vd, vs2, (rs1), vs3
```

Operation:

```

tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← minu(tmp[i], vs3[i])
if(vd != x0)

```

$$vd[i] \leftarrow tmp[i]$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamominuw.v x0, vs2, (rs1), vs3, v0.t`.

When *wd* is 0 and *vm* is 1, this instruction is not written back to *vd* or masked. The corresponding assembler instruction is `vamominuw.v x0, vs2, (rs1), vs3`.

When *wd* is 1 and *vm* is 0, this instruction is written back to *vd* and masked. The corresponding assembler instruction is `vamominuw.v vd, vs2, (rs1), vs3, v0.t`.

When *wd* and *vm* are both 1, this instruction is written back to *vd* and is not masked. The corresponding assembler instruction is `vamominuw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
11000		wd	vm	vs2			rs1		110		vs3/vd		0101111	

13.7.22 VAMOORD.V: a vector atomic doubleword bitwise OR instruction**Syntax:**
`vamoord.v vd, vs2, (rs1), vs3`
Operation:

$$tmp[i] \leftarrow mem[rs1 + vs2[i]]$$

$$mem[rs1 + vs2[i]] \leftarrow tmp[i] \mid vs3[i]$$

$$\text{if}(vd \neq x0)$$

$$vd[i] \leftarrow tmp[i]$$
Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When *wd* and *vm* are both 0, this instruction is not written back to *vd* and is masked. The corresponding assembler instruction is `vamoord.v x0, vs2, (rs1), vs3, v0.t`.

When *wd* is 0 and *vm* is 1, this instruction is not written back to *vd* or masked. The corresponding assembler instruction is `vamoord.v x0, vs2, (rs1), vs3`.

When *wd* is 1 and *vm* is 0, this instruction is written back to *vd* and masked. The corresponding assembler instruction is `vamoord.v vd, vs2, (rs1), vs3, v0.t`.

When *wd* and *vm* are both 1, this instruction is written back to *vd* and is not masked. The corresponding assembler instruction is `vamoord.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000			<i>wd</i>	<i>vm</i>	<i>vs2</i>		<i>rs1</i>		111		<i>vs3/vd</i>		0101111

13.7.23 VAMORW.V: a vector atomic word bitwise OR instruction

Syntax:

`vamoorw.v vd, vs2, (rs1), vs3`

Operation:

```
tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← tmp[i] | vs3[i]
if(vd != x0)
    vd[i] ← tmp[i]
```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamoorw.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamoorw.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamoorw.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamoorw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01000		<code>wd</code>	<code>vm</code>	<code>vs2</code>		<code>rs1</code>		110		<code>vs3/vd</code>		0101111	

13.7.24 VAMOSWAPD.V: a vector atomic doubleword swap instruction**Syntax:**

`vamoswapd.v vd, vs2, (rs1), vs3`

Operation:

$\text{tmp}[i] \leftarrow \text{mem}[\text{rs1} + \text{vs2}[i]]$

$\text{mem}[\text{rs1} + \text{vs2}[i]] \leftarrow \text{vs3}[i]$

if(`vd` != `x0`)

$\text{vd}[i] \leftarrow \text{tmp}[i]$

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamoswapd.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamoswapd.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamoswapd.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamoswapd.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00001		<code>wd</code>	<code>vm</code>	<code>vs2</code>	<code>rs1</code>		111		<code>vs3/vd</code>			0101111	

13.7.25 VAMOSWAPW.V: a vector atomic word swap instruction

Syntax:

`vamoswapw.v vd, vs2, (rs1), vs3`

Operation:

```
tmp[i] ← mem[rs1 + vs2[i]]
mem[rs1 + vs2[i]] ← vs3[i]
if(vd != x0)
    vd[i] ← tmp[i]
```

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamoswapw.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamoswapw.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamoswapw.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked.

The corresponding assembler instruction is `vamoswapw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00001		<code>wd</code>	<code>vm</code>	<code>vs2</code>		<code>rs1</code>		110		<code>vs3/vd</code>		0101111	

13.7.26 VAMOXORD.V: a vector atomic doubleword bitwise XOR instruction

Syntax:

`vamoxord.v vd, vs2, (rs1), vs3`

Operation:

$\text{tmp}[i] \leftarrow \text{mem}[\text{rs1} + \text{vs2}[i]]$

$\text{mem}[\text{rs1} + \text{vs2}[i]] \leftarrow \text{tmp}[i] \wedge \text{vs3}[i]$

if(`vd` != `x0`)

$\text{vd}[i] \leftarrow \text{tmp}[i]$

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When `wd` and `vm` are both 0, this instruction is not written back to `vd` and is masked. The corresponding assembler instruction is `vamoxord.v x0, vs2, (rs1), vs3, v0.t`.

When `wd` is 0 and `vm` is 1, this instruction is not written back to `vd` or masked. The corresponding assembler instruction is `vamoxord.v x0, vs2, (rs1), vs3`.

When `wd` is 1 and `vm` is 0, this instruction is written back to `vd` and masked. The corresponding assembler instruction is `vamoxord.v vd, vs2, (rs1), vs3, v0.t`.

When `wd` and `vm` are both 1, this instruction is written back to `vd` and is not masked. The corresponding assembler instruction is `vamoxord.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100				wd	vm	vs2	rs1	111	vs3/vd			0101111	

13.7.27 VAMOXORW.V: a vector atomic doubleword bitwise XOR instruction

Syntax:

vamoxorw.v vd, vs2, (rs1), vs3

Operation:

$tmp[i] \leftarrow mem[rs1 + vs2[i]]$

$mem[rs1 + vs2[i]] \leftarrow tmp[i] \wedge vs3[i]$

if(vd != x0)

$vd[i] \leftarrow tmp[i]$

Permission:

M mode/S mode/U mode

Exception:

Atomic instruction unaligned access, atomic instruction access error, atomic instruction page error, or illegal instruction exceptions.

Affected flag bits:

None.

Notes:

When wd and vm are both 0, this instruction is not written back to vd and is masked. The corresponding assembler instruction is `vamoxorw.v x0, vs2, (rs1), vs3, v0.t`.

When wd is 0 and vm is 1, this instruction is not written back to vd or masked. The corresponding assembler instruction is `vamoxorw.v x0, vs2, (rs1), vs3`.

When wd is 1 and vm is 0, this instruction is written back to vd and masked. The corresponding assembler instruction is `vamoxorw.v vd, vs2, (rs1), vs3, v0.t`.

When wd and vm are both 1, this instruction is written back to vd and is not masked. The corresponding assembler instruction is `vamoxorw.v vd, vs2, (rs1), vs3`.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
00100				wd	vm	vs2	rs1	110	vs3/vd			0101111	

13.7.28 VAND.VI: a vector-immediate bitwise AND instruction

Syntax:

vand.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow vs2[i] \& \text{sign_extend}(imm)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vand.vi vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vand.vi vd, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001001		vm	vs2		imm		011		vd		1010111	

13.7.29 VAND.VV: a vector bitwise AND instruction

Syntax:

vand.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] \& vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vand.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vand.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001001			vm	vs2		vs1		000		vd		1010111

13.7.30 VAND.VX: a vector-scalar bitwise AND instruction**Syntax:**

`vand.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] \& rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vand.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vand.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001001			vm	vs2		rs1		100		vd		1010111

13.7.31 VASUB.VV: a vector integer subtract-average instruction**Syntax:**

`vasub.vv vd, vs2, vs1, vm`

Operation:

$$vd[i] \leftarrow (vs2[i] - vs1[i] + \text{round}) \gg 1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vasub.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vasub.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100110		vm	vs2		vs1		000		vd		1010111	

13.7.32 VASUB.VX: a vector-scalar integer subtract-average instruction**Syntax:**
`vasub.vx vd, vs2, rs1, vm`
Operation:

$$vd[i] \leftarrow (vs2[i] - rs1 + \text{round}) \gg 1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vasub.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vasub.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100110		vm	vs2	rs1		100		vd		1010111		

13.7.33 VCOMPRESS.VM: a vector integer element compress instruction**Syntax:**

vcompress.vm vd, vs2, vs1

Operation:

j=0, vd[i]=0

for(i=0; i<vl; i=i+1){

if(vs1[i]==1)

vd[j] = vs2[i]

j=j++

}

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
010111			1	vs2			vs1			010	vd		1010111	

13.7.34 VDIV.VV: an integer vector signed divide instruction

Syntax:

vdiv.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) / \text{signed}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vdiv.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vdiv.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
100001			vm	vs2			vs1			010	vd		1010111	

13.7.35 VDIV.VX: a vector-scalar integer signed divide instruction

Syntax:

vdiv.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) / \text{signed}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vdiv.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vdiv.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100001	vm	vs2	rs1	110	vd	1010111

13.7.36 VDIVU.VV: an integer vector unsigned divide instruction

Syntax:

`vdivu.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) / \text{unsigned}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

None.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vdivu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vdivu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100000			vm	vs2		vs1		010		vd		1010111

13.7.37 VDIVU.VX: a vector-scalar integer unsigned divide instruction

Syntax:

`vdivu.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) / \text{unsigned}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vdivu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vdivu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100000			vm	vs2		rs1		110		vd		1010111

13.7.38 VEXT.X.V: an integer vector get element instruction

Syntax:

`vext.x.v rd, vs2, rs1`

Operation:

if ($\text{unsigned}(rs1) \geq \text{VLEN}/\text{SEW}$)

$rd = 0$

else

$rd = vs2[\text{unsigned}(rs1)]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
001100			1	vs2			rs1		010		vd		1010111	

13.7.39 VFADD.VF: a vector-scalar floating-point add instruction**Syntax:**

vfadd.vf vd, vs2, fs1, vm

Operation: $rd[i] \leftarrow vs2[i] + fs1$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VF.

- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfadd.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfadd.vf vd, vs2, fs1`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000			vm	vs2		fs1		101		vd		1010111

13.7.40 VFADD.VV: a vector floating-point add instruction

Syntax:

`vfadd.vv vd, vs2, vs1, vm`

Operation:

$rd[i] \leftarrow vs2[i] + vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VV.

- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFADD.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfadd.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfadd.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000			vm	vs2		vs1		001		vd	1010111	

13.7.41 VFCLASS.V: a vector floating-point classify instruction

Syntax: v

`vfclass.v vd, vs2, vm`

Operation:

if (*vs2*[*i*] = -inf)

vd[*i*] ← 64' h1

if (*vs2*[*i*] = -norm)

vd[*i*] ← 64' h2

if (*vs2*[*i*] = -subnorm)

vd[*i*] ← 64' h4

if (*vs2*[*i*] = -zero)

vd[*i*] ← 64' h8

if (*vs2*[*i*] = +zero)

vd[*i*] ← 64' h10

if (*vs2*[*i*] = +subnorm)

vd[*i*] ← 64' h20

if (*vs2*[*i*] = +norm)

vd[*i*] ← 64' h40

if (*vs2*[*i*] = +inf)

vd[*i*] ← 64' h80

```

if ( vs2[i] = sNaN)
    vd[i] ← 64' h100
if ( vs2[i] = qNaN)
    vd[i] ← 64' h200

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfclass.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfclass.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011			vm	vs2		10000		001		vd		1010111

13.7.42 VFCVT.F.X.V: a single-width floating-point/integer type-convert instruction that converts signed integers to floating-point values

Syntax:

```
vfcvt.f.x.v vd, vs2, vm
```

Operation:

```
vd [i] ← integer_convert_to_float(vs2[i])
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NX and OF

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.X.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.X.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.X.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfcvt.f.x.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfcvt.f.x.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		<i>vm</i>	<i>vs2</i>		00011		001		<i>vd</i>		1010111	

13.7.43 VFCVT.F.XU.V: a single-width floating-point/integer type-convert instruction that converts unsigned vector integers to floating-point values

Syntax:

`vfcvt.f.xu.v vd, vs2, vm`

Operation:

$vd[i] \leftarrow \text{unsigned_integer_convert_to_float}(vs2[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NX and OF

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.XU.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.XU.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.F.XU.V.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfcvt.f.xu.v vd, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfcvt.f.xu.v vd, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100010	vm	vs2	00010	001	vd	1010111

13.7.44 VFCVT.X.F.V: a single-width floating-point/integer type-convert instruction that converts vector floating-point values to signed integers

Syntax:

`vfcvt.x.f.v vd, vs2, vm`

Operation:

`vd [i] ← float_convert_to_signed_integer(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.X.F.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.X.F.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.X.F.V.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfcvt.x.f.v vd, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfcvt.x.f.v vd, vs2`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2	00001		001		vd		1010111		

13.7.45 VFCVT.XU.F.V: a single-width floating-point/integer type-convert instruction that converts vector floating-point values to unsigned integers

Syntax:

`vfcvt.xu.f.v vd, vs2, vm`

Operation:

$vd[i] \leftarrow \text{float_convert_to_unsigned_integer}(vs2[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.XU.F.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.XU.F.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFCVT.XU.F.V.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfcvt.xu.f.v vd, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfcvt.xu.f.v vd, vs2`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		00000		001		vd		1010111

13.7.46 VFDIV.VF: a vector-scalar floating-point divide instruction

Syntax:

`vfdiv.vf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] / fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfdiv.vf vd, vs2, fs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfdiv.vf vd, vs2, fs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100001			vm	vs2		fs1		101		vd		1010111

13.7.47 VFDIV.VV: a vector floating-point divide instruction

Syntax:

vfdiv.vv vd, vs2, vs1, vm

Operation:

$rd[i] \leftarrow vs2[i] / vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFDIV.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfdiv.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfdiv.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100000		vm	vs2	vs1		001		rd		1010111		

13.7.48 VFMACC.VF: an FP multiply-accumulate instruction that overwrites addends

Syntax:

vfmacc.vf vd, fs1, vs2, vm

Operation:

$vd[i] \leftarrow fs1 * vs2[i] + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmac.vf vd, fs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfmac.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101100	vm	vs2	fs1	101	vd	1010111

13.7.49 VFMAcc.VV: an FP multiply-accumulate instruction that overwrites addends

Syntax:

`vfmac.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow vs1[i] * vs2[i] + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMAcc.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmaccc.vv vd, vs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfmaccc.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101100	vm	vs2	vs1	001	rd	1010111

13.7.50 VFMAcc.VF: an FP multiply-add instruction that overwrites multiplicands

Syntax:

`vfmaccc.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow fs1 * vd[i] + vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vmadd.vf vd, fs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vmadd.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101000	vm	vs2	fs1	101	vd	1010111

13.7.51 VFMADD.VV: an FP multiply-add instruction that overwrites multiplicands

Syntax:

`vmadd.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow vs1[i] * vd[i] + vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMADD.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmax.vf vd, vs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfmax.vf vd, vs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101000	vm	vs2	vs1	101	vd	1010111

13.7.52 VFMAX.VF: a vector-scalar floating-point MAX instruction

Syntax:

`vfmax.vf vd, vs2, fs1, vm`

Operation:

if(`fs1 >= vs2[i]`)

`vd[i] ← fs1`

else

`vd[i] ← vs2[i]`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmax.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmax.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000110			vm	vs2		fs1		101		vd		1010111

13.7.53 VFMAX.VV: a vector floating-point MAX instruction**Syntax:**`vfmax.vv vd, vs2, vs1, vm`**Operation:** $\text{if}(vs1[i] \geq vs2[i])$ $vd[i] \leftarrow vs1[i]$

else

 $vd[i] \leftarrow vs2[i]$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmax.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmmax.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
000110			vm	vs2			vs1			001		vd	1010111	

13.7.54 VFMERGE.VFM: a vector floating-point element select instruction

Syntax: `vfmerge.vfm vd, vs2, fs1, v0`

Operation: `if(v0[i].LSB == 1)`

`vd[i] ← fs1`

else

`vd[i] ← vs2[i]`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
010111			0	vs2			fs1			101		vd	1010111	

13.7.55 VFMIN.VF: a vector-scalar floating-point MIN instruction

Syntax:

`vfmin.vf vd, vs2, fs1, vm`

Operation:

`if(fs1 <= vs2[i])`

`vd[i] ← fs1`

else

`vd[i] ← vs2[i]`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmin.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmin.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
000100			vm	vs2			fs1		101		vd	1010111	

13.7.56 VFMIN.VV: a vector floating-point MIN instruction**Syntax:**
`vfmin.vv vd, vs2, vs1, vm`
Operation:

$$\text{if}(vs1[i] \leq vs2[i])$$

$$vd[i] \leftarrow vs1[i]$$

$$\text{else}$$

$$vd[i] \leftarrow vs2[i]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmmin.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmmin.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000100		vm	vs2		vs1		001		vd		1010111	

13.7.57 VFMSAC.VF: a vector-scalar floating-point multiply-sub instruction that overwrites subtrahends

Syntax:

`vfmsac.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow fs1 * vs2[i] - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VF.

- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsac.vf vd, fs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsac.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101110			vm	vs2		fs1		101		vd		1010111

13.7.58 VFMSAC.VV: a vector floating-point multiply-sub instruction that overwrites subtrahends

Syntax:

`vfmsac.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow vs1[i] * vs2[i] - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VV.

- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSAC.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsac.vv vd, vs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsac.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101110		<i>vm</i>	<i>vs2</i>		<i>vs1</i>		001		<i>vd</i>		1010111	

13.7.59 VFMSUB.VF: a vector-scalar floating-point multiply-sub instruction that overwrites multiplicands

Syntax:

`vfmsub.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow fs1 * vd[i] - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.

- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSUB.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSUB.VF.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsub.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsub.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101010		vm	vs2		fs1		101	vd		1010111		

13.7.60 VFMSUB.VV: a vector floating-point multiply-sub instruction that overwrites multiplicands

Syntax:

`vfmsub.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow (vs1[i] * vs2[i]) - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.

- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSUB.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMSUB.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsub.vv vd, vs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsub.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101010	vm			vs2			vs1			001	vd	1010111

13.7.61 VFMUL.VF: a vector-scalar floating-point multiply instruction

Syntax:

`vfmul.vf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] * fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Note: Dynamically rounds off based on the *rm* bit of the floating-point register FCSR.

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.

- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMUL.VF.
- 3' b110: invalid value. When it is set to this value, an illegal instruction exception occurs for the execution of VFMUL.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMUL.VF.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmul.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmul.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100100		vm	vs2		fs1		101		vd		1010111	

13.7.62 VFMUL.VV: a vector floating-point multiply instruction

Syntax:

`vfmul.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] * vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.

- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMUL.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMUL.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFMUL.VV.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmul.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmul.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100100	vm	vs2	vs1	001	vd	1010111

13.7.63 VFMV.F.S: an instruction that moves element 0 of a vector to a floating-point scalar

Syntax:

`vfmv.f.s rd, vs2`

Operation:

`rd = vs2[0]`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
001100	1	vs2	00000	001	rd	1010111

13.7.64 VFMV.S.F: an instruction that moves a floating-point scalar to element 0 of a vector

Syntax:

`vfmv.s.f vd, fs1`

Operation:

`vd[0] = fs1`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001101	1	00000	fs1	101	vd	1010111						

13.7.65 VFMV.V.F: an instruction that moves a floating-point scalar to a vector

Syntax:

`vfmv.v.f vd, fs1`

Operation:

`vd[i] = fs1`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111	1	vs2	fs1	101	vd	1010111						

13.7.66 VFNCVT.F.F.V: a vector floating-point reduction instruction

Syntax:

`vfncvt.f.f.v vd, vs2, vm`

Operation:

`vd [i] ← double_width_float_convert_to_float(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2 \times \text{SEW}$ for `vs2` and `SEW` for `vd`.

Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.F.V`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.F.V`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.F.V`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfncvt.f.f.v vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfncvt.f.f.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		10100		001		vd		1010111

13.7.67 VFNCVT.F.XU.V: a vector reduction-type instruction that converts unsigned integers to floating-point values

Syntax:

`vfncvt.f.xu.v vd, vs2, vm`

Operation:

`vd [i] ← double_width_unsigned_integer_convert_to_float(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NX and OF

Notes:

The element width is 2*SEW for vs2 and SEW for vd.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.F.XU.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.F.XU.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.F.XU.V.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfncvt.f.xu.v vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfnvcvt.f.xu.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2		10010		001		vd		1010111	

13.7.68 VFNCVT.F.X.V: a vector reduction-type instruction that converts signed integers to floating-point values

Syntax:

`vfnvcvt.f.x.v vd, vs2, vm`

Operation:

`vd [i] ← double_width_signed_integer_convert_to_float(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NX and OF

Notes:

The element width is $2 \times \text{SEW}$ for `vs2` and `SEW` for `vd`.

Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.X.V`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.X.V`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.F.X.V`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnvcvt.f.x.v vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfnvcvt.f.x.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		10011		001		vd		1010111

13.7.69 VFNCVT.X.F.V: a vector reduction-type instruction that converts floating-point values to signed integers

Syntax:

`vfnvcvt.x.f.v vd, vs2, vm`

Operation:

`vd [i] ← double_width_float_convert_to_signed_integer(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

The element width is $2 \times \text{SEW}$ for `vs2` and `SEW` for `vd`.

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.X.F.V`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNCVT.X.F.V`.

- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.X.F.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnvcvt.x.f.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnvcvt.x.f.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2		10001		001		vd		1010111	

13.7.70 VFNCVT.XU.F.V: a vector reduction-type instruction that converts floating-point values to unsigned integers

Syntax:

`vfnvcvt.xu.f.v vd, vs2, vm`

Operation:

`vd [i] ← double_width_float_convert_to_unsigned_integer(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

The element width is 2*SEW for *vs2* and SEW for *vd*.

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.XU.F.V.

- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.XU.F.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNCVT.XU.F.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnvcvt.xu.f.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnvcvt.xu.f.v vd, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100010	<i>vm</i>	<i>vs2</i>	10000	001	<i>vd</i>	1010111

13.7.71 VFNMACC.VF: a vector-scalar floating-point negate-(multiply-add) instruction that overwrites subtrahends

Syntax:

`vfnmacc.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(fs1 * vs2[i]) - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.

- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VF.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnmacc.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfnmacc.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101		vm	vs2		fs1		101		vd		1010111	

13.7.72 VFNMACC.VV: a vector floating-point negate-(multiply-add) instruction that overwrites subtrahends

Syntax:

`vfnmacc.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(vs1[i] * vs2[i]) - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.

- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMACC.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnmacc.vv vd, vs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnmacc.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101101	vm	vs2	vs1	001	vd	1010111

13.7.73 VFNMADD.VF: a vector-scalar floating-point negate-(multiply-add) instruction that overwrites multiplicands

Syntax:

`vfnmadd.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(fs1 * vd[i]) - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.

- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnmadd.vf vd, fs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnmadd.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101001		vm	vs2		fs1		101		rd		1010111	

13.7.74 VFNMADD.VV: a vector floating-point negate-(multiply-add) instruction that overwrites multiplicands

Syntax:

`vfnmadd.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(vs1[i] * vd[i]) - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.

- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMADD.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnmadd.vv vd, vs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnmadd.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101001			vm	vs2		fs1		001		vd		1010111

13.7.75 VFNMSAC.VF: a vector-scalar floating-point negate-(multiply-sub) instruction that overwrites minuends

Syntax:

`vfnmsac.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(fs1 * vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.

- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VF.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsac.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsac.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101111		vm	vs2	fs1		101		vd		1010111		

13.7.76 VFNMSAC.VV: a vector floating-point negate-(multiply-sub) instruction that overwrites minuends

Syntax:

`vfmsac.vv vd, vs1, vs2, vm`

Operation:

$$vd[i] \leftarrow -(vs1[i] * vs2[i]) + vd[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSAC.VV.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfmsac.vv vd, vs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsac.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101111	vm	vs2	vs1	001	vd	1010111

13.7.77 VFNMSUB.VF: a vector-scalar floating-point negate-(multiply-sub) instruction that overwrites multiplicands

Syntax:

`vfmsub.vf vd, fs1, vs2, vm`

Operation:

$$vd[i] \leftarrow -(fs1 * vd[i]) + vs2[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VF.

If the value of *vm* is 1, the instruction will not be masked. The, corresponding assembler instruction is `vfmsub.vf vd, fs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfmsub.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
101011	vm	vs2	fs1	101	vd	1010111

13.7.78 VFNMSUB.VV: a vector floating-point negate-(multiply-sub) instruction that overwrites multiplicands

Syntax:

`vfmsub.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(vs1[i] * vd[i]) + vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, UF, and IX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFNMSUB.VV.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfnmsub.vv vd, vs1, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfnmsub.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011		vm	vs2		vs1		001		vd		101011	

13.7.79 VFRDIV.VF: a scalar-vector floating-point divide instruction**Syntax:**

`vfrdiv.vf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow fs1 / vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFRDIV.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFRDIV.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFRDIV.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfrdiv.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfrdiv.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100001			<i>vm</i>	<i>vs2</i>		<i>fs1</i>		101		<i>vd</i>		1010111

13.7.80 VFREDMAX.VS: a vector floating-point reduction instruction that obtains the MAX value

Syntax:

`vfredmax.vs. vd, vs2, vs1, vm`

Operation:

`tmp = vs1[0]`

`for(i=0; i<vl; i++) {`

`tmp = max(tmp, vs2[i])`

`}`

`vd[0]= tmp`

`vd[VLEN/SEW-1:1] = 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

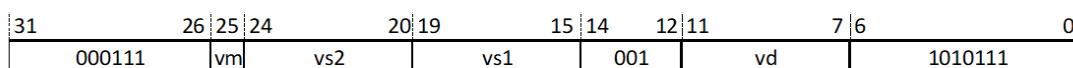
Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfredmax.vs.vf vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfredmax.vs vd, vs2, vs1, v0.t`.

Instruction format:

13.7.81 VFREDMIN.VS: a vector floating-point reduction instruction that obtains the MIN value

Syntax:

```
vfredmin.vs. vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
for( i=0; i < vl; i++) {
    tmp = min(vs2[i])
}
vd[0] = tmp
vd[VLEN/SEW-1:1] = 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfredmin.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfredmin.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000101			vm	vs2		vs1		001		vd		1010111

13.7.82 VFREDOSUM.VS: a vector floating-point reduction instruction that sums values in element order

Syntax:

`vfredosum.vs. vd, vs2, vs1, vm`

Operation:

```
tmp = vs1[0]
for( i=0; i < vl; i++) {
    tmp = tmp + vs2[i]
}
```

`vd[0]= tmp`

`vd[VLEN/SEW-1:1] = 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfredosum.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding to the assembler instruction is `vfredosum.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000011			vm	vs2		vs1		001		vd		1010111

13.7.83 VFREDSUM.VS: a vector floating-point reduction instruction that sums values in any order

Syntax:

vfredsum.vs. vd, vs2, vs1, vm

Operation:

tmp=unorder_sum(vs2[i])

vd[0]= tmp + vs1[0];

vd[VLEN/SEW-1:1] = 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfredsum.vs vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfredsum.vs vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000001			vm	vs2		vs1		001		vd		1010111

13.7.84 VFRSUB.VF: a vector-scalar floating-point subtract instruction

Syntax:

vfrsub.vf. vd, vs2, fs1, vm

Operation:

$$vd[i] = fs1 - vs2[i];$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, DZ, OF, UF, and NX

Notes: Dynamically rounds off based on the rm bit of the floating-point register FCSR: 3' b000: Rounds off to the nearest even number. 3' b001: Rounds off to zero. 3' b010: Rounds off to negative infinity. 3' b011: Rounds off to positive infinity. 3' b100: Rounds off to the nearest large value. 3' b101 to 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFRSUB.VF.

If vm is 1, the instruction is not masked. The corresponding assembler instruction is vfrsub.vf vd, vs2, fs1. If vm is 0, the instruction is masked. The corresponding assembler instruction is vfrsub.vf vd, vs2, fs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100111		vm	vs2	fs1		101		vd		1010111		

13.7.85 VFSGNJ.VF: a vector-scalar floating-point sign-injection instruction

Syntax:

vfgnj.vf vd, vs2, fs1,vm

Operation:

$$vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$$

$$vd[i][MSB] \leftarrow fs1[MSB]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnj.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnj.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
001000	vm	vs2	fs1	101	vd	1010111

13.7.86 VFSGNJ.VV: a vector floating-point sign-injection instruction

Syntax:

`vfsgnj.vv vd, vs2, vs1, vm`

Operation:

$vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$

$vd[i][MSB] \leftarrow vs1[i][MSB]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnj.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnj.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
001000	vm	vs2	vs1	001	vd	1010111

13.7.87 VFSGNJN.VF: a vector-scalar floating-point NOT-sign-injection instruction

Syntax:

`vfsgnjn.vf vd, vs2, fs1, vm`

Operation:

$$vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$$

$$vd[i][MSB] \leftarrow !fs1[MSB]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnjn.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnjn.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001001		vm	vs2	fs1		101		vd		1010111		

13.7.88 VFSGNJN.VV: a vector floating-point NOT-sign-injection instruction

Syntax:
`vfsgnjn.vv vd, vs2, vs1,vm`
Operation:

$$vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$$

$$vd[i][MSB] \leftarrow !vs1[i][MSB]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnjn.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnjn.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001001		vm	vs2		vs1		001		vd		1010111	

13.7.89 VFSGNJX.VF: a vector-scalar floating-point XOR-sign-injection instruction

Syntax:

`vfsgnjx.vf vd, vs2, fs1,vm`

Operation:

$vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$

$vd[i][MSB] \leftarrow fs1[MSB] \wedge vs2[i][MSB]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnjx.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnjx.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001010		vm	vs2		fs1		101		vd		1010111	

13.7.90 VFSGNJX.VV: a vector floating-point XOR-sign-injection instruction

Syntax:

`vfsgnjx.vv vd, vs2, vs1,vm`

Operation: $vd[i][MSB-1:0] \leftarrow vs2[i][MSB-1:0]$

$vd[i][MSB] \leftarrow vs1[i][MSB] \wedge vs2[i][MSB]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsgnjx.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfsgnjx.vv vd, vs2, vs1`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001010		vm	vs2		vs1		001		vd		1010111	

13.7.91 VFSQRT.V: a vector floating-point square-root instruction

Syntax:

`vfsqrt.v vd, vs2, vm`

Operation:

$vd[i] \leftarrow \text{sqrt}(vs2[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.

- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSQRT.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSQRT.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSQRT.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsqrt.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfsqrt.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011	vm	vs2		00000		001		vd		1010111		

13.7.92 VFSUB.VF: a vector-scalar floating-point subtract instruction

Syntax:

`vfsub.vf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] - fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.

- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfsb.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfsb.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000010			<i>vm</i>	<i>vs2</i>		<i>fs1</i>		101		<i>vd</i>		1010111

13.7.93 VFSUB.VV: a vector floating-point subtract instruction

Syntax:

`vfsb.vv vd, vs2, vs1, vm`

Operation:

$rd[i] \leftarrow vs2[i] - vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and NX

Notes:

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.

- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFSUB.VF.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsub.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsub.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000010	vm		vs2	vs1		001		vd		1010111		

13.7.94 VFWADD.VF: a vector-scalar floating-point widening add instruction

Syntax:

`vfwadd.vf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] + fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

The element width is $2 * SEW$ for *vd* and *SEW* for *vs2* or *fs1*.

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.VF.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwadd.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwadd.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
110000	vm	vs2	fs1	101	vd	1010111

13.7.95 VFWADD.VV: a vector floating-point widening add instruction

Syntax:

`vfwadd.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] + vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

The element width is $2*SEW$ for vd and SEW for $vs2$ or $vs1$.

