



DRAFT

rva Profile Family

Version 2.0, 2024-09-06

Table of Contents

Licensing and Acknowledgements	1
Copyright and license information	2
Acknowledgements	3
1. RISC-V Profiles	4
1.1. Profiles versus Platforms	4
1.2. Components of a Profile	5
1.2.1. Profile Family	5
1.2.2. Profile Privilege Mode	5
1.2.3. Profile ISA Features	6
2. Profiles in the rva family	8
3. Family description	9
3.1. Extension summary	9
4. RVA20S64 Profile	11
4.1. Extensions	11
4.1.1. Mandatory Extensions	11
4.1.2. Optional Extensions	12
5. RVA20U64 Profile	14
5.1. Extensions	14
5.1.1. Mandatory Extensions	14
5.1.2. Optional Extensions	15
6. RVA22S64 Profile	16
6.1. Extensions	16
6.1.1. Mandatory Extensions	16
6.1.2. Optional Extensions	18
7. RVA22U64 Profile	20
7.1. Extensions	20
7.1.1. Mandatory Extensions	20
7.1.2. Optional Extensions	22
8. Profile Parameters	24
Appendix A: Extension Specifications	36
A.1. I Extension	37
A.1.1. Synopsys	37
A.1.2. Instructions	37
A.1.3. Parameters	38
A.2. A Extension	44
A.2.1. Synopsys	44
A.2.2. Specifying Ordering of Atomic Instructions	44
A.2.3. Instructions	45
A.2.4. Parameters	45
A.3. C Extension	47
A.3.1. Synopsys	47

A.3.2. Overview	47
A.3.3. Compressed Instruction Formats	49
A.3.4. Parameters	50
A.4. D Extension	52
A.4.1. Synopsys	52
A.4.2. D Register State	52
A.4.3. NaN Boxing of Narrower Values	52
A.4.4. Parameters	53
A.5. F Extension	54
A.5.1. Synopsys	54
A.5.2. F Register State	54
Floating-Point Control and Status Register	55
A.5.3. NaN Generation and Propagation	57
A.5.4. Subnormal Arithmetic	57
A.5.5. Instructions	58
A.5.6. Parameters	58
A.6. M Extension	59
A.6.1. Synopsys	59
A.6.2. Instructions	59
A.6.3. Parameters	59
A.7. U Extension	61
A.7.1. Synopsys	61
A.7.2. Parameters	61
A.8. Zicntr Extension	62
A.8.1. Synopsys	62
A.9. Ziccif Extension	63
A.10. Ziccrse Extension	64
A.11. Ziccamoa Extension	65
A.12. Za128rs Extension	66
A.13. Zicclsm Extension	67
A.14. S Extension	68
A.14.1. Synopsys	68
A.14.2. Instructions	68
A.14.3. Parameters	68
A.15. Zifencei Extension	72
A.15.1. Instructions	73
A.16. Svbare Extension	74
A.17. Sv39 Extension	75
A.17.1. Synopsys	75
A.18. Svade Extension	76
A.18.1. Synopsys	76
A.19. Ssccptr Extension	77
A.19.1. Synopsys	77

A.20. Sstvecd Extension	78
A.20.1. Synopsys	78
A.21. Sstvala Extension.....	79
A.21.1. Synopsys.....	79
A.22. Zihpm Extension.....	80
A.22.1. Synopsys.....	80
A.22.2. Parameters	80
A.23. Sv48 Extension.....	81
A.23.1. Synopsys	81
A.24. Zihintpause Extension.....	82
A.25. Zba Extension	84
A.25.1. Synopsys.....	84
A.25.2. Instructions	84
A.26. Zbb Extension.....	85
A.26.1. Synopsys	85
A.26.2. Instructions.....	85
A.27. Zbs Extension	87
A.27.1. Synopsys	87
A.27.2. Instructions.....	87
A.28. Zic64b Extension.....	88
A.28.1. Synopsys	88
A.29. Zicbom Extension.....	89
A.29.1. Synopsys	89
A.29.2. Instructions.....	89
A.29.3. Parameters	89
A.30. Zicbop Extension.....	90
A.30.1. Synopsys	90
A.30.2. Parameters	90
A.31. Zicboz Extension.....	91
A.31.1. Synopsys.....	91
A.31.2. Instructions	91
A.31.3. Parameters	91
A.32. Zfhmin Extension.....	92
A.32.1. Synopsys	92
A.32.2. Instructions.....	92
A.33. Zkt Extension	94
A.33.1. Synopsys.....	94
Scope and Goal.....	94
Background.....	95
Specific Instruction Rationale.....	96
Programming Information.....	96
Zkt listings.....	96
RVI (Base Instruction Set)	97

RVM (Multiply).....	97
RVC (Compressed).....	98
RVK (Scalar Cryptography).....	98
RVB (Bitmanip).....	99
A.34. Sscounterenw Extension.....	101
A.34.1. Synopsys.....	101
A.35. Svpbmt Extension.....	102
A.36. Svinval Extension.....	103
A.36.1. Synopsys.....	103
A.36.2. Instructions.....	104
A.37. Sv57 Extension.....	105
A.37.1. Synopsys.....	105
A.38. Sstc Extension.....	106
A.38.1. Synopsys.....	106
A.39. Sscofpmf Extension.....	107
A.39.1. Synopsys.....	107
A.40. H Extension.....	108
A.40.1. Synopsys.....	108
A.40.2. Privilege Modes.....	109
A.40.3. Parameters.....	110
Appendix B: Instruction Specifications	114
B.1. add.....	115
B.1.1. Encoding.....	115
B.1.2. Synopsis.....	115
B.1.3. Access.....	115
B.1.4. Decode Variables.....	115
B.1.5. Execution.....	115
B.1.6. Exceptions.....	116
B.2. add.uw.....	117
B.2.1. Encoding.....	117
B.2.2. Synopsis.....	117
B.2.3. Access.....	117
B.2.4. Decode Variables.....	117
B.2.5. Execution.....	117
B.2.6. Exceptions.....	118
B.3. addi.....	119
B.3.1. Encoding.....	119
B.3.2. Synopsis.....	119
B.3.3. Access.....	119
B.3.4. Decode Variables.....	119
B.3.5. Execution.....	119
B.3.6. Exceptions.....	120
B.4. addiw.....	121

B.4.1. Encoding	121
B.4.2. Synopsis	121
B.4.3. Access	121
B.4.4. Decode Variables	121
B.4.5. Execution	121
B.4.6. Exceptions	122
B.5. addw	123
B.5.1. Encoding	123
B.5.2. Synopsis	123
B.5.3. Access	123
B.5.4. Decode Variables	123
B.5.5. Execution	123
B.5.6. Exceptions	124
B.6. amoad.d	125
B.6.1. Encoding	125
B.6.2. Synopsis	125
B.6.3. Access	125
B.6.4. Decode Variables	125
B.6.5. Execution	125
B.6.6. Exceptions	127
B.7. amoad.w	128
B.7.1. Encoding	128
B.7.2. Synopsis	128
B.7.3. Access	128
B.7.4. Decode Variables	128
B.7.5. Execution	128
B.7.6. Exceptions	130
B.8. amoand.d	131
B.8.1. Encoding	131
B.8.2. Synopsis	131
B.8.3. Access	131
B.8.4. Decode Variables	131
B.8.5. Execution	131
B.8.6. Exceptions	133
B.9. amoand.w	134
B.9.1. Encoding	134
B.9.2. Synopsis	134
B.9.3. Access	134
B.9.4. Decode Variables	134
B.9.5. Execution	134
B.9.6. Exceptions	136
B.10. amomax.d	137
B.10.1. Encoding	137

B.10.2. Synopsis	137
B.10.3. Access	137
B.10.4. Decode Variables	137
B.10.5. Execution	137
B.10.6. Exceptions	139
B.11. amomax.w	140
B.11.1. Encoding	140
B.11.2. Synopsis	140
B.11.3. Access	140
B.11.4. Decode Variables	140
B.11.5. Execution	140
B.11.6. Exceptions	142
B.12. amomaxu.d	143
B.12.1. Encoding	143
B.12.2. Synopsis	143
B.12.3. Access	143
B.12.4. Decode Variables	143
B.12.5. Execution	143
B.12.6. Exceptions	145
B.13. amomaxu.w	146
B.13.1. Encoding	146
B.13.2. Synopsis	146
B.13.3. Access	146
B.13.4. Decode Variables	146
B.13.5. Execution	146
B.13.6. Exceptions	148
B.14. amomin.d	149
B.14.1. Encoding	149
B.14.2. Synopsis	149
B.14.3. Access	149
B.14.4. Decode Variables	149
B.14.5. Execution	149
B.14.6. Exceptions	151
B.15. amomin.w	152
B.15.1. Encoding	152
B.15.2. Synopsis	152
B.15.3. Access	152
B.15.4. Decode Variables	152
B.15.5. Execution	152
B.15.6. Exceptions	154
B.16. amominu.d	155
B.16.1. Encoding	155
B.16.2. Synopsis	155

B.16.3. Access	155
B.16.4. Decode Variables	155
B.16.5. Execution	155
B.16.6. Exceptions	157
B.17. amominu.w	158
B.17.1. Encoding	158
B.17.2. Synopsis	158
B.17.3. Access	158
B.17.4. Decode Variables	158
B.17.5. Execution	158
B.17.6. Exceptions	160
B.18. amoor.d	161
B.18.1. Encoding	161
B.18.2. Synopsis	161
B.18.3. Access	161
B.18.4. Decode Variables	161
B.18.5. Execution	161
B.18.6. Exceptions	163
B.19. amoor.w	164
B.19.1. Encoding	164
B.19.2. Synopsis	164
B.19.3. Access	164
B.19.4. Decode Variables	164
B.19.5. Execution	164
B.19.6. Exceptions	166
B.20. amoswap.d	167
B.20.1. Encoding	167
B.20.2. Synopsis	167
B.20.3. Access	167
B.20.4. Decode Variables	167
B.20.5. Execution	167
B.20.6. Exceptions	168
B.21. amoswap.w	169
B.21.1. Encoding	169
B.21.2. Synopsis	169
B.21.3. Access	169
B.21.4. Decode Variables	169
B.21.5. Execution	169
B.21.6. Exceptions	170
B.22. amoxor.d	171
B.22.1. Encoding	171
B.22.2. Synopsis	171
B.22.3. Access	171

B.22.4. Decode Variables.....	171
B.22.5. Execution.....	171
B.22.6. Exceptions.....	173
B.23. amoxor.w.....	174
B.23.1. Encoding.....	174
B.23.2. Synopsis.....	174
B.23.3. Access.....	174
B.23.4. Decode Variables.....	174
B.23.5. Execution.....	174
B.23.6. Exceptions.....	176
B.24. and.....	177
B.24.1. Encoding.....	177
B.24.2. Synopsis.....	177
B.24.3. Access.....	177
B.24.4. Decode Variables.....	177
B.24.5. Execution.....	177
B.24.6. Exceptions.....	178
B.25. andi.....	179
B.25.1. Encoding.....	179
B.25.2. Synopsis.....	179
B.25.3. Access.....	179
B.25.4. Decode Variables.....	179
B.25.5. Execution.....	179
B.25.6. Exceptions.....	180
B.26. andn.....	181
B.26.1. Encoding.....	181
B.26.2. Synopsis.....	181
B.26.3. Access.....	181
B.26.4. Decode Variables.....	181
B.26.5. Execution.....	181
B.26.6. Exceptions.....	182
B.27. auipc.....	183
B.27.1. Encoding.....	183
B.27.2. Synopsis.....	183
B.27.3. Access.....	183
B.27.4. Decode Variables.....	183
B.27.5. Execution.....	183
B.27.6. Exceptions.....	184
B.28. bclr.....	185
B.28.1. Encoding.....	185
B.28.2. Synopsis.....	185
B.28.3. Access.....	185
B.28.4. Decode Variables.....	185

B.28.5. Execution.....	185
B.28.6. Exceptions	186
B.29. bclri	187
B.29.1. Encoding.....	187
B.29.2. Synopsis	187
B.29.3. Access.....	187
B.29.4. Decode Variables.....	187
B.29.5. Execution	188
B.29.6. Exceptions	189
B.30. beq	190
B.30.1. Encoding	190
B.30.2. Synopsis.....	190
B.30.3. Access	190
B.30.4. Decode Variables	190
B.30.5. Execution	190
B.30.6. Exceptions.....	191
B.31. bext.....	192
B.31.1. Encoding.....	192
B.31.2. Synopsis.....	192
B.31.3. Access	192
B.31.4. Decode Variables	192
B.31.5. Execution	192
B.31.6. Exceptions	193
B.32. bexti.....	194
B.32.1. Encoding.....	194
B.32.2. Synopsis	194
B.32.3. Access.....	194
B.32.4. Decode Variables.....	194
B.32.5. Execution	195
B.32.6. Exceptions	196
B.33. bge.....	197
B.33.1. Encoding.....	197
B.33.2. Synopsis	197
B.33.3. Access.....	197
B.33.4. Decode Variables.....	197
B.33.5. Execution	197
B.33.6. Exceptions	198
B.34. bgeu.....	199
B.34.1. Encoding	199
B.34.2. Synopsis	199
B.34.3. Access	199
B.34.4. Decode Variables	199
B.34.5. Execution	199