Dynamically rounds off based on the rm bit of the floating-point register $FCSR$:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of $VFWADD.VV$.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of $VFWADD.VV$.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of $VFWADD.VV$.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is $vfwadd.vv\ vd,\ vs2,\ vs1$.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is $vfwadd.vv\ vd,\ vs2,\ vs1,\ v0.t$.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110000		vm	vs2	vs1		001		vd		1010111		

13.7.96 VFWADD.WF: a widening vector-scalar floating-point widening add instruction

Syntax:

$vfwadd.wf\ vd,\ vs2,\ fs1,\ vm$

Operation:

$vd[i] \leftarrow vs2[i] + fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 \times \text{SEW}$ for vd or vs2 and SEW for fs1.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwadd.wf vd, vs2, fs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwadd.wf vd, vs2, fs1, v0.t.

Instruction format:

31		26	25	24		20	19		15	14	12	11		7	6		0
	110100		vm		vs2			fs1		101			vd			1010111	

13.7.97 VFWADD.WF: a widening vector floating-point widening add instruction

Syntax:

vfwadd.wv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] + vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is 2*SEW for vd or vs2 and SEW for vs1.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWADD.WF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwadd.wv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwadd.wv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110100		vm	vs2		fs1		001		vd		1010111	

13.7.98 VFWCVT.F.F.V: a vector floating-point widening type-convert instruction**Syntax:**

vfwcvt.f.f.v vd, vs2, vm

Operation:

vd [i] ← float_convert_to_double_width_float(vs2[i])

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd and SEW for vs2.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2		01100		001		vd		1010111	

13.7.99 VFWCVT.F.X.V: a vector widening type-convert instruction that converts signed integers to floating-point values

Syntax:

`vfwcvt.f.x.v vd, vs2, vm`

Operation:

`vd [i] ← signed_integer_convert_to_double_width_float(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd and SEW for vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.X.V.

- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.X.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.X.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwcvt.f.x.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfwcvt.f.x.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2		01011		001		vd		1010111	

13.7.100 VFWCVT.F.XU.V: a vector widening type-convert instruction that converts unsigned integers to floating-point values

Syntax:

`vfwcvt.f.xu.v vd, vs2, vm`

Operation:

`vd [i] ← unsigned_integer_convert_to_double_width_float(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2 \times \text{SEW}$ for *vd* and *SEW* for *vs2*.

Dynamically rounds off based on the *rm* bit of the floating-point register *FCSR*:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.

- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.XU.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.XU.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.F.XU.V.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwcvt.f.xu.v vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwcvt.f.xu.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		01010		001		vd		1010111

13.7.101 VFWCVT.X.F.V: a vector widening type-convert instruction that converts floating-point values to signed integers

Syntax:

`vfwcvt.x.f.v vd, vs2, vm`

Operation:

`vd [i] ← float_convert_to_double_width_signed_integer(vs2[i])`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

The element width is $2 \times \text{SEW}$ for `vd` and `SEW` for `vs2`.

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.

- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.X.F.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.X.F.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.X.F.V.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwcvt.x.f.v vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vfwcvt.x.f.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2		01001		001		vd		1010111	

13.7.102 VFWCVT.XU.F.V: a vector widening type-convert instruction that converts floating-point values to unsigned integers

Syntax:

`vfwcvt.xu.f.v vd, vs2, vm`

Operation:

$vd[i] \leftarrow \text{float_convert_to_double_width_unsigned_integer}(vs2[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and NX

Notes:

The element width is $2 \times \text{SEW}$ for *vd* and SEW for *vs2*.

Dynamically rounds off based on the *rm* bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.

- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.XU.F.V.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.XU.F.V.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWCVT.XU.F.V.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwcvt.xu.f.v vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwcvt.xu.f.v vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2	01000		001		vd		1010111		

13.7.103 VFWMACC.VF: a vector-scalar floating-point widening multiply-add instruction that overwrites addends

Syntax:

`vfwmaccc.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow fs1 * vs2[i] + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 \times \text{SEW}$ for `vd` and `SEW` for `fs1` or `vs2`.

Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNMACC.VF`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNMACC.VF`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFNMACC.VF`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwmacc.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwmacc.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111100			vm	vs2		fs1		101		vd		1010111

13.7.104 VFWMACC.VV: a vector floating-point widening multiply-add instruction that overwrites addends

Syntax:

`vfwmacc.vv vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow vs1[i] * vs2[i] + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is 2*SEW for vd and SEW for vs1 or vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMACC.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMACC.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMACC.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwmac.vv vd, vs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwmac.vv vd, vs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111100		vm	vs2	vs1		001		vd		1010111		

13.7.105 VFWMSAC.VF: a vector floating-point widening multiply-sub instruction that overwrites addends

Syntax:

vfwmsac.vf vd, fs1, vs2, vm

Operation:

$vd[i] \leftarrow fs1 * vs2[i] - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is 2*SEW for vd and SEW for fs1 or vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwmsac.vf vd, fs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwmsac.vf vd, fs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111110		vm	vs2	fs1	101	vd	1010111					

13.7.106 VFWMSAC.VV: a vector floating-point widening multiply-sub instruction that overwrites addends

Syntax:

vfwmsac.vv vd, vs1, vs2, vm

Operation:

$vd[i] \leftarrow vs1[i] * vs2[i] - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 * SEW$ for vd and SEW for vs1 or vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMSAC.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwmsac.vv vd, vs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwmsac.vv vd, vs1, vs2, v0.t.

Instruction format:

31		26	25	24		20	19		15	14	12	11		7	6		0
	111110		vm		vs2			vs1		001			vd			1010111	

13.7.107 VFWMUL.VF: a vector-scalar widening floating-point multiply instruction

Syntax:

vfwmul.vf vd, vs2, fs1, vm

Operation:

$vd[i] \leftarrow vs2[i] * fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

The element width is 2*SEW for vd and SEW for fs1 or vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwmul.vf vd, vs2, fs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwmul.vf vd, vs2, fs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111000		vm	vs2	fs1		101		vd		1010111		

13.7.108 VFWMUL.VV: a vector widening floating-point multiply instruction**Syntax:**

vfwmul.vv vd, vs2, vs1, vm

Operation:

$$vd[i] \leftarrow vs2[i] * vs1[i]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

The element width is 2*SEW for vd and SEW for vs1 or vs2.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWMUL.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwmul.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwmul.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
111000	vm	vs2	vs1	001	vd	1010111

13.7.109 VFWNMACC.VF: a vector-scalar floating-point widening negate-(multiply-add) instruction that overwrites addends

Syntax:

`vfwnmacc.vf vd, fs1, vs2, vm`

Operation:

$$vd[i] \leftarrow -(fs1 * vs2[i]) - vd[i]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is 2*SEW for `vd` and SEW for `fs1` or `vs2`.

Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VF`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VF`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VF`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwnmacc.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwnmacc.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111101			vm	vs2		fs1		101		vd		1010111

13.7.110 VFWNMACC.VV: a vector floating-point widening negate-(multiply-add) instruction that overwrites addends

Syntax:

`vfwnmacc.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(vs1[i] * vs2[i]) - vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 * SEW$ for `vd` and SEW for `vs1` or `vs2`.

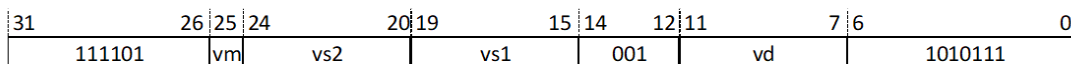
Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VV`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VV`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMACC.VV`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwnmacc.vv vd, vs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwnmacc.vv vd, vs1, vs2, v0.t`.

Instruction format:



13.7.111 VFWNMSAC.VF: a vector-scalar floating-point widening negate-(multiply-sub) instruction that overwrites addends

Syntax:

`vfwnmsac.vf vd, fs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(fs1 * vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 * SEW$ for `vd` and SEW for `fs1` or `vs2`.

Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VF`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VF`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VF`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwnmsac.vf vd, fs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwnmvsac.vf vd, fs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111111		vm	vs2		fs1		101		vd		1010111	

13.7.112 VFWNMSAC.VV: a vector floating-point widening negate-(multiply-sub) instruction that overwrites addends

Syntax:

`vfwnmvsac.vv. vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(vs1[i] * vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2 * SEW$ for `vd` and SEW for `vs1` or `vs2`.

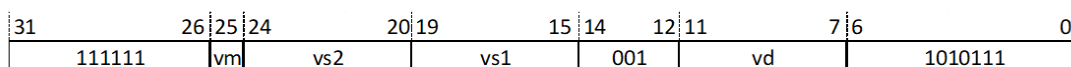
Dynamically rounds off based on the `rm` bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VF`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VV`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWNMSAC.VV`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwnmsac.vv vd, vs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwnmsac.vv vd, vs1, vs2, v0.t`.

Instruction format:



13.7.113 VFWREDOSUM.VS: a vector widening floating-point reduction instruction that sums values in element order

Syntax:

`vfwredosum.vs. vd, vs2, vs1, vm`

Operation:

```
tmp = vs1[0]
for( i=0; i < vl; i++) {
    tmp = tmp + widen_to_2sew(vs2[i])
}
vd[0]= tmp
vd[VLEN/SEW-1:1] = 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is $2*SEW$ for `vd` or `vs1` and `SEW` for `vs2`. The elements in `vs2` are promoted to $2*SEW$ bits before computation.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwredosum.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwredosum.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110011			vm	vs2		vs1		001		vd		1010111

13.7.114 VFWRDVSUM.VS: a vector widening floating-point reduction instruction that sums values in any order

Syntax:

vwredsum.vs. vd, vs2, vs1, vm

Operation:

tmp = unorder_sum(widen_to_2sew(vs2[i]))

vd[0]= vs1[0] + tmp

vd[VLEN/SEW-1:1] = 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV, OF, and IX

Notes:

The element width is 2*SEW for vd or vs1 and SEW for vs2. The elements in vs2 are promoted to 2*SEW bits before computation.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwredsum.vs vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwredsum.vs vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110001			vm	vs2		vs1		001		vd		1010111

13.7.115 VFWSUB.VF: a vector-scalar widening floating-point subtract instruction

Syntax:

vwsub.vf vd, vs2, fs1, vm

Operation:

$$vd[i] \leftarrow vs2[i] - fs1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:The element width is $2 * SEW$ for vd and SEW for $vs2$ or $fs1$.Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VF.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VF.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VF.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwsb.vf vd, vs2, fs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfwsb.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110010		vm	vs2		fs1		101		vd		1010111	

13.7.116 VFWSUB.VV: a vector widening floating-point subtract instruction**Syntax:**

vfwsb.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] - vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

The element width is $2 * SEW$ for vd and SEW for vs2 or vs1.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.VV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vfwsb.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vfwsb.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110010		vm	vs2		vs1		001		vd		1010111	

13.7.117 VFWSUB.WF: a widening vector-scalar floating-point widening subtract instruction

Syntax:

`vfwsb.wf vd, vs2, fs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] - fs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and OF

Notes:

The element width is $2 \times \text{SEW}$ for `vd` or `vs2` and `SEW` for `fs1`.

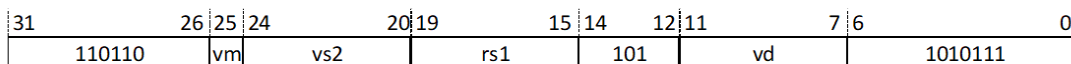
Dynamically rounds off based on the `rm` bit of the floating-point register `FCSR`:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWSUB.WF`.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWSUB.WF`.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of `VFWSUB.WF`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwsb.wf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vfwsb.wf vd, vs2, fs1, v0.t`.

Instruction format:



13.7.118 VFWSUB.WV: a widening vector floating-point widening subtract instruction

Syntax:

vfwsb.wv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] - vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bits NV and OF

Notes:

The element width is $2 * SEW$ for vd or vs2 and SEW for vs1.

Dynamically rounds off based on the rm bit of the floating-point register FCSR:

- 3' b000: Rounds off to the nearest even number.
- 3' b001: Rounds off to 0.
- 3' b010: Rounds off to negative infinity.
- 3' b011: Rounds off to positive infinity.
- 3' b100: Rounds off to the nearest large value.
- 3' b101: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.WV.
- 3' b110: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.WV.
- 3' b111: invalid value. When this value is used, an illegal instruction exception occurs for the execution of VFWSUB.WV.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vfwsb.wv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vfwsb.wv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
110110			vm	vs2		vs1		001		vd		1010111	

13.7.119 VID.V: a vector element index instruction that writes each element's index to the destination

Syntax:

vid.v vd, vm

Operation:

vd[i] = i

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vid.v vd.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vid.v vd, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
010110			vm	vs2		10001		010		vd		1010111	

13.7.120 VIOTA.M: a vector instruction that gets destination offsets of active elements

Syntax:

viota.m vd, vs2, vm

Operation:

```
for(i=0;i<VLMAX;i++){
    if vs2[i] is active{
```

```

count = 0
for(j=0;j<i;j++){
    if vs2[i] is active
        count = count + 1
    }
vd[i] = count
}
else
    if vs2[i] is inactive{
        vd[i] = vd[i]
    }
    else{
        vd[i] = 0
    }
}

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. corresponding to the assembler instruction `viota.m vd, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `viota.m vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010110	vm	vs2	10000	010	vd	1010111						

13.7.121 VLB.V: a vector signed byte load instruction

Syntax:

`vlb.v vd, (rs1), vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1}+i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlb.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlb.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	100	vm	00000			rs1		000		vd			0000111	

13.7.122 VLBF.V: a vector fault-only-first (FOF) signed byte load instruction:

Syntax:

`vlbff.v vd, (rs1), vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1}+i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlbff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlbff.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	100	vm	10000	rs1	000	vd	0000111

13.7.123 VLBU.V: a vector unsigned byte load instruction

Syntax:

`vlbu.v vd, (rs1), vm`

Operation:

`vd[i] = zero_extend(mem[rs1+i])`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlbu.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlbu.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	00000			rs1		000		vd			0000111	

13.7.124 VLBUFF.V: a vector FOF unsigned byte load instruction

Syntax:

`vlbuff.v vd, (rs1), vm`

Operation:

`vd[i] = zero_extend(mem[rs1+i])`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlbuff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlbuff.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	10000			rs1		000		vd			0000111	

13.7.125 VLE.V: a vector element load instruction

Syntax:

vle.v vd, (rs1), vm

Operation:

$vd[i] = mem[rs1+i*SEW/8]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vle.v vd, (rs1).

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vle.v vd, (rs1), v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	0000	rs1	111	vd	0000111							

13.7.126 VLEFF.V: a vector FOF unsigned element load instruction

Syntax:

vleff.v vd, (rs1), vm

Operation:

$vd[i] = mem[rs1+i*SEW/8]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vleff.v vd, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vleff.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	000	vm	10000	rs1	111	vd	0000111

13.7.127 VLH.V: a vector signed halfword load instruction**Syntax:**

`vlh.v vd, (rs1), vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[rs1+2*i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlh.v vd, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlh.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	100	vm	00000	rs1	101	vd	0000111

13.7.128 VLHFF.V: a vector FOF signed halfword load instruction

Syntax:

```
vlhff.v vd, (rs1), vm
```

Operation:

$$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1} + 2 * i])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlhff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlhff.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	100	vm	10000			rs1		101		vd			0000111	

13.7.129 VLHU.V: a vector unsigned halfword load instruction

Syntax:

```
vlhu.v vd, (rs1), vm
```

Operation:

$$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + 2 * i])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlhu.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlhu.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	000	vm	00000	rs1	101	vd	0000111

13.7.130 VLHUFF.V: a vector FOF unsigned halfword load instruction**Syntax:**

`vlhuff.v vd, (rs1), vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + 2 * i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlhuff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlhuff.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000		000	vm		10000		rs1		101		vd			0000111

13.7.131 VLSB.V: a vector strided signed byte load instruction

Syntax:

`vlsb.v vd, (rs1), rs2, vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[rs1+i*rs2])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsb.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlsb.v vd, (rs1), rs2, v0.t`.

Instruction format:

31 29 28 26 25 24 20 19 15 14 12 11 7 6 0

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000		110	vm		rs2		rs1		000		vd			0000111

13.7.132 VLSBU.V: a vector strided unsigned byte load instruction

Syntax:

```
vlsbu.v vd, (rs1), rs2, vm
```

Operation:

$$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + i * \text{rs2}])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsbu.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlsbu.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	010	vm	rs2	rs1	000	vd	0000111							

13.7.133 VLSE.V: a vector strided element load instruction

Syntax:

```
vlse.v vd, (rs1), rs2, vm
```

Operation:

$$vd[i] = \text{mem}[\text{rs1} + i * \text{rs2}]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits: None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlse.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlse.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	010	vm	rs2	rs1	111	vd				0000111				

13.7.134 VLSEG<NF>B.V: a vector SEGMENT signed byte load instruction

Syntax:

`vlseg<nf>b.v vd, (rs1), vm`

Operation:

for(`k=0; k<=nf-1; k++`) {

$V_{d+k}[i] = \text{mem}[\text{rs1} + k + i * \text{nf}]$

}

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vd=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vlseg4b.v v8, (x5)`. This instruction performs the following operations:

$v8[0] = \text{mem}[x5], v8[1] = \text{mem}[x5+4] \dots v8[i] = \text{mem}[x5+4*i]$

$v9[0] = \text{mem}[x5+1], v9[1] = \text{mem}[x5+5] \dots v9[i] = \text{mem}[x5+1+4*i]$

$v10[0] = \text{mem}[x5+2], v10[1] = \text{mem}[x5+6] \dots v10[i] = \text{mem}[x5+2+4*i]$

$v11[0] = \text{mem}[x5+3], v11[1] = \text{mem}[x5+7] \dots v11[i] = \text{mem}[x5+3+4*i]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>b.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>b.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	100		vm	00000			rs1		000		vd		0000111	

13.7.135 VLSEG<NF>BFF.V: a vector FOF SEGMENT signed byte load instruction**Syntax:**
`vlseg<nf>bff.v vd, (rs1), vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + k + i *nf]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vd=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vlseg4bff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+4] \dots v8[i]=mem[x5+4*i]$$

$$v9[0]=mem[x5+1], v9[1]=mem[x5+5] \dots v9[i]=mem[x5+1+4*i]$$

$$v10[0]=mem[x5+2], v10[1]=mem[x5+6] \dots v10[i]=mem[x5+2+4*i]$$

$$v11[0]=mem[x5+3], v11[1]=mem[x5+7] \dots v11[i]=mem[x5+3+4*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>bff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>bff.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	100	vm	10000	rs1	000	vd	0000111

13.7.136 VLSEG<NF>BU.V: a vector SEGMENT unsigned byte load instruction

Syntax:

`vlseg<nf>bu.v vd, (rs1), vm`

Operation:

```
for( k=0; k<=nf-1 ; k++) {
    Vd+k[i] = mem[rs1 + k +i *nf]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vd=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vlseg4bu.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=mem[x5],v8[1]=mem[x5+4] \dots v8[i]=mem[x5+4*i]$$

$$v9[0]=mem[x5+1],v9[1]=mem[x5+5] \dots v9[i]=mem[x5+1+4*i]$$

$$v10[0]=mem[x5+2],v10[1]=mem[x5+6] \dots v10[i]=mem[x5+2+4*i]$$

$$v11[0]=mem[x5+3],v11[1]=mem[x5+7] \dots v11[i]=mem[x5+3+4*i]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>bu.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>bu.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000		vm	00000			rs1		000		vd		0000111	

13.7.137 VLSEG<NF>BUFF.V: a vector FOF SEGMENT unsigned byte load instruction

Syntax:
`vlseg<nf>buff.v vd, (rs1), vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + k + i *nf]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vd=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vlseg4buff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+4] \dots v8[i]=mem[x5+4*i]$$

$$v9[0]=mem[x5+1], v9[1]=mem[x5+5] \dots v9[i]=mem[x5+1+4*i]$$

$$v10[0]=mem[x5+2], v10[1]=mem[x5+6] \dots v10[i]=mem[x5+2+4*i]$$

$$v11[0]=mem[x5+3], v11[1]=mem[x5+7] \dots v11[i]=mem[x5+3+4*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>buff.v vd, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>buff.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000		vm	10000			rs1		000		vd		0000111	

13.7.138 VLSEG<NF>E.V: a vector SEGMENT element load instruction**Syntax:**

```
vlseg<nf>e.v vd, (rs1), vm
```

Operation:

```
offset=sew/8
```

```
for( k=0; k<=nf-1; k++) {
```

```
    Vd+k[i] = mem[rs1 + k*offset + i*nf*offset]
```

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, *SEW*=32, and *lmul*=1, the instruction is `vlseg4e.v v8, (x5), offset=32/8=4`. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+4*4] \dots v8[i]=mem[x5+16*i]$$

$$v9[0]=mem[x5+4], v9[1]=mem[x5+5*4] \dots v9[i]=mem[x5+4+16*i]$$

$$v10[0]=mem[x5+2*4], v10[1]=mem[x5+6*4] \dots v10[i]=mem[x5+8+16*i]$$

$$v11[0]=mem[x5+3*4], v11[1]=mem[x5+7*4] \dots v11[i]=mem[x5+12+16*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000		vm	00000			rs1		111		vd		0000111	

13.7.139 VLSEG<NF>EFF.V: a vector FOF SEGMENT element load instruction**Syntax:**

`vlseg<nf>eff.v vd, (rs1), vm`

Operation:

`offset=sew/8`

for(*k*=0; *k*<=*nf*-1; *k*++) {

$V_{d+k}[i] = \text{mem}[\text{rs1} + k * \text{offset} + i * \text{nf} * \text{offset}]$

}

nf+1 specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, SEW=32, and *lmul*=1, the instruction is `lseg4eff.v v8, (x5), offset=32/8=4`. This instruction performs the following operations:

$v8[0] = \text{mem}[x5], v8[1] = \text{mem}[x5+4*4] \dots v8[i] = \text{mem}[x5+16*i]$

$v9[0] = \text{mem}[x5+4], v9[1] = \text{mem}[x5+5*4] \dots v9[i] = \text{mem}[x5+4+16*i]$

$v10[0] = \text{mem}[x5+2*4], v10[1] = \text{mem}[x5+6*4] \dots v10[i] = \text{mem}[x5+8+16*i]$

$v11[0] = \text{mem}[x5+3*4], v11[1] = \text{mem}[x5+7*4] \dots v11[i] = \text{mem}[x5+12+16*i]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>eff.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>eff.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000	vm	10000				rs1	111				vd	0000111	

13.7.140 VLSEG<NF>H.V: a vector SEGMENT signed halfword load instruction

Syntax:

`vlseg<nf>h.v vd, (rs1), vm`

Operation:

for(*k*=0; *k*<=*nf*-1; *k*++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4h.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+8] \cdots v8[i]=\text{mem}[x5+8*i]$$

$$v9[0]=\text{mem}[x5+2], v9[1]=\text{mem}[x5+10] \cdots v9[i]=\text{mem}[x5+2+8*i]$$

$$v10[0]=\text{mem}[x5+4], v10[1]=\text{mem}[x5+12] \cdots v10[i]=\text{mem}[x5+4+8*i]$$

$$v11[0]=\text{mem}[x5+6], v11[1]=\text{mem}[x5+14] \cdots v11[i]=\text{mem}[x5+6+8*i]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>h.v vd, (rs1)`.If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>h.v vd, (rs1), v0.t`.**Instruction format:**

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	100	vm	00000	rs1	101	vd	0000111

13.7.141 VLSEG<NF>HFF.V: a vector FOF SEGMENT signed halfword load instruction

Syntax:
`vlseg<nf>hff.v vd, (rs1), vm`
Operation:for(*k*=0; *k*<=*nf*-1; *k*++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4hff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0] = \text{mem}[x5], v8[1] = \text{mem}[x5+8] \cdots v8[i] = \text{mem}[x5+8*i]$$

$$v9[0] = \text{mem}[x5+2], v9[1] = \text{mem}[x5+10] \cdots v9[i] = \text{mem}[x5+2+8*i]$$

$$v10[0] = \text{mem}[x5+4], v10[1] = \text{mem}[x5+12] \cdots v10[i] = \text{mem}[x5+4+8*i]$$

$$v11[0] = \text{mem}[x5+6], v11[1] = \text{mem}[x5+14] \cdots v11[i] = \text{mem}[x5+6+8*i]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>hff.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>hff.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	100	vm	10000	rs1	101	vd	0000111

13.7.142 VLSEG<NF>HU.V: a vector SEGMENT unsigned halfword load instruction**Syntax:**
`vlseg<nf>hu.v vd, (rs1), vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
```

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4hu.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+8] \cdots v8[i]=\text{mem}[x5+8*i]$$

$$v9[0]=\text{mem}[x5+2], v9[1]=\text{mem}[x5+10] \cdots v9[i]=\text{mem}[x5+2+8*i]$$

$$v10[0]=\text{mem}[x5+4], v10[1]=\text{mem}[x5+12] \cdots v10[i]=\text{mem}[x5+4+8*i]$$

$$v11[0]=\text{mem}[x5+6], v11[1]=\text{mem}[x5+14] \cdots v11[i]=\text{mem}[x5+6+8*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>hu.v vd, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>hu.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	000	vm	00000	rs1	101	vd	0000111

13.7.143 VLSEG<NF>HUFF.V: a vector FOF SEGMENT unsigned halfword load instruction

Syntax:
`vlseg<nf>huff.v vd, (rs1), vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
```

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4huff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+8] \cdots v8[i]=\text{mem}[x5+8*i]$$

$$v9[0]=\text{mem}[x5+2], v9[1]=\text{mem}[x5+10] \cdots v9[i]=\text{mem}[x5+2+8*i]$$

$$v10[0]=\text{mem}[x5+4], v10[1]=\text{mem}[x5+12] \cdots v10[i]=\text{mem}[x5+4+8*i]$$

$$v11[0]=\text{mem}[x5+6], v11[1]=\text{mem}[x5+14] \cdots v11[i]=\text{mem}[x5+6+8*i]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>huff.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>huff.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000	vm	10000			rs1		101		vd			0000111	

13.7.144 VLSEG<NF>W.V: a vector SEGMENT signed word load instruction**Syntax:**

`vlseg<nf>w.v vd, (rs1), vm`

Operation:

for(*k*=0; *k*<=*nf*-1; *k*++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 4*k + 4*i * \text{nf}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4w.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+16] \cdots v8[i]=\text{mem}[x5+16*i]$$

$$v9[0]=\text{mem}[x5+4], v9[1]=\text{mem}[x5+20] \cdots v9[i]=\text{mem}[x5+4+16*i]$$

$$v10[0]=\text{mem}[x5+8], v10[1]=\text{mem}[x5+24] \cdots v10[i]=\text{mem}[x5+8+16*i]$$

$$v11[0]=\text{mem}[x5+12], v11[1]=\text{mem}[x5+28] \cdots v11[i]=\text{mem}[x5+12+16*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>w.v vd, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>w.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	100	vm	00000			rs1		110		vd			0000111	

13.7.145 VLSEG<NF>WFF.V: a vector FOF SEGMENT signed word load instruction**Syntax:**

`vlseg<nf>wff.v vd, (rs1), vm`

Operation:

for(*k*=0; *k*<=*nf*-1; *k*++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vlseg4wff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+8] \cdots v8[i]=\text{mem}[x5+8*i]$$

$$v9[0]=\text{mem}[x5+2], v9[1]=\text{mem}[x5+10] \cdots v9[i]=\text{mem}[x5+2+8*i]$$

$$v10[0]=\text{mem}[x5+4], v10[1]=\text{mem}[x5+12] \cdots v10[i]=\text{mem}[x5+4+8*i]$$

$$v11[0]=\text{mem}[x5+6], v11[1]=\text{mem}[x5+14] \cdots v11[i]=\text{mem}[x5+6+8*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlseg<nf>wff.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlseg<nf>wff.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		100		vm	10000			rs1	111		vd		0000111	

13.7.146 VLSEG<NF>WU.V: a vector SEGMENT unsigned word load instruction**Syntax:**
`vlseg<nf>wu.v vd, (rs1), vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
```

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 4*k + 4*i * \text{nf}]$$

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf* is 4, *vd* is 8, *rs1* is 5, *vm* is 1, and *lmul* is 1, the instruction is `vlseg4wu.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+16] \cdots v8[i]=\text{mem}[x5+16*i]$$

$$v9[0]=\text{mem}[x5+4], v9[1]=\text{mem}[x5+20] \cdots v9[i]=\text{mem}[x5+4+16*i]$$

$$v10[0]=\text{mem}[x5+8], v10[1]=\text{mem}[x5+24] \cdots v10[i]=\text{mem}[x5+8+16*i]$$

$$v11[0]=\text{mem}[x5+12], v11[1]=\text{mem}[x5+28] \cdots v11[i]=\text{mem}[x5+12+16*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If *vm* is 1, the instruction is not masked. The corresponding assembler instruction is `vlseg<nf>wu.v vd, (rs1)`.

If *vm* is 0, the instruction is masked. The corresponding assembler instruction is `vlseg<nf>wu.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		000		vm	00000			rs1	110		vd	0000111		

13.7.147 VLSEG<NF>WUFF.V: a vector FOF SEGMENT unsigned word load instruction

Syntax:

`vlseg<nf>wuff.v vd, (rs1), vm`

Operation:

for(*k*=0; *k*<=*nf*-1; *k*++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 2*k + 2*i * \text{nf}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf* is 4, *vd* is 8, *rs1* is 5, *vm* is 1, and *lmul* is 1, the instruction is `vlseg4wuff.v v8, (x5)`. This instruction performs the following operations:

$$v8[0]=\text{mem}[x5], v8[1]=\text{mem}[x5+8] \cdots v8[i]=\text{mem}[x5+8*i]$$

$$v9[0]=\text{mem}[x5+2], v9[1]=\text{mem}[x5+10] \cdots v9[i]=\text{mem}[x5+2+8*i]$$

$$v10[0]=\text{mem}[x5+4], v10[1]=\text{mem}[x5+12] \cdots v10[i]=\text{mem}[x5+4+8*i]$$

$$v11[0]=\text{mem}[x5+6], v11[1]=\text{mem}[x5+14] \cdots v11[i]=\text{mem}[x5+6+8*i]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If *vm* is 1, the instruction is not masked. The corresponding assembler instruction is `vlsseg<nf>wuff.v vd, (rs1), rs2`.