B.34.6. Exceptions	200
B.35. binv	201
B.35.1. Encoding	201
B.35.2. Synopsis	201
B.35.3. Access	201
B.35.4. Decode Variables	201
B.35.5. Execution	201
B.35.6. Exceptions	202
B.36. binvi	203
B.36.1. Encoding	203
B.36.2. Synopsis	203
B.36.3. Access	203
B.36.4. Decode Variables	203
B.36.5. Execution	204
B.36.6. Exceptions	205
B.37. blt	206
B.37.1. Encoding	206
B.37.2. Synopsis	206
B.37.3. Access	206
B.37.4. Decode Variables	206
B.37.5. Execution	206
B.37.6. Exceptions	207
B.38. bltu	208
B.38.1. Encoding	208
B.38.2. Synopsis	208
B.38.3. Access	208
B.38.4. Decode Variables	208
B.38.5. Execution	208
B.38.6. Exceptions	209
B.39. bne	210
B.39.1. Encoding	210
B.39.2. Synopsis	210
B.39.3. Access	210
B.39.4. Decode Variables	210
B.39.5. Execution	210
B.39.6. Exceptions	211
B.40. bset	212
B.40.1. Encoding	212
B.40.2. Synopsis	212
B.40.3. Access	212
B.40.4. Decode Variables	212
B.40.5. Execution	212
B.40.6. Exceptions	213

B.41. bseti	214
B.41.1. Encoding	214
B.41.2. Synopsis	214
B.41.3. Access	214
B.41.4. Decode Variables	214
B.41.5. Execution	215
B.41.6. Exceptions	216
B.42. cbo.clean	217
B.42.1. Encoding	217
B.42.2. Synopsis	217
B.42.3. Access	217
B.42.4. Decode Variables	218
B.42.5. Execution	218
B.42.6. Exceptions	219
B.43. cbo.flush	220
B.43.1. Encoding	220
B.43.2. Synopsis	220
B.43.3. Access	220
B.43.4. Decode Variables	221
B.43.5. Execution	221
B.43.6. Exceptions	222
B.44. cbo.inval	223
B.44.1. Encoding	223
B.44.2. Synopsis	223
B.44.3. Access	225
B.44.4. Decode Variables	225
B.44.5. Execution	225
B.44.6. Exceptions	226
B.45. cbo.zero	227
B.45.1. Encoding	227
B.45.2. Synopsis	227
B.45.3. Access	227
B.45.4. Decode Variables	228
B.45.5. Execution	228
B.45.6. Exceptions	229
B.46. clz	230
B.46.1. Encoding	230
B.46.2. Synopsis	230
B.46.3. Access	230
B.46.4. Decode Variables	230
B.46.5. Execution	230
B.46.6. Exceptions	231
B.47. clzw	232

B.47.1. Encoding	232
B.47.2. Synopsis	232
B.47.3. Access	232
B.47.4. Decode Variables	232
B.47.5. Execution	232
B.47.6. Exceptions	233
B.48. cpop	234
B.48.1. Encoding	234
B.48.2. Synopsis	234
B.48.3. Access	234
B.48.4. Decode Variables	234
B.48.5. Execution	234
B.48.6. Exceptions	236
B.49. cpopw	237
B.49.1. Encoding	237
B.49.2. Synopsis	237
B.49.3. Access	237
B.49.4. Decode Variables	237
B.49.5. Execution	237
B.49.6. Exceptions	239
B.50. ctz	240
B.50.1. Encoding	240
B.50.2. Synopsis	240
B.50.3. Access	240
B.50.4. Decode Variables	240
B.50.5. Execution	240
B.50.6. Exceptions	241
B.51. ctzw	242
B.51.1. Encoding	242
B.51.2. Synopsis	242
B.51.3. Access	242
B.51.4. Decode Variables	242
B.51.5. Execution	242
B.51.6. Exceptions	243
B.52. div	244
B.52.1. Encoding	244
B.52.2. Synopsis	244
B.52.3. Access	244
B.52.4. Decode Variables	244
B.52.5. Execution	244
B.52.6. Exceptions	246
B.53. divu	247
B.53.1. Encoding	247

B.53.2. Synopsis	247
B.53.3. Access.....	247
B.53.4. Decode Variables	247
B.53.5. Execution.....	247
B.53.6. Exceptions	248
B.54. divuw.....	249
B.54.1. Encoding.....	249
B.54.2. Synopsis.....	249
B.54.3. Access	249
B.54.4. Decode Variables	249
B.54.5. Execution	249
B.54.6. Exceptions	251
B.55. divw	252
B.55.1. Encoding.....	252
B.55.2. Synopsis	252
B.55.3. Access.....	252
B.55.4. Decode Variables.....	252
B.55.5. Execution.....	252
B.55.6. Exceptions	254
B.56. ebreak	255
B.56.1. Encoding.....	255
B.56.2. Synopsis	255
B.56.3. Access.....	255
B.56.4. Decode Variables	255
B.56.5. Execution.....	255
B.56.6. Exceptions.....	256
B.57. ecall	257
B.57.1. Encoding.....	257
B.57.2. Synopsis	257
B.57.3. Access.....	257
B.57.4. Decode Variables	257
B.57.5. Execution.....	257
B.57.6. Exceptions	259
B.58. fcvt.h.s.....	260
B.58.1. Encoding.....	260
B.58.2. Synopsis	260
B.58.3. Access	260
B.58.4. Decode Variables	260
B.58.5. Execution	260
B.58.6. Exceptions.....	262
B.59. fcvt.s.h.....	263
B.59.1. Encoding	263
B.59.2. Synopsis.....	263

B.59.3. Access.....	263
B.59.4. Decode Variables	263
B.59.5. Execution	263
B.59.6. Exceptions	265
B.60. fence.....	266
B.60.1. Encoding.....	266
B.60.2. Synopsis	266
B.60.3. Access	267
B.60.4. Decode Variables	267
B.60.5. Execution.....	268
B.60.6. Exceptions.....	270
B.61. fence.i	271
B.61.1. Encoding.....	271
B.61.2. Synopsis	271
B.61.3. Access.....	271
B.61.4. Decode Variables.....	271
B.61.5. Execution.....	271
B.61.6. Exceptions.....	272
B.62. flh	273
B.62.1. Encoding.....	273
B.62.2. Synopsis.....	273
B.62.3. Access	273
B.62.4. Decode Variables.....	273
B.62.5. Execution	273
B.62.6. Exceptions	274
B.63. fmv.h.x.....	275
B.63.1. Encoding.....	275
B.63.2. Synopsis.....	275
B.63.3. Access	275
B.63.4. Decode Variables.....	275
B.63.5. Execution	275
B.63.6. Exceptions.....	276
B.64. fmv.w.x	277
B.64.1. Encoding.....	277
B.64.2. Synopsis	277
B.64.3. Access.....	277
B.64.4. Decode Variables.....	277
B.64.5. Execution.....	277
B.64.6. Exceptions	278
B.65. fmv.x.h.....	279
B.65.1. Encoding.....	279
B.65.2. Synopsis	279
B.65.3. Access.....	279

B.65.4. Decode Variables.....	279
B.65.5. Execution.....	279
B.65.6. Exceptions.....	280
B.66. fsh.....	281
B.66.1. Encoding.....	281
B.66.2. Synopsis.....	281
B.66.3. Access.....	281
B.66.4. Decode Variables.....	281
B.66.5. Execution.....	281
B.66.6. Exceptions.....	282
B.67. hinval.gvma.....	283
B.67.1. Encoding.....	283
B.67.2. Synopsis.....	283
B.67.3. Access.....	283
B.67.4. Decode Variables.....	283
B.67.5. Execution.....	283
B.67.6. Exceptions.....	285
B.68. hinval.vvma.....	286
B.68.1. Encoding.....	286
B.68.2. Synopsis.....	286
B.68.3. Access.....	286
B.68.4. Decode Variables.....	286
B.68.5. Execution.....	286
B.68.6. Exceptions.....	288
B.69. jal.....	289
B.69.1. Encoding.....	289
B.69.2. Synopsis.....	289
B.69.3. Access.....	289
B.69.4. Decode Variables.....	289
B.69.5. Execution.....	289
B.69.6. Exceptions.....	290
B.70. jalr.....	291
B.70.1. Encoding.....	291
B.70.2. Synopsis.....	291
B.70.3. Access.....	291
B.70.4. Decode Variables.....	291
B.70.5. Execution.....	291
B.70.6. Exceptions.....	292
B.71. lb.....	293
B.71.1. Encoding.....	293
B.71.2. Synopsis.....	293
B.71.3. Access.....	293
B.71.4. Decode Variables.....	293

B.71.5. Execution	293
B.71.6. Exceptions	294
B.72. lbu	295
B.72.1. Encoding	295
B.72.2. Synopsis	295
B.72.3. Access	295
B.72.4. Decode Variables	295
B.72.5. Execution	295
B.72.6. Exceptions	296
B.73. ld	297
B.73.1. Encoding	297
B.73.2. Synopsis	297
B.73.3. Access	297
B.73.4. Decode Variables	297
B.73.5. Execution	297
B.73.6. Exceptions	298
B.74. lh	299
B.74.1. Encoding	299
B.74.2. Synopsis	299
B.74.3. Access	299
B.74.4. Decode Variables	299
B.74.5. Execution	299
B.74.6. Exceptions	300
B.75. lhu	301
B.75.1. Encoding	301
B.75.2. Synopsis	301
B.75.3. Access	301
B.75.4. Decode Variables	301
B.75.5. Execution	301
B.75.6. Exceptions	302
B.76. lr.d	303
B.76.1. Encoding	303
B.76.2. Synopsis	303
B.76.3. Access	304
B.76.4. Decode Variables	304
B.76.5. Execution	304
B.76.6. Exceptions	305
B.77. lr.w	306
B.77.1. Encoding	306
B.77.2. Synopsis	306
B.77.3. Access	307
B.77.4. Decode Variables	307
B.77.5. Execution	307

B.77.6. Exceptions.....	309
B.78. lui.....	310
B.78.1. Encoding.....	310
B.78.2. Synopsis.....	310
B.78.3. Access.....	310
B.78.4. Decode Variables.....	310
B.78.5. Execution.....	310
B.78.6. Exceptions.....	311
B.79. lw.....	312
B.79.1. Encoding.....	312
B.79.2. Synopsis.....	312
B.79.3. Access.....	312
B.79.4. Decode Variables.....	312
B.79.5. Execution.....	312
B.79.6. Exceptions.....	313
B.80. lwu.....	314
B.80.1. Encoding.....	314
B.80.2. Synopsis.....	314
B.80.3. Access.....	314
B.80.4. Decode Variables.....	314
B.80.5. Execution.....	314
B.80.6. Exceptions.....	315
B.81. max.....	316
B.81.1. Encoding.....	316
B.81.2. Synopsis.....	316
B.81.3. Access.....	316
B.81.4. Decode Variables.....	316
B.81.5. Execution.....	316
B.81.6. Exceptions.....	316
B.82. maxu.....	317
B.82.1. Encoding.....	317
B.82.2. Synopsis.....	317
B.82.3. Access.....	317
B.82.4. Decode Variables.....	317
B.82.5. Execution.....	317
B.82.6. Exceptions.....	318
B.83. min.....	319
B.83.1. Encoding.....	319
B.83.2. Synopsis.....	319
B.83.3. Access.....	319
B.83.4. Decode Variables.....	319
B.83.5. Execution.....	319
B.83.6. Exceptions.....	320

B.84. minu.....	321
B.84.1. Encoding.....	321
B.84.2. Synopsis	321
B.84.3. Access.....	321
B.84.4. Decode Variables.....	321
B.84.5. Execution.....	321
B.84.6. Exceptions	322
B.85. mret.....	323
B.85.1. Encoding.....	323
B.85.2. Synopsis	323
B.85.3. Access.....	323
B.85.4. Decode Variables.....	323
B.85.5. Execution.....	323
B.85.6. Exceptions.....	323
B.86. mul.....	324
B.86.1. Encoding.....	324
B.86.2. Synopsis	324
B.86.3. Access.....	324
B.86.4. Decode Variables.....	324
B.86.5. Execution.....	324
B.86.6. Exceptions.....	325
B.87. mulh.....	326
B.87.1. Encoding.....	326
B.87.2. Synopsis	326
B.87.3. Access.....	326
B.87.4. Decode Variables.....	326
B.87.5. Execution.....	326
B.87.6. Exceptions	328
B.88. mulhsu.....	329
B.88.1. Encoding.....	329
B.88.2. Synopsis.....	329
B.88.3. Access.....	329
B.88.4. Decode Variables	329
B.88.5. Execution.....	329
B.88.6. Exceptions	331
B.89. mulhu.....	332
B.89.1. Encoding	332
B.89.2. Synopsis.....	332
B.89.3. Access.....	332
B.89.4. Decode Variables	332
B.89.5. Execution.....	332
B.89.6. Exceptions.....	334
B.90. mulw	335

B.90.1. Encoding	335
B.90.2. Synopsis	335
B.90.3. Access	335
B.90.4. Decode Variables	335
B.90.5. Execution	335
B.90.6. Exceptions	337
B.91. or	338
B.91.1. Encoding	338
B.91.2. Synopsis	338
B.91.3. Access	338
B.91.4. Decode Variables	338
B.91.5. Execution	338
B.91.6. Exceptions	339
B.92. orc.b	340
B.92.1. Encoding	340
B.92.2. Synopsis	340
B.92.3. Access	340
B.92.4. Decode Variables	340
B.92.5. Execution	340
B.92.6. Exceptions	340
B.93. ori	341
B.93.1. Encoding	341
B.93.2. Synopsis	341
B.93.3. Access	341
B.93.4. Decode Variables	341
B.93.5. Execution	341
B.93.6. Exceptions	343
B.94. orn	344
B.94.1. Encoding	344
B.94.2. Synopsis	344
B.94.3. Access	344
B.94.4. Decode Variables	344
B.94.5. Execution	344
B.94.6. Exceptions	345
B.95. rem	346
B.95.1. Encoding	346
B.95.2. Synopsis	346
B.95.3. Access	346
B.95.4. Decode Variables	346
B.95.5. Execution	346
B.95.6. Exceptions	348
B.96. remu	349
B.96.1. Encoding	349

B.96.2. Synopsis.....	349
B.96.3. Access.....	349
B.96.4. Decode Variables	349
B.96.5. Execution	349
B.96.6. Exceptions.....	350
B.97. remuw.....	351
B.97.1. Encoding.....	351
B.97.2. Synopsis	351
B.97.3. Access.....	351
B.97.4. Decode Variables.....	351
B.97.5. Execution.....	351
B.97.6. Exceptions	353
B.98. remw	354
B.98.1. Encoding.....	354
B.98.2. Synopsis	354
B.98.3. Access.....	354
B.98.4. Decode Variables.....	354
B.98.5. Execution.....	354
B.98.6. Exceptions	356
B.99. rev8.....	357
B.99.1. Encoding.....	357
B.99.2. Synopsis	357
B.99.3. Access.....	357
B.99.4. Decode Variables.....	357
B.99.5. Execution	358
B.99.6. Exceptions	359
B.100. rol.....	360
B.100.1. Encoding.....	360
B.100.2. Synopsis.....	360
B.100.3. Access	360
B.100.4. Decode Variables.....	360
B.100.5. Execution	360
B.100.6. Exceptions.....	361
B.101. rolw	362
B.101.1. Encoding.....	362
B.101.2. Synopsis.....	362
B.101.3. Access	362
B.101.4. Decode Variables.....	362
B.101.5. Execution	362
B.101.6. Exceptions.....	363
B.102. ror	364
B.102.1. Encoding	364
B.102.2. Synopsis.....	364

B.102.3. Access.....	364
B.102.4. Decode Variables	364
B.102.5. Execution.....	364
B.102.6. Exceptions	365
B.103. rori.....	366
B.103.1. Encoding.....	366
B.103.2. Synopsis.....	366
B.103.3. Access.....	366
B.103.4. Decode Variables.....	366
B.103.5. Execution.....	367
B.103.6. Exceptions	368
B.104. roriw	369
B.104.1. Encoding.....	369
B.104.2. Synopsis.....	369
B.104.3. Access	369
B.104.4. Decode Variables	369
B.104.5. Execution	369
B.104.6. Exceptions	369
B.105. rorw	370
B.105.1. Encoding.....	370
B.105.2. Synopsis.....	370
B.105.3. Access.....	370
B.105.4. Decode Variables.....	370
B.105.5. Execution	370
B.105.6. Exceptions.....	371
B.106. sb	372
B.106.1. Encoding.....	372
B.106.2. Synopsis.....	372
B.106.3. Access.....	372
B.106.4. Decode Variables.....	372
B.106.5. Execution.....	372
B.106.6. Exceptions.....	373
B.107. sc.d.....	374
B.107.1. Encoding.....	374
B.107.2. Synopsis.....	374
B.107.3. Access	376
B.107.4. Decode Variables.....	376
B.107.5. Execution	376
B.107.6. Exceptions	376
B.108. sc.w.....	377
B.108.1. Encoding.....	377
B.108.2. Synopsis.....	377
B.108.3. Access.....	379

B.108.4. Decode Variables	379
B.108.5. Execution.....	379
B.108.6. Exceptions	381
B.109. sd.....	382
B.109.1. Encoding.....	382
B.109.2. Synopsis.....	382
B.109.3. Access	382
B.109.4. Decode Variables	382
B.109.5. Execution	382
B.109.6. Exceptions	383
B.110. sext.b	384
B.110.1. Encoding.....	384
B.110.2. Synopsis.....	384
B.110.3. Access	384
B.110.4. Decode Variables	384
B.110.5. Execution	384
B.110.6. Exceptions	385
B.111. sext.h.....	386
B.111.1. Encoding.....	386
B.111.2. Synopsis.....	386
B.111.3. Access	386
B.111.4. Decode Variables	386
B.111.5. Execution	386
B.111.6. Exceptions.....	387
B.112. sfence.inval.ir	388
B.112.1. Encoding.....	388
B.112.2. Synopsis.....	388
B.112.3. Access.....	388
B.112.4. Decode Variables.....	388
B.112.5. Execution.....	388
B.112.6. Exceptions.....	389
B.113. sfence.vma	390
B.113.1. Encoding.....	390
B.113.2. Synopsis.....	390
B.113.3. Access.....	393
B.113.4. Decode Variables	393
B.113.5. Execution.....	393
B.113.6. Exceptions.....	394
B.114. sfence.w.inval	395
B.114.1. Encoding.....	395
B.114.2. Synopsis	395
B.114.3. Access.....	395
B.114.4. Decode Variables.....	395

B.114.5. Execution.....	395
B.114.6. Exceptions.....	396
B.115. sh	397
B.115.1. Encoding.....	397
B.115.2. Synopsis	397
B.115.3. Access	397
B.115.4. Decode Variables.....	397
B.115.5. Execution	397
B.115.6. Exceptions.....	398
B.116. sh1add	399
B.116.1. Encoding.....	399
B.116.2. Synopsis.....	399
B.116.3. Access	399
B.116.4. Decode Variables.....	399
B.116.5. Execution	399
B.116.6. Exceptions.....	400
B.117. sh1add.uw.....	401
B.117.1. Encoding.....	401
B.117.2. Synopsis.....	401
B.117.3. Access.....	401
B.117.4. Decode Variables.....	401
B.117.5. Execution.....	401
B.117.6. Exceptions.....	402
B.118. sh2add.....	403
B.118.1. Encoding.....	403
B.118.2. Synopsis.....	403
B.118.3. Access	403
B.118.4. Decode Variables	403
B.118.5. Execution	403
B.118.6. Exceptions.....	404
B.119. sh2add.uw.....	405
B.119.1. Encoding.....	405
B.119.2. Synopsis.....	405
B.119.3. Access	405
B.119.4. Decode Variables.....	405
B.119.5. Execution	405
B.119.6. Exceptions	406
B.120. sh3add.....	407
B.120.1. Encoding.....	407
B.120.2. Synopsis	407
B.120.3. Access.....	407
B.120.4. Decode Variables	407
B.120.5. Execution.....	407

B.120.6. Exceptions	408
B.121. sh3add.uw.....	409
B.121.1. Encoding	409
B.121.2. Synopsis	409
B.121.3. Access	409
B.121.4. Decode Variables	409
B.121.5. Execution	409
B.121.6. Exceptions	410
B.122. sinval.vma	411
B.122.1. Encoding.....	411
B.122.2. Synopsis	411
B.122.3. Access	411
B.122.4. Decode Variables.....	411
B.122.5. Execution	411
B.122.6. Exceptions	413
B.123. sll.....	414
B.123.1. Encoding	414
B.123.2. Synopsis	414
B.123.3. Access	414
B.123.4. Decode Variables	414
B.123.5. Execution	414
B.123.6. Exceptions	415
B.124. slli	416
B.124.1. Encoding.....	416
B.124.2. Synopsis	416
B.124.3. Access	416
B.124.4. Decode Variables	416
B.124.5. Execution	416
B.124.6. Exceptions	418
B.125. slli.uw.....	419
B.125.1. Encoding.....	419
B.125.2. Synopsis	419
B.125.3. Access	419
B.125.4. Decode Variables.....	419
B.125.5. Execution	419
B.125.6. Exceptions	420
B.126. slliw.....	421
B.126.1. Encoding.....	421
B.126.2. Synopsis	421
B.126.3. Access	421
B.126.4. Decode Variables.....	421
B.126.5. Execution	421
B.126.6. Exceptions	422

B.127. sllw	423
B.127.1. Encoding	423
B.127.2. Synopsis	423
B.127.3. Access	423
B.127.4. Decode Variables	423
B.127.5. Execution	423
B.127.6. Exceptions	424
B.128. slt	425
B.128.1. Encoding	425
B.128.2. Synopsis	425
B.128.3. Access	425
B.128.4. Decode Variables	425
B.128.5. Execution	425
B.128.6. Exceptions	426
B.129. slti	427
B.129.1. Encoding	427
B.129.2. Synopsis	427
B.129.3. Access	427
B.129.4. Decode Variables	427
B.129.5. Execution	427
B.129.6. Exceptions	428
B.130. sltiu	429
B.130.1. Encoding	429
B.130.2. Synopsis	429
B.130.3. Access	429
B.130.4. Decode Variables	429
B.130.5. Execution	429
B.130.6. Exceptions	430
B.131. sltu	431
B.131.1. Encoding	431
B.131.2. Synopsis	431
B.131.3. Access	431
B.131.4. Decode Variables	431
B.131.5. Execution	431
B.131.6. Exceptions	432
B.132. sra	433
B.132.1. Encoding	433
B.132.2. Synopsis	433
B.132.3. Access	433
B.132.4. Decode Variables	433
B.132.5. Execution	433
B.132.6. Exceptions	434
B.133. srai	435

B.133.1. Encoding.....	435
B.133.2. Synopsis.....	435
B.133.3. Access	435
B.133.4. Decode Variables	435
B.133.5. Execution	436
B.133.6. Exceptions.....	437
B.134. sraiw.....	438
B.134.1. Encoding	438
B.134.2. Synopsis.....	438
B.134.3. Access	438
B.134.4. Decode Variables	438
B.134.5. Execution	438
B.134.6. Exceptions	439
B.135. saw.....	440
B.135.1. Encoding.....	440
B.135.2. Synopsis	440
B.135.3. Access.....	440
B.135.4. Decode Variables.....	440
B.135.5. Execution	440
B.135.6. Exceptions	441
B.136. sret.....	442
B.136.1. Encoding.....	442
B.136.2. Synopsis.....	442
B.136.3. Access	442
B.136.4. Decode Variables	443
B.136.5. Execution	443
B.136.6. Exceptions	445
B.137. srl.....	446
B.137.1. Encoding.....	446
B.137.2. Synopsis	446
B.137.3. Access.....	446
B.137.4. Decode Variables.....	446
B.137.5. Execution.....	446
B.137.6. Exceptions	447
B.138. srli.....	448
B.138.1. Encoding.....	448
B.138.2. Synopsis.....	448
B.138.3. Access	448
B.138.4. Decode Variables	448
B.138.5. Execution	448
B.138.6. Exceptions.....	450
B.139. srliw.....	451
B.139.1. Encoding.....	451

B.139.2. Synopsis	451
B.139.3. Access	451
B.139.4. Decode Variables	451
B.139.5. Execution	451
B.139.6. Exceptions	452
B.140. srlw	453
B.140.1. Encoding	453
B.140.2. Synopsis	453
B.140.3. Access	453
B.140.4. Decode Variables	453
B.140.5. Execution	453
B.140.6. Exceptions	454
B.141. sub	455
B.141.1. Encoding	455
B.141.2. Synopsis	455
B.141.3. Access	455
B.141.4. Decode Variables	455
B.141.5. Execution	455
B.141.6. Exceptions	456
B.142. subw	457
B.142.1. Encoding	457
B.142.2. Synopsis	457
B.142.3. Access	457
B.142.4. Decode Variables	457
B.142.5. Execution	457
B.142.6. Exceptions	458
B.143. sw	459
B.143.1. Encoding	459
B.143.2. Synopsis	459
B.143.3. Access	459
B.143.4. Decode Variables	459
B.143.5. Execution	459
B.143.6. Exceptions	460
B.144. wfi	461
B.144.1. Encoding	461
B.144.2. Synopsis	461
B.144.3. Access	461
B.144.4. Decode Variables	462
B.144.5. Execution	462
B.144.6. Exceptions	463
B.145. xnor	464
B.145.1. Encoding	464
B.145.2. Synopsis	464

B.145.3. Access.....	464
B.145.4. Decode Variables	464
B.145.5. Execution.....	464
B.145.6. Exceptions	465
B.146. xor	466
B.146.1. Encoding.....	466
B.146.2. Synopsis	466
B.146.3. Access.....	466
B.146.4. Decode Variables.....	466
B.146.5. Execution	466
B.146.6. Exceptions.....	467
B.147. xori.....	468
B.147.1. Encoding.....	468
B.147.2. Synopsis	468
B.147.3. Access.....	468
B.147.4. Decode Variables	468
B.147.5. Execution.....	468
B.147.6. Exceptions.....	469
B.148. zext.h	470
B.148.1. Encoding.....	470
B.148.2. Synopsis	470
B.148.3. Access	470
B.148.4. Decode Variables	470
B.148.5. Execution.....	471
B.148.6. Exceptions.....	471
Appendix C: CSR Specifications	472
C.1. cycle	473
C.1.1. Attributes.....	473
C.1.2. Format	473
C.2. cycleh	474
C.2.1. Attributes	474
C.2.2. Format.....	474
C.3. fcsr	475
C.3.1. Attributes.....	476
C.3.2. Format	476
C.4. hcounteren	477
C.4.1. Attributes	477
C.4.2. Format	477
C.5. hedeleg	478
C.5.1. Attributes.....	478
C.5.2. Format	478
C.6. hedeleg	479
C.6.1. Attributes	479