If *vm* is 0, the instruction is masked. The corresponding assembler instruction is `vlsseg<nf>wuff.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		000		vm	10000			rs1	111		vd	0000111		

13.7.148 VLSSEG<NF>B.V: a vector strided SEGMENT signed byte load instruction**Syntax:**
`vlsseg<nf>b.v vd, (rs1), rs2, vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + k + i *rs2]
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *rs2*=6, *vm*=1, and *lmul*=1, the instruction is `vlsseg4b.v v8, (x5), x6`. This instruction performs the following operations:

$$v8[0]=mem[x5],v8[1]=mem[x5+x6] \dots v8[i]=mem[x5+i*x6]$$

$$v9[0]=mem[x5+1],v9[1]=mem[x5+x6+1] \dots v9[i]=mem[x5+i*x6+1]$$

$$v10[0]=mem[x5+2],v10[1]=mem[x5+x6+2] \dots v10[i]=mem[x5+i*x6+2]$$

$$v11[0]=mem[x5+3],v11[1]=mem[x5+x6+3] \dots v11[i]=mem[x5+i*x6+3]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsseg<nf>b.v vd, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlsseg<nf>b.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	110		vm	00000			rs1	000		vd	0000111			

13.7.149 VLSSEG<NF>BU.V: a vector strided SEGMENT unsigned byte load instruction

Syntax:
`vlsseg<nf>bu.v vd, (rs1), rs2, vm`
Operation:

```
for( k=0; k<=nf-1; k++) {
```

$$V_{d+k}[i] = mem[rs1 + k + i * rs2]$$

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vd*=8, *rs1*=5, *rs2*=6, *vm*=1, and *lmul*=1, the instruction is `vlsseg4bu.v v8, (x5), x6`. This instruction performs the following operations:

$$v8[0]=mem[x5],v8[1]=mem[x5+x6] \dots v8[i]=mem[x5+i*x6]$$

$$v9[0]=mem[x5+1],v9[1]=mem[x5+x6+1] \dots v9[i]=mem[x5+i*x6+1]$$

$$v10[0]=mem[x5+2],v10[1]=mem[x5+x6+2] \dots v10[i]=mem[x5+i*x6+2]$$

$$v11[0]=mem[x5+3],v11[1]=mem[x5+x6+3] \dots v11[i]=mem[x5+i*x6+3]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsseg<nf>bu.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlsseg<nf>bu.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	010		vm	00000			rs1		000		vd		0000111	

13.7.150 VLSSEG<NF>E.V: a vector strided SEGMENT element load instruction

Syntax:
`vlsseg<nf>e.v vd, (rs1), rs2, vm`
Operation:
`offset=sew/8`
`for(k=0; k<=nf-1; k++) {`
`Vd+k[i] = mem[rs1 + k*offset + i*rs2]`
`}`

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vd=8`, `rs1=5`, `rs2=6`, `vm=1`, `SEW=32`, and `lmul=1`, the

instruction is `vlsseg4e.v v8, (x5), x6, offset=32/8=4`. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+x6] \dots v8[i]=mem[x5+x6]$$

$$v9[0]=mem[x5+4], v9[1]=mem[x5+x6+4] \dots v9[i]=mem[x5+i*x6+4]$$

$$v10[0]=mem[x5+8], v10[1]=mem[x5+x6+8] \dots v10[i]=mem[x5+i*x6+8]$$

$$v11[0]=mem[x5+12], v11[1]=mem[x5+x6+12] \dots v11[i]=mem[x5+i*x6+12]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsseg<nf>.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlsseg<nf>.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	010		vm	00000			rs1		111		vd		0000111	

13.7.151 VLSSEG<NF>H.V: a vector strided SEGMENT signed halfword load instruction

Syntax:

`vlsseg<nf>h.v vd, (rs1), rs2, vm`

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + 2*k + i *rs2]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vlsseg4h.v vd, (rs1), rs2. This instruction performs the following operations:

$$\begin{aligned}
 v8[0] &= \text{mem}[x5], v8[1] = \text{mem}[x5+x6] \dots v8[i] = \text{mem}[x5+i*x6] \\
 v9[0] &= \text{mem}[x5+2], v9[1] = \text{mem}[x5+x6+2] \dots v9[i] = \text{mem}[x5+i*x6+2] \\
 v10[0] &= \text{mem}[x5+4], v10[1] = \text{mem}[x5+x6+4] \dots v10[i] = \text{mem}[x5+i*x6+4] \\
 v11[0] &= \text{mem}[x5+6], v11[1] = \text{mem}[x5+x6+6] \dots v11[i] = \text{mem}[x5+i*x6+6]
 \end{aligned}$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlsseg<nf>h.v vd, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlsseg<nf>h.v vd, (rs1), rs2, v0.t.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	110	vm	00000	rs1	101	vd	0000111

13.7.152 VLSSEG<NF>HU.V: a vector strided SEGMENT unsigned halfword load instruction

Syntax:

vlsseg<nf>hu.v vd, (rs1), rs2, vm

Operation:

for(k=0; k<=nf-1; k++) {

$$V_{d+k}[i] = \text{mem}[rs1 + k + i * rs2]$$

}

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vlsseg4hu.v v8, (x5), x6. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+x6] \dots v8[i]=mem[x5+i*x6]$$

$$v9[0]=mem[x5+2], v9[1]=mem[x5+x6+2] \dots v9[i]=mem[x5+i*x6+2]$$

$$v10[0]=mem[x5+4], v10[1]=mem[x5+x6+4] \dots v10[i]=mem[x5+i*x6+4]$$

$$v11[0]=mem[x5+6], v11[1]=mem[x5+x6+6] \dots v11[i]=mem[x5+i*x6+6]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlsseg<nf>hu.v vd, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlsseg<nf>hu.v vd, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	010		vm	00000			rs1	101		vd		0000111		

13.7.153 VLSSEG<NF>W.V: a vector strided SEGMENT signed word load instruction

Syntax:

$$vlsseg<nf>w.v vd, (rs1), rs2, vm$$
Operation:

for(k=0; k<=nf-1; k++) {

$$V_{d+k}[i] = mem[rs1 + 4*k + i *rs2]$$

}

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vlsseg4w.v v8, (x5), x6. This instruction performs the following operations:

$$\begin{aligned} v8[0] &= \text{mem}[x5], v8[1] = \text{mem}[x5+x6] \dots v8[i] = \text{mem}[x5+i*x6] \\ v9[0] &= \text{mem}[x5+4], v9[1] = \text{mem}[x5+x6+4] \dots v9[i] = \text{mem}[x5+i*x6+4] \\ v10[0] &= \text{mem}[x5+8], v10[1] = \text{mem}[x5+x6+8] \dots v10[i] = \text{mem}[x5+i*x6+8] \\ v11[0] &= \text{mem}[x5+12], v11[1] = \text{mem}[x5+x6+12] \dots v11[i] = \text{mem}[x5+i*x6+12] \end{aligned}$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlsseg<nf>w.v vd, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlsseg<nf>w.v vd, (rs1), rs2, v0.t.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	110	vm	00000	rs1	110	vd	0000111

13.7.154 VLSSEG<NF>WU.V: a vector strided SEGMENT unsigned word load instruction

Syntax:

$$\text{vlsseg}\langle\text{nf}\rangle\text{wu.v vd, (rs1), rs2, vm}$$
Operation:

for(k=0; k<=nf-1; k++) {

$$V_{d+k}[i] = \text{mem}[\text{rs1} + 4*k + i * \text{rs2}]$$

}

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vlsseg4wu.v v8, (x5), x6. This instruction performs the following operations:

$$v8[0]=mem[x5], v8[1]=mem[x5+x6] \dots v8[i]=mem[x5+i*x6]$$

$$v9[0]=mem[x5+4], v9[1]=mem[x5+x6+4] \dots v9[i]=mem[x5+i*x6+4]$$

$$v10[0]=mem[x5+8], v10[1]=mem[x5+x6+8] \dots v10[i]=mem[x5+i*x6+8]$$

$$v11[0]=mem[x5+12], v11[1]=mem[x5+x6+12] \dots v11[i]=mem[x5+i*x6+12]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlsseg<nf>wu.v vd, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlsseg<nf>wu.v vd, (rs1), rs2, v0.t.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	010	vm	00000	rs1	110	vd	0000111

13.7.155 VLSH.V: a vector strided signed halfword load instruction**Syntax:**

vlsh.v vd, (rs1), rs2, vm

Operation:

$$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1}+i*\text{rs2}])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlsh.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlsh.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	110	vm	rs2	rs1	101	vd	0000111

13.7.156 VLSHU.V: a vector strided unsigned halfword load instruction

Syntax:

`vlshu.v vd, (rs1), rs2, vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + i * \text{rs2}])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes: The instruction reads halfwords and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlshu.v vd, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlshu.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000		010		vm	rs2		rs1		101		vd		0000111	

13.7.157 VLSW: a vector strided signed word load instruction**Syntax:**

vlsw.v vd, (rs1), rs2, vm

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[rs1+i*rs2])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlsw.v vd, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlsw.v vd, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000		110		vm	rs2		rs1		110		vd		0000111	

13.7.158 VLSWU.V: a vector strided unsigned word load instruction**Syntax:**

vlswu.v vd, (rs1), rs2, vm

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[rs1+i*rs2])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlswu.v vd, (rs1), rs2`.

If the `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlswu.v vd, (rs1), rs2, v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	010	vm	rs2	rs1	110	vd	0000111

13.7.159 VLW.V: a vector signed word load instruction**Syntax:**

`vlw.v vd, (rs1), vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[rs1+4*i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlw.v vd, (rs1)`.

If the `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlw.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	100	vm	00000	rs1	110	vd	0000111

13.7.160 VLWFF.V: a vector FOF signed word load instruction

Syntax:

`vlwff.v vd, (rs1), vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1} + 4 * i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlwff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlwff.v vd, (rs1), v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	100	vm	10000	rs1	110	vd	0000111

13.7.161 VLWU.V: a vector unsigned word load instruction

Syntax:

`vlwu.v vd, (rs1), vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1}+4*i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlwu.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlwu.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0	
000	000	vm	00000				rs1	110			vd				0000111

13.7.162 VLWUFF.V: a vector FOF unsigned word load instruction

Syntax:

`vlwuff.v vd, (rs1), vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1}+4*i])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW. The instruction only takes a trap on element 0. If an element greater than 0 raises a trap, the instruction will neither take the trap nor be executed on that element and all following elements, and the VL register will be updated to the index of the trapped element.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlwuff.v vd, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlwuff.v vd, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	10000			rs1		110		vd			0000111	

13.7.163 VLXB.V: a vector indexed signed byte load instruction

Syntax:

`vlxb.v vd, (rs1), vs2, vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxb.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxb.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2	rs1	000	vd	0000111							

13.7.164 VLXBU.V: a vector indexed unsigned byte load instruction

Syntax:

`vlxbu.v vd, (rs1), vs2, vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxbu.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxbu.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2	rs1	000	vd	0000111							

13.7.165 VLXE.V: a vector indexed element load instruction

Syntax:

`vlxe.v vd, (rs1), vs2, vm`

Operation:

$$vd[i] = \text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxe.v vd, (rs1), vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vlxe.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	011	vm	vs2	rs1	111	vd	0000111

13.7.166 VLXH.V: a vector indexed signed halfword load instruction

Syntax:
`vlxh.v vd, (rs1), vs2, vm`
Operation:

$$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxh.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxh.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2	rs1	101	vd	0000111							

13.7.167 VLXHU.V: a vector indexed unsigned halfword load instruction

Syntax:

`vlxhu.v vd, (rs1), vs2, vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxhu.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxhu.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2	rs1	101	vd	0000111							

13.7.168 VLXSEG<NF>B.V: a vector indexed SEGMENT signed byte load instruction

Syntax:

```
vlxseg<nf>b.v vd, (rs1), vs2, vm
```

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + k + vs2[i]]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4b.v v8, (x5), v16. This instruction performs the following operations:

```
v8[0]=mem[x5+v16[0] ] ...v8[i]=mem[x5+vs16[i] ]
v9[0]=mem[x5+v16[0]+1] ...v9[i]=mem[x5+vs16[i]+1]
v10[0]=mem[x5+v16[0]+2] ...v10[i]=mem[x5+vs16[i]+2]
v11[0]=mem[x5+v16[0]+3] ...v11[i]=mem[x5+vs16[i]+3]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>b.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>b.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	111		vm	00000			rs1	000		vd	0000111			

13.7.169 VLXSEG<NF>BU.V: a vector indexed SEGMENT unsigned byte load instruction

Syntax:

vlxseg<nf>bu.v vd, (rs1), vs2, vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + k + vs2[i]]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4bu.v v8, (x5), v16. This instruction performs the following operations:

```
v8[0]=mem[x5+v16[0] ] ...v8[i]=mem[x5+vs16[i] ]
v9[0]=mem[x5+v16[0]+1] ...v9[i]=mem[x5+vs16[i]+1]
v10[0]=mem[x5+v16[0]+2] ...v10[i]=mem[x5+vs16[i]+2]
v11[0]=mem[x5+v16[0]+3] ...v11[i]=mem[x5+vs16[i]+3]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads bytes and then unsign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>bu.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>bu.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	011	vm	00000	rs1	000	vd	0000111							

13.7.170 VLXSEG<NF>E.V: a vector indexed SEGMENT element load instruction

Syntax:

```
vlxseg<nf>e.v vd, (rs1), vs2, vm
```

Operation:

```
offset=sew/8
```

```
for( k=0; k<=nf-1; k++) {
```

```
    Vd+k[i] = mem[rs1 + vs2[i] + k*offset]
```

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=6, vm=1, SEW=32, and lmul=1, the instruction is vlxseg4e.v v8, (x5), v16, offset=32/8=4. This instruction performs the following operations:

```
v8[0]=mem[x5+v16[0]] ... v8[i]=mem[x5+v16[i]]
```

```
v9[0]=mem[x5+v16[0]+4] ... v9[i]=mem[x5+v16[i]+4]
```

```
v10[0]=mem[x5+v16[0]+8] ... v10[i]=mem[x5+v16[i]+8]
```

```
v11[0]=mem[x5+v16[0]+12] ... v11[i]=mem[x5+v16[i]+12]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads data with the length of the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>e.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>e.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	011	vm	00000	rs1	111	vd	0000111							

13.7.171 VLXSEG<NF>H.V: a vector indexed SEGMENT signed halfword load instruction

Syntax:

vlxseg<nf>h.v vd, (rs1), vs2, vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + vs2[i] + 2*k]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4h.v v8, (x5), v16. This instruction performs the following operations:

```
v8[0]=mem[x5+v16[0] ] ...v8[i]=mem[x5+vs16[i] ]
v9[0]=mem[x5+v16[0]+2] ...v9[i]=mem[x5+vs16[i]+2]
v10[0]=mem[x5+v16[0]+4] ...v10[i]=mem[x5+vs16[i]+4]
v11[0]=mem[x5+v16[0]+6] ...v11[i]=mem[x5+vs16[i]+6]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>h.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>h.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	111	vm	00000	rs1	101	vd	0000111							

13.7.172 VLXSEG<NF>HU.V: a vector indexed SEGMENT unsigned halfword load instruction

Syntax:

vlxseg<nf>hu.v vd, (rs1), vs2, vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + vs2[i] + 2*k]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4hu.v v8, (x5), v16. This instruction performs the following operations:

$$v8[0]=mem[x5+v16[0]] \cdots v8[i]=mem[x5+vs16[i]]$$

$$v9[0]=mem[x5+v16[0]+2] \cdots v9[i]=mem[x5+vs16[i]+2]$$

$$v10[0]=mem[x5+v16[0]+4] \cdots v10[i]=mem[x5+vs16[i]+4]$$

$$v11[0]=mem[x5+v16[0]+6] \cdots v11[i]=mem[x5+vs16[i]+6]$$

Permission:

M mode/S mode/U mode

Exception: Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads halfwords and then unsign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>hu.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>hu.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	011	vm	00000	rs1	101	vd	0000111

13.7.173 VLXSEG<NF>W.V: a vector indexed SEGMENT signed word load instruction

Syntax:

vlxseg<nf>w.v vd, (rs1), vs2, vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + vs2[i] + 4*k]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4w.v v8, (x5), v16. This instruction performs the following operations:

```
v8[0]=mem[x5+v16[0] ] ...v8[i]=mem[x5+vs16[i] ]
v9[0]=mem[x5+v16[0]+4] ...v9[i]=mem[x5+vs16[i]+4]
v10[0]=mem[x5+v16[0]+8] ...v10[i]=mem[x5+vs16[i]+8]
v11[0]=mem[x5+v16[0]+12] ...v11[i]=mem[x5+vs16[i]+12]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>w.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>w.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	111	vm	00000	rs1	110	vd	0000111							

13.7.174 VLXSEG<NF>WU.V: a vector indexed SEGMENT unsigned word load instruction

Syntax:

vlxseg<nf>wu.v vd, (rs1), vs2, vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    Vd+k[i] = mem[rs1 + vs2[i] + 4*k]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vd=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vlxseg4wu.v v8, (x5), v16. This instruction performs the following operations:

$$v8[0]=mem[x5+v16[0]] \cdots v8[i]=mem[x5+vs16[i]]$$

$$v9[0]=mem[x5+v16[0]+4] \cdots v9[i]=mem[x5+vs16[i]+4]$$

$$v10[0]=mem[x5+v16[0]+8] \cdots v10[i]=mem[x5+vs16[i]+8]$$

$$v11[0]=mem[x5+v16[0]+12] \cdots v11[i]=mem[x5+vs16[i]+12]$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then unsign-extends them to the current SEW.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vlxseg<nf>wu.v vd, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vlxseg<nf>wu.v vd, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	011		vm	00000			rs1	110		vd		0000111		

13.7.175 VLXW.V: a vector indexed signed word load instruction

Syntax:

`vlxw.v vd, (rs1), vs2, vm`

Operation:

$vd[i] = \text{sign_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction reads words and then sign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxw.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxw.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2			rs1		110		vd		0000111		

13.7.176 VLXWU.V: a vector indexed unsigned word load instruction

Syntax:

`vlxwu.v vd, (rs1), vs2, vm`

Operation:

$vd[i] = \text{zero_extend}(\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])])$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on load instructions, access error exceptions on load instructions, and page error exceptions on load instructions

Affected flag bits:

None.

Notes:

The instruction loads words and then unsign-extends them to the current SEW.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vlxwu.v vd, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vlxwu.v vd, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2			rs1			110	vd			0000111	

13.7.177 VMACC.VV: a vector lower-bit multiply-add instruction that overwrites addends

Syntax:

`vmacc.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{low_half}(vs1[i] \times vs2[i]) + vd[i];$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmacc.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmacc.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101			vm	vs2		vs1		010		vd		1010111

13.7.178 VMACC.VX: a vector-scalar lower-bit multiply-add instruction that overwrites addends

Syntax:

vmacc.vx vd, rs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{low_half}(rs1 \times vs2[i]) + vd[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmacc.vx vd, rs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmacc.vx vd, rs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101			vm	vs2		rs1		110		vd		1010111

13.7.179 VMADC.VVM: a vector integer add-with-carry instruction that produces the carry out

Syntax:

vmadc.vvm vd, vs2, vs1, v0

Operation:

$$vd[i] \leftarrow \text{carry_out}(vs2[i] + vs1[i] + v0[i].\text{LSB})$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010001		1	vs2		vs1		000		vd		1010111	

13.7.180 VMADC.VXM: a vector-scalar integer add-with-carry instruction that produces the carry out

Syntax:

vmadc.vxm vd, vs2, rs1, v0

Operation:

$vd[i] \leftarrow \text{carry_out}(vs2[i] + rs1 + v0[i].\text{LSB})$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010001		1	vs2		rs1		100		vd		1010111	

13.7.181 VMADC.VIM: a vector-immediate integer add-with-carry instruction that produces the carry out

Syntax:

vmadc.vim vd, vs2, imm, v0

Operation:

$vd[i] \leftarrow \text{carry_out}(vs2[i] + \text{sign_extend}(\text{imm}) + v0[i].\text{LSB})$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010001	1	vs2			imm		011	vd			1010111	

13.7.182 VMADD.VV: a vector lower-bit multiply-add instruction that overwrites multiplicands

Syntax:

vmadd.vv vd, vs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{low_half}(vs1[i] \times vd[i] + vs2[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmadd.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmadd.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101001	vm	vs2			vs1		010	vd			1010111	

13.7.183 VMADD.VX: a vector-scalar lower-bit multiply-add instruction that overwrites multiplicands

Syntax:

```
vmadd.vx vd, rs1, vs2, vm
```

Operation:

$$vd[i] \leftarrow \text{low_half}(rs1 \times vd[i] + vs2[i])$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vmadd.vx vd, rs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vmadd.vx vd, rs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101001		vm	vs2	rs1		110		vd		1010111		

13.7.184 VMAND.MM: a vector mask AND instruction

Syntax:

```
vmand.mm vd, vs2, vs1
```

Operation:

$$vd[i] \leftarrow \text{zero-extend}(vs2[i].\text{LSB} \&\& vs1[i].\text{LSB})$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
011001	1	vs2	vs1	010	vd	1010111

13.7.185 VMANDNOT.MM: a vector mask AND NOT instruction

Syntax:

vmandnot.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(vs2[i].\text{LSB} \&\& !vs1[i].\text{LSB})$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
011000	1	vs2	vs1	010	vd	1010111

13.7.186 VMAX.VV: a vector signed integer MAX instruction

Syntax:

vmax.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \max(\text{signed}(vs2[i]), \text{signed}(vs1[i]))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmax.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmax.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000111		vm	vs2		vs1		000		vd		1010111	

13.7.187 VMAX.VX: a vector-scalar signed integer MAX instruction

Syntax:

`vmax.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \max(\text{signed}(vs2[i]), \text{signed}(rs1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmax.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmax.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000111		vm	vs2		rs1		100		vd		1010111	

13.7.188 VMAXU.VV: a vector unsigned integer MAX instruction

Syntax:

vmaxu.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \max(\text{unsigned}(vs2[i]), \text{unsigned}(vs1[i]))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmaxu.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmaxu.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000110		vm	vs2		vs1		000		vd		1010111	

13.7.189 VMAXU.VX: a vector-scalar unsigned integer MAX instruction

Syntax:

vmaxu.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \max(\text{unsigned}(vs2[i]), \text{unsigned}(rs1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmaxu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmaxu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000110			vm	vs2		rs1		100		vd	1010111	

13.7.190 VMERGE.VVM: a vector element select instruction**Syntax:**

`vmerge.vvm vd, vs2, vs1, v0`

Operation:

$vd[i] \leftarrow v0[i].LSB ? vs1[i] : vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111			0	vs2		vs1		000		vd	1010111	

13.7.191 VMERGE.VXM: a vector-scalar element select instruction**Syntax:**

`vmerge.vxm vd, vs2, rs1, v0`

Operation:

$vd[i] \leftarrow v0[i].LSB ? rs1 : vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

This instruction is masked by default.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111		0	vs2	rs1		100		vd		1010111		

13.7.192 VMERGE.VIM: a vector-immediate element select instruction**Syntax:**

`vmerge.vim vd, vs2, imm, v0`

Operation:

$vd[i] \leftarrow v0[i].LSB ? sign_extend(imm) : vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111		0	vs2	imm		011		vd		1010111		

13.7.193 VMFEQ.VF: a vector-scalar floating-point compare equal instruction**Syntax:**

`vmfeq.vf vd, vs2, fs1,vm`

Operation:

$if(fs1 == vs2[i])$

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfeq.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfeq.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011000		vm	vs2		fs1		101		vd		1010111	

13.7.194 VMFEQ.VV: a vector floating-point compare equal instruction

Syntax:

`vmfeq.vv vd, vs2, vs1, vm`

Operation:

if($vs1 == vs2[i]$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfeq.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfeq.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011000		vm	vs2	vs1		001		vd		1010111		

13.7.195 VMFGE.VF: a vector-scalar floating-point compare greater than or equal to instruction

Syntax:

`vmfge.vf vd, vs2, fs1, vm`

Operation:

if($vs2[i] \geq fs1$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfge.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfge.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011111		vm	vs2	fs1		101		vd		1010111		

13.7.196 VMFGT.VF: a vector-scalar floating-point compare greater than instruction

Syntax:

```
vmfgt.vf vd, vs2, fs1, vm
```

Operation:

```
if(vs2[i] > fs1)
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfgt.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmfgt.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
011101			vm	vs2			fs1			101		vd	1010111	

13.7.197 VMFIRST: a vector mask find-first-set instruction

Syntax:

```
vmfirst.m rd, vs2, vm
```

Operation:

```
rd ← 0xffffffff
```

```
for (i=0; i < vl; i++){
```

```
    if(vs2[i].LSB == 1 && vs2[i] is active)
```

```
        rd ← i
```

```

        break
    }

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfirst.m rd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfirst.m rd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010101		vm	vs2		vs1		010		vd		1010111	

13.7.198 VMFLE.VF: a vector-scalar floating-point compare less than or equal to instruction

Syntax:

```
vmfle.vf vd, vs2, fs1,vm
```

Operation:

if($vs2[i] \leq fs1$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfle.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfle.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011001	vm			vs2			fs1			101	vd	1010111

13.7.199 VMFLE.VV: a vector floating-point compare less than or equal to instruction

Syntax:

`vmfle.vv vd, vs2, vs1, vm`

Operation:

if($vs2[i] \leq vs1$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfle.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfle.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011001	vm			vs2			vs1			001	vd	1010111

13.7.200 VMFLT.VF: a vector-scalar floating-point compare less than instruction

Syntax:

```
vmflt.vf vd, vs2, fs1,vm
```

Operation:

```
if(vs2[i] < fs1)
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmflt.vf vd, vs2, fs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmflt.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
011011			vm	vs2			fs1		101		vd		1010111	

13.7.201 VMFLT.VV: a vector floating-point compare less than instruction

Syntax:

```
vmflt.vv vd, vs2, vs1,vm
```

Operation:

```
if(vs2[i] < vs1)
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

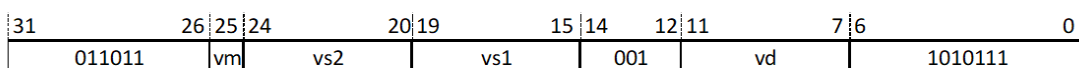
Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmflt.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmflt.vv vd, vs2, vs1, v0.t`.

Instruction format:

13.7.202 VMFNE.VF: a vector-scalar floating-point compare not equal instruction

Syntax:`vmfne.vf vd, vs2, fs1,vm`**Operation:**if(`fs1 != vs2[i]`)`vd[i] ← 1`

else

`vd[i] ← 0`**Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfne.vf vd, vs2, fs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfne.vf vd, vs2, fs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100		vm	vs2	fs1		101		vd		1010111		

13.7.203 VMFNE.VV: a vector floating-point compare not equal instruction

Syntax:

`vmfne.vv vd, vs2, vs1,vm`

Operation:

if($vs1 \neq vs2[i]$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmfne.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmfne.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100		vm	vs2	vs1		001		vd		1010111		

13.7.204 VMFORD.VF: a vector-scalar floating-point Not a Number (NaN) check instruction

Syntax:

vmford.vf vd, vs2, fs1,vm

Operation:

if(fs1 != NaN && vs2[i] != NaN)

vd[i] ← 1

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmford.vf vd, vs2, fs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmford.vf vd, vs2, fs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011010	vm	vs2		fs1		101		vd		1010111		

13.7.205 VMFORD.VV: a vector floating-point NaN check instruction

Syntax:

vmford.vv vd, vs2, vs1,vm

Operation:

if(vs1[i] != NaN && vs2[i] != NaN)

vd[i] ← 1

else

$$vd[i] \leftarrow 0$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

Floating-point status bit NV

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmford.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmford.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011010	vm		vs2		vs1		001		vd			1010111

13.7.206 VMIN.VV: a vector signed integer MIN instruction

Syntax:
`vmin.vv vd, vs2, vs1, vm`
Operation:

$$vd[i] \leftarrow \min(\text{signed}(vs2[i]), \text{signed}(vs1[i]))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmin.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmin.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000101			vm	vs2		vs1		000		vd		1010111

13.7.207 VMIN.VX: a vector-scalar signed integer MIN instruction**Syntax:**

vmin.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \min(\text{signed}(vs2[i]), \text{signed}(rs1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmin.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmin.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000101			vm	vs2		rs1		100		vd		1010111

13.7.208 VMINU.VV: a vector unsigned integer MIN instruction**Syntax:**

vminu.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \min(\text{unsigned}(vs2[i]), \text{unsigned}(vs1[i]))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vminu.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vminu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000100		vm	vs2	vs1		000		vd		1010111		

13.7.209 VMINU.VX: a vector-scalar unsigned integer MIN instruction

Syntax:

`vminu.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \min(\text{unsigned}(vs2[i]), \text{unsigned}(rs1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vminu.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vminu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000100		vm	vs2	rs1		100		vd		1010111		

13.7.210 VMNAND.MM: a vector mask NOT AND instruction

Syntax:

vmnand.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(\!(vs2[i].\text{LSB} \&\& vs1[i].\text{LSB}))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011101	1	vs2		vs1		010		vd		1010111		

13.7.211 VMNOR.MM: a vector mask NOT OR instruction

Syntax:

vmnor.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(\!(vs2[i].\text{LSB} \|\| vs1[i].\text{LSB}))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011110	1	vs2		vs1		010		vd		1010111		

13.7.212 VMOR.MM: a vector mask OR instruction**Syntax:**

vmor.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(vs2[i].\text{LSB} \parallel vs1[i].\text{LSB})$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011010	1	vs2		vs1		010		vd		1010111		

13.7.213 VMORNOT.MM: a vector mask OR NOT instruction**Syntax:**

vmornot.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(vs2[i].\text{LSB} \parallel !vs1[i].\text{LSB})$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100	1	vs2		vs1		010		vd		1010111		

13.7.214 VMPOPC.M: a vector mask population count instruction

Syntax:

vmopc.m rd, vs2, vm

Operation:

cnt = 0

for (i=0; i < vl; i++){

if(vs2[i].LSB == 1 && vs2[i] is active)

cnt←cnt+1

}

rd ←cnt

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmopc.m rd, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmopc.m rd, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010100	vm	vs2		vs1		010		vd		1010111		

13.7.215 VMSBC.VVM: a vector integer subtract-with-borrow instruction that produces the borrow out

Syntax:

vmsbc.vvm vd, vs2, vs1, v0

Operation:

$vd[i] \leftarrow borrow_out(vs2[i] - vs1[i] - v0[i].LSB)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010011		1	vs2	vs1		000		vd		1010111		

13.7.216 VMSBC.VXM: a vector-scalar integer subtract-with-borrow instruction that produces the borrow out

Syntax:

vmsbc.vxm vd, vs2, rs1, v0

Operation:

$vd[i] \leftarrow borrow_out(vs2[i] - rs1 - v0[i].LSB)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010011		1	vs2	rs1		100		vd		1010111		

13.7.217 VMSBF.M: a vector mask set-before-first instruction

Syntax:

vmsbf.m vd, vs2, vm

Operation:

index = active element && the first element in vs2 with LSB equals to 1

if (i < index)

vd[i] = 1

else

vd[i] = 0

}

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmsbf.m vd, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmsbf.m vd, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010110			vm	vs2		00001		010		vd		1010111

13.7.218 VMSIF.M: a vector mask set-including-first instruction

Syntax:

vmsif.m vd, vs2, vm

Operation:

index = active element && the first element in vs2 with LSB equals to 1

if (i <= index)

```

    vd[i] = 1
else
    vd[i] = 0
}

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsif.m vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmsif.m vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010110			vm	vs2		00011		010		vd		1010111

13.7.219 VMSOF.M: a vector mask set-only-first instruction**Syntax:**

```
vmsof.m vd, vs2, vm
```

Operation:

index = active element && the first element in *vs2* with LSB equals to 1

```
vd[index] = 1
```

```
vd[others] = 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsof.m vd, vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmsof.m vd, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010110		vm	vs2		00010		010		vd		1010111	

13.7.220 VMSEQ.VV: a vector integer compare equal instruction

Syntax:

`vmseq.vv vd, vs2, vs1, vm`

Operation:

if(`vs2[i] == vs1[i]`)

`vd[i] ← 1`

else

`vd[i] ← 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmseq.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmseq.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011000		vm	vs2		vs1		000		vd		1010111	

13.7.221 VMSEQ.VX: a vector-scalar integer compare equal instruction

Syntax:

```
vmseq.vx vd, vs2, rs1, vm
```

Operation:

```
if(vs2[i] == rs1)
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmseq.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmseq.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
011000			vm	vs2			rs1		100		vd		1010111	