C.6.2. Format	479
C.7. hgap	480
C.7.1. Attributes	481
C.7.2. Format	481
C.8. hpmcounter10	482
C.8.1. Attributes	482
C.8.2. Format	483
C.9. hpmcounter10h	484
C.9.1. Attributes	484
C.9.2. Format	484
C.10. hpmcounter11	485
C.10.1. Attributes	485
C.10.2. Format	486
C.11. hpmcounter11h	487
C.11.1. Attributes	487
C.11.2. Format	487
C.12. hpmcounter12	488
C.12.1. Attributes	488
C.12.2. Format	489
C.13. hpmcounter12h	490
C.13.1. Attributes	490
C.13.2. Format	490
C.14. hpmcounter13	491
C.14.1. Attributes	491
C.14.2. Format	492
C.15. hpmcounter13h	493
C.15.1. Attributes	493
C.15.2. Format	493
C.16. hpmcounter14	494
C.16.1. Attributes	494
C.16.2. Format	495
C.17. hpmcounter14h	496
C.17.1. Attributes	496
C.17.2. Format	496
C.18. hpmcounter15	497
C.18.1. Attributes	497
C.18.2. Format	498
C.19. hpmcounter15h	499
C.19.1. Attributes	499
C.19.2. Format	499
C.20. hpmcounter16	500
C.20.1. Attributes	500
C.20.2. Format	501

C.21. hpmcounter16h	502
C.21.1. Attributes	502
C.21.2. Format	502
C.22. hpmcounter17	503
C.22.1. Attributes	503
C.22.2. Format	504
C.23. hpmcounter17h	505
C.23.1. Attributes	505
C.23.2. Format	505
C.24. hpmcounter18	506
C.24.1. Attributes	506
C.24.2. Format	507
C.25. hpmcounter18h	508
C.25.1. Attributes	508
C.25.2. Format	508
C.26. hpmcounter19	509
C.26.1. Attributes	509
C.26.2. Format	510
C.27. hpmcounter19h	511
C.27.1. Attributes	511
C.27.2. Format	511
C.28. hpmcounter20	512
C.28.1. Attributes	512
C.28.2. Format	513
C.29. hpmcounter20h	514
C.29.1. Attributes	514
C.29.2. Format	514
C.30. hpmcounter21	515
C.30.1. Attributes	515
C.30.2. Format	516
C.31. hpmcounter21h	517
C.31.1. Attributes	517
C.31.2. Format	517
C.32. hpmcounter22	518
C.32.1. Attributes	518
C.32.2. Format	519
C.33. hpmcounter22h	520
C.33.1. Attributes	520
C.33.2. Format	520
C.34. hpmcounter23	521
C.34.1. Attributes	521
C.34.2. Format	522
C.35. hpmcounter23h	523

C.35.1. Attributes.....	523
C.35.2. Format	523
C.36. hpmcounter24.....	524
C.36.1. Attributes.....	524
C.36.2. Format	525
C.37. hpmcounter24h	526
C.37.1. Attributes	526
C.37.2. Format	526
C.38. hpmcounter25.....	527
C.38.1. Attributes	527
C.38.2. Format	528
C.39. hpmcounter25h	529
C.39.1. Attributes	529
C.39.2. Format	529
C.40. hpmcounter26	530
C.40.1. Attributes.....	530
C.40.2. Format.....	531
C.41. hpmcounter26h	532
C.41.1. Attributes.....	532
C.41.2. Format	532
C.42. hpmcounter27.....	533
C.42.1. Attributes.....	533
C.42.2. Format	534
C.43. hpmcounter27h	535
C.43.1. Attributes.....	535
C.43.2. Format	535
C.44. hpmcounter28.....	536
C.44.1. Attributes.....	536
C.44.2. Format	537
C.45. hpmcounter28h.....	538
C.45.1. Attributes.....	538
C.45.2. Format	538
C.46. hpmcounter29	539
C.46.1. Attributes	539
C.46.2. Format.....	540
C.47. hpmcounter29h	541
C.47.1. Attributes	541
C.47.2. Format.....	541
C.48. hpmcounter3	542
C.48.1. Attributes	542
C.48.2. Format.....	543
C.49. hpmcounter30	544
C.49.1. Attributes	544

C.49.2. Format.....	545
C.50. hpmcounter30h	546
C.50.1. Attributes	546
C.50.2. Format.....	546
C.51. hpmcounter31.....	547
C.51.1. Attributes	547
C.51.2. Format	548
C.52. hpmcounter31h	549
C.52.1. Attributes	549
C.52.2. Format	549
C.53. hpmcounter3h	550
C.53.1. Attributes	550
C.53.2. Format.....	550
C.54. hpmcounter4	551
C.54.1. Attributes.....	551
C.54.2. Format.....	552
C.55. hpmcounter4h	553
C.55.1. Attributes.....	553
C.55.2. Format	553
C.56. hpmcounter5	554
C.56.1. Attributes.....	554
C.56.2. Format	555
C.57. hpmcounter5h	556
C.57.1. Attributes.....	556
C.57.2. Format	556
C.58. hpmcounter6	557
C.58.1. Attributes.....	557
C.58.2. Format	558
C.59. hpmcounter6h	559
C.59.1. Attributes	559
C.59.2. Format.....	559
C.60. hpmcounter7	560
C.60.1. Attributes.....	560
C.60.2. Format	561
C.61. hpmcounter7h	562
C.61.1. Attributes	562
C.61.2. Format.....	562
C.62. hpmcounter8	563
C.62.1. Attributes	563
C.62.2. Format	564
C.63. hpmcounter8h.....	565
C.63.1. Attributes	565
C.63.2. Format.....	565

C.64. hpmcounter9	566
C.64.1. Attributes	566
C.64.2. Format	567
C.65. hpmcounter9h	568
C.65.1. Attributes	568
C.65.2. Format	568
C.66. hstatus	569
C.66.1. Attributes	569
C.66.2. Format	569
C.67. htimedelta	570
C.67.1. Attributes	570
C.67.2. Format	570
C.68. htimedeltah	571
C.68.1. Attributes	571
C.68.2. Format	571
C.69. instret	572
C.69.1. Attributes	572
C.69.2. Format	572
C.70. instreth	573
C.70.1. Attributes	573
C.70.2. Format	573
C.71. marchid	574
C.71.1. Attributes	574
C.71.2. Format	574
C.72. mcause	576
C.72.1. Attributes	576
C.72.2. Format	576
C.73. mconfigptr	577
C.73.1. Attributes	577
C.73.2. Format	577
C.74. mcounteren	579
C.74.1. Attributes	579
C.74.2. Format	580
C.75. mcountinhibit	581
C.75.1. Attributes	581
C.75.2. Format	581
C.76. mcycle	582
C.76.1. Attributes	582
C.76.2. Format	582
C.77. mcycleh	583
C.77.1. Attributes	583
C.77.2. Format	583
C.78. medeleg	584

C.78.1. Attributes	584
C.78.2. Format	584
C.79. medelegh	585
C.79.1. Attributes	585
C.79.2. Format	585
C.80. menvcfg	586
C.80.1. Attributes	587
C.80.2. Format	588
C.81. menvcfgh	589
C.81.1. Attributes	589
C.81.2. Format	589
C.82. mepc	590
C.82.1. Attributes	590
C.82.2. Format	590
C.83. mhartid	591
C.83.1. Attributes	591
C.83.2. Format	591
C.84. mhpmcounter10	592
C.84.1. Attributes	592
C.84.2. Format	592
C.85. mhpmcounter10h	593
C.85.1. Attributes	593
C.85.2. Format	593
C.86. mhpmcounter11	594
C.86.1. Attributes	594
C.86.2. Format	594
C.87. mhpmcounter11h	595
C.87.1. Attributes	595
C.87.2. Format	595
C.88. mhpmcounter12	596
C.88.1. Attributes	596
C.88.2. Format	596
C.89. mhpmcounter12h	597
C.89.1. Attributes	597
C.89.2. Format	597
C.90. mhpmcounter13	598
C.90.1. Attributes	598
C.90.2. Format	598
C.91. mhpmcounter13h	599
C.91.1. Attributes	599
C.91.2. Format	599
C.92. mhpmcounter14	600
C.92.1. Attributes	600

C.92.2. Format.....	600
C.93. mhpmcounter14h	601
C.93.1. Attributes	601
C.93.2. Format	601
C.94. mhpmcounter15	602
C.94.1. Attributes	602
C.94.2. Format	602
C.95. mhpmcounter15h	603
C.95.1. Attributes	603
C.95.2. Format	603
C.96. mhpmcounter16	604
C.96.1. Attributes	604
C.96.2. Format	604
C.97. mhpmcounter16h	605
C.97.1. Attributes	605
C.97.2. Format	605
C.98. mhpmcounter17	606
C.98.1. Attributes	606
C.98.2. Format	606
C.99. mhpmcounter17h	607
C.99.1. Attributes	607
C.99.2. Format	607
C.100. mhpmcounter18	608
C.100.1. Attributes	608
C.100.2. Format	608
C.101. mhpmcounter18h	609
C.101.1. Attributes	609
C.101.2. Format	609
C.102. mhpmcounter19	610
C.102.1. Attributes	610
C.102.2. Format	610
C.103. mhpmcounter19h	611
C.103.1. Attributes	611
C.103.2. Format	611
C.104. mhpmcounter20	612
C.104.1. Attributes	612
C.104.2. Format	612
C.105. mhpmcounter20h	613
C.105.1. Attributes	613
C.105.2. Format	613
C.106. mhpmcounter21	614
C.106.1. Attributes	614
C.106.2. Format	614

C.107. mhpmcounter21h	615
C.107.1. Attributes	615
C.107.2. Format	615
C.108. mhpmcounter22	616
C.108.1. Attributes	616
C.108.2. Format	616
C.109. mhpmcounter22h	617
C.109.1. Attributes	617
C.109.2. Format	617
C.110. mhpmcounter23	618
C.110.1. Attributes	618
C.110.2. Format	618
C.111. mhpmcounter23h	619
C.111.1. Attributes	619
C.111.2. Format	619
C.112. mhpmcounter24	620
C.112.1. Attributes	620
C.112.2. Format	620
C.113. mhpmcounter24h	621
C.113.1. Attributes	621
C.113.2. Format	621
C.114. mhpmcounter25	622
C.114.1. Attributes	622
C.114.2. Format	622
C.115. mhpmcounter25h	623
C.115.1. Attributes	623
C.115.2. Format	623
C.116. mhpmcounter26	624
C.116.1. Attributes	624
C.116.2. Format	624
C.117. mhpmcounter26h	625
C.117.1. Attributes	625
C.117.2. Format	625
C.118. mhpmcounter27	626
C.118.1. Attributes	626
C.118.2. Format	626
C.119. mhpmcounter27h	627
C.119.1. Attributes	627
C.119.2. Format	627
C.120. mhpmcounter28	628
C.120.1. Attributes	628
C.120.2. Format	628
C.121. mhpmcounter28h	629

C.121.1. Attributes	629
C.121.2. Format	629
C.122. mhpmmcounter29	630
C.122.1. Attributes	630
C.122.2. Format	630
C.123. mhpmmcounter29h	631
C.123.1. Attributes	631
C.123.2. Format	631
C.124. mhpmmcounter3	632
C.124.1. Attributes	632
C.124.2. Format	632
C.125. mhpmmcounter30	633
C.125.1. Attributes	633
C.125.2. Format	633
C.126. mhpmmcounter30h	634
C.126.1. Attributes	634
C.126.2. Format	634
C.127. mhpmmcounter31	635
C.127.1. Attributes	635
C.127.2. Format	635
C.128. mhpmmcounter31h	636
C.128.1. Attributes	636
C.128.2. Format	636
C.129. mhpmmcounter3h	637
C.129.1. Attributes	637
C.129.2. Format	637
C.130. mhpmmcounter4	638
C.130.1. Attributes	638
C.130.2. Format	638
C.131. mhpmmcounter4h	639
C.131.1. Attributes	639
C.131.2. Format	639
C.132. mhpmmcounter5	640
C.132.1. Attributes	640
C.132.2. Format	640
C.133. mhpmmcounter5h	641
C.133.1. Attributes	641
C.133.2. Format	641
C.134. mhpmmcounter6	642
C.134.1. Attributes	642
C.134.2. Format	642
C.135. mhpmmcounter6h	643
C.135.1. Attributes	643

C.135.2. Format.....	643
C.136. mhpcounter7	644
C.136.1. Attributes	644
C.136.2. Format.....	644
C.137. mhpcounter7h	645
C.137.1. Attributes	645
C.137.2. Format.....	645
C.138. mhpcounter8.....	646
C.138.1. Attributes.....	646
C.138.2. Format	646
C.139. mhpcounter8h	647
C.139.1. Attributes	647
C.139.2. Format.....	647
C.140. mhpcounter9	648
C.140.1. Attributes	648
C.140.2. Format	648
C.141. mhpcounter9h.....	649
C.141.1. Attributes.....	649
C.141.2. Format	649
C.142. mhpmevent10	650
C.142.1. Attributes	650
C.142.2. Format	650
C.143. mhpmevent10h	651
C.143.1. Attributes.....	651
C.143.2. Format	651
C.144. mhpmevent11.....	652
C.144.1. Attributes.....	652
C.144.2. Format.....	652
C.145. mhpmevent11h.....	653
C.145.1. Attributes.....	653
C.145.2. Format.....	653
C.146. mhpmevent12	654
C.146.1. Attributes.....	654
C.146.2. Format.....	654
C.147. mhpmevent12h	655
C.147.1. Attributes	655
C.147.2. Format.....	655
C.148. mhpmevent13.....	656
C.148.1. Attributes.....	656
C.148.2. Format	656
C.149. mhpmevent13h	657
C.149.1. Attributes.....	657
C.149.2. Format.....	657

C.150. mhpmevent14	658
C.150.1. Attributes	658
C.150.2. Format	658
C.151. mhpmevent14h	659
C.151.1. Attributes	659
C.151.2. Format	659
C.152. mhpmevent15	660
C.152.1. Attributes	660
C.152.2. Format	660
C.153. mhpmevent15h	661
C.153.1. Attributes	661
C.153.2. Format	661
C.154. mhpmevent16	662
C.154.1. Attributes	662
C.154.2. Format	662
C.155. mhpmevent16h	663
C.155.1. Attributes	663
C.155.2. Format	663
C.156. mhpmevent17	664
C.156.1. Attributes	664
C.156.2. Format	664
C.157. mhpmevent17h	665
C.157.1. Attributes	665
C.157.2. Format	665
C.158. mhpmevent18	666
C.158.1. Attributes	666
C.158.2. Format	666
C.159. mhpmevent18h	667
C.159.1. Attributes	667
C.159.2. Format	667
C.160. mhpmevent19	668
C.160.1. Attributes	668
C.160.2. Format	668
C.161. mhpmevent19h	669
C.161.1. Attributes	669
C.161.2. Format	669
C.162. mhpmevent20	670
C.162.1. Attributes	670
C.162.2. Format	670
C.163. mhpmevent20h	671
C.163.1. Attributes	671
C.163.2. Format	671
C.164. mhpmevent21	672

C.164.1. Attributes	672
C.164.2. Format	672
C.165. mhpmevent21h	673
C.165.1. Attributes	673
C.165.2. Format	673
C.166. mhpmevent22	674
C.166.1. Attributes	674
C.166.2. Format	674
C.167. mhpmevent22h	675
C.167.1. Attributes	675
C.167.2. Format	675
C.168. mhpmevent23	676
C.168.1. Attributes	676
C.168.2. Format	676
C.169. mhpmevent23h	677
C.169.1. Attributes	677
C.169.2. Format	677
C.170. mhpmevent24	678
C.170.1. Attributes	678
C.170.2. Format	678
C.171. mhpmevent24h	679
C.171.1. Attributes	679
C.171.2. Format	679
C.172. mhpmevent25	680
C.172.1. Attributes	680
C.172.2. Format	680
C.173. mhpmevent25h	681
C.173.1. Attributes	681
C.173.2. Format	681
C.174. mhpmevent26	682
C.174.1. Attributes	682
C.174.2. Format	682
C.175. mhpmevent26h	683
C.175.1. Attributes	683
C.175.2. Format	683
C.176. mhpmevent27	684
C.176.1. Attributes	684
C.176.2. Format	684
C.177. mhpmevent27h	685
C.177.1. Attributes	685
C.177.2. Format	685
C.178. mhpmevent28	686
C.178.1. Attributes	686

C.178.2. Format	686
C.179. mhpmevent28h	687
C.179.1. Attributes	687
C.179.2. Format	687
C.180. mhpmevent29	688
C.180.1. Attributes	688
C.180.2. Format	688
C.181. mhpmevent29h	689
C.181.1. Attributes	689
C.181.2. Format	689
C.182. mhpmevent3	690
C.182.1. Attributes	690
C.182.2. Format	690
C.183. mhpmevent30	691
C.183.1. Attributes	691
C.183.2. Format	691
C.184. mhpmevent30h	692
C.184.1. Attributes	692
C.184.2. Format	692
C.185. mhpmevent31	693
C.185.1. Attributes	693
C.185.2. Format	693
C.186. mhpmevent31h	694
C.186.1. Attributes	694
C.186.2. Format	694
C.187. mhpmevent3h	695
C.187.1. Attributes	695
C.187.2. Format	695
C.188. mhpmevent4	696
C.188.1. Attributes	696
C.188.2. Format	696
C.189. mhpmevent4h	697
C.189.1. Attributes	697
C.189.2. Format	697
C.190. mhpmevent5	698
C.190.1. Attributes	698
C.190.2. Format	698
C.191. mhpmevent5h	699
C.191.1. Attributes	699
C.191.2. Format	699
C.192. mhpmevent6	700
C.192.1. Attributes	700
C.192.2. Format	700

C.193. mhpmevent6h.....	701
C.193.1. Attributes	701
C.193.2. Format	701
C.194. mhpmevent7	702
C.194.1. Attributes	702
C.194.2. Format	702
C.195. mhpmevent7h.....	703
C.195.1. Attributes	703
C.195.2. Format	703
C.196. mhpmevent8	704
C.196.1. Attributes	704
C.196.2. Format	704
C.197. mhpmevent8h.....	705
C.197.1. Attributes	705
C.197.2. Format	705
C.198. mhpmevent9.....	706
C.198.1. Attributes	706
C.198.2. Format	706
C.199. mhpmevent9h	707
C.199.1. Attributes	707
C.199.2. Format	707
C.200. mideleg	708
C.200.1. Attributes	708
C.200.2. Format	709
C.201. mie	710
C.201.1. Attributes	712
C.201.2. Format	712
C.202. mimpid	713
C.202.1. Attributes	713
C.202.2. Format	713
C.203. minstret.....	714
C.203.1. Attributes	714
C.203.2. Format	714
C.204. minstreth.....	715
C.204.1. Attributes	715
C.204.2. Format	715
C.205. mip	716
C.205.1. Attributes	716
C.205.2. Format	716
C.206. misa	717
C.206.1. Attributes	717
C.206.2. Format	717
C.207. mscratch	718

C.207.1. Attributes	718
C.207.2. Format	718
C.208. msecfcg	719
C.208.1. Attributes	719
C.208.2. Format	719
C.209. mstatus	720
C.209.1. Attributes	720
C.209.2. Format	720
C.210. mstatush	721
C.210.1. Attributes	721
C.210.2. Format	721
C.211. mtval	722
C.211.1. Attributes	722
C.211.2. Format	722
C.212. mtvec	723
C.212.1. Attributes	723
C.212.2. Format	723
C.213. mvendorid	724
C.213.1. Attributes	724
C.213.2. Format	724
C.214. pmpaddr0	725
C.214.1. Attributes	725
C.214.2. Format	725
C.215. pmpaddr1	726
C.215.1. Attributes	726
C.215.2. Format	726
C.216. pmpaddr10	727
C.216.1. Attributes	727
C.216.2. Format	727
C.217. pmpaddr11	728
C.217.1. Attributes	728
C.217.2. Format	728
C.218. pmpaddr12	729
C.218.1. Attributes	729
C.218.2. Format	729
C.219. pmpaddr13	730
C.219.1. Attributes	730
C.219.2. Format	730
C.220. pmpaddr14	731
C.220.1. Attributes	731
C.220.2. Format	731
C.221. pmpaddr15	732
C.221.1. Attributes	732

C.221.2. Format	732
C.222. pmpaddr16	733
C.222.1. Attributes	733
C.222.2. Format	733
C.223. pmpaddr17	734
C.223.1. Attributes	734
C.223.2. Format	734
C.224. pmpaddr18	735
C.224.1. Attributes	735
C.224.2. Format	735
C.225. pmpaddr19	736
C.225.1. Attributes	736
C.225.2. Format	736
C.226. pmpaddr20	737
C.226.1. Attributes	737
C.226.2. Format	737
C.227. pmpaddr21	738
C.227.1. Attributes	738
C.227.2. Format	738
C.228. pmpaddr22	739
C.228.1. Attributes	739
C.228.2. Format	739
C.229. pmpaddr23	740
C.229.1. Attributes	740
C.229.2. Format	740
C.230. pmpaddr24	741
C.230.1. Attributes	741
C.230.2. Format	741
C.231. pmpaddr25	742
C.231.1. Attributes	742
C.231.2. Format	742
C.232. pmpaddr26	743
C.232.1. Attributes	743
C.232.2. Format	743
C.233. pmpaddr27	744
C.233.1. Attributes	744
C.233.2. Format	744
C.234. pmpaddr28	745
C.234.1. Attributes	745
C.234.2. Format	745
C.235. pmpaddr29	746
C.235.1. Attributes	746
C.235.2. Format	746

C.236. pmpaddr29	747
C.236.1. Attributes	747
C.236.2. Format	747
C.237. pmpaddr3	748
C.237.1. Attributes	748
C.237.2. Format	748
C.238. pmpaddr30	749
C.238.1. Attributes	749
C.238.2. Format	749
C.239. pmpaddr31	750
C.239.1. Attributes	750
C.239.2. Format	750
C.240. pmpaddr32	751
C.240.1. Attributes	751
C.240.2. Format	751
C.241. pmpaddr33	752
C.241.1. Attributes	752
C.241.2. Format	752
C.242. pmpaddr34	753
C.242.1. Attributes	753
C.242.2. Format	753
C.243. pmpaddr35	754
C.243.1. Attributes	754
C.243.2. Format	754
C.244. pmpaddr36	755
C.244.1. Attributes	755
C.244.2. Format	755
C.245. pmpaddr37	756
C.245.1. Attributes	756
C.245.2. Format	756
C.246. pmpaddr38	757
C.246.1. Attributes	757
C.246.2. Format	757
C.247. pmpaddr39	758
C.247.1. Attributes	758
C.247.2. Format	758
C.248. pmpaddr4	759
C.248.1. Attributes	759
C.248.2. Format	759
C.249. pmpaddr40	760
C.249.1. Attributes	760
C.249.2. Format	760
C.250. pmpaddr41	761

C.250.1. Attributes.....	761
C.250.2. Format.....	761
C.251. pmpaddr42.....	762
C.251.1. Attributes.....	762
C.251.2. Format.....	762
C.252. pmpaddr43.....	763
C.252.1. Attributes.....	763
C.252.2. Format.....	763
C.253. pmpaddr44.....	764
C.253.1. Attributes.....	764
C.253.2. Format.....	764
C.254. pmpaddr45.....	765
C.254.1. Attributes.....	765
C.254.2. Format.....	765
C.255. pmpaddr46.....	766
C.255.1. Attributes.....	766
C.255.2. Format.....	766
C.256. pmpaddr47.....	767
C.256.1. Attributes.....	767
C.256.2. Format.....	767
C.257. pmpaddr48.....	768
C.257.1. Attributes.....	768
C.257.2. Format.....	768
C.258. pmpaddr49.....	769
C.258.1. Attributes.....	769
C.258.2. Format.....	769
C.259. pmpaddr5.....	770
C.259.1. Attributes.....	770
C.259.2. Format.....	770
C.260. pmpaddr50.....	771
C.260.1. Attributes.....	771
C.260.2. Format.....	771
C.261. pmpaddr51.....	772
C.261.1. Attributes.....	772
C.261.2. Format.....	772
C.262. pmpaddr52.....	773
C.262.1. Attributes.....	773
C.262.2. Format.....	773
C.263. pmpaddr53.....	774
C.263.1. Attributes.....	774
C.263.2. Format.....	774
C.264. pmpaddr54.....	775
C.264.1. Attributes.....	775

C.264.2. Format.....	775
C.265. pmpaddr55.....	776
C.265.1. Attributes.....	776
C.265.2. Format.....	776
C.266. pmpaddr56.....	777
C.266.1. Attributes.....	777
C.266.2. Format.....	777
C.267. pmpaddr57.....	778
C.267.1. Attributes.....	778
C.267.2. Format.....	778
C.268. pmpaddr58.....	779
C.268.1. Attributes.....	779
C.268.2. Format.....	779
C.269. pmpaddr59.....	780
C.269.1. Attributes.....	780
C.269.2. Format.....	780
C.270. pmpaddr6.....	781
C.270.1. Attributes.....	781
C.270.2. Format.....	781
C.271. pmpaddr60.....	782
C.271.1. Attributes.....	782
C.271.2. Format.....	782
C.272. pmpaddr61.....	783
C.272.1. Attributes.....	783
C.272.2. Format.....	783
C.273. pmpaddr62.....	784
C.273.1. Attributes.....	784
C.273.2. Format.....	784
C.274. pmpaddr63.....	785
C.274.1. Attributes.....	785
C.274.2. Format.....	785
C.275. pmpaddr7.....	786
C.275.1. Attributes.....	786
C.275.2. Format.....	786
C.276. pmpaddr8.....	787
C.276.1. Attributes.....	787
C.276.2. Format.....	787
C.277. pmpaddr9.....	788
C.277.1. Attributes.....	788
C.277.2. Format.....	788
C.278. pmpcfg0.....	789
C.278.1. Attributes.....	789
C.278.2. Format.....	789