13.7.222 VMSEQ.VI: a vector-immediate integer compare equal instruction

Syntax:

```
vmseq.vi vd, vs2, imm, vm
```

Operation:

```
if(signed(vs2[i]) == signed(sign_extend(imm)))
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```


Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

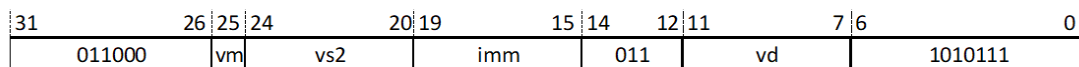
Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmseq.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmseq.vi vd, vs2, imm, v0.t`.

Instruction format:

13.7.223 VMSGT.VX: a vector-scalar signed integer compare greater than instruction

Syntax:

`vmsgt.vx vd, vs2, rs1, vm`

Operation:

if(`signed(vs2[i]) > signed(rs1)`)

`vd[i] ← 1`

else

`vd[i] ← 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsgt.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsgt.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011111		vm	vs2		rs1		100		vd		1010111	

13.7.224 VMSGT.VI: a vector-immediate signed integer compare greater than instruction

Syntax:

`vmsgt.vi vd, vs2, imm, vm`

Operation:

`if(signed(vs2[i]) > signed(sign_extend(imm)))`

`vd[i] ← 1`

else

`vd[i] ← 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsgt.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsgt.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011111		vm	vs2		imm		011		vd		1010111	

13.7.225 VMSGTU.VX: a vector-scalar unsigned integer compare greater than instruction

Syntax:

vmsgtu.vx vd, vs2, rs1, vm

Operation:

if(unsigned(vs2[i] > unsigned(rs1))

vd[i] ← 1

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmsgtu.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmsgtu.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011110		vm	vs2		rs1		100		vd		1010111	

13.7.226 VMSGTU.VI: a vector-immediate unsigned integer compare greater than instruction

Syntax:

vmsgtu.vi vd, vs2, imm, vm

Operation:

if(unsigned(vs2[i] > unsigned(sign_extend(imm)))

vd[i] ← 1

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsgtu.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmsgtu.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011110	vm		vs2		imm		011		vd		1010111	

13.7.227 VMSLE.VV: a vector signed integer compare less than or equal to instruction

Syntax:

`vmsle.vv vd, vs2, vs1, vm`

Operation:

if($\text{signed}(vs2[i]) \leq \text{signed}(vs1[i])$)

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsle.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsle.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011101	vm			vs2			vs1	000			vd	1010111

13.7.228 VMSLE.VX: a vector-scalar signed integer compare less than or equal to instruction

Syntax:

`vmsle.vx vd, vs2, rs1, vm`

Operation:

if(signed(vs2[i]) <= signed(rs1))

vd[i] ← 1

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsle.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsle.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011101	vm			vs2			rs1	100			vd	1010111

13.7.229 VMSLE.VI: a vector-immediate signed integer compare less than or equal to instruction

Syntax:

vmsle.vi vd, vs2, imm, vm

Operation:

if(signed(vs2[i]) <= signed(sign_extend(imm)))

vd[i] ← 1

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmsle.vi vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmsle.vi vd, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011101	vm	vs2		imm		011		vd		1010111		

13.7.230 VMSLEU.VV: a vector unsigned integer compare less than or equal to instruction

Syntax:

vmsleu.vv vd, vs2, vs1, vm

Operation:

if(unsigned(vs2[i]) <= unsigned(vs1[i]))

vd[i] ← 1

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsleu.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmsleu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100		vm	vs2		vs1		000		vd		1010111	

13.7.231 VMSLEU.VX: a vector-scalar unsigned integer compare less than or equal to instruction

Syntax:

`vmsleu.vx vd, vs2, rs1, vm`

Operation:

$\text{if}(\text{unsigned}(vs2[i]) \leq \text{unsigned}(rs1))$

$vd[i] \leftarrow 1$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsleu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsleu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100		vm	vs2		rs1		100		vd		1010111	

13.7.232 VMSLEU.VI: a vector-immediate unsigned integer compare less than or equal to instruction

Syntax:

`vmsleu.vi vd, vs2, imm, vm`

Operation:

$\text{if}(\text{unsigned}(\text{vs2}[i]) \leq \text{unsigned}(\text{sign_extend}(\text{imm})))$

$\text{vd}[i] \leftarrow 1$

else

$\text{vd}[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsleu.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsleu.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011100			vm	vs2		imm		011		vd		1010111

13.7.233 VMSLT.VV: a vector signed integer compare less than instruction

Syntax:

vmslt.vv vd, vs2, vs1, vm

Operation:

if(signed(vs2[i]) < signed(vs1[i]))

vd[i] ← 1

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vmslt.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vmslt.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011011			vm	vs2		vs1		000		vd		1010111

13.7.234 VMSLT.VX: a vector-scalar signed integer compare less than instruction

Syntax:

vmslt.vx vd, vs2, rs1, vm

Operation:

if(signed(vs2[i]) < signed(rs1))

$$vd[i] \leftarrow 1$$

else

$$vd[i] \leftarrow 0$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

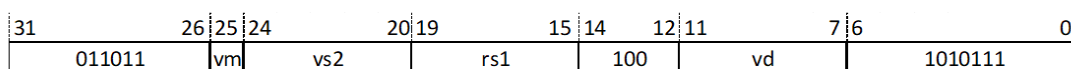
Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmslt.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmslt.vx vd, vs2, rs1, v0.t`.

Instruction format:

13.7.235 VMSLTU.VV: a vector unsigned integer compare less than instruction

Syntax:

$$vmsltu.vv vd, vs2, vs1, vm$$
Operation:

if(unsigned(vs2[i]) < unsigned(vs1[i]))

$$vd[i] \leftarrow 1$$

else

$$vd[i] \leftarrow 0$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsltu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsltu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011010			vm	vs2		vs1		000		vd		1010111

13.7.236 VMSLTU.VX: a vector-scalar unsigned integer compare less than instruction

Syntax:

`vmsltu.vx vd, vs2, rs1, vm`

Operation:

if(`signed(vs2[i]) < signed(rs1)`)

`vd[i] ← 1`

else

`vd[i] ← 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsltu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsltu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011010			vm	vs2		rs1		100		vd		1010111

13.7.237 VMSNE.VV: a vector integer compare not equal instruction

Syntax:

```
vmsne.vv vd, vs2, vs1, vm
```

Operation:

```
if(vs2[i] != vs1[i])
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsne.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmsne.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
011001			vm	vs2			vs1			000		vd	1010111	

13.7.238 VMSNE.VX: a vector-scalar integer compare not equal instruction

Syntax:

```
vmsne.vx vd, vs2, rs1, vm
```

Operation:

```
if(vs2[i] != rs1)
```

```
    vd[i] ← 1
```

```
else
```

```
    vd[i] ← 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsne.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsne.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011001	vm		vs2		rs1		100		vd			1010111

13.7.239 VMSNE.VI: a vector-immediate integer compare not equal instruction**Syntax:**

$$\text{vmsne.vi vd, vs2, imm, vm}$$
Operation:

$$\text{if}(\text{signed}(\text{vs2}[i]) \neq \text{signed}(\text{sign_extend}(\text{imm})))$$

$$\text{vd}[i] \leftarrow 1$$

else

$$\text{vd}[i] \leftarrow 0$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmsne.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmsne.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
011001			vm	vs2			imm			011		vd	1010111	

13.7.240 VMUL.VV: a vector integer multiply instruction that returns lower bits

Syntax:

`vmul.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{low_half}(vs2[i] \times vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmul.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmul.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
100101			vm	vs2			vs1			010		vd	1010111	

13.7.241 VMUL.VX: a vector-scalar integer multiply instruction that returns lower bits

Syntax:

`vmul.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{low_half}(vs2[i] \times rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmul.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmul.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100101			vm	vs2		rs1		110		vd		1010111

13.7.242 VMULH.VV: a vector signed integer multiply instruction that returns upper bits

Syntax:
`vmulh.vv vd, vs2, vs1, vm`
Operation:

$$vd[i] \leftarrow \text{high_half}(\text{signed}(vs2[i]) \times \text{signed}(vs1[i]))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulh.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmulh.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100111		vm	vs2	vs1		010		vd		1010111		

13.7.243 VMULH.VX: a vector-scalar signed integer multiply instruction that returns upper bits

Syntax:

`vmulh.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{high_half}(\text{signed}(vs2[i]) \times \text{signed}(rs1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulh.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmulh.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100111		vm	vs2	rs1		110		vd		1010111		

13.7.244 VMULHU.VV: a vector unsigned integer multiply instruction that returns upper bits

Syntax:

`vmulhu.vv vd, vs2, vs1, vm`

Operation:

$$vd[i] \leftarrow \text{high_half}(\text{unsigned}(vs2[i]) \times \text{unsigned}(vs1[i]))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulhu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmulhu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100100		vm	vs2	vs1		010		vd		1010111		

13.7.245 VMULHU.VX: a vector-scalar unsigned integer multiply instruction that returns upper bits

Syntax:
`vmulhu.vx vd, vs2, rs1, vm`
Operation:

$$vd[i] \leftarrow \text{high_half}(\text{unsigned}(vs2[i]) \times \text{unsigned}(rs1))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulhu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmulhu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100100		vm	vs2		rs1		110		vd		1010111	

13.7.246 VMULHSU.VV: a vector signed-unsigned integer multiply instruction that returns upper bits

Syntax:

`vmulhsu.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{high_half}(\text{signed}(vs2[i]) \times \text{unsigned}(vs1[i]))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulhsu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vmulhsu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100110		vm	vs2		vs1		010		vd		1010111	

13.7.247 VMULHSU.VX: a vector-scalar signed-unsigned integer multiply instruction that returns upper bits

Syntax:

`vmulhsu.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{high_half}(\text{signed}(vs2[i]) \times \text{unsigned}(rs1))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vmulhsu.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vmulhsu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100110	vm		vs2		rs1		110		vd			1010111

13.7.248 VMV.V.V: a vector element move instruction**Syntax:**
`vmv.vv vd, vs1`
Operation:

$$vd[i] \leftarrow vs1[i]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111	1		00000		vs1		000		vd			1010111

13.7.249 VMV.V.X: an instruction that moves an integer scalar to a vector

Syntax:

vmv.v.x vd, rs1

Operation:

vd[i] ← rs1

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111	1	00000	rs1	100	vd	1010111						

13.7.250 VMV.V.I: an instruction that moves an immediate to a vector

Syntax:

vmv.v.i vd, imm

Operation:

vd[i] ← sign_extend(imm)

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010111	1	00000	imm	011	vd	1010111						

13.7.251 VMV.S.X: an instruction that moves an integer scalar to element 0 of a vector

Syntax:

vmv.s.x vd, rs1

Operation:

vd[0] = rs1

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001101	1	00000	rs1	110	vd	1010111						

13.7.252 VMXOR.MM: a vector mask XOR instruction

Syntax:

vmxor.mm vd, vs2, vs1

Operation:

vd[i] ← zero-extend(vs2[i].LSB ^ vs1[i].LSB)

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011011		1	vs2	vs1		010		vd		1010111		

13.7.253 VMXNOR.MM: a vector mask XNOR instruction

Syntax:

vmxnor.mm vd, vs2, vs1

Operation:

$vd[i] \leftarrow \text{zero-extend}(!(\text{vs2}[i].\text{LSB} \wedge \text{vs1}[i].\text{LSB}))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

vs2 and vs1 are ordered in the mask mode for operations.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
011111		1	vs2	vs1		010		vd		1010111		

13.7.254 VNCLIP.VV: a vector narrowing signed arithmetic right shift instruction with result saturated when necessary

Syntax:

vnclip.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{clip}(\text{sign_extend}((\text{vs2}[i]) + \text{round})) \gg \text{vs1}[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vnclip.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vnclip.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101111			vm	vs2		vs1		000		vd		1010111

13.7.255 VNCLIP.VX: a vector-scalar narrowing signed arithmetic right shift instruction with result saturated when necessary

Syntax:

vnclip.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{clip}(\text{sign_extend}((vs2[i] + \text{round})) \gg rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.

- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnclip.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnclip.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
101111			vm	vs2			rs1			100		vd	1010111	

13.7.256 VNCLIP.VI: a vector-immediate narrowing signed arithmetic right shift instruction with result saturated when necessary

Syntax:

`vnclip.vi vd, vs2, imm, vm`

Operation:

$vd[i] \leftarrow clip(sign_extend((vs2[i]) + round)) \gg unsigned(imm)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (`vrxm`):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnclip.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnclip.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
101111			vm	vs2			imm		011		vd	1010111	

13.7.257 VNCLIPU.VV: a vector narrowing unsigned arithmetic right shift instruction with result saturated when necessary

Syntax:

vnclipu.vv vd, vs2, vs1, vm

Operation:

$$vd[i] \leftarrow \text{clipu}(\text{zero_extend}((vs2[i] + \text{round})) \gg vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vnclipu.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vnclipu.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
101110			vm	vs2			vs1		000		vd	1010111	

13.7.258 VNCLIPU.VX: a vector narrowing unsigned arithmetic right shift instruction with result saturated when necessary

Syntax:

`vnclipu.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{clipu}(\text{zero_extend}((vs2[i]) + \text{round})) \gg rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnclipu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnclipu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101110		vm	vs2		rs1		100		vd		1010111	

13.7.259 VNCLIPU.VI: a vector-immediate narrowing unsigned arithmetic right shift instruction with result saturated when necessary

Syntax:

`vnclipu.vi vd, vs2, imm, vm`

Operation:

$$vd[i] \leftarrow \text{clipu}(\text{zero_extend}((vs2[i] + \text{round})) \gg \text{unsigned}(\text{imm}))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vnclipu.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vnclipu.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101110		vm	vs2		imm		011		vd		1010111	

13.7.260 VNMSAC.VV: a vector lower-bit multiply-sub instruction that overwrites minuends

Syntax:
`vnmsac.vv vd, vs1, vs2, vm`
Operation:

$$vd[i] \leftarrow - (\text{low_half}(vs1[i] \times vs2[i])) + vd[i]$$

Execution permissions: M-mode/S-mode/U-mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnmsac.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnmsac.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
1011111			vm	vs2			vs1			010	vd		1010111	

13.7.261 VNMSAC.VX: a vector-scalar lower-bit multiply-sub instruction that overwrites minuends

Syntax:

`vnmsac.vx vd, rs1, vs2, vm`

Operation:

$vd[i] \leftarrow -(low_half(rs1 \times vs2[i])) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnmsac.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnmsac.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
1011111			vm	vs2			rs1			110	vd		1010111	

13.7.262 VNMSUB.VV: a vector lower-bit negate-(multiply-sub) instruction that overwrites multiplicands

Syntax:

vnmsub.vv vd, vs1, vs2, vm

Operation:

$$vd[i] \leftarrow -(low_half(vs1[i] \times vd[i]) + vs2[i])$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vnmsub.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vnmsub.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011		vm	vs2		vs1		010		vd		1010111	

13.7.263 VNMSUB.VX: a vector-scalar lower-bit negate-(multiply-sub) instruction that overwrites multiplicands

Syntax:

vnmsub.vx vd, rs1, vs2, vm

Operation:

$$vd[i] \leftarrow -(low_half(rs1 \times vd[i])) + vs2[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnmsub.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnmsub.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011	vm		vs2		rs1		110		vd			1010111

13.7.264 VNSRA.VV: a vector narrowing arithmetic right shift instruction**Syntax:**
`vnsra.vv vd, vs2, vs1, vm`
Operation:

$$vd[i] \leftarrow \text{sign_extend}(vs2[i]) \gg vs1[i]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnrsra.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnsra.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101	vm		vs2		vs1		000		vd			1010111

13.7.265 VNSRA.VX: a vector-scalar narrowing arithmetic right shift instruction

Syntax:

```
vnsra.vx vd, vs2, rs1, vm
```

Operation:

$$vd[i] \leftarrow \text{sign_extend}(vs2[i]) \gg rs1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnsra.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnsra.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101		vm	vs2		rs1		100		vd		1010111	

13.7.266 VNSRA.VI: a vector-immediate narrowing arithmetic right shift instruction

Syntax:

```
vnsra.vi vd, vs2, imm, vm
```

Operation:

$$vd[i] \leftarrow \text{sign_extend}(vs2[i]) \gg \text{unsigned}(imm)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnrsla.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnsra.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101101			vm	vs2		imm		011		vd		1010111

13.7.267 VNSRL.VV: a vector narrowing logical right shift instruction**Syntax:**

`vnsrl.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{zero_extend}(vs2[i]) \gg\gg vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnsrl.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnsrl.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101100			vm	vs2		vs1		000		vd		1010111

13.7.268 VNSRL.VX: a vector-scalar narrowing logical right shift instruction**Syntax:**

`vnsrl.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{zero_extend}(vs2[i]) \gg \gg rs1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnrsl.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnrsl.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101100		vm	vs2		rs1		100		vd		1010111	

13.7.269 VNSRL.VI: a vector-immediate narrowing logical right shift instruction

Syntax:

$$vnsrl.vi\ vd, vs2, imm, vm$$
Operation:

$$vd[i] \leftarrow \text{zero_extend}(vs2[i]) \gg \gg \text{unsigned}(imm)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vnrsl.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vnsrl.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101100		vm	vs2		imm		011		vd		1010111	

13.7.270 VOR.VV: a vector bitwise OR instruction

Syntax:

`vor.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] \mid vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vor.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vor.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001010		vm	vs2		vs1		000		vd		1010111	

13.7.271 VOR.VX: a vector-scalar bitwise OR instruction

Syntax:

`vor.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] \mid rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vor.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vor.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001010		vm	vs2		rs1		100		vd		1010111	

13.7.272 VOR.VI: a vector-immediate bitwise OR instruction**Syntax:**`vor.vi vd, vs2, imm, vm`**Operation:** $vd[i] \leftarrow vs2[i] \mid \text{sign_extend}(imm)$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vor.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vor.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001010		vm	vs2		imm		011		vd		1010111	

13.7.273 VREDAND.VS: a vector reduction bitwise AND instruction**Syntax:**

```
vredand.vs vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
```

```
for( i=0; i<vl; i++) {
```

```
    tmp = and(tmp, vs2[i])
```

```
}
```

```
vd[0]= tmp
```

```
vd[VLEN/SEW-1:1] = 0
```

Permission:

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

Exception:

```
Invalid instruction.
```

Affected flag bits:

```
None.
```

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredand.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredand.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000001		vm	vs2		vs1		010		vd		1010111	

13.7.274 VREDMAX.VS: a vector reduction signed MAX instruction**Syntax:**

```
vredmax.vs vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
for( i=0; i<vl; i++) {
    tmp = max(tmp, vs2[i])
}
vd[0]= tmp
vd[VLEN/SEW-1:1] = 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredmax.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredmax.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
000111	vm	vs2	vs1	010	vd	1010111

13.7.275 VREDMAXU.VS: a vector reduction unsigned MAX instruction

Syntax:

```
vredmaxu.vs vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
for( i=0; i<vl; i++) {
    tmp = maxu(tmp, vs2[i])
}
```

```

}
vd[0]= tmp
vd[VLEN/SEW-1:1] = 0

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredmaxu.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredmaxu.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000110	vm			vs2			vs1	010			vd	1010111

13.7.276 VREDMIN.VS: a vector reduction signed MIN instruction**Syntax:**

```
vredmin.vs vd, vs2, vs1, vm
```

Operation:

```

tmp = vs1[0]
for( i=0; i<vl; i++) {
    tmp = min(tmp, vs2[i])
}

```

```

vd[0]= tmp
vd[VLEN/SEW-1:1] = 0

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredmin.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredmin.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000101	vm			vs2			vs1	010			vd	1010111

13.7.277 VREDMINU.VS: a vector reduction unsigned MIN instruction

Syntax:

`vredmaxu.vs vd, vs2, vs1, vm`

Operation:

`tmp = vs1[0]`

`for(i=0; i<vl; i++) {`

`tmp = minu(tmp, vs2[i])`

`}`

`vd[0]= tmp`

`vd[VLEN/SEW-1:1] = 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredminu.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredminu.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000101		vm	vs2	vs1		010		vd		1010111		

13.7.278 VREDOR.VS: a vector reduction bitwise OR instruction

Syntax:

`vredor.vs vd, vs2, vs1, vm`

Operation:

`tmp = vs1[0]`

`for(i=0; i<v1; i++) {`

`tmp = or(tmp, vs2[i])`

`}`

`vd[0]= tmp`

`vd[VLEN/SEW-1:1] = 0`

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredor.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredor.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000010		vm	vs2	vs1		010		vd		1010111		

13.7.279 VREDXOR.VS: a vector reduction bitwise XOR instruction

Syntax:

```
vredxor.vs vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
for( i=0; i<vl; i++) {
    tmp = xor(tmp, vs2[i])
}
vd[0]= tmp
vd[VLEN/SEW-1:1] = 0
```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredxor.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredxor.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000011		vm	vs2	vs1		010		vd		1010111		

13.7.280 VREDSUM.VS: a vector reduction sum instruction

Syntax:

```
vredsum.vs vd, vs2, vs1, vm
```

Operation:

```
tmp = vs1[0]
for( i=0; i<vl; i++) {
```

```

    tmp = tmp + vs2[i]
}
vd[0] = tmp
vd[VLEN/SEW-1:1] = 0

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vredsum.vs vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vredsum.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000000			vm	vs2		vs1		010		vd		1010111

13.7.281 VREM.VV: a vector signed integer remainder instruction**Syntax:**

```
vrem.vv vd, vs2, vs1, vm
```

Operation:

$$vd[i] \leftarrow \text{signed}(vs2[i]) \% \text{signed}(vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vrem.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vrem.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011		vm	vs2		vs1		010		vd		1010111	

13.7.282 VREM.VX: a vector-scalar signed integer remainder instruction

Syntax:

`vrem.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) \% \text{signed}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vrem.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vrem.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011		vm	vs2		rs1		110		vd		1010111	

13.7.283 VREMU.VV: a vector unsigned integer remainder instruction

Syntax:

`vremu.vv vd, vs2, vs1, vm`

Operation:

$$vd[i] \leftarrow \text{unsigned}(vs2[i]) \% \text{unsigned}(vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vremu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vremu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010		vm	vs2	vs1		010		vd		1010111		

13.7.284 VREMU.VX: a vector-scalar unsigned integer remainder instruction

Syntax:
`vremu.vx vd, vs2, rs1, vm`
Operation:

$$vd[i] \leftarrow \text{unsigned}(vs2[i]) \% \text{unsigned}(rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vremu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vremu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		rs1		110		vd		1010111

13.7.285 VRGATHER.VV: a vector index-based gather instruction for integer elements**Syntax:**

vrgather.vv vd, vs2, vs1, vm

Operation:

$vd[i] = (vs1[i] \ll VLMAX) \ ?0 : vs2[vs1[i]]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vrgather.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vrgather.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001100			vm	vs2		vs1		000		vd		1010111

13.7.286 VRGATHER.VX: a vector-scalar index-based gather instruction for integer elements**Syntax:**

vrgather.vx vd, vs2, rs1, vm

Operation:

$vd[i] = (\text{unsigned}(rs1) \ll VLAMX) \ ?0 : vs2[\text{unsigned}(rs1)]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vrgather.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vrgather.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
001100			vm	vs2			rs1		100		vd	1010111	

13.7.287 VRGATHER.VI: a vector-immediate index-based gather instruction for integer elements

Syntax:

`vrgather.vi vd, vs2, imm, vm`

Operation:

$vd[i] = (\text{unsigned}(\text{imm}) \ll \text{VLMAX}) \ ? 0 : vs2[\text{unsigned}(\text{imm})]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vrgather.vi vd, vs2,imm, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vrgather.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001100			vm	vs2		imm		011		vd		1010111

13.7.288 VRSUB.VX: a vector-scalar integer subtract instruction

Syntax:

vrsub.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow rs1 - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vrsub.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vrsub.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000011			vm	vs2		rs1		100		vd		1010111

13.7.289 VRSUB.VI: an immediate-vector integer subtract instruction

Syntax:

vrsub.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow \text{sign_extend}(\text{imm}) - vs2[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vrsb.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vrsb.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000011		vm	vs2		imm		011		vd		1010111	

13.7.290 VSADD.VV: a vector saturating signed integer add instruction

Syntax:

`vsadd.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] + vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsadd.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsadd.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100001		vm	vs2		vs1		000		vd		1010111	

13.7.291 VSADD.VX: a vector-scalar saturating signed integer add instruction

Syntax:

vsadd.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] + rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsadd.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsadd.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100001		vm	vs2	rs1		100		vd		1010111		

13.7.292 VSADD.VI: a vector-immediate saturating signed integer add instruction

Syntax:

vsadd.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] + \text{sign_extend}(\text{imm}))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsadd.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsadd.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100001		vm	vs2		imm		011		vd		1010111	

13.7.293 VSADDU.VV: a vector saturating unsigned integer add instruction**Syntax:**

`vsaddu.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] + vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsaddu.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsaddu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100000		vm	vs2		vs1		000		vd		1010111	

13.7.294 VSADDU.VX: a vector-scalar saturating unsigned integer add instruction**Syntax:**

`vsaddu.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{sat}(vs2[i] + rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsaddu.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsaddu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26 25 24	20 19	15 14	12 11	7 6	0
100000	vm	vs2	rs1	100	vd	1010111

13.7.295 VSADDU.VI: a vector-immediate saturating unsigned integer add instruction

Syntax:
`vsaddu.vi vd, vs2, imm, vm`
Operation:

$$vd[i] \leftarrow \text{sat}(vs2[i] + \text{sign_extend}(\text{imm}))$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsaddu.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsaddu.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
100000			vm	vs2			imm		011		vd		1010111	

13.7.296 VSB.V: a vector byte store instruction

Syntax:

`vsb.v vs3, (rs1), vm`

Operation:

$\text{mem}[\text{rs1}+i] = \text{vs3}[i][7:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsb.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsb.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0			
000			000			vm	00000			rs1		000		vs3		0100111	

13.7.297 VSBC.VVM: a vector integer subtract-with-borrow instruction

Syntax:

`vsbc.vvm vd, vs2, vs1, v0`

Operation:

$\text{vd}[i] \leftarrow \text{vs2}[i] - \text{vs1}[i] - \text{v0}[i].\text{LSB}$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010010		1	vs2	vs1		000		vd		1010111		

13.7.298 VSBC.VXM: a vector-scalar integer subtract-with-borrow instruction**Syntax:**

vsbc.vxm vd, vs2, rs1, v0

Operation: $vd[i] \leftarrow vs2[i] - rs1 - v0[i].LSB$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
010010		1	vs2	rs1		100		vd		1010111		

13.7.299 VSE.V: a vector element store instruction**Syntax:**

vse.v vs3, (rs1), vm

Operation: $mem[rs1+i*SEW/8] = vs3[i]$ **Permission:**

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vse.v vs3, (rs1)`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vse.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	00000			rs1		111		vd		0100111		

13.7.300 VSETVL: an instruction that sets *vtype* and *vl* CSRs

Syntax:

`vsetvl rd, rs1, rs2`

Operation:

`rd` =new *vl*, `rs1`=AVL, `rs2`=new *vtype* value

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `rs1` is `x0`, *vl* is set to VLMAX.

When SEW is set to 128 or another unsupported *vtype* value, the VILL bit is set to 1.

Instruction format:

31	30	25	24	20	19	15	14	12	11	7	6	0
1	000000			rs2		rs1		111		rd		1010111

13.7.301 VSETVLI: an instruction that sets vl and vtype with immediate values

Syntax:

vsetvli rd, rs1, vtypei

Operation:

rd =new vl, rs1=AVL, vtypei =new vtype setting

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of rs1 is x0, vl is set to VLMAX.

When SEW is set to 128 or another unsupported vtype value, the VILL bit is set to 1.

Instruction format:

31	30	20	19	15	14	12	11	7	6	0
1	imm			rs1		111		rd		1010111

13.7.302 VSH.V: a vector halfword store instruction

Syntax:

vsh.v vs3, (rs1), vm

Operation:

mem[rs1+2*i] = vs3[i][15:0]

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsh.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsh.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	00000			rs1			101	vd			0100111	

13.7.303 VSLIDEDOWN.VX: a vector slide instruction that moves elements down

Syntax:

`vslidedown.vx vd, vs2, rs1, vm`

Operation:

if($(i + \text{unsigned}(rs)) < \text{VLMAX}$)

$vd[i] \leftarrow vs2[i + \text{unsigned}(rs1)]$

else

$vd[i] \leftarrow 0$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vslidedown.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vslidedown.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001111		vm	vs2		rs1		100		vd		1010111	

13.7.304 VSLIDEDOWN.VI: a vector-immediate slide instruction that moves elements down

Syntax:

vslidedown.vi vd, vs2, imm, vm

Operation:

if((i + unsigned(imm)) < VLMAX)

vd[i] ← vs2[i + unsigned(imm)]

else

vd[i] ← 0

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vslidedown.vx vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vslidedown.vx vd, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001111		vm	vs2		imm		011		vd		1010111	

13.7.305 VSLIDE1DOWN.VX: a vector slide instruction that moves elements down by 1 index

Syntax:

vslide1down.vx vd, vs2, rs1, vm

Operation:

$vd[i] = vs2[i+1]$

$vd[vl-1] = rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vslide1down.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vslide1down.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001111			vm	vs2		rs1		110		vd		1010111

13.7.306 VSLIDEUP.VX: a vector slide instruction that moves elements up**Syntax:**

vslideup.vx vd, vs2, rs1, vm

Operation:

if($i < \text{unsigned}(rs1)$)

$vd[i] \leftarrow vd[i]$

else

$vd[i] \leftarrow vs2[i - \text{unsigned}(rs1)]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vslideup.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vslideup.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001110	vm		vs2		rs1		100		vd			1010111

13.7.307 VSLIDEUP.VI: a vector-immediate slide instruction that moves elements up**Syntax:**
`vslideup.vi vd, vs2, imm, vm`
Operation:
`if((i < unsigned(imm)))`
`vd[i] ← vd[i]`
`else`
`vd[i] ← vs2[i - unsigned(imm)]`
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vslideup.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vslideup.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001110			vm	vs2			imm		011	vd		1010111

13.7.308 VSLIDE1UP.VX: a vector slide instruction that moves elements up by 1 index

Syntax:

vslide1up.vx vd, vs2, rs1, vm

Operation:

vd[0] = rs1

vd[i+1] = vs2[i]

Permission:

M-mode/S-mode/U-mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vslide1up.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vslide1up.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001110			vm	vs2			rs1		110	vd		1010111

13.7.309 VSLL.VV: a vector logical left shift instruction

Syntax:

vsll.vv vd, vs2, vs1, vm

Operation:

vd[i] ← unsigned(vs2[i]) << vs1[i]

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsl.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsl.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100101	vm			vs2			vs1	000		vd	1010111	

13.7.310 VSL.VX: a vector-scalar logical left shift instruction

Syntax:

`vsl.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \ll rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsl.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsl.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100101	vm			vs2			rs1	100		vd	1010111	

13.7.311 VSLL.VI: a vector-immediate logical left shift instruction

Syntax:

vsll.vi vd, vs2, imm, vm

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \ll \text{unsigned}(imm)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsll.vi vd, vs2, imm.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsll.vi vd, vs2, imm, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100101	vm	vs2		imm		011		vd		1010111		

13.7.312 VSMUL.VV: a vector saturating multiply instruction

Syntax:

vsmul.vv vd, vs2, vs1, vm

Operation:

$vd[i] = \text{clip}((vs2[i] \times vs1[i] + \text{round}) \gg (\text{SEW} - 1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vsmul.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vsmul.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100111		vm	vs2		vs1		000		vd		1010111	

13.7.313 VSMUL.VX: a vector-scalar saturating multiply instruction**Syntax:**

`vsmul.vx vd, vs2, rs1, vm`

Operation:

$vd[i] = \text{clip}((vs2[i] \times rs1 + \text{round}) \gg (\text{SEW} - 1))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

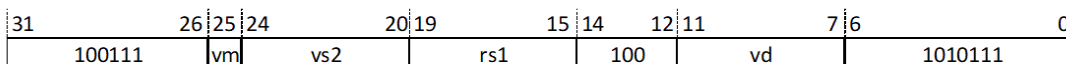
Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsmul.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsmul.vx vd, vs2, rs1, v0.t`.

Instruction format:



13.7.314 VSRA.VV: a vector arithmetic right shift instruction

Syntax:

`vsra.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) \ggg vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

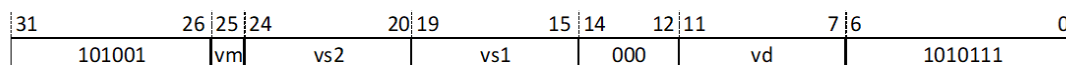
None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsra.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsra.vv vd, vs2, vs1, v0.t`.

Instruction format:



13.7.315 VSRA.VX: a vector-scalar arithmetic right shift instruction

Syntax:

`vsra.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{signed}(vs2[i]) \gg\gg rs1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsra.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsra.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101001		vm	vs2	rs1		100		vd		1010111		

13.7.316 VSRA.VI: a vector-immediate arithmetic right shift instruction

Syntax:
`vsra.vi vd, vs2, imm, vm`
Operation:

$$vd[i] \leftarrow \text{signed}(vs2[i]) \gg\gg \text{unsigned}(imm)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsra.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsra.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0	
101001			vm	vs2			imm		011		vd	1010111	

13.7.317 VSRL.VV: a vector logical right shift instruction**Syntax:**

vsrl.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \gg vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsrl.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsrl.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101000			vm	vs2		vs1		000		vd	1010111	

13.7.318 VSRL.VX: a vector-scalar logical right shift instruction**Syntax:**

vsrl.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \gg rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsrl.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsrl.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101000		vm	vs2		rs1		100		vd		1010111	

13.7.319 VSRL.VI: a vector-immediate logical right shift instruction

Syntax:

`vsrl.vi vd, vs2, imm, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \gg \text{unsigned}(imm)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsrl.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsrl.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101000		vm	vs2		imm		011		vd		1010111	

13.7.320 VSSB.V: a vector strided byte store instruction

Syntax:

$$\text{vssb.v vs3, (rs1), rs2, vm}$$
Operation:

$$\text{mem}[\text{rs1} + i * \text{rs2}] = \text{vs3}[i][7:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vssb.v vs3, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vssb.v vs3, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	010	vm	rs2			rs1			000	vd			0100111	

13.7.321 VSSE.V: a vector strided element store instruction

Syntax:

$$\text{vsse.v vs3, (rs1), rs2, vm}$$
Operation:

$$\text{mem}[\text{rs1} + i * \text{rs2}] = \text{vs3}[i][\text{SEW}-1:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsse.v vs3, (rs1), rs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsse.v vs3, (rs1), rs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100101	vm		vs2		imm		011		vd			1010111

13.7.322 VSSEG<NF>B.V: a vector SEGMENT byte store instruction**Syntax:**

$$\text{vsseg}\langle\text{nf}\rangle\text{b.v vs3, (rs1), vm}$$
Operation:

```
for( k=0; k<=nf-1; k++) {
  mem[rs1 + k + i *nf] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. *nf* ranges from 2 to 8. For example, when *nf*=4, *vs3*=8, *rs1*=5, *vm*=1, and *lmul*=1, the instruction is `vsseg4b.v v8, (x5)`. This instruction performs the following operations:

$$\begin{aligned} \text{mem}[\text{x5}] &= \text{v8}[0], \text{mem}[\text{x5}+4] = \text{v8}[1] \dots \text{mem}[\text{x5}+4*i] = \text{v8}[i] \\ \text{mem}[\text{x5}+1] &= \text{v9}[0], \text{mem}[\text{x5}+5] = \text{v9}[1] \dots \text{mem}[\text{x5}+1+4*i] = \text{v9}[i] \\ \text{mem}[\text{x5}+2] &= \text{v10}[0], \text{mem}[\text{x5}+6] = \text{v10}[1] \dots \text{mem}[\text{x5}+2+4*i] = \text{v10}[i] \\ \text{mem}[\text{x5}+3] &= \text{v11}[0], \text{mem}[\text{x5}+7] = \text{v11}[1] \dots \text{mem}[\text{x5}+3+4*i] = \text{v11}[i] \end{aligned}$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsseg<nf>b.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsseg<nf>b.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000		vm	00000			rs1		000		vs3		0100111	

13.7.323 VSSEG<NF>E.V: a vector SEGMENT element store instruction**Syntax:**

`vsseg<nf>e.v vs3, (rs1), vm`

Operation:

`offset=sew/8`

for(`k=0; k<=nf-1; k++`) {

`mem[rs1 + k*offset + i *nf*offset] = vs3d+k[i]`

}

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vs3=8`, `rs1=5`, `vm=1`, `SEW=32`, and `lmul=1`, the value of `offset` is 4 (`32/8`) and the instruction is `vsseg4e.v v8, (x5)`. This instruction performs the following operations:

`mem[x5]=v8[0], mem[x5+16]=v8[1] ... mem[x5+16*i]=v8[i]`

`mem[x5+4]=v9[0], mem[x5+20]=v9[1] ... mem[x5+4+16*i]=v9[i]`

`mem[x5+8]=v10[0], mem[x5+24]=v10[1] ... mem[x5+8+16*i]=v10[i]`

`mem[x5+12]=v11[0], mem[x5+28]=v11[1] ... mem[x5+12+16*i]=v11[i]`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsseg<nf>.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsseg<nf>.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	000		vm	00000			rs1		111		vs3		0100111	

13.7.324 VSSEG<NF>H.V: a vector SEGMENT halfword store instruction**Syntax:**

`vsseg<nf>h.v vs3, (rs1), vm`

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + k*2 + 2*i *nf] = vs3d+k[i]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vs3=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vsseg4h.v v8, (x5)`. This instruction performs the following operations:

$$\begin{aligned} \text{mem}[x5] &= v8[0], \text{mem}[x5+8] = v8[1] \dots \text{mem}[x5+8*i] = v8[i] \\ \text{mem}[x5+2] &= v9[0], \text{mem}[x5+10] = v9[1] \dots \text{mem}[x5+2+8*i] = v9[i] \\ \text{mem}[x5+4] &= v10[0], \text{mem}[x5+12] = v10[1] \dots \text{mem}[x5+4+8*i] = v10[i] \\ \text{mem}[x5+6] &= v11[0], \text{mem}[x5+14] = v11[1] \dots \text{mem}[x5+6+8*i] = v11[i] \end{aligned}$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsseg<nf>h.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsseg<nf>h.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0		
nf-1			000		vm	00000			rs1		101		vs3		0100111	

13.7.325 VSSEG<NF>W.V: a vector SEGMENT word store instruction

Syntax:

`vsseg<nf>w.v vs3, (rs1), vm`

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + k*4 +4*i *nf] = vs3d+k[i]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vs3=8`, `rs1=5`, `vm=1`, and `lmul=1`, the instruction is `vsseg4w.v v8, (x5)`. This instruction performs the following operations:

$$\begin{aligned} \text{mem}[x5] &= v8[0], \text{mem}[x5+16] = v8[1] \dots \text{mem}[x5+8*i] = v8[i] \\ \text{mem}[x5+4] &= v9[0], \text{mem}[x5+20] = v9[1] \dots \text{mem}[x5+4+8*i] = v9[i] \\ \text{mem}[x5+8] &= v10[0], \text{mem}[x5+24] = v10[1] \dots \text{mem}[x5+8+8*i] = v10[i] \\ \text{mem}[x5+12] &= v11[0], \text{mem}[x5+28] = v11[1] \dots \text{mem}[x5+12+8*i] = v11[i] \end{aligned}$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsseg<nf>w.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssseg<nf>w.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		000		vm	00000		rs1		110		vs3		0100111	

13.7.326 VSSSEG<NF>B.V: a vector strided SEGMENT byte store instruction

Syntax:

`vssseg<nf>b.v vs3, (rs1),rs2, vm`

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + k + i *nf] = vs3d+k[i]
}
```

`nf` specifies the number of fields in each segment and the number of destination registers. `nf` ranges from 2 to 8. For example, when `nf=4`, `vs3=8`, `rs1=5`, `rs2=6`, `vm=1`, and `lmul=1`, the instruction is `vssseg4b.v v8, (x5), x6`. This instruction performs the following operations:

```
mem[x5]=v8[0], mem[x5+x6+1]=v8[1] ... mem[x5+i*x6+1]=v8[i]
mem[x5+1]=v9[0], mem[x5+x6+2]=v9[1] ... mem[x5+i*x6+2]=v9[i]
mem[x5+2]=v10[0], mem[x5+x6+3]=v10[1] ... mem[x5+i*x6+3]=v10[i]
mem[x5+3]=v11[0], mem[x5+x6+4]=v11[1] ... mem[x5+i*x6+4]=v11[i]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssseg<nf>b.v vs3, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssseg<nf>b.v vs3, (rs1), rs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		010		vm	rs2		rs1		000		vs3		0100111	

13.7.327 VSSSEG<NF>E.V: a vector strided SEGMENT element store instruction**Syntax:**

```
vssseg<nf>e.v vs3, (rs1),rs2, vm
```

Operation:

```
offset=sew/32
```

```
for( k=0; k<=nf-1; k++) {
```

```
    mem[rs1 + k*offset +i *nf*offset] = vs3d+k[i]
```

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, rs2=6, vm=1, lmul=1, and SEW=32, the value of offset is 4 (32/8) and the instruction is vssseg4e.v v8, (x5), x6. This instruction performs the following operations:

```
mem[x5]=v8[0], mem[x5+x6]=v8[1] ... mem[x5+i*x6]=v8[i]
```

```
mem[x5+4]=v9[0], mem[x5+x6+4]=v9[1] ... mem[x5+i*x6+4]=v9[i]
```

```
mem[x5+8]=v10[0], mem[x5+x6+8]=v10[1] ... mem[x5+i*x6+8]=v10[i]
```

```
mem[x5+12]=v11[0], mem[x5+x6+12]=v11[1] ... mem[x5+i*x6+12]=v11[i]
```

Permission:

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

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

```
None.
```

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vssseg<nf>e.v vs3, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vssseg<nf>e.v vs3, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		010		vm	rs2		rs1		111		vs3		0100111	

13.7.328 VSSSEG<NF>H.V: a vector strided SEGMENT halfword store instruction**Syntax:**

vssseg<nf>h.v vs3, (rs1),rs2 vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 +2* k +2*i *nf] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vssseg4h.v v8, (x5), x6. This instruction performs the following operations:

$$\begin{aligned} \text{mem}[x5] &= v8[0], \text{mem}[x5+x6] = v8[1] \dots \text{mem}[x5+i*x6] = v8[i] \\ \text{mem}[x5+2] &= v9[0], \text{mem}[x5+x6+2] = v9[1] \dots \text{mem}[x5+i*x6+2] = v9[i] \\ \text{mem}[x5+4] &= v10[0], \text{mem}[x5+x6+4] = v10[1] \dots \text{mem}[x5+i*x6+4] = v10[i] \\ \text{mem}[x5+6] &= v11[0], \text{mem}[x5+x6+6] = v11[1] \dots \text{mem}[x5+i*x6+6] = v11[i] \end{aligned}$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vssseg<nf>h.v vs3, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vssseg<nf>h.v vs3, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	010	vm	rs2			rs1		101				vs3		0100111

13.7.329 VSSSEG<NF>W.V: a vector strided SEGMENT word store instruction

Syntax:

vssseg<nf>w.v vs3, (rs1),rs2,vm

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 +4* k +4*i *nf] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, rs2=6, vm=1, and lmul=1, the instruction is vssseg4w.v v8, (x5), x6. This instruction performs the following operations:

$$\begin{aligned} \text{mem}[x5] &= v8[0], \text{mem}[x5+x6] = v8[1] \dots \text{mem}[x5+i*x6] = v8[i] \\ \text{mem}[x5+4] &= v9[0], \text{mem}[x5+x6+4] = v9[1] \dots \text{mem}[x5+i*x6+4] = v9[i] \\ \text{mem}[x5+8] &= v10[0], \text{mem}[x5+x6+8] = v10[1] \dots \text{mem}[x5+i*x6+8] = v10[i] \\ \text{mem}[x5+12] &= v11[0], \text{mem}[x5+x6+12] = v11[1] \dots \text{mem}[x5+i*x6+12] = v11[i] \end{aligned}$$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. This corresponding assembler instruction is vssseg<nf>w.v vs3, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vssseg<nf>w.v vs3, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	010	vm	rs2			rs1		110				vs3		0100111

13.7.330 VSXSEG<NF>B.V: a vector indexed SEGMENT byte store instruction

Syntax:

```
vsxseg<nf>b.v vs3, (rs1),vs2, vm
```

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + k +vs2[i]] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vsxseg4b.v v8, (x5), v16. This instruction performs the following operations:

```
mem[x5+v16[0]]=v8[0] ... mem[x5+v16[i]]=v8[i]
mem[x5+v16[0]+1]=v9[0] ... mem[x5+v16[i]+1]=v9[i]
mem[x5+v16[0]+2]=v10[0] ... mem[x5+v16[i]+2]=v10[i]
mem[x5+v16[0]+3]=v11[0] ... mem[x5+v16[i]+3]=v11[i]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsxseg<nf>b.v vs3, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsxseg<nf>b.v vs3, (rs1), vs2, v0.t.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
nf-1	011	vm	vs2	rs1	000	vs3	0100111

13.7.331 VSXSEG<NF>E.V: a vector indexed SEGMENT element store instruction

Syntax:

```
vsxseg<nf>e.v vs3, (rs1),vs2, vm
```

Operation:

```
offset = sew/8
```

```
for( k=0; k<=nf-1; k++) {
```

```
    mem[rs1 + k*offset +vs2[i]] = vs3d+k[i]
```

```
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, vs2=16, vm=1, lmul=1, and SEW=32, the value of offset is 4 (32/8) and the instruction is vsxseg4e.v v8, (x5), v16. This instruction performs the following operations:

```
mem[x5+v16[0]]=v8[0] ... mem[x5+v16[i]]=v8[i]
```

```
mem[x5+v16[0]+4]=v9[0] ... mem[x5+v16[i]+4]=v9[i]
```

```
mem[x5+v16[0]+8]=v10[0] ... mem[x5+v16[i]+8]=v10[i]
```

```
mem[x5+v16[0]+12]=v11[0] ... mem[x5+v16[i]+12]=v11[i]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsxseg<nf>e.v vs3, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsxseg<nf>e.v vs3, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	0	1	vm	vs2		rs1		111		vs3		0100111		

13.7.332 VSXSEG<NF>H.V: a vector indexed SEGMENT halfword store instruction

Syntax:

```
vsxseg<nf>h.v vs3, (rs1),vs2, vm
```

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + 2*k +vs2[i]] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vsxseg4h.v v8, (x5), v16. This instruction performs the following operations:

```
mem[x5+v16[0]]=v8[0] ... mem[x5+v16[i]]=v8[i]
mem[x5+v16[0]+2]=v9[0] ... mem[x5+v16[i]+2]=v9[i]
mem[x5+v16[0]+4]=v10[0] ... mem[x5+v16[i]+4]=v10[i]
mem[x5+v16[0]+6]=v11[0] ... mem[x5+v16[i]+6]=v11[i]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsxseg<nf>h.v vs3, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsxseg<nf>h.v vs3, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1		011		vm	vs2		rs1		101		vs3		0100111	

13.7.333 VSXSEG<NF>W.V: a vector indexed SEGMENT word store instruction

Syntax:

```
vsxseg<nf>w.v vs3, (rs1),vs2, vm
```

Operation:

```
for( k=0; k<=nf-1; k++) {
    mem[rs1 + 4*k +vs2[i]] = vs3d+k[i]
}
```

nf specifies the number of fields in each segment and the number of destination registers. nf ranges from 2 to 8. For example, when nf=4, vs3=8, rs1=5, vs2=16, vm=1, and lmul=1, the instruction is vsxseg4w.v v8, (x5), v16. This instruction performs the following operations:

```
mem[x5+v16[0]]=v8[0] ... mem[x5+v16[i]]=v8[i]
mem[x5+v16[0]+4]=v9[0] ... mem[x5+v16[i]+4]=v9[i]
mem[x5+v16[0]+8]=v10[0] ... mem[x5+v16[i]+8]=v10[i]
mem[x5+v16[0]+12]=v11[0] ... mem[x5+v16[i]+12]=v11[i]
```

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vsxseg<nf>w.v vs3, (rs1), vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vsxseg<nf>w.v vs3, (rs1), vs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
nf-1	011	vm	vs2	rs1	110	vs3	0100111							

13.7.334 VSSH.V: a vector strided halfword store instruction

Syntax:

vssh.v vs3, (rs1), rs2, vm

Operation:

$\text{mem}[\text{rs1} + i * \text{rs2}] = \text{vs3}[i][15:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes: If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vssh.v vs3, (rs1), rs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vssh.v vs3, (rs1), rs2, v0.t.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	010	vm	rs2	rs1	101	vs3	0100111							

13.7.335 VSSRA.VV: a vector scaling arithmetic right shift instruction

Syntax:

vssra.vv vd, vs2, vs1, vm

Operation:

$\text{vd}[i] \leftarrow (\text{signed}(\text{vs2}[i] + \text{round}) \gg \gg \text{vs1}[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vssra.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vssra.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011		vm	vs2		vs1		000		vd		1010111	

13.7.336 VSSRA.VX: a vector-scalar scaling arithmetic right shift instruction**Syntax:**

`vssra.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow (\text{signed}(vs2[i] + \text{round}) \gg \gg rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssra.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssra.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011	vm			vs2			vs1	100			vd	1010111

13.7.337 VSSRA.VI: a vector-immediate scaling arithmetic right shift instruction

Syntax:

`vssra.vi vd, vs2, imm, vm`

Operation:

$vd[i] \leftarrow (\text{signed}(vs2[i] + \text{round}) \gg \text{unsigned}(imm))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (`vrxm`):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssra.vi vd, vs2, imm`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssra.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101011	vm			vs2			vs1	011			vd	1010111

13.7.338 VSSRL.VV: a vector scaling logical right shift instruction

Syntax:

```
vssrl.vv vd, vs2, vs1, vm
```

Operation:

$$vd[i] \leftarrow (\text{signed}(vs2[i] + \text{round}) \gg vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vssrl.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vssrl.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101010		vm	vs2		vs1		000		vd		1010111	

13.7.339 VSSRL.VX: a vector-scalar scaling logical right shift instruction

Syntax:

```
vssrl.vx vd, vs2, rs1, vm
```

Operation:

$$vd[i] \leftarrow (\text{signed}(vs2[i] + \text{round}) \gg rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vssrl.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vssrl.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101010			vm	vs2		vs1		100		vd		1010111

13.7.340 VSSRL.VI: a vector-immediate scaling logical right shift instruction

Syntax:

`vssrl.vi vd, vs2, imm, vm`

Operation:

$vd[i] \leftarrow (\text{signed}(vs2[i] + \text{round}) \gg \text{unsigned}(imm))$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vssrl.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vssrl.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
101010		vm	vs2		vs1		011		vd		1010111	

13.7.341 VSSUB.VV: a vector saturating subtract instruction for signed integers

Syntax:

`vssub.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] - vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vssub.vv vd, vs2, vs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vssub.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011		vm	vs2		vs1		000		vd		1010111	

13.7.342 VSSUB.VX: a vector-scalar saturating subtract instruction for signed integers

Syntax:

vssub.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] - rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vssub.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vssub.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100011		vm	vs2		rs1		100		vd		1010111	

13.7.343 VSSUBU.VV: a vector saturating unsigned-integer subtract instruction

Syntax:

vssubu.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] - vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssubu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssubu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		vs1		000		vd		1010111

13.7.344 VSSUBU.VX: a vector-scalar saturating unsigned-integer subtract instruction**Syntax:**

`vssub.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{sat}(vs2[i] - rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssubu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssubu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
100010			vm	vs2		rs1		100		vd		1010111

13.7.345 VSSW.V: a vector strided word store instruction**Syntax:**

`vssw.v vs3, (rs1), rs2, vm`

Operation:

$$\text{mem}[\text{rs1} + \text{i} * \text{rs2}] = \text{vs3}[\text{i}][31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vssw.v vs3, (rs1), rs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vssw.v vs3, (rs1), rs2, v0.t`.

Instruction format:

31	29 28	26 25 24	20 19	15 14	12 11	7 6	0
000	010	vm	rs2	rs1	110	vs3	0100111

13.7.346 VSUB.VV: a vector integer subtract instruction**Syntax:**
`vsub.vv vd, vs2, vs1, vm`
Operation:

$$\text{vd}[\text{i}] \leftarrow \text{vs2}[\text{i}] - \text{vs1}[\text{i}]$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsub.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsub.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000010		vm	vs2		vs1		000		vd		1010111	

13.7.347 VSUB.VX: a vector-scalar integer subtract instruction

Syntax:

`vsub.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] - rs1$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsub.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsub.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
000010		vm	vs2		rs1		100		vd		1010111	

13.7.348 VSUXB.V: a vector unordered-indexed byte store instruction

Syntax:

`vsuxb.v vs3, (rs1), vs2, vm`

Operation:

$$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[\text{i}])] = \text{vs3}[\text{i}][7:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in an unordered way.

- If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsuxb.v vs3, (rs1), vs2`.
- If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsuxb.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2				rs1				000	vs3	0100111	

13.7.349 VSUXE.V: a vector unordered-indexed element store instruction**Syntax:**
`vsuxe.v vs3, (rs1), vs2, vm`
Operation:

$$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[\text{i}])] = \text{vs3}[\text{i}][\text{SEW}-1:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in an unordered way.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsuxe.v vs3, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsuxe.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2			rs1		111		vs3				0100111

13.7.350 VSUXH.V: a vector unordered-indexed halfword store instruction

Syntax:

`vsuxh.v vs3, (rs1), vs2, vm`

Operation:

$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[\text{i}])] = \text{vs3}[\text{i}][15:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in an unordered way.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsuxh.v vs3, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsuxh.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2			rs1		101		vs3				0100111

13.7.351 VSUXW.V: a vector unordered-indexed word store instruction

Syntax:

$$\text{vsuxw.v vs3, (rs1), vs2, vm}$$
Operation:

$$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[\text{i}])] = \text{vs3}[\text{i}][31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in an unordered way.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsuxw.v vs3, (rs1), vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsuxw.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	111	vm	vs2	rs1	110	vs3	0100111							

13.7.352 VSW.V: a vector word store instruction

Syntax:

$$\text{vsw.v vs3, (rs1), vm}$$
Operation:

$$\text{mem}[\text{rs1} + 4 * \text{i}] = \text{vs3}[\text{i}][31:0]$$
Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsw.v vs3, (rs1)`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsw.v vs3, (rs1), v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	000	vm	00000			rs1		110		vd		0100111		

13.7.353 VSXB.V: a vector ordered-indexed byte store instruction

Syntax:

`vsxb.v vs3, (rs1), vs2, vm`

Operation:

$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[\text{i}])] = \text{vs3}[\text{i}][7:0]$

The instruction writes the lowest bytes of each element in the source register to a memory location in order, with `rs1` as the base address and `vs2[i]` as the offset.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in order.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsxb.v vs3, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsxb.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2	rs1	000	vs3	0100111							

13.7.354 VSXE.V: a vector ordered-indexed element store instruction

Syntax:

`vsxe.v vs3, (rs1), vs2, vm`

Operation:

$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])] = \text{vs3}[i][\text{SEW}-1:0]$

Permission:

M mode/S mode/U mode

Exception: Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in order.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsxe.v vs3, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsxe.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2	rs1	111	vs3	0100111							

13.7.355 VSXH.V: a vector ordered-indexed halfword store instruction

Syntax:

`vsxh.v vs3, (rs1), vs2, vm`

Operation:

$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])] = \text{vs3}[i][15:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in order.

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vsxh.v vs3, (rs1), vs2`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vsxh.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000	011	vm	vs2			rs1			101	vs3			0100111	

13.7.356 VSXW.V: a vector ordered-indexed word store instruction**Syntax:**

`vsxw.v vs3, (rs1), vs2, vm`

Operation:

$\text{mem}[\text{rs1} + \text{sign_extend}(\text{vs2}[i])] = \text{vs3}[i][31:0]$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction exceptions, unaligned access exceptions on store instructions, access error exceptions on store instructions, and page error exceptions on store instructions

Affected flag bits:

None.

Notes:

This instruction writes data to the memory in order.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsxw.v vs3, (rs1), vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsxw.v vs3, (rs1), vs2, v0.t`.

Instruction format:

31	29	28	26	25	24	20	19	15	14	12	11	7	6	0
000			011		vm	vs2		rs1		110		vs3		0100111

13.7.357 VWADD.VV: a vector widening signed-integer add instruction

Syntax:

`vwadd.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{sign_extend}(vs2[i]) + \text{sign_extend}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2 * SEW$ for `vd` and SEW for `vs2` or `vs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwadd.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwadd.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110001			vm	vs2	vs1	010		vd	1010111			

13.7.358 VWADD.VX: a vector-scalar widening signed-integer add instruction

Syntax:

`vwadd.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow \text{sign_extend}(vs2[i]) + \text{sign_extend}(rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:The element width is $2*SEW$ for vd and SEW for $vs2$ or $rs1$.If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwadd.vx vd, vs2, rs1`.If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwadd.vx vd, vs2, rs1, v0.t`.**Instruction format:**

31	26 25 24	20 19	15 14	12 11	7 6	0
110001	vm	vs2	rs1	110	vd	1010111

13.7.359 VWADDU.VV: a vector widening unsigned-integer add instruction**Syntax:**
`vwaddu.vv vd, vs2, vs1, vm`
Operation:

$$vd[i] \leftarrow \text{zero_extend}(vs2[i]) - \text{zero_extend}(rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for `vd` and SEW for `vs2` or `vs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `waddu.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwaddu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110000		vm	vs2	vs1		110		vd		1010111		

13.7.360 VWADDU.VX: a vector-scalar widening unsigned-integer add instruction

Syntax:

`vwaddu.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{zero_extend}(vs2[i]) - \text{zero_extend}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for `vd` and SEW for `vs2` or `rs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwaddu.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwaddu.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110000		vm	vs2	rs1		110		vd		1010111		

13.7.361 VWADD.WV: a widening vector widening signed-integer add instruction

Syntax:

```
vwadd.wv vd, vs2, vs1, vm
```

Operation:

$$vd[i] \leftarrow vs2[i] + \text{sign_extend}(vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

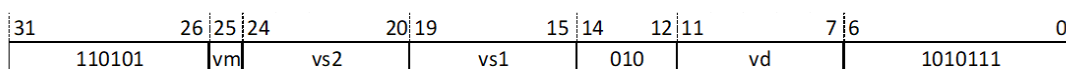
Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwadd.wv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwadd.wv vd, vs2, vs1, v0.t`.

Instruction format:

13.7.362 VWADD.WX: a widening vector-scalar widening signed-integer add instruction

Syntax:

```
vwadd.wx vd, vs2, rs1, vm
```

Operation:

$$vd[i] \leftarrow vs2[i] + \text{sign_extend}(rs1)$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd or $vs2$ and SEW for $rs1$.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwadd.wx vd, vs2, rs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwadd.wx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110101		vm	vs2		rs1		110		vd		1010111	

13.7.363 VWADDU.WV: a widening vector widening unsigned-integer add instruction

Syntax:

`vwaddu.wv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] + \text{zero_extend}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd or $vs2$ and SEW for $vs1$.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwaddu.wv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwaddu.wv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110100		vm	vs2		vs1		010		vd		1010111	

13.7.364 VWADDU.WX: a widening vector-scalar widening unsigned-integer add instruction

Syntax:

`vwaddu.wx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] + \text{zero_extend}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for `vd` or `vs2` and SEW for `rs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwaddu.wx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwaddu.wx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
110100			vm	vs2			rs1			110		vd	1010111	

13.7.365 VWMACC.VV: a vector widening signed-integer multiply-add instruction that overwrites addends

Syntax:

`vwmacc.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{signed}(vs1[i]) \times \text{signed}(vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

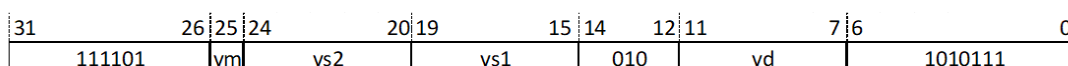
Notes:

The element width is 2*SEW for vd and SEW for vs2 or vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmacc.vv vd, vs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwmacc.vv vd, vs1, vs2, v0.t`.

Instruction format:



13.7.366 VWMACC.VX: a vector-scalar widening signed-integer multiply-add instruction that overwrites addends

Syntax:

`vwmacc.vx vd, rs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{signed}(rs1) \times \text{signed}(vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

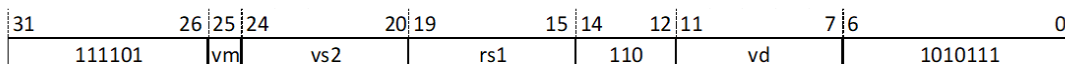
Notes:

The element width is 2*SEW for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmacc.vx vd, rs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwmacc.vx vd, rs1, vs2, v0.t`.

Instruction format:



13.7.367 VWMACCSU.VV: a vector widening signed-unsigned-integer multiply-add instruction that overwrites addends

Syntax:

vwmaccsu.vv vd, vs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{signed}(vs1[i]) \times \text{unsigned}(vs2[i]) + vd[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

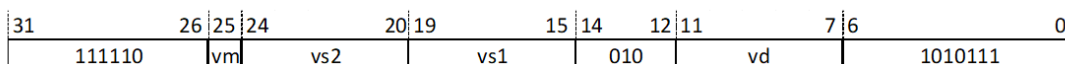
Notes:

The element width is 2*SEW for vd and SEW for vs2 or vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmaccsu.vv vd, vs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmaccsu.vv vd, vs1, vs2, v0.t.