C.279. pmpcf1	790
C.279.1. Attributes	790
C.279.2. Format	790
C.280. pmpcf10	791
C.280.1. Attributes	791
C.280.2. Format	791
C.281. pmpcf11	792
C.281.1. Attributes	792
C.281.2. Format	792
C.282. pmpcf12	793
C.282.1. Attributes	793
C.282.2. Format	793
C.283. pmpcf13	794
C.283.1. Attributes	794
C.283.2. Format	794
C.284. pmpcf14	795
C.284.1. Attributes	795
C.284.2. Format	795
C.285. pmpcf15	796
C.285.1. Attributes	796
C.285.2. Format	796
C.286. pmpcf2	797
C.286.1. Attributes	797
C.286.2. Format	797
C.287. pmpcf3	798
C.287.1. Attributes	798
C.287.2. Format	798
C.288. pmpcf4	799
C.288.1. Attributes	799
C.288.2. Format	799
C.289. pmpcf5	800
C.289.1. Attributes	800
C.289.2. Format	800
C.290. pmpcf6	801
C.290.1. Attributes	801
C.290.2. Format	801
C.291. pmpcf7	802
C.291.1. Attributes	802
C.291.2. Format	802
C.292. pmpcf8	803
C.292.1. Attributes	803
C.292.2. Format	803
C.293. pmpcf9	804

C.293.1. Attributes	804
C.293.2. Format	804
C.294. satp	805
C.294.1. Attributes	805
C.294.2. Format	805
C.295. scause	806
C.295.1. Attributes	806
C.295.2. Format	806
C.296. scounteren	807
C.296.1. Attributes	807
C.296.2. Format	807
C.297. senvcfg	808
C.297.1. Attributes	808
C.297.2. Format	808
C.298. sepc	809
C.298.1. Attributes	809
C.298.2. Format	809
C.299. sip	810
C.299.1. Attributes	810
C.299.2. Format	810
C.300. sscratch	811
C.300.1. Attributes	811
C.300.2. Format	811
C.301. sstatus	812
C.301.1. Attributes	812
C.301.2. Format	812
C.302. stval	813
C.302.1. Attributes	813
C.302.2. Format	813
C.303. stvec	814
C.303.1. Attributes	814
C.303.2. Format	814
C.304. time	815
C.304.1. Attributes	815
C.304.2. Format	815
C.305. vscause	816
C.305.1. Attributes	816
C.305.2. Format	816
C.306. vsepc	817
C.306.1. Attributes	817
C.306.2. Format	817
C.307. vsstatus	818
C.307.1. Attributes	818

C.307.2. Format	818
C.308. vstval	819
C.308.1. Attributes	819
C.308.2. Format	819
C.309. vstvec	820
C.309.1. Attributes	820
C.309.2. Format	820

DRAFT

Licensing and Acknowledgements

TODO: revmark

DRAFT

Copyright and license information

This document is released under the [Creative Commons Attribution 4.0 International License](#).

Copyright 2024 by RISC-V International.

DRAFT

Acknowledgements

Contributors to the RVA20S64 Profile (in alphabetical order) include:

- Krste Asanovic <krste@sifive.com> (SiFive)

Contributors to the RVA20U64 Profile (in alphabetical order) include:

- Krste Asanovic <krste@sifive.com> (SiFive)

Contributors to the RVA22S64 Profile (in alphabetical order) include:

- Krste Asanovic <krste@sifive.com> (SiFive)

Contributors to the RVA22U64 Profile (in alphabetical order) include:

- Krste Asanovic <krste@sifive.com> (SiFive)

We express our gratitude to everyone that contributed to, reviewed or improved this specification through their comments and questions.

DRAFT

Chapter 1. RISC-V Profiles

RISC-V was designed to provide a highly modular and extensible instruction set, and includes a large and growing set of standard extensions. In addition, users may add their own custom extensions. This flexibility can be used to highly optimize a specialized design by including only the exact set of ISA features required for an application, but the same flexibility also leads to a combinatorial explosion in possible ISA choices. Profiles specify a much smaller common set of ISA choices that capture the most value for most users, and which thereby enable the software community to focus resources on building a rich software ecosystem with application and operating system portability across different implementations.



Another pragmatic concern is the long and unwieldy ISA strings required to encode common sets of extensions, which will continue to grow as new extensions are defined.

Each profile is built on a standard base ISA plus a set of mandatory ISA extensions, and provides a small set of standard ISA options to extend the mandatory components. Profiles provide a convenient shorthand for describing the ISA portions of hardware and software platforms, and also guide the development of common software toolchains shared by different platforms that use the same profile. The intent is that the software ecosystem focus on supporting the profiles' mandatory base and standard options, instead of attempting to support every possible combination of individual extensions. Similarly, hardware vendors should aim to structure their offerings around standard profiles to increase the likelihood their designs will have mainstream software support.



Profiles are not intended to prohibit the use of combinations of individual ISA extensions or the addition of custom extensions, which can continue to be used for more specialized applications albeit without the expectation of widespread software support or portability between hardware platforms.



As RISC-V evolves over time, the set of ISA features will grow, and new platforms will be added that may need different profiles. To manage this evolution, RISC-V is adopting a model of regular annual releases of new ISA profiles, following an ISA roadmap managed by the RISC-V Technical Steering Committee. The architecture profiles will also be used for branding and to advertise compatibility with the RISC-V standard.

1.1. Profiles versus Platforms

Profiles only describe ISA features, not a complete execution environment.

A *software platform* is a specification for an execution environment, in which software targeted for that software platform can run.

A *hardware platform* is a specification for a hardware system (which can be viewed as a physical realization of an execution environment).

Both software and hardware platforms include specifications for many features beyond details of the ISA used by RISC-V, such as in the platform (e.g., boot process, calling convention, behavior of environment calls, discovery mechanism, presence of certain memory-mapped hardware devices, etc.). Architecture profiles factor out ISA-specific definitions from platform definitions to allow ISA profiles to be reused across different platforms, and to be used by tools (e.g., compilers) that are common across many different platforms.

A platform can add additional constraints on top of those in a profile. For example, mandating an extension that is a standard option in the underlying profile, or constraining some implementation-

specific parameter in the profile to lie within a certain range.

A platform cannot remove mandates or reduce other requirements in a profile.



A new profile should be proposed if existing profiles do not match the needs of a new platform.

1.2. Components of a Profile

1.2.1. Profile Family

Every profile is a member of a *profile family*. A profile family is a set of profiles that share the same base ISA but which vary in highest-supported privilege mode. The initial two types of family are:

- generic unprivileged instructions (I)
- application processors running rich operating systems (A)



More profile families may be added over time.

A profile family may be updated no more than annually, and the release calendar year is treated as part of the profile family name.

Each profile family is described in more detail below.

1.2.2. Profile Privilege Mode

RISC-V has a layered architecture supporting multiple privilege modes, and most RISC-V platforms support more than one privilege mode. Software is usually written assuming a particular privilege mode during execution. For example, application code is written assuming it will be run in user mode, and kernel code is written assuming it will be run in supervisor mode.



Software can be run in a mode different than the one for which it was written. For example, privileged code using privileged ISA features can be run in a user-mode execution environment, but will then cause traps into the enclosing execution environment when privileged instructions are executed. This behavior might be exploited, for example, to emulate a privileged execution environment using a user-mode execution environment.

The profile for a privilege mode describes the ISA features for an execution environment that has the eponymous privilege mode as the most-privileged mode available, but also includes all supported lower-privilege modes. In general, available instructions vary by privilege mode, and the behavior of RISC-V instructions can depend on the current privilege mode. For example, an S-mode profile includes U-mode as well as S-mode and describes the behavior of instructions when running in different modes in an S-mode execution environment, such as how an [ecall](#) instruction in U-mode causes a contained trap into an S-mode handler whereas an [ecall](#) in S-mode causes a requested trap out to the execution environment.

A profile may specify that certain conditions will cause a requested trap (such as an [ecall](#) made in the highest-supported privilege mode) or fatal trap to the enclosing execution environment. The profile does not specify the behavior of the enclosing execution environment in handling requested or fatal traps.



In particular, a profile does not specify the set of ECALLs available in the outer execution environment. This should be documented in the appropriate binary interface to the outer execution environment (e.g., Linux user ABI, or RISC-V SEE).



In general, a profile can be implemented by an execution environment using any hardware or software technique that provides compatible functionality, including pure software emulation.

A profile does not specify any invisible traps.



In particular, a profile does not constrain how invisible traps to a more-privileged mode can be used to emulate profile features.

A more-privileged profile can always support running software to implement a less-privileged profile from the same profile family. For example, a platform supporting the S-mode profile can run a supervisor-mode operating system that provides user-mode execution environments supporting the U-mode profile.



Instructions in a U-mode profile, which are all executed in user mode, have potentially different behaviors than instructions executed in user mode in an S-mode profile. For this reason, a U-mode profile cannot be considered a subset of an S-mode profile.

1.2.3. Profile ISA Features

An architecture profile has a mandatory ratified base instruction set (RV32I or RV64I for the current profiles). The profile also includes ratified ISA extensions placed into two categories:

1. Mandatory
2. Optional

As the name implies, *Mandatory ISA extensions* are a required part of the profile. Implementations of the profile must provide these. The combination of the profile base ISA plus the mandatory ISA extensions are termed the profile *mandates*, and software using the profile can assume these always exist.

The *Optional* category (also known as *options*) contains extensions that may be added as options, and which are expected to be generally supported as options by the software ecosystem for this profile.



The level of "support" for an Optional extension will likely vary greatly among different software components supporting a profile. Users would expect that software claiming compatibility with a profile would make use of any available supported options, but as a bare minimum software should not report errors or warnings when supported options are present in a system.

An optional extension may comprise many individually named and ratified extensions but a profile option requires all constituent extensions are present. In particular, unless explicitly listed as a profile option, individual extensions are not by themselves a profile option even when required as part of a profile option. For example, the Zbkb extension is not by itself a profile option even though it is a required component of the Zkn option.



Profile optional extensions are intended to capture the granularity at which the broad software ecosystem is expected to cope with combinations of extensions.

All components of a ratified profile must themselves have been ratified.

Platforms may provide a discovery mechanism to determine what optional extensions are present.

Extensions that are not explicitly listed in the mandatory or optional categories are termed *non-profile* extensions, and are not considered parts of the profile. Some non-profile extensions can be added to an implementation without conflicting with the mandatory or optional components of a profile. In this case, the implementation is still compatible with the profile even though additional non-profile extensions are present. Other non-profile extensions added to an implementation might alter or conflict with the behavior of the mandatory or optional extensions in a profile, in which case the implementation would not be compatible with the profile.



Extensions that are released after a given profile is released are by definition non-profile extensions. For example, mandatory or optional profile extensions for a new profile might be prototyped as non-profile extensions on an earlier profile.

DRAFT

Chapter 2. Profiles in the rva family

The following profiles are defined in this family:

Name

RVA20S64

State

Name

RVA20U64

State

ratified

Name

RVA22S64

State

Name

RVA22U64

State

DRAFT

Chapter 3. Family description

The RVA profiles are intended to be used for 64-bit application processors running rich OS stacks. Only user-mode and supervisor-mode profiles are specified in this family.



There is no machine-mode profile currently defined for application processor families. A machine-mode profile for application processors would only be used in specifying platforms for portable machine-mode software. Given the relatively low volume of portable M-mode software in this domain, the wide variety of potential M-mode code, and the very specific needs of each type of M-mode software, we are not specifying individual M-mode ISA requirements in the A-family profiles.



Only XLEN=64 application processor profiles are currently defined. It would be possible to also define very similar XLEN=32 variants.

3.1. Extension summary

The RVA Profile Family references 40 extensions.

Table 1. Status

Extension	RVA20S64	RVA20U64	RVA22S64	RVA22U64
A	Mandatory, = 2.1	Mandatory, = 2.1	Mandatory, = 2.1	Mandatory, = 2.1
C	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2
D	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2
F	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2	Mandatory, = 2.2
H	-	-	Optional, ~> 1.0	-
I	Mandatory, ~> 2.1	Mandatory, ~> 2.1	Mandatory, ~> 2.1	Mandatory, ~> 2.1
M	Mandatory, = 2.0	Mandatory, = 2.0	Mandatory, = 2.0	Mandatory, = 2.0
S	Mandatory, = 1.11	-	Mandatory, = 1.12	-
Ssccptr	Mandatory, = 1.0	-	-	-
Sscofpmf	-	-	Optional, ~> 1.0	-
Sscounterenw	-	-	Mandatory, = 1.0	-
Sstc	-	-	Optional, ~> 1.0	-
Sstvala	Mandatory, = 1.0	-	-	-
Sstvecd	Mandatory, = 1.0	-	-	-
Sv39	Mandatory, = 1.11	-	-	-
Sv48	Optional, = 1.11	-	-	-
Sv57	-	-	Optional, ~> 1.12	-
Svade	Mandatory, ~> 1.0	-	-	-
Svbare	Mandatory, = 1.0	-	-	-
Svinval	-	-	Mandatory, ~> 1.0	-
Svpbmt	-	-	Mandatory, ~> 1.0	-
U	Mandatory, ~> 2.0	Mandatory, ~> 2.0	Mandatory, ~> 2.0	Mandatory, ~> 2.0
Za128rs	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0

Zba	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zbb	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zbs	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zfhmin	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zic64b	-	-	Mandatory, = 1.0	Mandatory, = 1.0
Zicbom	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zicbop	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Zicboz	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0
Ziccamoa	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0
Ziccif	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0
Zicclsm	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0
Ziccrse	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0	Mandatory, = 1.0
Zicntr	Mandatory, = 2.0	Mandatory, = 2.0	Mandatory, = 2.0	Mandatory, = 2.0
Zifencei	Mandatory, = 2.0	-	-	-
Zihintpause	-	-	Mandatory, = 2.0	Mandatory, = 2.0
Zihpm	Optional, = 2.0	Optional, = 2.0	Mandatory, = 2.0	Mandatory, = 2.0
Zkt	-	-	Mandatory, ~> 1.0	Mandatory, ~> 1.0

Chapter 4. RVA20S64 Profile

The RVA20S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA20S64 is based on privileged architecture version 1.11.