Instruction format:



13.7.368 VWMACCSU.VX: a vector-scalar widening signed-unsigned-integer multiply-add instruction that overwrites addends

Syntax:

vwmaccsu.vx vd, rs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{signed}(rs1) \times \text{unsigned}(vs2[i]) + vd[i]$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:The element width is $2*SEW$ for vd and SEW for $vs2$ or $rs1$.If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmaccsu.vx vd, rs1, vs2`.If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwmaccsu.vx vd, rs1, vs2, v0.t`.**Instruction format:**

31	26	25	24	20	19	15	14	12	11	7	6	0
111110		vm	vs2		rs1		110		vd		1010111	

13.7.369 VWMACCU.VV: a vector widening unsigned-integer multiply-add instruction that overwrites addends

Syntax:`vwmaccu.vv vd, vs1, vs2, vm`**Operation:** $vd[i] \leftarrow \text{unsigned}(vs1[i]) \times \text{unsigned}(vs2[i]) + vd[i]$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:The element width is $2*SEW$ for vd and SEW for $vs2$ or $vs1$.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmaccu.vv vd, vs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwmaccu.vv vd, vs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111100		vm	vs2		vs1		010		vd		1010111	

13.7.370 VWMACCU.VX: a vector-scalar widening unsigned-integer multiply-add instruction that overwrites addends

Syntax:

`vwmaccu.vx vd, rs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(rs1) \times \text{unsigned}(vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2 \times \text{SEW}$ for `vd` and `SEW` for `vs2` or `rs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmaccu.vx vd, rs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwmaccu.vx vd, rs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111100		vm	vs2		rs1		110		vd		1010111	

13.7.371 VWMACCUS.VX: a vector-scalar widening unsigned-signed-integer multiply-add instruction that overwrites addends

Syntax:

vwmaccus.vx vd, rs1, vs2, vm

Operation:

$vd[i] \leftarrow \text{unsigned}(rs1) \times \text{signed}(vs2[i]) + vd[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmaccus.vx vd, rs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmaccus.vx vd, rs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
111111			vm	vs2			rs1		110		vd		1010111	

13.7.372 VWMUL.VV: a vector widening signed-integer multiply instruction

Syntax:

vwmul.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) \times \text{signed}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd and SEW for vs2 or vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmul.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmul.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111011			vm	vs2		vs1		010		vd		1010111

13.7.373 VWMUL.VX: a vector-scalar widening signed-integer multiply instruction**Syntax:**

vwmul.vx vd, vs2, rs1, vm

Operation: $vd[i] \leftarrow \text{signed}(vs2[i]) \times \text{signed}(rs1)$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmul.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmul.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111011			vm	vs2		rs1		110		vd		1010111

13.7.374 VWMULSU.VV: a vector widening signed-unsigned integer multiply instruction

Syntax:

vwmulsu.vv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) \times \text{unsigned}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd and SEW for vs2 or vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmulsu.vv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmulsu.vv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
111010			vm	vs2			vs1			010		vd	1010111	

13.7.375 VWMULSU.VX: a vector-scalar widening signed-unsigned integer multiply instruction

Syntax:

vwmulsu.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{signed}(vs2[i]) \times \text{unsigned}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

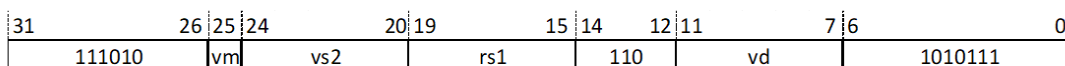
Notes:

The element width is 2*SEW for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmulsu.vx vd, vs2, rs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwmulsu.vx vd, vs2, rs1, v0.t`.

Instruction format:



13.7.376 VWMULU.VV: a vector widening unsigned integer multiply instruction

Syntax:

`vwmulu.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \times \text{unsigned}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd and SEW for vs2 or vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwmulu.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwmulu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111000			vm	vs2		vs1		010		vd		1010111

13.7.377 VWMULU.VX: a vector-scalar widening unsigned integer multiply instruction

Syntax:

vwmulu.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{unsigned}(vs2[i]) \times \text{unsigned}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwmulu.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwmulu.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111000			vm	vs2		rs1		110		vd		1010111

13.7.378 VWREDSUM.VS: a vector widening reduction instruction that sign-extends vector elements before summing them

Syntax:

vwredsum.vs vd, vs2, vs1, vm

Operation:

tmp = vs1[0]

for(i=0; i<vl; i++) {

```

    tmp = tmp + sign_extend(vs2[i])
}
vd[0]= tmp
vd[VLEN/SEW-1:1] = 0

```

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd or vs1 and SEW for vs2. This instruction sign-extends all elements in vs2 to 2*SEW before summing them.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwredsum.vs vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwredsum.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110001	vm		vs2		vs1	000		vd		1010111		

13.7.379 VWREDSUMU.VS: a vector widening reduction instruction that unsign-extends vector elements before summing them

Syntax:

```
vwredsumu.vs vd, vs2, vs1, vm
```

Operation:

```

tmp = vs1[0]
for( i=0; i<vl; i++) {
    tmp = tmp + unsign_extend(vs2[i])
}

```


$$vd[0] = tmp$$

$$vd[VLEN/SEW-1:1] = 0$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd or $vs1$ and SEW for $vs2$. This instruction unsign-extends all elements in $vs2$ to $2*SEW$ before summing them.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwredsumu.vs vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwredsumu.vs vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110000			vm	vs2		vs1		000		vd	1010111	

13.7.380 VWSMACC.VV: a vector widening signed-integer saturating scaled multiply-add instruction

Syntax:
`vwsmacc.vv vd, vs1, vs2, vm`
Operation:

$$vd[i] \leftarrow \text{clip}(((vs1[i] \times vs2[i] + \text{round}) \gg SEW/2) + vd[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

The element width is $2*SEW$ for vd and SEW for $vs2$ or $vs1$.

Dynamically rounds off based on the fixed-point rounding mode register ($vxrm$):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsnacc.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwsnacc.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111101			vm	vs2		vs1		000		vd		1010111

13.7.381 VWSNACC.VX: a vector-scalar widening signed-integer saturating scaled multiply-add instruction

Syntax:

`vwsnacc.vx vd, rs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{clip}(((rs1 \times vs2[i] + \text{round}) \gg SEW/2) + vd[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

The element width is $2*SEW$ for vd and SEW for $vs2$ or $rs1$.

Dynamically rounds off based on the fixed-point rounding mode register ($vxrm$):

- 2' b00: Rounds off to the nearest large value.

- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vsmacc.vx vd, rs1, vs2`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vsmacc.vx vd, rs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111101	vm		vs2	rs1		100		vd		1010111		

13.7.382 VWSMACCSU.VV: a vector widening signed-unsigned-integer saturating scaled negate-(multiply-sub) instruction

Syntax:

`vwsmaccsu.vv vd, vs1, vs2, vm`

Operation:

$vd[i] \leftarrow \text{clip}(-((\text{signed}(vs1[i]) \times \text{unsigned}(vs2[i]) + \text{round}) \gg \text{SEW}/2) + vd[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

The element width is $2 * \text{SEW}$ for `vd` and `SEW` for `vs2` or `vs1`.

Dynamically rounds off based on the fixed-point rounding mode register (`vrxm`):

- 2' b00: Rounds off to the nearest large value.
- 2' b01: Rounds off to the nearest even number.
- 2' b10: Rounds off to zero.
- 2' b11: Rounds off to an odd number.

31	26	25	24	20	19	15	14	12	11	7	6	0
111110		vm	vs2		rs1		100		vd		1010111	

13.7.384 VWSMACCU.VV: a vector widening unsigned-integer saturating scaled multiply-add instruction

Syntax:

vwsmaccu.vv vd, vs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{clipu}(((vs1[i] \times vs2[i] + \text{round}) \gg \text{SEW}/2) + vd[i])$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

The element width is $2 * \text{SEW}$ for vd and SEW for vs2 or vs1.

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2^7 b00: Rounds off to the nearest large value.
- 2^7 b01: Rounds off to the nearest even number.
- 2^7 b10: Rounds off to zero.
- 2^7 b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwsmaccu.vv vd, vs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwsmaccu.vv vd, vs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111100		vm	vs2		vs1		000		vd		1010111	

13.7.385 VWSMACCU.VX: a vector-scalar widening unsigned-integer saturating scaled multiply-add instruction

Syntax:

vwsmaccu.vx vd, rs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{clipu}(((rs1 \times vs2[i] + \text{round}) \gg SEW/2) + vd[i])$$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:

The element width is $2*SEW$ for vd and SEW for vs2 or rs1.

Dynamically rounds off based on the fixed-point rounding mode register (vxrm):

- 2^7 b00: Rounds off to the nearest large value.
- 2^7 b01: Rounds off to the nearest even number.
- 2^7 b10: Rounds off to zero.
- 2^7 b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwsmaccu.vv vd, rs1, vs2.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwsmaccu.vv vd, rs1, vs2, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
111100			vm	vs2			rs1		100		vd		1010111	

13.7.386 VWSMACCUS.VX: a vector-scalar widening unsigned-signed-integer saturating scaled multiply-sub instruction

Syntax:

vswmaccus.vx vd, rs1, vs2, vm

Operation:

$$vd[i] \leftarrow \text{clip}(-((\text{unsigned}(rs1) \times \text{signed}(vs2[i]) + \text{round}) \gg SEW/2) + vd[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

VXSAT bit

Notes:The element width is $2 * SEW$ for vd and SEW for $vs2$ or $rs1$.Dynamically rounds off based on the fixed-point rounding mode register ($vfrm$):

- 2^7 b00: Rounds off to the nearest large value.
- 2^7 b01: Rounds off to the nearest even number.
- 2^7 b10: Rounds off to zero.
- 2^7 b11: Rounds off to an odd number.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vswmaccus.vx vd, rs1, vs2`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vswmaccus.vx vd, rs1, vs2, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
111111			vm	$vs2$		$rs1$		100		vd		1010111

13.7.387 VWSUB.VV: a vector widening signed-integer subtract instruction**Syntax:**
`vwsb.vv vd, vs2, vs1, vm`
Operation:

$$vd[i] \leftarrow \text{sign_extend}(vs2[i]) - \text{sign_extend}(vs1[i])$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

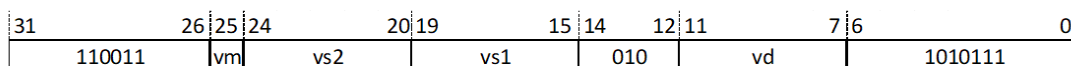
Notes:

The element width is $2 \times \text{SEW}$ for `vd` and `SEW` for `vs2` or `vs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsb.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwsb.vv vd, vs2, vs1, v0.t`.

Instruction format:



13.7.388 VWSUB.VX: a vector-scalar widening signed-integer subtract instruction

Syntax:

`vwsb.vx vd, vs2, rs1, vm`

Operation:

$vd[i] \leftarrow \text{sign_extend}(vs2[i]) - \text{sign_extend}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

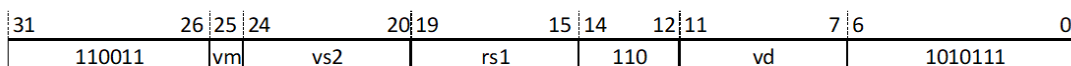
Notes:

The element width is $2 \times \text{SEW}$ for `vd` and `SEW` for `vs2` or `rs1`.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsb.vx vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwsb.vx vd, vs2, rs1, v0.t`.

Instruction format:



13.7.389 VWSUB.WV: a widening vector widening signed-integer subtract instruction

Syntax:

vwsb.wv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] - \text{sign_extend}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

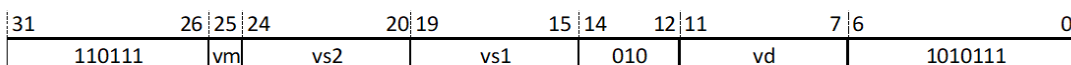
Notes:

The element width is $2*SEW$ for vd or vs2 and SEW for vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwsb.wv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwsb.wv vd, vs2, vs1, v0.t.

Instruction format:



13.7.390 VWSUB.WX: a widening vector-scalar widening signed-integer subtract instruction

Syntax:

vwsb.wx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow vs2[i] - \text{sign_extend}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd or $vs2$ and SEW for $rs1$.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsb.wx vd, vs2, rs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwsb.wx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0		
110111			vm	vs2			rs1		110		vd		1010111	

13.7.391 VWSUBU.VV: a vector widening unsigned-integer subtract instruction

Syntax:

`vwsbu.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow \text{unsign_extend}(vs2[i]) - \text{unsign_extend}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2*SEW$ for vd and SEW for $vs2$ or $vs1$.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsbu.vv vd, vs2, vs1`.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is `vwsbu.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110010			vm	vs2		vs1		010		vd		1010111

13.7.392 VWSUBU.VX: a vector-scalar widening unsigned-integer subtract instruction

Syntax:

vwsubu.vx vd, vs2, rs1, vm

Operation:

$vd[i] \leftarrow \text{unsign_extend}(vs2[i]) - \text{unsign_extend}(rs1)$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is $2 * SEW$ for vd and SEW for vs2 or rs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwsubu.vx vd, vs2, rs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwsubu.vx vd, vs2, rs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110010			vm	vs2		vs1		110		vd		1010111

13.7.393 VWSUBU.WV: a widening vector widening unsigned-integer subtract instruction

Syntax:

vwsubu.wv vd, vs2, vs1, vm

Operation:

$vd[i] \leftarrow vs2[i] - \text{unsign_extend}(vs1[i])$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd or vs2 and SEW for vs1.

If the value of vm is 1, the instruction will not be masked. The corresponding assembler instruction is vwsu.wv vd, vs2, vs1.

If the value of vm is 0, the instruction will be masked. The corresponding assembler instruction is vwsu.wv vd, vs2, vs1, v0.t.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110110		vm	vs2		vs1		010		vd		1010111	

13.7.394 VWSUBU.WX: a widening vector-scalar widening unsigned-integer subtract instruction

Syntax:

vwsu.wx vd, vs2, rs1, vm

Operation: $vd[i] \leftarrow vs2[i] - \text{zero_extend}(rs1)$ **Permission:**

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

The element width is 2*SEW for vd or vs2 and SEW for rs1.

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vwsu.w vd, vs2, rs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vwsu.w vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
110110		vm	vs2		rs1		110		vd		1010111	

13.7.395 VXOR.VV: a vector bitwise XOR instruction

Syntax:

`vxor.vv vd, vs2, vs1, vm`

Operation:

$vd[i] \leftarrow vs2[i] \wedge vs1[i]$

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of `vm` is 1, the instruction will not be masked. The corresponding assembler instruction is `vxor.vv vd, vs2, vs1`.

If the value of `vm` is 0, the instruction will be masked. The corresponding assembler instruction is `vxor.vv vd, vs2, vs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001011		vm	vs2		vs1		000		vd		1010111	

13.7.396 VXOR.VX: a vector-scalar bitwise XOR instruction

Syntax:

`vxor.vx vd, vs2, rs1, vm`

Operation:

$$vd[i] \leftarrow vs2[i] \wedge rs1$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vxor.vx vd, vs2, rs1`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vxor.vx vd, vs2, rs1, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001011		vm	vs2		rs1		100		vd		1010111	

13.7.397 VXOR.VI: a vector-immediate bitwise XOR instruction**Syntax:**
`vxor.vi vd, vs2, imm, vm`
Operation:

$$vd[i] \leftarrow vs2[i] \wedge \text{sign_extend}(\text{imm})$$
Permission:

M mode/S mode/U mode

Exception:

Invalid instruction.

Affected flag bits:

None.

Notes:

If the value of *vm* is 1, the instruction will not be masked. The corresponding assembler instruction is `vxor.vi vd, vs2, imm`.

If the value of *vm* is 0, the instruction will be masked. The corresponding assembler instruction is `vxor.vi vd, vs2, imm, v0.t`.

Instruction format:

31	26	25	24	20	19	15	14	12	11	7	6	0
001011	vm		vs2		imm		011		vd			1010111

13.8 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.
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, xs, 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.

Continued on next page

Table 13.1 – continued from previous page

Pseudo instruction	Base instruction	Meaning
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.
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.

Continued on next page

Table 13.1 – continued from previous page

Pseudo instruction	Base instruction	Meaning
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
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

In addition to GCV instruction sets defined in the standard, C906 implements custom instruction sets, including the cache instructions, arithmetic operation instructions, bit operation instructions, store instructions, and floating-point half-precision instructions.

Cache instructions, synchronization instructions, arithmetic operation instructions, bit operation instructions, and store instructions can be executed only when `mxstatus.theadisae` is set to 1. Otherwise, illegal instruction exceptions will occur. Floating-point half-precision instructions can be executed only when `mstatus.fs` is set to 2' b00. Otherwise, illegal instruction exceptions will occur.

The instructions are described in the following sections by instruction set.

14.1 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.

14.1.1 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.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		00001		00000		000		00000		0001011	

14.1.2 DCACHE.CVA: an instruction that clears dirty page table entries in the D-Cache that match the specified virtual address.

Syntax:

`dcache.cva rs1`

Operation:

Writes the page table entries in the D-Cache that matches the virtual address specified by `rs1` 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:

Invalid instruction or error page during instruction loading.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an illegal instruction exception.
- 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 invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00100		rs1		000		00000		0001011	

14.1.3 DCACHE.CPA: an instruction that clears dirty page table entries in the D-Cache that match the specified physical address.

Syntax:

```
dcache.cpa rs1
```

Operation:

Writes the page table entries in the D-Cache that match the physical address specified by 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:

Invalid instruction.

Notes:

If the value of mxstatus.theadisae is 0, executing this instruction causes an exception of invalid instruction.

If the value of mxstatus.theadisae is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		01000		rs1		000		00000		0001011	

14.1.4 DCACHE.CSW: 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.csw rs1
```

Operation:

Writes the dirty page table entries in the L1 D-Cache back into the next-level storage based on the way and set specified in rs1.

Permission:

M-mode/S-mode

Exception:

Invalid instruction.

Notes:

C906 D-Cache is a 4-way set-associative cache. $rs1[31:30]$ specifies the way, and $rs1[w:6]$ 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.

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an illegal instruction exception.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000001			00001		rs1		000		00000		0001011	

14.1.5 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 D-Cache.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an illegal instruction exception.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000000			00010		00000		000		00000		0001011	

14.1.6 DCACHE.IVA: an instruction that invalidates page table entries in the D-Cache that match the specified virtual address.

Syntax:

dcache.iva rs1

Operation:

Invalidates page table entries in the D-Cache that match the virtual address specified by rs1.

Permission:

M mode/S mode

Exception:

Invalid instruction or error page during instruction loading.

Notes:

If the value of mxstatus.theadisae is 0, executing this instruction causes an exception of invalid instruction.

If the value of mxstatus.theadisae is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00110		rs1		000		00000		0001011	

14.1.7 DCACHE.IPA: an instruction that invalidates page table entries in the D-Cache that match the specified physical addresses.

Syntax:

dcache.ipa rs1

Operation:

Invalidates the page table entries in the D-Cache that match the physical address specified by rs1.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

- If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an illegal instruction exception.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000001			01010		rs1		000		00000		0001011	

14.1.8 DCACHE.ISW: an instruction that invalidates page table entries in the D-Cache based on the specified way and set.

Syntax:

```
dcache.isw rs1
```

Operation:

Invalidates the page table entries in the D-Cache based on the specified set and way.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

C906 D-Cache is a 2-way set-associative cache. `rs1[31:30]` specifies the way, and `rs1[w: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 illegal instruction exception.
- If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0	
0000001			00010		rs1		000		00000		0001011	

14.1.9 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 L1 D-Cache back into the next-level storage and invalidates all these page table entries.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000			00011		00000		000		00000		0001011

□

14.1.10 DCACHE.CIVA: an instruction that clears dirty page table entries in the D-Cache that match the specified virtual address and invalidates the D-Cache.

Syntax:

`dcache.civa rs1`

Operation:

Writes the page table entry in the D-Cache that matches the virtual address specified by `rs1` back into the next-level storage and invalidates this page table entry.

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction or error page during instruction loading.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid 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 invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		00111		rs1		000		00000		0001011	

14.1.11 DCACHE.CIPA: an instruction that clears dirty page table entries in the D-Cache that match the specified physical address and invalidates the D-Cache.

Syntax:

`dcache.cipa rs1`

Operation:

Writes the page table entry that matches the specified physical address from the D-Cache of the `rs1` register back into the next-level storage and invalidates this page table entry.

Permission:

M-mode/S-mode

Exception:

Invalid instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		01011		rs1		000		00000		0001011	

14.1.12 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 in the L1 D-Cache that matches the way and set specified in rs1 back into the next-level storage and invalidates this page table entry.

Permission:

M mode/S mode

Exception:

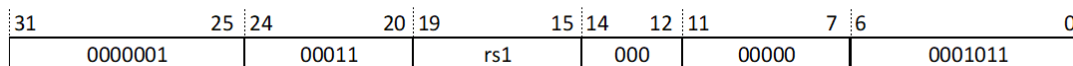
Invalid instruction.

Notes:

C906 D-Cache is a 4-way set-associative cache. rs1[31:30] specifies the way, and rs1[w: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.

- If the value of mxstatus.theadisaee is 0, executing this instruction causes an illegal instruction exception.
- If the value of mxstatus.theadisaee is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:



14.1.13 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.

Permission:

M mode/S mode

Exception:

Invalid instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		10000		00000		000		00000		0001011	

14.1.14 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:

Invalid instruction.

Notes:

If the value of `mxstatus.theadisaee` is 0, executing this instruction causes an exception of invalid instruction.

If the value of `mxstatus.theadisaee` is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000000		10001		00000		000		00000		0001011	

14.1.15 ICACHE.IVA: an instruction that invalidates page table entries in the I-Cache that match the specified virtual address.

Syntax:

`icache.iva rs1`

Operation:

Invalidates page table entries in the I-Cache that match the virtual address specified by rs1.

Permission:

M mode/S mode/U mode

Exception:

Invalid instruction or error page during instruction loading.

Notes:

If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of invalid 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 invalid instruction.

Instruction format:

31	25	24	20	19	15	14	12	11	7	6	0
0000001		10000		rs1		000		00000		0001011	

14.1.16 ICACHE.IPA: an instruction that invalidates page table entries in the I-Cache that match the specified physical address.

Syntax:

icache.ipa rs1

Operation:

Invalidates the page table entries in the I-Cache that match the physical address specified by rs1. You can perform this operation on all cores.

Permission:

M-mode/S-mode

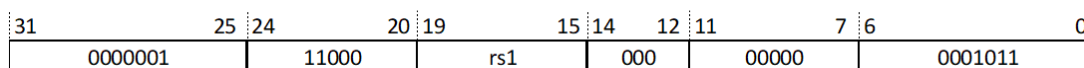
Exception:

Invalid instruction.

Notes:

If the value of mxstatus.theadisaee is 0, executing this instruction causes an exception of invalid instruction.

If the value of mxstatus.theadisaee is 1, executing this instruction in U mode causes an exception of invalid instruction.

Instruction format:

14.2 Appendix B-2 Multi-core synchronization instructions

The bit width of each synchronous instruction is 32 bits.

14.2.1 SYNC: an instruction that performs the synchronization operation

Syntax:

sync

Operation:

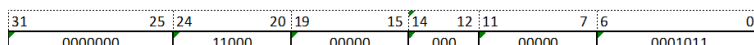
Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

14.2.2 SYNC.I: an instruction that synchronizes the clearing operation.

Syntax:

sync.i

Operation:

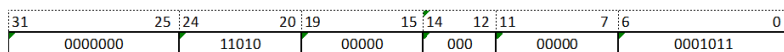
Ensures that all preceding instructions retire earlier than this instruction and all subsequent instructions retire later than this instruction, and clears the pipeline when this instruction retires.

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

14.3 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.

14.3.1 ADDSL: an add register instruction that shifts registers

Syntax:

addsl rd rs1, rs2, imm2

Operation:

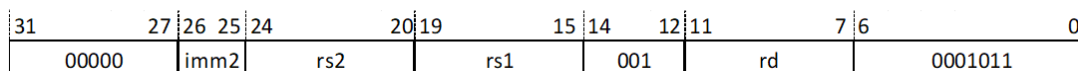
$rd \leftarrow rs1 + rs2 \ll imm2$

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

Instruction format:

14.3.2 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	

14.3.3 MULA: a multiply-add instruction that operates on the lower 16 bits

Syntax:

mulah rd, rs1, rs2

Operation:

$$tmp[31:0] \leftarrow rd[31:0] + (rs1[15:0] * rs[15:0])$$

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

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		00		rs2		rs1		001		rd		0001011	

14.3.4 MULAW: a multiply-add instruction that operates on the lower 32 bits

Syntax:

mulaw rd, rs1, rs2

Operation:

$$tmp[31:0] \leftarrow rd[31:0] + (rs1[31:0] * rs[31:0])[31:0]$$

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

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
00100		10		rs2	rs1		001		rd		0001011		

14.3.5 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		

14.3.6 MULSH: a multiply-subtract instruction that operates on the lower 16 bits

Syntax:

mul_{sh} rd, rs1, rs2

Operation:

$tmp[31:0] \leftarrow rd[31:0] - (rs1[15:0] * rs[15:0])$

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

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	

14.3.7 MULSW: a multiply-subtract instruction that operates on the lower 32 bits

Syntax:

mulaw rd, rs1, rs2

Operation:

$\text{tmp}[31:0] \leftarrow \text{rd}[31:0] - (\text{rs1}[31:0] * \text{rs}[31:0])$

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

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	

14.3.8 MVEQZ: an instruction that sends a message when the register is 0

Syntax:

mveqz rd, rs1, rs2

Operation: if (rs2 == 0)

rd \leftarrow rs1

else

rd \leftarrow 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	

14.3.9 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	

14.3.10 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	

14.3.11 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

14.4 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.

14.4.1 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

14.4.2 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

14.4.3 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	

14.4.4 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	

14.4.5 REV: an instruction that reverses the byte order in a word stored in the register**Syntax:**

rev rd, rs1

Operation:

rd[63:56] ← rs1[7:0]

rd[55:48] ← rs1[15:8]

rd[47:40] ← rs1[23:16]

rd[39:32] ← rs1[31:24]

rd[31:24] ← rs1[39:32]

rd[23:16] ← rs1[47:40]

rd[15:8] ← rs1[55:48]

rd[7:0] ← 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	

14.4.6 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	

14.4.7 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

14.4.8 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

14.5 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.

14.5.1 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	

14.5.2 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	

14.5.3 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	

14.5.4 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	

14.5.5 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	

14.5.6 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.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
01000		imm2		rs2		rs1		111		rd		0001011	

14.5.7 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.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
01110		imm2		rs2		rs1		111		rd		0001011	

14.5.8 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.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
01010		imm2		rs2		rs1		111		rd		0001011	

14.5.9 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	

14.5.10 LBIB: a load byte instruction that auto-increments the base address and extends signed bits

Syntax:

lbib rd, (rs1), imm5,imm2

Operation:

$$rs1 \leftarrow rs1 + \text{sign_extend}(imm5 \ll imm2)$$

$$rd \leftarrow \text{sign_extend}(\text{mem}[rs1+7:rs1])$$

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
00001		imm2		imm5		rs1		100		rd		0001011	

14.5.11 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	

14.5.12 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	

14.5.13 LDD: an instruction that loads double registers

Syntax:

ldd rd1,rd2, (rs1),imm2,4

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

14.5.14 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

14.5.15 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	

14.5.16 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	

14.5.17 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	

14.5.18 LHUIA: a base-address auto-increment instruction that extends zero bits and loads halfwords

Syntax:

lhuiA 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	

14.5.19 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	

14.5.20 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	

14.5.21 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	

14.5.22 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	

14.5.23 LRH: a load halfword instruction that shifts registers and extends signed bits

Syntax:

l_{rh} rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2 \ll imm2)+1: (rs1+rs2 \ll imm2)])$

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
00100		imm2		rs2		rs1		100		rd		0001011	

14.5.24 LRHU: a load halfword instruction that shifts registers and extends zero bits

Syntax:

l_{rhu} rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2 \ll imm2)+1: (rs1+rs2 \ll imm2)])$

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
10100		imm2		rs2		rs1		100		rd		0001011	

14.5.25 LRW: a load word instruction that shifts registers and extends signed bits

Syntax:

l_{rw} rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2 \ll imm2)+3: (rs1+rs2 \ll imm2)])$

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
01000		imm2		rs2		rs1		100		rd		0001011	

14.5.26 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	

14.5.27 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	

14.5.28 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	

14.5.29 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	

14.5.30 LURH: a load halfword instruction that shifts low 32 bits of registers and extends signed bits

Syntax:

```
lurh rd, rs1, rs2, imm2
```

Operation:

$$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)+1: \\ (rs1+rs2[31:0] \ll imm2)])$$
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
00110		imm2		rs2		rs1		100		rd		0001011	

14.5.31 LURHU: a load halfword instruction that shifts low 32 bits of registers and extends zero bits

Syntax:

```
lurhu rd, rs1, rs2, imm2
```

Operation:

$$rd \leftarrow \text{zero_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)+1: \\ (rs1+rs2[31:0] \ll imm2)])$$
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
10110		imm2		rs2		rs1		100		rd		0001011	

14.5.32 LURW: a load word instruction that shifts low 32 bits of registers and extends signed bits

Syntax:

lurw rd, rs1, rs2, imm2

Operation:

$rd \leftarrow \text{sign_extend}(\text{mem}[(rs1+rs2[31:0] \ll imm2)+3]:$

$(rs1+rs2[31:0] \ll imm2))$

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.

Instruction format:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
01010		imm2		rs2		rs1		100		rd		0001011	

14.5.33 LURWU: a load word instruction that shifts 32 bits of registers and extends zero bits

Syntax:

lwd rd1, rd2, (rs1),imm2

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{imm2} \ll 3)$$

$$\text{rd1} \leftarrow \text{sign_extend}(\text{mem}[\text{address}+3: \text{address}])$$

$$\text{rd2} \leftarrow \text{sign_extend}(\text{mem}[\text{address}+7: \text{address}+4])$$
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	

14.5.34 LWD: a load word instruction that loads double registers and extends signed bits

Syntax:

lwd rd, imm7(rs1)

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm7})$$

$$\text{rd} \leftarrow \text{sign_extend}(\text{mem}[\text{address}+31: \text{address}])$$

$$\text{rd}+1 \leftarrow \text{sign_extend}(\text{mem}[\text{address}+63: \text{address}+32])$$
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	

14.5.35 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	

14.5.36 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	

14.5.37 LWUD: a load word instruction that loads double registers and extends zero bits

Syntax:

```
lwud rd1,rd2, (rs1),imm2
```

Operation:

$$\text{address} \leftarrow \text{rs1} + \text{zero_extend}(\text{imm2} \ll 3)$$

$$\text{rd1} \leftarrow \text{zero_extend}(\text{mem}[\text{address}+3: \text{address}])$$

$$\text{rd2} \leftarrow \text{zero_extend}(\text{mem}[\text{address}+7: \text{address}+4])$$
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	

14.5.38 LWUIA: a base-address auto-increment instruction that extends zero bits and loads words

Syntax:

```
lwuia rd, (rs1), imm5,imm2
```

Operation:

$$\text{rd} \leftarrow \text{zero_extend}(\text{mem}[\text{rs1}+3:\text{rs1}])$$

$$\text{rs1} \leftarrow \text{rs1} + \text{sign_extend}(\text{imm5} \ll \text{imm2})$$
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			
11011				imm2		imm5			rs1		100		rd		0001011	

14.5.39 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	

14.5.40 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	

14.5.41 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	

14.5.42 SDD: an instruction that stores double registers**Syntax:**

sdd rd1,rd2, (rs1),imm2,4

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	

14.5.43 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	

14.5.44 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	

14.5.45 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	

14.5.46 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	

14.5.47 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	

14.5.48 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	

14.5.49 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		

14.5.50 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		

14.5.51 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	

14.5.52 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	

14.5.53 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	

14.5.54 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	

14.5.55 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	

14.5.56 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	

14.5.57 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	

14.6 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.

14.6.1 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	

14.6.2 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	

14.6.3 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	

14.6.4 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	

14.6.5 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.

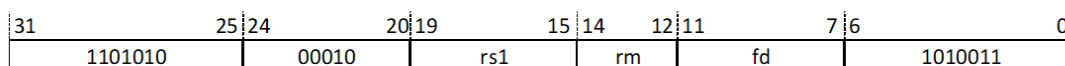
Affected flag bits:

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:

14.6.6 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(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:

31	25 24	20 19	15 14	12 11	7 6	0
1101010	00011	rs1	rm	fd	1010011	

14.6.7 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` \leftarrow `single_convert_to_half(fs1)`

Permission:

M mode/S mode/U mode

Exception:

Illegal instruction.

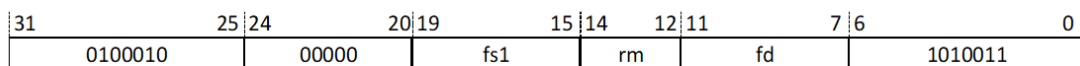
Affected flag bits:

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 `fcsr` register. The corresponding assembler instruction is `fcvt.h.s fd, fs1`.

Instruction format:

14.6.8 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	

14.6.9 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.

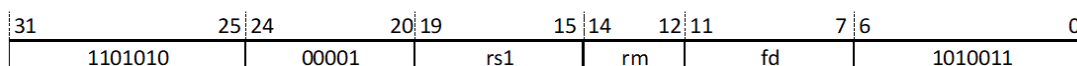
Affected flag bits:

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:

14.6.10 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.

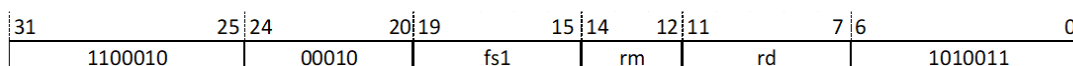
Affected flag bits:

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:

14.6.11 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	

14.6.12 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

14.6.13 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:

$\text{tmp}[31:0] \leftarrow \text{half_convert_to_signed_int}(\text{fs1})$

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

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	

14.6.14 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[31:0] ← half_convert_to_unsigned_int(fs1)
```

```
rd ← sign_extend(tmp[31:0])
```

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	

14.6.15 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:

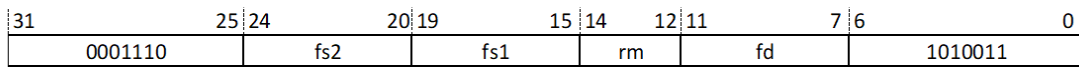
Floating-point status bits NV, DZ, OF, UF, and NX

Notes:

RM determines the round-off mode:

- 3' b000: Rounds off to the nearest even number. The corresponding assembler instruction is `fdiv.h 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:**14.6.16 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:

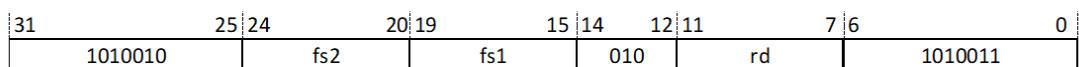
```
M mode/S mode/U mode
```

Exception:

```
Illegal instruction.
```

Affected flag bits:

```
Floating-point status bit NV
```

Instruction format:**14.6.17 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

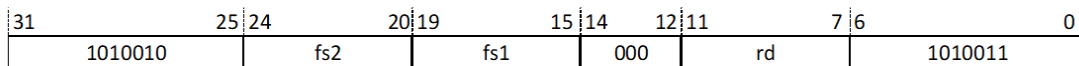
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



14.6.18 FLH: an instruction that loads half-precision floating-point data

Syntax:

flh fd, imm12(rs1)

Operation:

$address \leftarrow rs1 + sign_extend(imm12)$

$fd[15:0] \leftarrow mem[(address+1):address]$

$fd[63:16] \leftarrow 48' \text{ hfffffffffff}$

Permission:

M mode/S mode/U mode

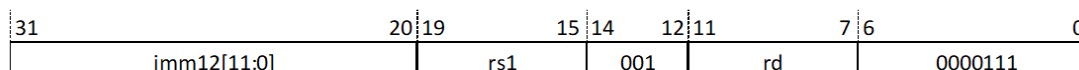
Exception:

Unaligned access, access error, page error, or illegal instruction.