4.1. Extensions

The RVA20S64 Profile has 21 mandatory extensions and 3 optional extensions.

4.1.1. Mandatory Extensions

- **I** Base integer ISA

Version ~> 2.1

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

Version = 2.2

- **D** Double-precision floating-point

Version = 2.2

- **F** Single-precision floating-point

Version = 2.2

- **M** Integer multiply and divide instructions

Version = 2.0

- **U** User-level privilege mode

Version ~> 2.0

- **Zicntr** Architectural performance counters

Version = 2.0

- **Ziccif** Main memory fetch requirement for RVA profiles

Version = 1.0

- **Ziccrse** Main memory reservability requirement for RVA profiles

Version = 1.0

- **Ziccamao** Main memory atomicity requirement for RVA profiles

Version = 1.0

- **Za128rs** Reservation set requirement for RVA profiles

Version = 1.0

- **Zicclsm** Main memory misaligned requirement for RVA profiles

Version = 1.0

- **S** Supervisor mode

Version = 1.11

- **Zifencei** Instruction fence

Version = 2.0



Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA20 application processors. A new instruction-cache coherence mechanism is under development which might be added as an option in the future.

- **Svbare** Bare virtual addressing

Version = 1.0



Svbare is a new extension name introduced with RVA20.

- **Sv39** 39-bit virtual address translation (3 level)

Version = 1.11

- **Svade** Exception on PTE A/D Bits

Version ~> 1.0



Svbare is a new extension name introduced with RVA20.

It is subsequently defined in more detail with the ratification of Svadu.

- **Ssccptr** Cacheable and coherent main memory page table reads

Version = 1.0



Ssccptr is a new extension name introduced with RVA20.

- **Sstvecd** Direct exception vectoring

Version = 1.0



Sstvecd is a new extension name introduced with RVA20.

- **Sstvala** **stval** requirements for RVA profiles

Version = 1.0



Sstvala is a new extension name introduced with RVA20.

4.1.2. Optional Extensions

- **Zihpm** Programmable hardware performance counters

Version= 2.0

- **Sv48** 48-bit virtual address translation (4 level)

Version= 1.11

- **Ssu64xl**

Version= 1.0



Ssu64xl is a new extension name introduced with RVA20.

DRAFT

Chapter 5. RVA20U64 Profile

The RVA20U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within the application processor family in terms of the amount of software that targets this profile.

5.1. Extensions

The RVA20U64 Profile has 13 mandatory extensions and 1 optional extensions.

5.1.1. Mandatory Extensions

- I Base integer ISA

Version ~> 2.1

RVI is the mandatory base ISA for RVA, and is little-endian.

As per the unprivileged architecture specification, the [ecall](#) instruction causes a requested trap to the execution environment.

*The **fence.tso** instruction is mandatory.*

*The **fence.tso** instruction was incorrectly described as optional in the 2019 ratified specifications. However, **fence.tso** is encoded within the standard [fence](#) encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that **fence.tso** is being exploited by a hardware implementation, there is no advantage to making the instruction a profile option. Later versions of the unprivileged ISA specifications correctly indicate that **fence.tso** is mandatory.*

- A Atomic instructions

Version = 2.1

- C Compressed instructions

Version = 2.2

- D Double-precision floating-point

Version = 2.2

- F Single-precision floating-point

Version = 2.2

- M Integer multiply and divide instructions

Version = 2.0

- **U** User-level privilege mode

Version ~> 2.0

- **Zicntr** Architectural performance counters

Version = 2.0

- **Ziccif** Main memory fetch requirement for RVA profiles

Version = 1.0



Ziccif is a new extension name introduced with RVA20. The fetch atomicity requirement facilitates runtime patching of aligned instructions.

- **Ziccrse** Main memory reservability requirement for RVA profiles

Version = 1.0



Ziccrse is a new extension name introduced with RVA20.

- **Ziccamao** Main memory atomicity requirement for RVA profiles

Version = 1.0



Ziccamao is a new extension name introduced with RVA20.

- **Za128rs** Reservation set requirement for RVA profiles

Version = 1.0



Za128rs is a new extension name introduced with RVA20. The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

- **Zicclsm** Main memory misaligned requirement for RVA profiles

Version = 1.0



Zicclsm is a new extension name introduced with RVA20. This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

5.1.2. Optional Extensions

- **Zihpm** Programmable hardware performance counters

Version= 2.0



The number of counters is platform-specific.

Chapter 6. RVA22S64 Profile

The RVA22S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA22S64 is based on privileged architecture version 1.12.

6.1. Extensions

The RVA22S64 Profile has 33 mandatory extensions and 10 optional extensions.

6.1.1. Mandatory Extensions

- **I** Base integer ISA

Version ~> 2.1

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

Version = 2.2

- **D** Double-precision floating-point

Version = 2.2

- **F** Single-precision floating-point

Version = 2.2

- **M** Integer multiply and divide instructions

Version = 2.0

- **U** User-level privilege mode

Version ~> 2.0

- **Zicntr** Architectural performance counters

Version = 2.0

- **Ziccif** Main memory fetch requirement for RVA profiles

Version = 1.0

- **Ziccrse** Main memory reservability requirement for RVA profiles

Version = 1.0

- **Ziccamaoa** Main memory atomicity requirement for RVA profiles

Version = 1.0

- **Za128rs** Reservation set requirement for RVA profiles
Version = 1.0
- **Zicclsm** Main memory misaligned requirement for RVA profiles
Version = 1.0
- **Zihpm** Programmable hardware performance counters
Version = 2.0
- **Zihintpause** PAUSE instruction
Version = 2.0
- **Zba** Address generation instructions
Version ~> 1.0
- **Zbb** Basic bit manipulation
Version ~> 1.0
- **Zbs** Single-bit instructions
Version ~> 1.0
- **Zic64b** 64-byte cache blocks
Version = 1.0
- **Zicbom** Cache block management instructions
Version ~> 1.0
- **Zicbop** Cache block prefetch
Version ~> 1.0
- **Zicboz** Cache block zero instruction
Version ~> 1.0
- **Zfhmin** Minimal half-precision Floating-point
Version ~> 1.0
- **Zkt** Data-independent execution latency
Version ~> 1.0
- **S** Supervisor mode
Version = 1.12

- **Sscounterenw** Supervisor counter enable

Version = 1.0



Sstvala is a new extension name introduced with RVA22.

- **Svpbmt** Page-based memory types

Version ~> 1.0

- **Svinval** Fine-grained address-translation cache invalidation

Version ~> 1.0

- **Ssstateen**

Version ~> 1.0



Ssstateen is a new extension name introduced with RVA22.

- **Shvstvala**

Version ~> 1.0



Shvstvala is a new extension name introduced with RVA22.

- **Shtvala**

Version ~> 1.0



Shtvala is a new extension name introduced with RVA22.

- **Shvstvecd**

Version ~> 1.0



Shvstvecd is a new extension name introduced with RVA22.

- **Shgatpa**

Version ~> 1.0



Shgatpa is a new extension name introduced with RVA22.

6.1.2. Optional Extensions

- **Zfh**

Version ~> 1.0

- **V**

Version ~> 1.0

- **Zkn**

Version~> 1.0

- **Zks**

Version~> 1.0

- **Sv57** 57-bit virtual address translation (5 level)

Version~> 1.12

- **Svnapot**

Version~> 1.0



It is expected that Svnapot will be mandatory in the next profile release.

- **Sstc** Supervisor mode timer interrupts

Version~> 1.0



Sstc was not made mandatory in RVA22S64 as it is a more disruptive change affecting system-level architecture, and will take longer for implementations to adopt. It is expected to be made mandatory in the next profile release.

- **Sscofpmf** Counter Overflow and Privilege Mode Filtering

Version~> 1.0



Platforms may choose to mandate the presence of Sscofpmf.

- **Zkr**

Version~> 1.0



Technically, Zk is also a privileged-mode option capturing that Zkr, Zkn, and Zkt are all implemented. However, the Zk rollup is less descriptive than specifying the individual extensions explicitly.

- **H** Hypervisor

Version~> 1.0



The following extensions become mandatory when H is implemented:

- **Ssstateen**
- **Shcounterenw**
- **Shvstvala**
- **Shtvala**
- **Shvstvecd**
- **Shgatpa**

Chapter 7. RVA22U64 Profile

The RVA22U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within the application processor family in terms of the amount of software that targets this profile.

7.1. Extensions

The RVA22U64 Profile has 24 mandatory extensions and 4 optional extensions.

7.1.1. Mandatory Extensions

- **I** Base integer ISA

Version ~> 2.1

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

Version = 2.2

- **D** Double-precision floating-point

Version = 2.2

- **F** Single-precision floating-point

Version = 2.2

- **M** Integer multiply and divide instructions

Version = 2.0

- **U** User-level privilege mode

Version ~> 2.0

- **Zicntr** Architectural performance counters

Version = 2.0

- **Ziccif** Main memory fetch requirement for RVA profiles

Version = 1.0

- **Ziccrse** Main memory reservability requirement for RVA profiles

Version = 1.0

- **Ziccamao** Main memory atomicity requirement for RVA profiles

Version = 1.0

- **Za128rs** Reservation set requirement for RVA profiles

Version = 1.0

- **Zicclsm** Main memory misaligned requirement for RVA profiles

Version = 1.0

- **Zihpm** Programmable hardware performance counters

Version = 2.0

- **Zihintpause** PAUSE instruction

Version = 2.0



While the **pause** instruction is a HINT can be implemented as a NOP and hence trivially supported by hardware implementers, its inclusion in the mandatory extension list signifies that software should use the instruction whenever it would make sense and that implementors are expected to exploit this information to optimize hardware execution.

- **Zba** Address generation instructions

Version ~> 1.0

- **Zbb** Basic bit manipulation

Version ~> 1.0

- **Zbs** Single-bit instructions

Version ~> 1.0

- **Zic64b** 64-byte cache blocks

Version = 1.0



This is a new extension name for this feature. While the general RISC-V specifications are agnostic to cache block size, selecting a common cache block size simplifies the specification and use of the following cache-block extensions within the application processor profile. Software does not have to query a discovery mechanism and/or provide dynamic dispatch to the appropriate code. We choose 64 bytes as it is effectively an industry standard. Implementations may use longer cache blocks to reduce tag cost provided they use 64-byte sub-blocks to remain compatible. Implementations may use shorter cache blocks provided they sequence cache operations across the multiple cache blocks comprising a 64-byte block to remain compatible.

- **Zicbom** Cache block management instructions

Version ~> 1.0

- **Zicbop** Cache block prefetch

Version ~> 1.0



As with other HINTS, the inclusion of prefetches in the mandatory set of extensions indicates that software should generate these instructions where they are expected to be useful, and hardware is expected to exploit that information.

- **Zicboz** Cache block zero instruction

Version ~> 1.0

- **Zfhmin** Minimal half-precision Floating-point

Version ~> 1.0



Zfhmin is a small extension that adds support to load/store and convert IEEE 754 half-precision numbers to and from the IEEE 754 single-precision format. The hardware cost for this extension is low, and mandating the extension avoids adding an option to the profile.

- **Zkt** Data-independent execution latency

Version ~> 1.0



Zkt requires a certain subset of integer instructions execute with data-independent latency. Mandating this feature enables portable libraries for safe basic cryptographic operations. It is expected that application processors will naturally have this property and so implementation cost is low, if not zero, in most systems that would support RVA22.

7.1.2. Optional Extensions

- **Zfh**

Version ~> 1.0



A future profile might mandate V.

- **V**

Version ~> 1.0



The smaller vector extensions (Zve32f, Zve32x, Zve64d, Zve64f, Zve64x) are not provided as separately supported profile options. The full V extension is specified as the only supported profile option.

A future profile might mandate V.

- **Zkn**

Version ~> 1.0



The scalar crypto extensions are expected to be superseded by vector crypto standards in future profiles, and the scalar extensions may be removed as supported options once vector crypto is present.

The smaller component scalar crypto extensions (Zbc, Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh) are not provided as separate options in the profile. Profile implementers should provide all of the instructions in a given algorithm suite as part of the Zkn or Zks supported options.

- Zks

Version~> 1.0

The scalar crypto extensions are expected to be superseded by vector crypto standards in future profiles, and the scalar extensions may be removed as supported options once vector crypto is present.



The smaller component scalar crypto extensions (Zbc, Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh) are not provided as separate options in the profile. Profile implementers should provide all of the instructions in a given algorithm suite as part of the Zkn or Zks supported options.

DRAFT

Chapter 8. Profile Parameters

ARCH_ID

Vendor-specific architecture ID in [marchid](#)

CONFIG_PTR_ADDRESS

Physical address of the unified discovery configuration data structure this address is reported in the [mconfigptr](#) CSR

COUNTINHIBIT_EN

Indicates which counters can be disabled from [mcountinhibit](#)

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `COUNTINHIBIT_EN[3]` to true.