Affected flag bits:

None

Instruction format:



14.6.19 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

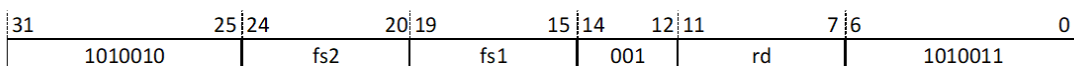
Exception:

Illegal instruction.

Affected flag bits:

Floating-point status bit NV

Instruction format:



14.6.20 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:

31	27	26	25	24	20	19	15	14	12	11	7	6	0
fs3			10		fs2		fs1		rm		fd		1000011

14.6.21 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	

14.6.22 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	

14.6.23 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 UX

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	

14.6.24 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	

14.6.25 FMV.H.X: a half-precision floating-point write transmit instruction**Syntax:**

`fmv.h.x fd, rs1`

Operation:

`fd[15:0] ← rs1[15:0]`

`fd[63:16] ← 48' hfffffffffff`

Permission:

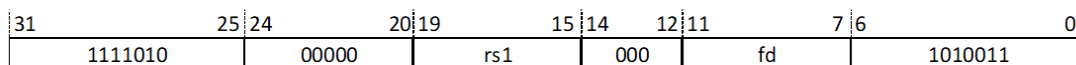
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

14.6.26 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:

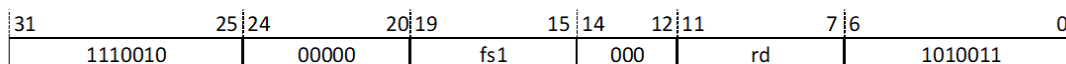
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

14.6.27 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

14.6.28 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	

14.6.29 FSGNJ.H: a half-precision floating-point sign-injection instruction

Syntax:

`fsgnj.h fd, fs1, fs2`

Operation:

`fd[14:0] ← fs1[14:0]`

`fd[15] ← fs2[15]`

`fd[63:16] ← 48' hfffffffffff`

Permission:

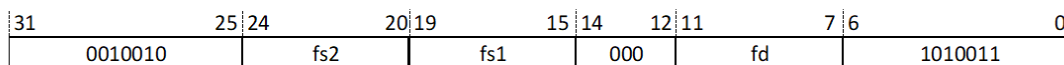
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

14.6.30 FSGNJN.H: a half-precision floating-point sign-injection negate instruction

Syntax:

fsgnfn.h fd, fs1, fs2

Operation: $fd[14:0] \leftarrow fs1[14:0]$ $fd[15] \leftarrow ! fs2[15]$ $fd[63:16] \leftarrow 48' \text{ hfffffffffff}$ **Permission:**

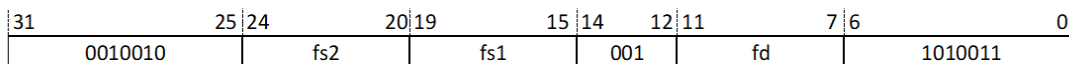
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

14.6.31 FSGNJX.H: a half-precision floating-point sign-injection XOR instruction

Syntax:

fsgnjx.h fd, fs1, fs2

Operation: $fd[14:0] \leftarrow fs1[14:0]$ $fd[15] \leftarrow fs1[15] \wedge fs2[15]$ $fd[63:16] \leftarrow 48' \text{ hfffffffffff}$ **Permission:**

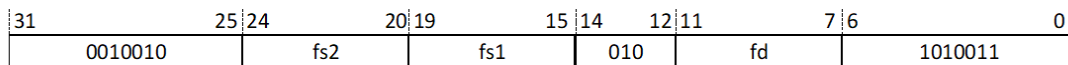
M mode/S mode/U mode

Exception:

Illegal instruction.

Affected flag bits:

None

Instruction format:

14.6.32 FSH: an instruction that stores half-precision floating point numbers

Syntax:

fsh fs2, imm12(fs1)

Operation:

address ← fs1 + sign_extend(imm12)

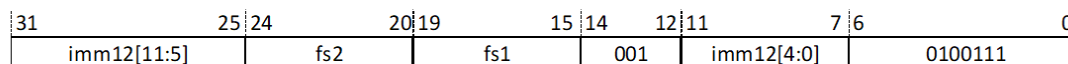
mem[(address+1):address] ← fs2[15:0]

Permission:

M mode/S mode/U mode

Exception:

Unaligned access, access error, page error, or illegal instruction.

Instruction format:

14.6.33 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 `fcscr` 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	

14.6.34 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	

This chapter describes the M-mode control registers, S-mode control registers, and U-mode control registers.

15.1 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, M-mode counter configuration registers, extended M-mode CPU control and status extension registers, extended M-mode cache access registers, and M-mode CPU model registers.

15.1.1 M-mode information register group

15.1.1.1 Machine vendor ID register (MVENDORID)

The MVENDORID register stores the vendor IDs that JEDEC allocates to T-Head Semiconductor Co., Ltd. The register value is fixed to 64' h5B7.

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.

15.1.1.2 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. Currently, C906 does not define this register, and its value is fixed to 64' h0.

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.

15.1.1.3 Machine implementation ID register (MIMPID)

The MIMPID register stores hardware implementation IDs of CPU cores. Currently, C906 does not define this register, and its value is fixed to 64' h0.

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.

15.1.1.4 Machine hart ID register (MHARTID)

The MHARTID register stores hart IDs of CPU cores. The three least significant bits of the MHARTID register specify the core ID of a multi-core CPU. This register value is fixed to 64' h0 in C906.

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.

15.1.2 M-mode exception configuration register group

15.1.2.1 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.

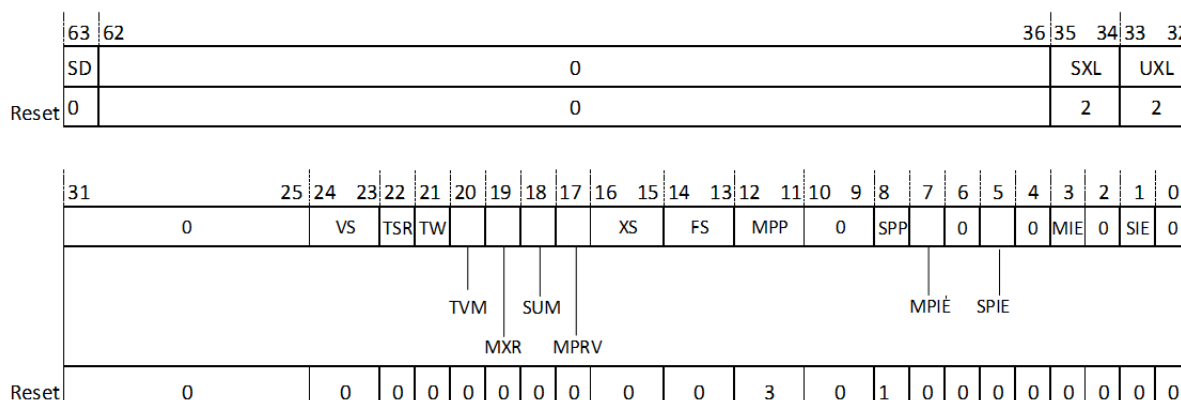


Fig. 15.1: Machine status register (MSTATUS)

SIE: global S-mode 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 to respond to interrupts, and is set to the value of SPIE when the CPU exits the interrupt service program.

MIE: global M-mode 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 CPU responds to interrupts in M-mode, 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 CPU is downgraded to the S-mode to respond to interrupts.

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 accesses to related floating-point registers will cause illegal instruction exceptions.
- 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.

This bit will be reset to 2' b00.

XS: extension status bit

C906 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.

This bit will be reset to 1' b0.

SUM: allow accesses to U-mode virtual memory spaces in S-mode

- When SUM is 1, load, store, and instruction fetch requests can be initiated in S-mode to access virtual memory spaces marked as U-mode.
- When SUM is 0, load, store, and instruction fetch requests cannot be initiated in S-mode to access virtual memory spaces marked as U-mode.

This bit will be reset to 1' b0.

MXR: allow initiation of load requests to access memory spaces marked as executable

- When MXR is 1, load requests can be initiated to access virtual memory spaces marked as executable or readable.
- When MXR is 0, load requests can be initiated only to access virtual memory spaces marked as readable.

This bit will be reset to 1' b0.

TVM: trap virtual memory

- When TVM is 1, an illegal instruction exception occurs for reads and writes to the STAP control register and for the execution of the SFENCE instruction in S-mode.
- When TVM is 0, reads and writes to the STAP control register and the execution of the SFENCE instruction are allowed in S-mode.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

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 illegal instruction 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 fixed to 0.

UXL: U-mode register bit width

This bit is read-only and always 2, which means that the register is 64 bits wide in U-mode.

SXL: S-mode register bit 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.

This bit will be reset to 1' b0.

15.1.2.2 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.

C906 supports the RV64GC instruction set architecture, and the reset value of the MISA register is 64 ‘h8000_0000_0094_112d. For more information about the assignment rules, see the official document of RISC-V riscv-privileged.

C906 does not support the dynamic configuration of the MISA register. Writes to this register do not take effect.

15.1.2.3 M-mode exception downgrade control register (MEDELEG)

The MEDELEG register allows the CPU to downgrade to the S-mode to handle exceptions that occur in S-mode and U-mode. The lower 16 bits of the MEDELEG register are in one-to-one correspondence to exception vector IDs in Table 3.9 exception and interrupt vector assignment. Exceptions can be downgraded to the S-mode for handling.

This register is 64 bits wide and is reset to 64’ h0. This register is readable and writable in M-mode. Accesses in non-M-mode will cause an illegal instruction exception.

15.1.2.4 M-mode interrupt downgrade control register (MIDELEG)

The MIDELEG register allows the CPU to downgrade to the S-mode to handle S-mode interrupts.

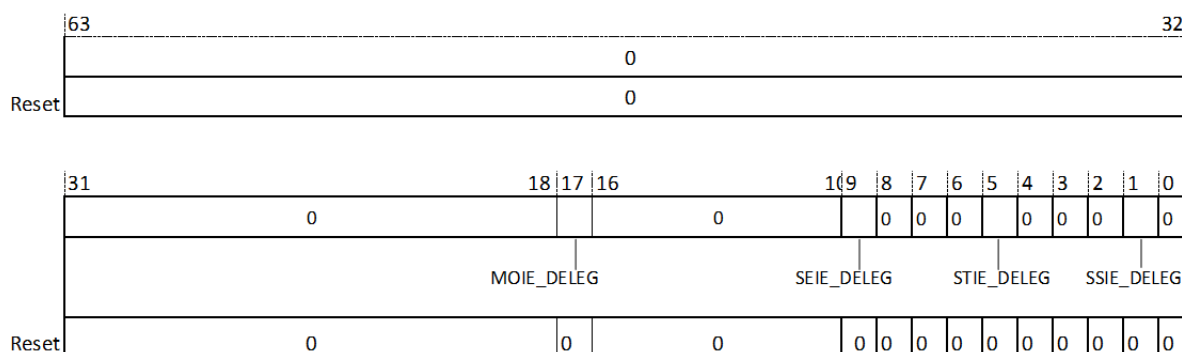


Fig. 15.2: M-mode interrupt downgrade control register (MIDELEG)

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.

SSIE_DELEG: S-mode software interrupt

- When SSIE_DELEG is 1, the CPU can downgrade to the S-mode to handle S-mode software interrupts.
- When SSIE_DELEG is 0, the CPU can handle S-mode software interrupts only in M-mode.

This bit will be reset to 1’ b0.

STIE_DELEG: S-mode timer interrupt

- When STIE_DELEG is 1, the CPU can downgrade to the S-mode to handle S-mode timer interrupts.
- When STIE_DELEG is 0, the CPU can handle S-mode timer interrupts only in M-mode.

This bit will be reset to 1' b0.

SEIE_DELEG: S-mode external interrupt

- When SEIE_DELEG is 1, the CPU can downgrade to the S-mode to handle S-mode external interrupts.
- When SEIE_DELEG is 0, the CPU can handle S-mode external interrupts only in M-mode.

This bit will be reset to 1' b0.

MOIE_DELEG: HPM event counter overflow interrupt

- When MOIE_DELEG is 1, the CPU can be downgraded to the S-mode to handle HPM event counter overflow interrupts.
- When MOIE_DELEG is 0, the CPU can handle HPM event counter overflow interrupts only in M-mode.

This bit will be reset to 1' b0.

15.1.2.5 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.

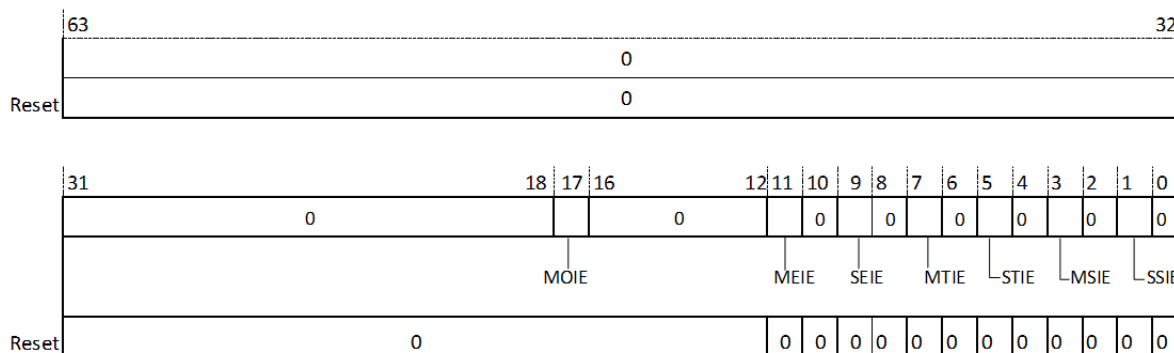


Fig. 15.3: M-mode interrupt-enable register (MIE)

SSIE: S-mode software interrupt enable bit

- When SEIE is 0, S-mode software external interrupts are not enabled.

- When SEIE is 1, S-mode software external interrupts are enabled.

This bit will be reset to 1' b0.

MSIE: M-mode software interrupt enable bit

- When MSIE is 0, M-mode software interrupts are not enabled.
- When MSIE is 1, M-mode software interrupts are enabled.

This bit will be reset to 1' b0.

STIE: S-mode timer interrupt enable bit

- When STIE is 0, S-mode timer interrupts are not enabled.
- When STIE is 1, S-mode timer interrupts are enabled.

This bit will be reset to 1' b0.

MTIE: M-mode timer interrupt enable bit

- When MTIE is 0, M-mode timer interrupts are not enabled.
- When MTIE is 1, M-mode timer interrupts are enabled.

This bit will be reset to 1' b0.

SEIE: S-mode external interrupt enable bit

- When SEIE is 0, S-mode external interrupts are not enabled.
- When SEIE is 1, S-mode external interrupts are enabled.

This bit will be reset to 1' b0.

MEIE: M-mode external interrupt enable bit

- When MEIE is 0, M-mode external interrupts are not enabled.
- When MEIE is 1, M-mode external interrupts are enabled.

This bit will be reset to 1' b0.

MOIE: HPM M-mode event counter overflow interrupt enable bit

- When MOIE is 0, M-mode counter overflow interrupts are not enabled.
- When MOIE is 1, M-mode counter overflow interrupts are enabled.

This bit will be reset to 1' b0.

15.1.2.6 M-mode 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.

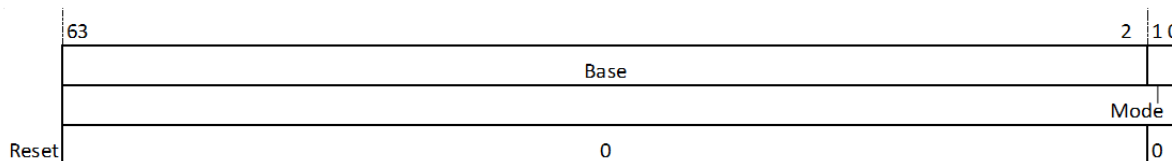


Fig. 15.4: M-mode vector base address register (MTVEC)

BASE: vector base address bit

The BASE bit indicates the upper 37 bits of the entry address of the exception service program. Combining this base address with 2' b00 obtains the entry address of the exception service program.

This bit will be reset to 62' h0.

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*Exception Code is used as the entry address for interrupts.

This bit will be reset to 2' b00.

15.1.2.7 MCOUNTEREN register

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

For more information, see ref:*performance_test*.

15.1.3 M-mode exception handling register group**15.1.3.1 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.

15.1.3.2 M-mode exception program counter register (MEPC)

The MEPC register stores the program counter value (PC value) to be returned when the CPU exits the exception service program. C906 supports RVC instruction sets. The MEPC value is aligned with 16 bits,

and the least significant 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.

15.1.3.3 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.

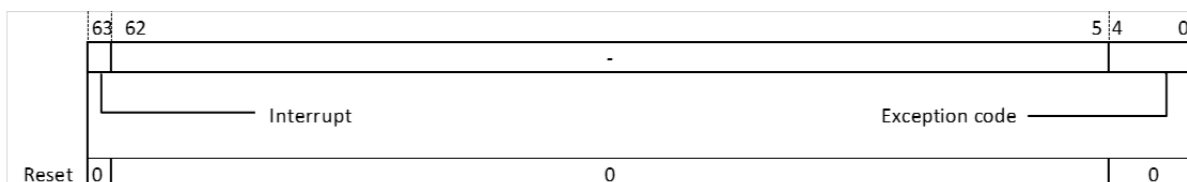


Fig. 15.5: M-mode cause register (MCAUSE)

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.

This bit will be reset to 1' b0.

Exception Code: exception vector number

When the CPU handles an exception or interrupt, this field is updated to the exception vector number. For more information, see [Table 3.9](#).

This bit will be reset to 5' b0.

15.1.3.4 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.

This register is 64 bits wide. Accesses in non-M-mode will cause an illegal instruction exception.

SSIP: supervisor software interrupt pending bit

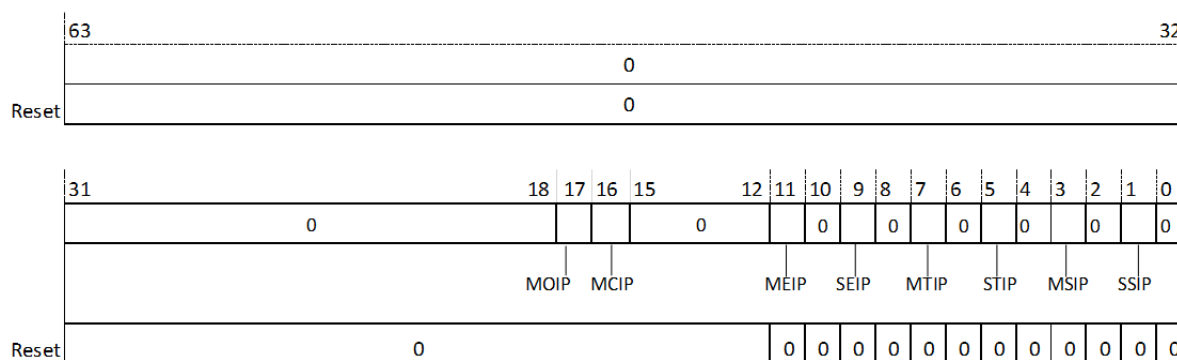


Fig. 15.6: M-mode interrupt pending register (MIP)

- 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.

This bit will be reset to 1' b0 and is readable and writable in M-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 will be reset to 1' b0 and is read-only in M-mode.

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.

This bit will be reset to 1' b0 and is readable and writable in M-mode.

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.

This bit will be reset to 1' b0 and is read-only in M-mode.

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.

This bit will be reset to 1' b0 and is readable and writable in M-mode.

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.

This bit will be reset to 1' b0 and is read-only in M-mode.

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.

This bit will be reset to 1' b0 and is read-only in M-mode.

15.1.4 M-mode memory protection register group

M-mode memory protection registers are related to the settings of the memory protection unit.

15.1.4.1 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.

For more information, see *Physical memory protection configuration (PMPCFG) register*.

15.1.4.2 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 *Physical memory protection address (PMPADDR) register*.

15.1.5 M-mode counter register group

M-mode counter registers belong to the HPM and collect software and hardware information during a program operation for software development personnel to optimize programs.

15.1.5.1 M-mode cycle counter (MCYCLE)

The MCYCLE counter stores the clock 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. In debug mode, this counter stops counting.

The MCYCLE counter is 64 bits wide and will be reset to 0.

For more information, see *Event counters*.

15.1.5.2 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 counter does not count for instructions executed in the debug mode.

The minstret counter is 64 bits wide and will be reset to 0.

For more information, see *Event counters*.

15.1.5.3 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 *Event counters*.

15.1.6 M-mode counter configuration register group

The M-mode counter configuration register selects events for M-mode event counters.

15.1.6.1 M-mode event selector (MHPMEVENTn)

The MHPMEVENTn register selects events for M-mode event counters. M-mode HPM events 3 to 17 have one-to-one correspondence with M-mode HPM counters 3 to 17. In C906, event counters can count only specified events. Therefore, only specified values can be written to MHPMEVENT 3-17.

The MHPMEVENTn counter is 64 bits wide and will be reset to 0.

For more information, see *MHPMCR register*.

15.1.7 M-mode CPU control and status extension register group

C906 extends some registers for the CPU and status, including the M-mode extension status register (MXSTATUS), M-mode hardware control register (MHCR), M-mode hardware operation register (MCOR), M-mode implicit operation register (MHINT), M-mode reset vector base address register (MRVBR), M-mode counter write enable register (MCOUNTERWEN), M-mode event counter overflow interrupt enable register (MCOUNTERINTEN), and M-mode event counter overflow mark register (MCOUNTEROF).

15.1.7.1 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 C906.

UCME: execute extended cache instructions in U-mode

- When UCME is 0, extended cache instructions cannot be executed in U-mode. Otherwise, illegal instruction exceptions may occur.
- When UCME is 1, extended DCACHE.CIVA, DCACHE.CVA, and ICACHE.IVA instructions can be executed in U-mode.

This bit will be reset to 1' b0.

CLINTEE: CLINT timer/software interrupt supervisor extension enable bit

- When CLINTEE is 0, S-mode software interrupts and timer interrupts initiated by CLINT are not responded to.
- When CLINTEE is 1, S-mode software interrupts and timer interrupts initiated by CLINT can be responded to.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

THEADISAEE: enables extended instruction sets

- When THEADISAEE is 0, executing C906 extended instructions causes illegal instruction exceptions.
- When THEADISAEE is 1, C906 extended instructions can be executed.

This bit will be reset to 1' b0.

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.

This bit will be reset to 2' b11.

15.1.7.2 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.

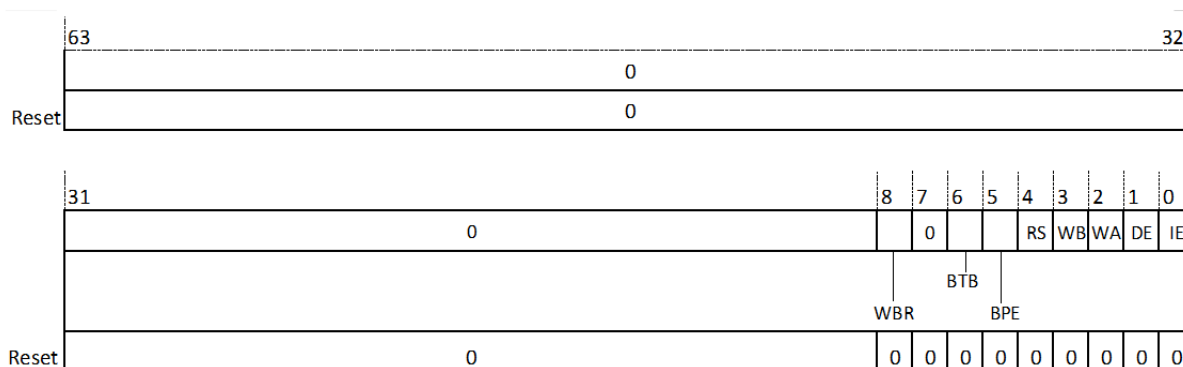


Fig. 15.8: M-mode hardware configuration register (MHCR)

IE: I-Cache enable bit

- When IE is 0, I-Cache is disabled.
- When IE is 1, I-Cache is enabled.

This bit will be reset to 1' b0.

DE: D-Cache enable bit

- When DE is 0, D-Cache is disabled.
- When DE is 1, D-Cache is enabled.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

WB: cache writeback set bit

- When WB is 0, the D-Cache is in write-through mode.
- When WB is 1, the D-Cache is in writeback mode.

C906 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.

This bit will be reset to 1' b0.

BPE: branch prediction enable bit

- When BPE is 0, branch prediction is disabled.
- When BPE is 1, branch prediction is enabled.

This bit will be reset to 1' b0.

BTB: branch target prediction enable bit

- When BTB is 0, branch target prediction is disabled.
- When BTB is 1, branch target prediction is enabled.

This bit will be reset to 1' b0.

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.

In C906, WBR is fixed to 1.

15.1.7.3 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.

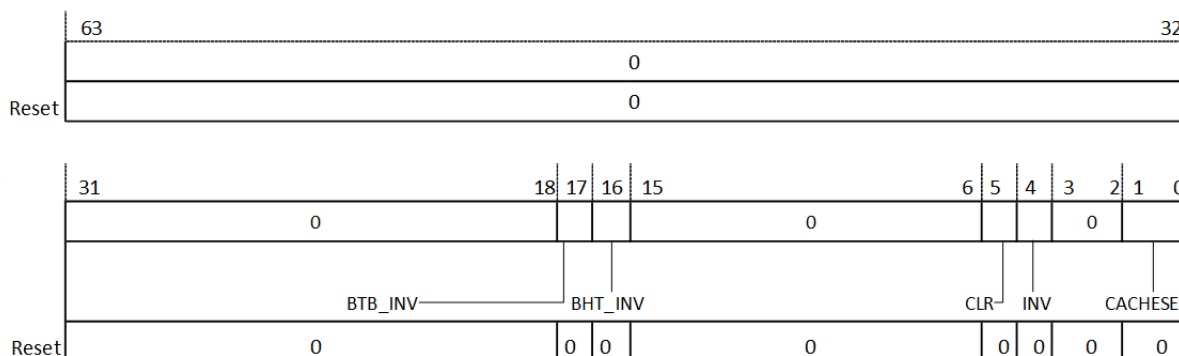


Fig. 15.9: M-mode hardware operation register (MCOR)

CACHE_SEL: cache select bit

- When CACHE_SEL is 2' b01, the I-Cache is selected.
- When CACHE_SEL is 2' b10, the D-Cache is selected.
- When CACHE_SEL is 2' b11, the I-Cache and D-Cache are selected.

This bit will be reset to 2' b00.

INV: cache invalidate bit

- When INV is 0, caches are not being invalidated.
- When INV is 1, caches are being invalidated.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

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.

15.1.7.4 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.

DPLD: D-Cache prefetch enable bit

- When DPLD is 0, D-Cache prefetch is disabled.

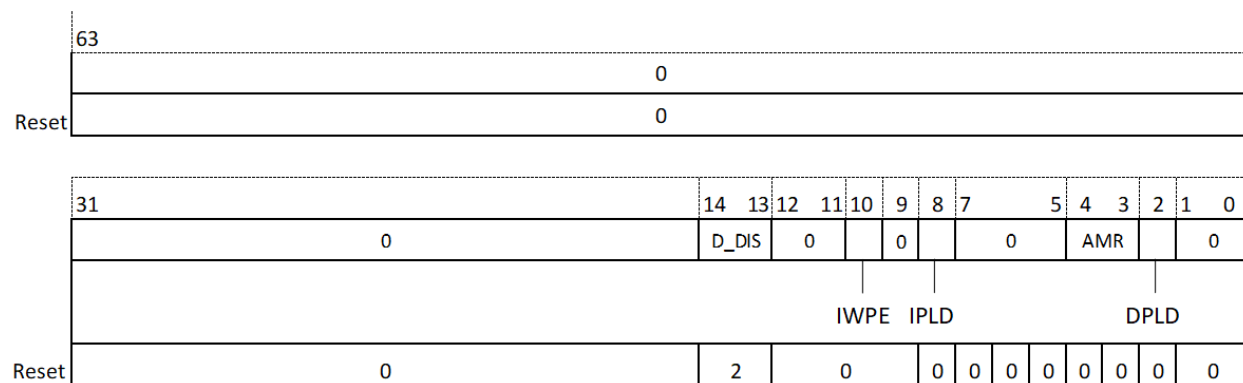


Fig. 15.10: M-mode implicit operation register (MHINT)

- When DPLD is 1, D-Cache prefetch is enabled.

This bit will be reset to 1' b0.

AMR: write allocate policy automatic adjustment enable bit for L1 D-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 3 consecutive cache lines are stored, storage operations of subsequent continuous addresses are no longer written to L1 Cache.
- When AMR is 2, if 64 consecutive cache lines are stored, storage operations of subsequent continuous addresses is no longer written to L1 Cache.
- When AMR is 3, if 128 consecutive cache lines are stored, storage operations of subsequent continuous addresses is no longer written to L1 Cache.

This bit will be reset to 2' b0.

IPLD: I-Cache prefetch enable bit

- When IPLD is 0, I-Cache prefetch is disabled.
- When IPLD is 1, I-Cache prefetch is enabled.

This bit will be reset to 1' b0.

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.

This bit will be reset to 1' b0.

D_DIS: number of prefetched cache lines in D-Cache

- When D_DIS is 0, 2 cache lines are prefetched.

- When D_DIS is 1, 4 cache lines are prefetched.
- When D_DIS is 2, 8 cache lines are prefetched.
- When D_DIS is 3, 16 cache lines are prefetched.

This bit will be reset to 2' b10.

15.1.7.5 M-mode reset vector base address register (MRVBR)

The MRVBR register stores base addresses of reset exception vectors.

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.

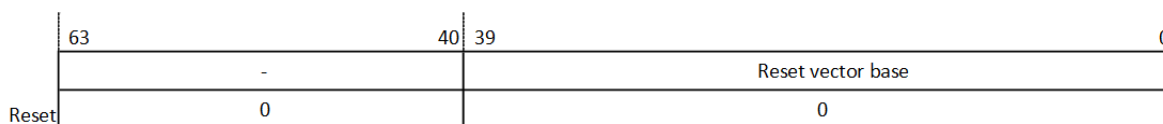


Fig. 15.11: M-mode reset vector base address register (MRVBR)

Reset vector base: reset base address

The value of this register is the reset start address transferred to C906 during SoC system integration. It is not writable in M-mode.

15.1.7.6 MCOUNTERWEN register

The MCOUNTERWEN register determines whether the CPU can write image registers, such as SHPM-COUNTERn of M-mode counters 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.

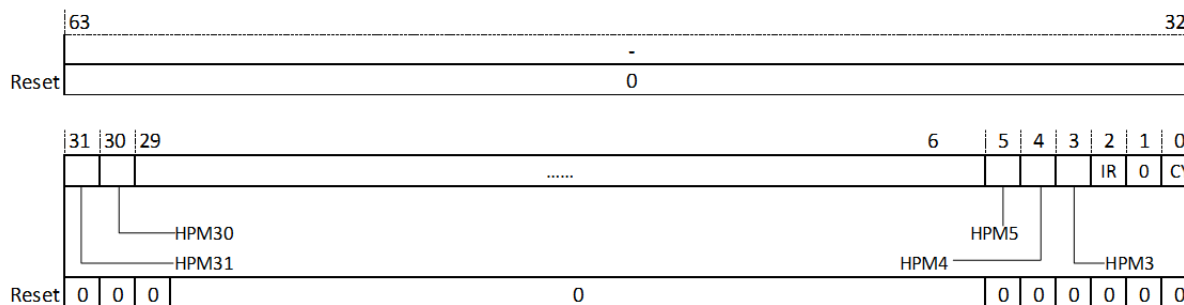


Fig. 15.12: S-mode counter write enable register (MCOUNTERWEN)

- When `MCOUNTERWEN.bit[n]` is 1, the CPU is allowed to write the `SHPMCOUNTERn`, `SINSTRET`, and `SCYCLE` registers in S-mode.
- When `MCOUNTERWEN.bit[n]` is 0, the CPU is not allowed to write the `SHPMCOUNTERn`, `SINSTERT`, or `SCYCLE` register in S-mode. Otherwise, an illegal instruction exception occurs.

15.1.7.7 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. For more information, see *MCOUNTERWEN register*.

15.1.7.8 M-mode event counter overflow mark register (MCOUNTEROF)

The `MCOUNTEROF` 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. For more information, see *HPM event overflow interrupt*.

15.1.7.9 M-mode device address upper bit register (MAPBADDR)

The `MAPBADDR` register marks the upper 13 bits of the register base addresses of PLIC and other in-core modules mounted to the APB. This address is specified when C906 is used during SoC integration. The register is 64 bits wide and read-only in M-mode.

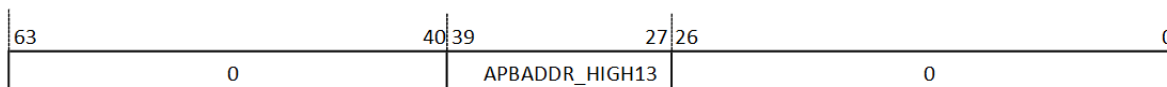


Fig. 15.13: M-mode device address upper bit register (MAPBADDR)

15.1.8 M-mode cache access extension register group

M-mode cache access extension registers directly read content in L1 Cache for cache commissioning.

15.1.8.1 M-mode cache instruction register (MCINS)

The `MCINS` register initiates read requests to L1 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.

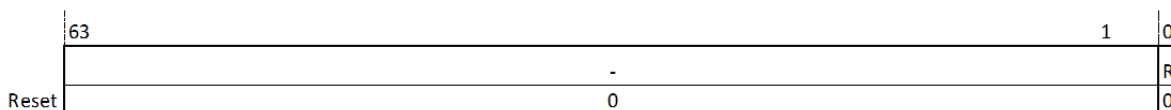


Fig. 15.14: M-mode cache instruction register (MCINS)

R: cache read access

- When R is 0, cache read requests are not initiated.
- When R is 1, cache read requests are initiated.

This bit will be reset to 1' b0.

15.1.8.2 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.

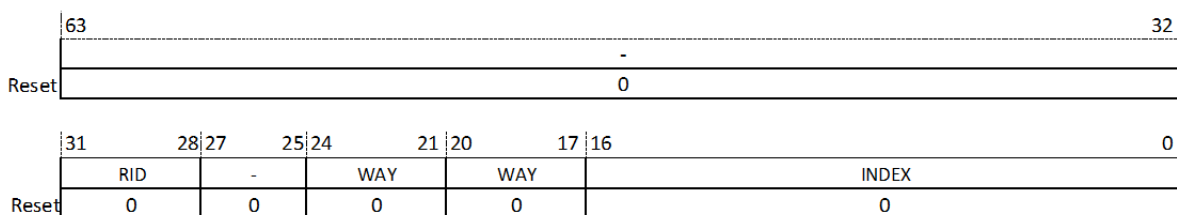


Fig. 15.15: M-mode cache access index register (MCINDEX)

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.

This bit will be reset to 4' b0.

WAY: cache way information

This bit specifies the RAM access way.

This bit will be reset to 4' b0.

INDEX: cache index

This bit specifies the RAM access index. This field is written based on the address format.

For the data array, 16 bytes are read each time and the four least significant bits of the index will be ignored.

For the tag array, the six least significant bits of the index will be ignored because the C906 cache line size is 64 bytes.

This bit will be reset to 21' b0.

15.1.8.3 M-mode cache data register (MCDATA0/1)

The MCDATA0/1 register records data read from L1 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.

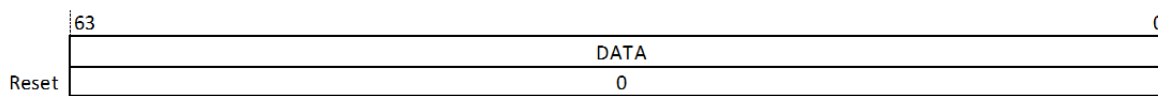


Fig. 15.16: M-mode cache access data register (MCDATA)

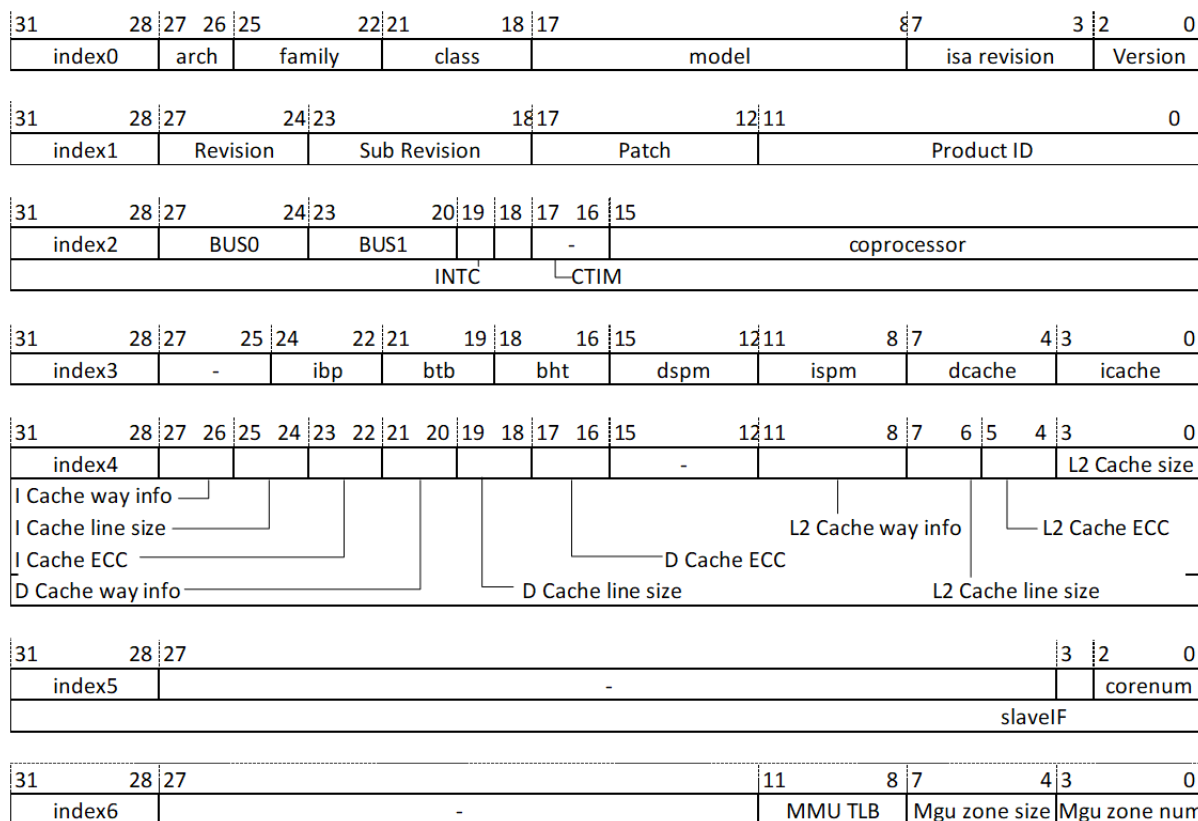
Table 15.1: Correspondence between cache data content and RAM types

RAM type	CDATA content
ICACHE TAG	CDATA0[39:12]: TAG CDATA0[0]: VALID
ICACHE DATA	CDATA1: DATA[127:64] CDATA0: DATA[63:0]
DCACHE TAG	CDATA0[39:12]: TAG CDATA0[2]: DIRTY CDATA0[0]: VALID
DCACHE DATA	CDATA1: DATA[127:64] CDATA0: DATA[63:0]

15.1.9 M-mode CPU model register group

15.1.9.1 M-mode CPU model register (MCPUID)

The MCPUID register stores CPU models. Its reset value is determined by the product and complies with T-Head product definition specifications to facilitate software identification. Continuous read of the MCPUID register can obtain up to seven different return values to present C906 product information, as shown in Fig. ??.



When DebugServer of the XuanTie CPU connects to a chip that contains the XuanTie CPU, it automatically parses the preceding information and prints it for users to identify.

15.2 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, S-mode address translation registers, extended S-mode CPU status and control registers, and extended S-mode MMU registers.

15.2.1 S-mode exception configuration registers

When exceptions and interrupts are downgraded to the S-mode for handling, exceptions must be configured through the S-mode exception configuration registers, like in M-mode.

15.2.1.1 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 M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

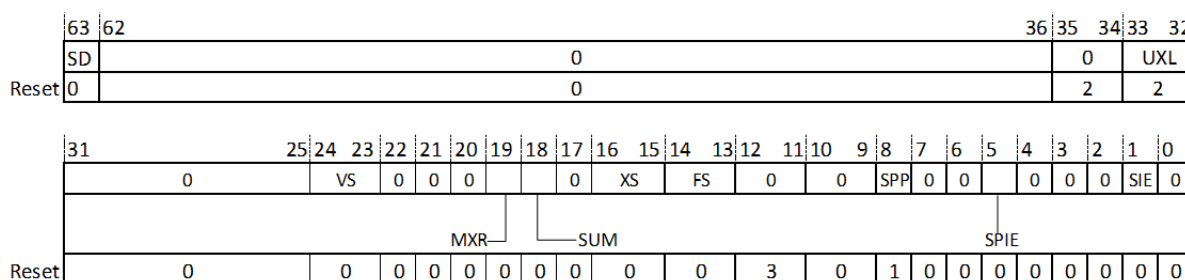


Fig. 15.17: S-mode status register (SSTATUS)

For more information, see *Machine status register (MSTATUS)*.

15.2.1.2 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, readable and writable in M-mode, and read-only in S-mode. The write permission in S-mode is determined by the corresponding bit of the MIDELEG register. Accesses in U-mode will cause an illegal instruction exception.

For more information, see *M-mode interrupt-enable register (MIE)*.

15.2.1.3 S-mode vector base address register (STVEC)

The STVEC register stores the entry address of the exception service program. It is used when exceptions and interrupts are downgraded to the S-mode for handling.

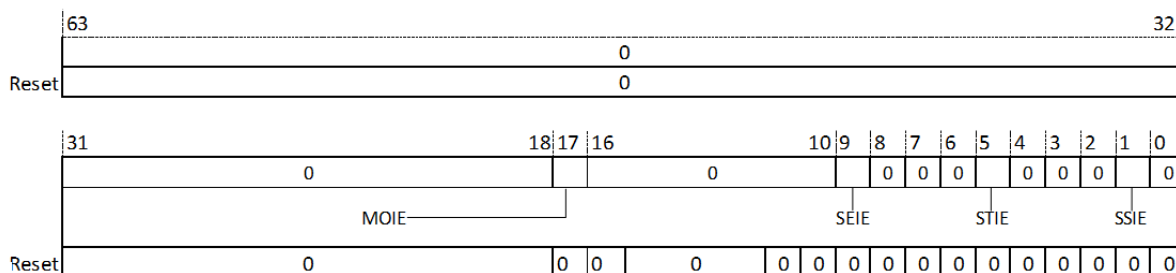


Fig. 15.18: S-mode interrupt enable register (SIE)

This register is 64 bits wide and is readable and writable in M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

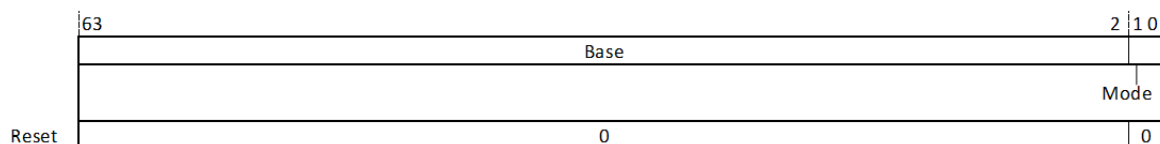


Fig. 15.19: S-mode vector base address register (STVEC)

For more information, see *M-mode vector base address register (MTVEC)*. All fields have the same meaning.

15.2.1.4 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 *SCOUNTEREN register*.

15.2.2 S-mode exception handling registers

15.2.2.1 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 M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

15.2.2.2 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. C906 supports RVC instruction sets. The values of SEPC are aligned with 16 bits and the least significant bit is 0.

This register is 64 bits wide and is readable and writable in M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

15.2.2.3 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 M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

15.2.2.4 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, is readable and writable in M-mode, and read-only in S-mode. The write permission in S-mode is determined by the corresponding bit of the MIDELEG register. Accesses in U-mode will cause an illegal instruction exception.

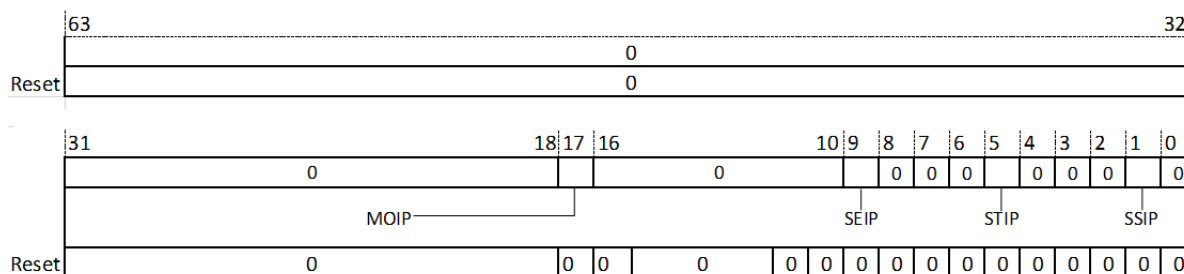


Fig. 15.20: S-mode interrupt-pending register (SIP)

15.2.2.5 S-mode event cause register (STVAL)

The STVAL register stores the causes of events that trigger exceptions. The causes 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 and S-mode. Accesses in U-mode will cause an illegal instruction exception.

15.2.3 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.

15.2.3.1 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 M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see *MMU address translation register (SATP)*.

15.2.4 Extended S-mode CPU control and status registers

15.2.4.1 Extended S-mode status register (SXSTATUS)

The SXSTATUS register is the mapping of the MXSTATUS register.

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. In S-mode, the MM/PMDS/PMDU bit is writable. Accesses in U-mode will cause an illegal instruction exception.

For more information, see *M-mode extension status register (MXSTATUS)*.

15.2.4.2 S-mode hardware control register (SHCR)

The SHCR register is the mapping of the MHCR register.

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see *M-mode hardware configuration register (MHCR)*.

15.2.4.3 S-mode event overflow interrupt enable register (SCOUNTERINTEN)

The SCOUNTERINTEN register is the mapping of the MCOUNTERINTEN register. For more information, see *Event counters*.

This register is 64 bits wide and is readable and writable in M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

When MCOUNTERWEN.bit[n] is 0, SCOUNTERINTEN.bit[n] is read as 0, and write operations to this bit are invalid. When MCOUNTERWEN.bit[n] is 1, write operations to SCOUNTERINTEN.bit[n] are

valid. When `SCOUNTERINTEN.bit[n]` is 1, overflow interrupts on the `SHPMCOUNTERn`, `SCYCLE`, and `SINSTRET` counters are enabled.

15.2.4.4 S-mode event overflow mark register (SCOUNTEROF)

The `SCOUNTEROF` register is the mapping of the `MCOUNTEROF` register. For more information, see *Event counters*.

This register is 64 bits wide and is readable and writable in M-mode and S-mode. Accesses in U-mode will cause an illegal instruction exception.

When `MCOUNTERWEN.bit[n]` is 1, `SCOUNTEROF.bit[n]` indicates whether overflow interrupts occur on the `SHPMCOUNTERn`, `SCYCLE`, and `SINSTRET` counters. When `MCOUNTERWEN.bit[n]` is 0, `SCOUNTEROF.bit[n]` is read as 0 in S-mode. Write operations to `SCOUNTEROF.bit[n]` in S-mode are invalid.

15.2.4.5 S-mode cycle counter (SCYCLE)

The `SCYCLE` counter stores the clock 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 `SCYCLE` counter is 64 bits wide and will be reset to 0.

For more information, see *Event counters*.

15.2.4.6 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 *Event counters*.

15.2.4.7 S-mode event counter (SHPMCOUNTERn)

The `SHPMCOUNTERn` counter is the mapping of the `MHPMCOUNTERn` counter.

For more information, see *Event counters*.

15.2.5 Extended S-mode MMU registers

C906 extends MMU-related registers to implement software writeback. Software can directly write and read the TLB.

15.2.5.1 S-mode MMU control register (SMCIR)

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see `virtual_mem_manage_smcir`.

15.2.5.2 S-mode MMU control register (SMIR)

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see `virtual_mem_manage_smir`.

15.2.5.3 S-mode MMU control register (SMEH)

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see `virtual_mem_manage_smeh`.

15.2.5.4 S-mode MMU control register (SMEL)

This register is 64 bits wide, readable and writable in M-mode, and read-only in S-mode. Accesses in U-mode will cause an illegal instruction exception.

For more information, see `virtual_mem_manage_smel`.

15.3 Appendix C-3 U-mode control registers

U-mode control registers are classified by function into U-mode floating-point control registers, U-mode event counter registers, and extended U-mode floating-point control registers.

15.3.1 U-mode floating-point control registers

15.3.1.1 Floating-point accrued exceptions register (FFLAGS)

The FFLAGS register is the mapping of accrued exceptions of the FCSR register. For more information, see *Floating-point control and status register (FCSR)*.

15.3.1.2 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 *Floating-point control and status register (FCSR)*.

15.3.1.3 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.

	63						32	
Reset	0							
	31	8	7	5	4	0		
	-			Rounding Mode(frm)	Accrued Exceptions(fflags)			
				NV	DZ	OF	UF	NX
Reset	0			0	0	0	0	0

Fig. 15.21: Floating-point control and status register (FCSR)

NX: imprecise exception

- When NX is 0, no imprecise exception occurs.
- When NX is 1, imprecise exceptions occur.

This bit will be reset to 1' b0.

UF: underflow exception

- When UF is 0, no underflow exception occurs.
- When UF is 1, underflow exceptions occur.

This bit will be reset to 1' b0.

OF: overflow exception

- When OF is 0, no overflow exception occurs.
- When OF is 1, overflow exceptions occur.

This bit will be reset to 1' b0.

DZ: division by zero exception

- When DZ is 0, no division by zero exception occurs.

- When DZ is 1, division by zero exceptions occur.

This bit will be reset to 1' b0.

NV: invalid instruction operand exception

- When NV is 0, no exception of invalid instruction operands occurs.
- When NV is 1, exceptions of invalid instruction operands occur.

This bit will be reset to 1' b0.

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.

This bit will be reset to 3' b0.

VXSAT: vector overflow flag bit

This register is the mapping of the least significant bit.

This bit will be reset to 1' b0.

VXRM: vector rounding mode bit

This register is the mapping of the two least significant bits.

This bit will be reset to 2' b0.

15.3.2 U-mode event counter registers

15.3.2.1 User-mode 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.

This register is 64 bits wide, readable and writable in M-mode and S-mode, and read-only in U-mode. This register will be reset to 0.

For more information, see *ref:performance_test_cont*.

15.3.2.2 U-mode timer register (TIME)

The TIME register is the read-only mapping of the MTIME register. It is readable and writable in M-mode and S-mode.

For more information, see *Event counters*.

15.3.2.3 U-mode 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 INSTRET counter is 64 bits wide, readable and writable in M-mode and S-mode, and read-only in U-mode. The INSTRET counter will be reset to 0.

For more information, see *Event counters*.

15.3.2.4 U-mode event counter (HPMCOUNTERn)

The HPMCOUNTERn counter is the read-only mapping of the MHPMCOUNTERn counter and is readable and writable in M-mode and S-mode.

For more information, see *Event counters*.

15.3.3 Extended U-mode floating-point control registers

15.3.3.1 Extended U-mode floating-point control register (FXCR)

The FXCR register controls the floating-point extension function and floating-point exception accrue bit.

	63															32																																																																																																																																																																				
Reset	0																																																																																																																																																																																			
	31															0																																																																																																																																																																				
	27															26															24															23															22															6															5															4															3															2															1															0														
	BF16															-															RM															-															FE															NV															DZ															OF															UF															NX																																												
	DQNaN																																																																																																																																																																																			
Reset	0															0															0															0															0															0															0															0															0															0															0																													

Fig. 15.22: Floating-point extension control register (FXCR)

NX: imprecise exception

It is the mapping of the corresponding bit of the FCSR register.

UF: underflow exception

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: invalid 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.

This bit will be reset to 1' b0.

DQNaN: output QNaN mode bit

When DQNaN is 0, the output QNaN value is the default value specified in RISC-V. That is, the signed bit is 0, all bits of the index are 1, the most significant bit of the end number is 1, and other bits of it are 0.

When DQNaN is 1, the output QNaN value is consistent with the IEEE754 standard.

This bit will be reset to 1' b0.

RM: rounding mode

It is the mapping of the corresponding bit of the FCSR register.

BF16: select BF16 format

When this bit is 1, the 16-bit floating-point data format will be handled as the BF16 format instead of the half-precision floating-point data format.

When this bit is 0, the 16-bit floating-point data format will be handled as the half-precision floating-point data format.

This bit will be reset to 1' b0.

15.3.4 Extended vector registers

15.3.4.1 Vector start position register (VSTART)

The VSTART register specifies the position of the start element when a vector instruction is executed. The VSTART register will be reset to 0 after each vector instruction is executed.

15.3.4.2 Fixed-point overflow flag bit register (VXSAT)

The VXSAT register specifies whether any fixed-point instruction overflows.

15.3.4.3 Fixed-point rounding mode register (VXRM)

The VXRM register specifies the rounding mode used by fixed-point instructions.

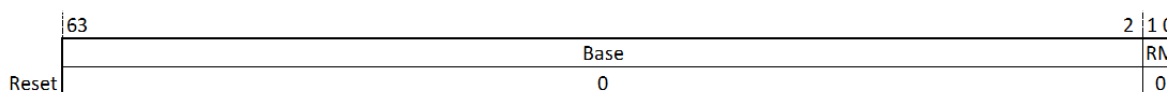


Fig. 15.23: Fixed-point rounding mode register (VXRM)

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.

15.3.4.4 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 or equal to 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.

15.3.4.5 Vector data type register (VTYPE)

The VTYPE register specifies the data type and elements of the vector registers.

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

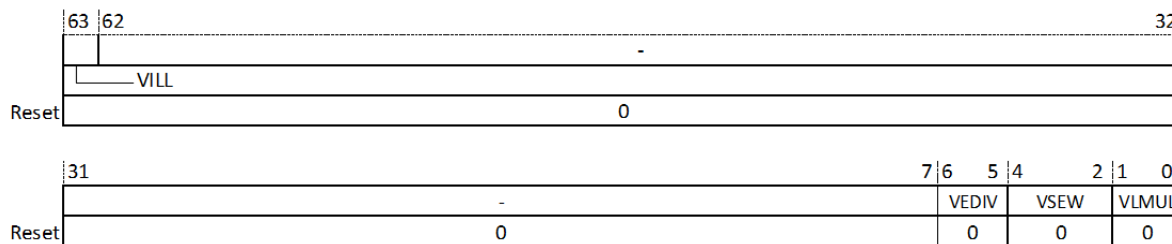


Fig. 15.24: Fixed-point type register (VTYPE)

This bit is set only when the `vsetvli/vsetvl` instruction updates the VTYPE register with a value not supported by C906. 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

C906 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). Table 15.2 describes the vector element widths supported by C906.

Table 15.2: Vector element widths

VSEW[2:0]			Element width
0	0	0	8
0	0	1	16
0	1	0	32

When VSEW takes other values, executing C906 vector instructions causes illegal 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 determines the number of vector registers (LMUL) in the vector register group, as described in Table 15.3.

Table 15.3: Number of registers in the vector register group

VLMUL[1:0]		LMUL
0	0	1
0	1	2
1	0	4
1	1	8

15.3.4.6 Vector register width (unit: byte) register (VLENB)

The VLENB register specifies the data width of the vector registers, that is, the number of bytes obtained by dividing the actual width with 8. C906 vector registers are 128 bits wide. Therefore, the VLENB register value is fixed to 16.

This register is 64 bits wide and read-only in U-mode.

This chapter describes various program examples, including the MMU setting example, PMP setting example, cache setting example, PLIC setting example, HPM setting example, and CPU power-off software process setting example.

16.1 MMU setting example

```
/******  
  
* Function: An example of set C906 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: {vpn2} <-> ppn: {ppn2}  
*  
*****/  
  
/*C906 will invalid all MMU TLB Entry automatically when reset*/  
/*you can use sfence.vma to invalid all MMU TLB Entry if necessarily*/  
sfence.vma x0, x0
```

(continues on next page)

(continued from previous page)

```

/* 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

/*2. cfig 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. cfig 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}

```

(continues on next page)

(continued from previous page)

```

sd x6, 0(x5)
/*4. cfig 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
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. cfig 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

```

(continues on next page)

(continued from previous page)

```

/*store pte at first-level page addr*/
li x6, {44'b BASE2, 10'b1}
sd x6, 0(x5)

/*3. cfig 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*/
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. cfig 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

```

(continues on next page)

(continued from previous page)

```
/*store pte at first-level page addr*/
li x6, { theadflag, ppn2, 9'b0, 9'b0, 9'b flags,1'b1}
sd x6, 0(x5)
```

16.2 PMP setting example

```

/*****
* Function: An example of set C906 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 address regions requiring execution permissions are configured, no permission
↳should be configured for the remaining address regions. 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

```

(continues on next page)

(continued from previous page)

```

# 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 ~
0xffffffff. The minimum PMP granularity is 4 KB in C906. An NAPOT entry must be
↳configured to mask the last 4 KB space of a 1 TB space.

```

16.3 Cache setting example

16.3.1 Cache enabling example

```

/*C906 will invalid all I-cache automatically when reset*/
/*you can invalid I-cache by yourself if necessarily*/
/*invalid I-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x11
csrs mcor, x3
// it can also use icache instruciton to replace the invalid sequence if theadisae is
↳enabled.
//icache.iall
//sync.is
/*enable I-cache*/
li x3, 0x1
csrs mhcr, x3
/*C906 will invalid all D-cache automatically when reset*/
/*you can invalid D-cache by yourself if necessarily*/
/*invalid D-cache*/
li x3, 0x33
csrc mcor, x3
li x3, 0x12
csrs mcor, x3
// it can also use dcache instruciton to replace the invalid sequence if theadisae is
↳enabled.
// dcache.iall

```

(continues on next page)

(continued from previous page)

```
// sync.is
/*enable D-cache*/
li x3, 0x2
csrs mhcr, x3
/*C906 will invalid all D-cache automatically when reset*/
/*you can invalid D-cache by yourself if necessarily*/
```

16.3.2 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 instrcutioin.
sync.s      // ensure completion of clean operation.
icache.iva r0 // invalid icache according to shareable configuraiton.
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
```

16.3.3 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
```

16.4 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
```

(continues on next page)

(continued from previous page)

```

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*/

```

16.5 HPM setting example

```

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

/*2.C906 will initial all HPM counters when reset*/
/*you can initial HPM counters manually if necessarily*/
csrwr mcycle, x0
csrwr minstret, x0
csrwr mhpmcounter3, x0
.....
csrwr mhpmcounter31, x0

```

(continues on next page)

(continued from previous page)

```

/*3.configure mhpmevent*/
li x3, 0x1
csrw mhpmevent3, x3 // mhpcounter3 count event: L1 ICache Access Counter
li x3, 0x2
csrw mhpmevent4, x3 // mhpcounter4 count event: L1 ICache 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

```

16.6 CPU power-off software process setting example

```

/*1.clear MIE(or SIE)*/
/*ensure you are in M/S mode and the external debug request is forbidden*/
li x3, 0x20aaa
csrs mie, x3

/*2.clear mie in MSTATUS(or sie in STATUS)*/
li x3, 0x8
csrci mstatus, x3

/*3.close data prefetch*/
li x3, 0x4
csrci mhint, x3

/*4. clear DCACHE and close it*/
dcache.call

li x3, 0x2

```

(continues on next page)

(continued from previous page)

```
csrci mhcr, x3  
  
/*5. execute wfi*/  
wfi
```