`COUNTINHIBIT_EN[1]` can never be true, since it corresponds to [mcountinhibit](#), which is always read-only-0.

`COUNTINHIBIT_EN[3:31]` must all be false if `Zihpm` is not implemented.

IMP_ID

Vendor-specific implementation ID in [mimpid](#)

MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE

The maximum granule size, in bytes, that the hart can atomically perform a misaligned load/store/AMO without raising a Misaligned exception. When `MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE` is 0, the hart cannot atomically perform a misaligned load/store/AMO. When a power of two, the hart can atomically load/store/AMO a misaligned access that is fully contained in a `MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE`-aligned region.



Even if the hart is capable of performing a misaligned load/store/AMO atomically, a misaligned exception may still occur if the access does not have the appropriate Misaligned Atomicity Granule PMA set.

MCOUNTENABLE_EN

Indicates which counters can delegated via [mcounteren](#)

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `MCOUNTENABLE_EN[3]` to true.

`MCOUNTENABLE_EN[0:2]` must all be false if `Zicntr` is not implemented. `MCOUNTENABLE_EN[3:31]` must all be false if `Zihpm` is not implemented.

MISALIGNED_LDST

Whether or not the implementation supports non-atomic misaligned loads and stores in main memory (does **not** affect misaligned support to device memory)

MISALIGNED_LDST_EXCEPTION_PRIORITY

The relative priority of a load/store/AMO exception vs. load/store/AMO page-fault or access-fault exceptions.

May be one of:

high	Misaligned load/store/AMO exceptions are always higher priority than load/store/AMO page-fault and access-fault exceptions.
low	Misaligned load/store/AMO exceptions are always lower priority than load/store/AMO page-fault and access-fault exceptions.

MISALIGNED_LDST_EXCEPTION_PRIORITY cannot be "high" when MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE is non-zero, since the atomicity of an access cannot be determined in that case until after address translation.

MISALIGNED_SPLIT_STRATEGY

when misaligned accesses are supported, this determines the **order** in the implementation appears to process the load/store, which determines how/which exceptions will be reported

Options: * by_byte: The load/store appears to be broken into byte-sized accesses that processed sequentially from smallest address to largest address * custom: Something else. Will result in a call to unpredictable() in the execution

MTVAL_WIDTH

The number of implemented bits in [mtval](#).

Must be greater than or equal to $\max(\text{PHYS_ADDR_WIDTH}, \text{VA_SIZE})$

M_MODE_ENDIANESS

Endianness of data in M-mode. Can be one of:

little	M-mode data is always little endian
big	M-mode data is always big endian
dynamic	M-mode data can be either little or big endian, depending on the CSR field mstatus.MBE

NUM_PMP_ENTRIES

Number of implemented PMP entries. Can be any value between 0-64, inclusive.

The architecture mandates that the number of implemented PMP registers must appear to be 0, 16, or 64.

Therefore, pmp registers will behave as follows according to NUM_PMP_ENTRIES:

NUM_PMP_ENTRIES	pmpaddr<0-15> / pmpcfg<0-3>	pmpaddr<16-63> / pmpcfg<4-15>
0	N	N
1-16	Y	N
17-64	Y	Y

- N = Not implemented; access will cause **IllegalInstruction** if TRAP_ON_UNIMPLEMENTED_CSR is true
- Y = Implemented; access will not cause an exception (from M-mode), but register may be read-

only-zero if NUM_PMP_ENTRIES is less than the corresponding register



pmpcfgN for an odd N never exists when XLEN == 64

When NUM_PMP_ENTRIES is not exactly 0, 16, or 64, some extant pmp registers, and associated pmpNcfg, will be read-only zero (but will never cause an exception).

PHYS_ADDR_WIDTH

Number of bits in the physical address space.

PMA_GRANULARITY

log2 of the smallest supported PMA region.

Generally, for systems with an MMU, should not be smaller than 12, as that would preclude caching PMP results in the TLB along with virtual memory translations

PMP_GRANULARITY

log2 of the smallest supported PMP region.

Generally, for systems with an MMU, should not be smaller than 12, as that would preclude caching PMP results in the TLB along with virtual memory translations

Note that PMP_GRANULARITY is equal to G+2 (not G) as described in the privileged architecture.

REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION

When true, **mtval** is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false **mtval** is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_MTVAL_ON_BREAKPOINT

When true, **mtval** is written with the virtual PC of the EBREAK instruction (same information as **mepc**).

When false, **mtval** is written with 0 on an EBREAK instruction.

Regardless, **mtval** is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, **mtval** is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, **mtval** is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED

When true, **mtval** is written with the virtual PC when an instruction fetch is misaligned.

When false, **mtval** is written with 0 when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT

When true, `mtval` is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, `mtval` is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT

When true, `mtval` is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, `mtval` is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED

When true, `mtval` is written with the virtual address of a load instruction when the address is misaligned and **MISALIGNED_LDST** is false.

When false, `mtval` is written with 0 when a load address is misaligned and **MISALIGNED_LDST** is false.

REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT

When true, `mtval` is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, `mtval` is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT

When true, `mtval` is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, `mtval` is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED

When true, `mtval` is written with the virtual address of a store instruction when the address is misaligned and **MISALIGNED_LDST** is false.

When false, `mtval` is written with 0 when a store address is misaligned and **MISALIGNED_LDST** is false.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT

When true, `mtval` is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, `mtval` is written with 0 when a store causes a **StoreAmoPageFault**.

TRAP_ON_ILLEGAL_WLRL

When true, writing an illegal value to a WLRL CSR field raises an **IllegalInstruction** exception.

When false, writing an illegal value to a WLRL CSR field is **unpredictable**.

TRAP_ON_UNIMPLEMENTED_CSR

When true, accessing an unimplemented CSR (via a Zicsr instruction) will cause an **IllegalInstruction** exception.

When false, accessing an unimplemented CSR (via a Zicsr instruction) is **unpredictable**.

TRAP_ON_UNIMPLEMENTED_INSTRUCTION

When true, fetching an unimplemented instruction will cause an **IllegalInstruction** exception.

When false, fetching an unimplemented instruction is **unpredictable**.

VENDOR_ID_BANK

JEDEC Vendor ID bank, for [mvendorid](#)

VENDOR_ID_OFFSET

Vendor JEDEC code offset, for [mvendorid](#)

XLEN

XLEN in M-mode (*i.e.*, *MXLEN*)

LRSC_FAIL_ON_NON_EXACT_LRSC

Whether or not a Store Conditional fails if its physical address and size do not exactly match the physical address and size of the last Load Reserved in program order (independent of whether or not the SC is in the current reservation set)

LRSC_FAIL_ON_VA_SYNONYM

Whether or not an **sc.l/sc.d** will fail if its VA does not match the VA of the prior **lr.l/lr.d**, even if the physical address of the SC and LR are the same

LRSC_MISALIGNED_BEHAVIOR

What to do when an LR/SC address is misaligned and MISALIGNED_AMO == false.

- 'always raise misaligned exception': self-explanatory
- 'always raise access fault': self-explanatory
- 'custom': Custom behavior; misaligned LR/SC may sometimes raise a misaligned exception and sometimes raise a access fault. Will lead to an 'unpredictable' call on any misaligned LR/SC access

LRSC_RESERVATION_STRATEGY

Strategy used to handle reservation sets.

- "reserve naturally-aligned 64-byte region": Always reserve the 64-byte block containing the LR/SC address
- "reserve naturally-aligned 128-byte region": Always reserve the 128-byte block containing the LR/SC address
- "reserve exactly enough to cover the access": Always reserve exactly the LR/SC access, and no more
- "custom": Custom behavior, leading to an 'unpredictable' call on any LR/SC

MISALIGNED_AMO

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A

When the A extensions is supported, indicates whether or not the extension can be disabled in the misa.A bit.

MUTABLE_MISA_C

Indicates whether or not the C extension can be disabled with the misa.C bit.

MUTABLE_MISA_D

Indicates whether or not the D extension can be disabled with the misa.D bit.

MUTABLE_MISA_F

Indicates whether or not the F extension can be disabled with the misa.F bit.

MUTABLE_MISA_M

Indicates whether or not the M extension can be disabled with the misa.M bit.

MUTABLE_MISA_U

Indicates whether or not the U extension can be disabled with the misa.U bit.

UXLEN

Set of XLENs supported in U-mode. Can be one of:

- 32: SXLEN is always 32
- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via mstatus.UXL) between 32 and 64

U_MODE_ENDIANNESS

Endianness of data in U-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field mstatus.UBE

ASID_WIDTH

Number of implemented ASID bits. Maximum is 16 for XLEN==64, and 9 for XLEN==32

MUTABLE_MISA_S

Indicates whether or not the S extension can be disabled with the misa.S bit.

REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION

When true, `stval` is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false `stval` is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_STVAL_ON_BREAKPOINT

When true, **stval** is written with the virtual PC of the EBREAK instruction (same information as **mepc**).

When false, **stval** is written with 0 on an EBREAK instruction.

Regardless, **stval** is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, **stval** is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, **stval** is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED

When true, **stval** is written with the virtual PC when an instruction fetch is misaligned.

When false, **stval** is written with 0 when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT

When true, **stval** is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, **stval** is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT

When true, **stval** is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, **stval** is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED

When true, **stval** is written with the virtual address of a load instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, **stval** is written with 0 when a load address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT

When true, **stval** is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, **stval** is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT

When true, **stval** is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, `stval` is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED

When true, `stval` is written with the virtual address of a store instruction when the address is misaligned and `MISALIGNED_LDST` is false.

When false, `stval` is written with 0 when a store address is misaligned and `MISALIGNED_LDST` is false.

REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT

When true, `stval` is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, `stval` is written with 0 when a store causes a **StoreAmoPageFault**.

SCOUNTENABLE_EN

Indicates which counters can be delegated via `scounteren`.

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `SCOUNTENABLE_EN[3]` to true.

`SCOUNTENABLE_EN[0:2]` must all be false if `Zicntr` is not implemented.
`SCOUNTENABLE_EN[3:31]` must all be false if `Zihpm` is not implemented.

STVAL_WIDTH

The number of implemented bits in `stval`.

Must be greater than or equal to $\max(\text{PHYS_ADDR_WIDTH}, \text{VA_SIZE})$.

STVEC_MODE_DIRECT

Whether or not `stvec.MODE` supports Direct (0).

STVEC_MODE_VECTORED

Whether or not `stvec.MODE` supports Vectored (1).

SV_MODE_BARE

Whether or not writing `mode=Bare` is supported in the `satp` register.

SXLEN

Set of XLENs supported in S-mode. Can be one of:

- 32: SXLEN is always 32
- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via `mstatus.SXL`) between 32 and 64

S_MODE_ENDIANESS

Endianess of data in S-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian

- dynamic: M-mode data can be either little or big endian, depending on the CSR field `mstatus.SBE`

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY

For implementations that make `satp.MODE` read-only zero (always Bare, *i.e.*, no virtual translation is implemented), attempts to execute an `SFENCE.VMA` instruction might raise an illegal-instruction exception.

`TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY` indicates whether or not that exception occurs.

`TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY` has no effect when some virtual translation mode is supported.

HPM_COUNTER_EN

List of HPM counters that are enabled. There is one entry for each `hpmcounter`.

The first three entries **must** be false (as they correspond to CY, IR, TM in, *e.g.* **mhmpcountinhibit**) Index 3 in `HPM_COUNTER_EN` corresponds to `hpmcounter3`. Index 31 in `HPM_COUNTER_EN` corresponds to `hpmcounter31`.

HPM_EVENTS

List of defined event numbers that can be written into `hpmeventN`

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

GSTAGE_MODE_BARE

Whether or not writing `mode=Bare` is supported in the `hgatp` register.

HCOUNTENABLE_EN

Indicates which counters can be delegated via `hcounteren`

An unimplemented counter cannot be specified, *i.e.*, if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `HCOUNTENABLE_EN[3]` to true.

`HCOUNTENABLE_EN[0:2]` must all be false if `Zicntr` is not implemented.
`HCOUNTENABLE_EN[3:31]` must all be false if `Zihpm` is not implemented.

MUTABLE_MISA_H

Indicates whether or not the H extension can be disabled with the `misa.H` bit.

NUM_EXTERNAL_GUEST_INTERRUPTS

Number of supported virtualized guest interrupts

Corresponds to the **GEILEN** parameter in the RVI specs

REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION

When true, `vstval` is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false `vstval` is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_VSTVAL_ON_BREAKPOINT

When true, `vstval` is written with the virtual PC of the EBREAK instruction (same information as `mepc`).

When false, `vstval` is written with 0 on an EBREAK instruction.

Regardless, `vstval` is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, `vstval` is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, `vstval` is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED

When true, `vstval` is written with the virtual PC when an instruction fetch is misaligned.

When false, `vstval` is written with 0 when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT

When true, `vstval` is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, `vstval` is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT

When true, `vstval` is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, `vstval` is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED

When true, `vstval` is written with the virtual address of a load instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, `vstval` is written with 0 when a load address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT

When true, `vstval` is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, `vstval` is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT

When true, `vstval` is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, `vstval` is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED

When true, `vstval` is written with the virtual address of a store instruction when the address is misaligned and `MISALIGNED_LDST` is false.

When false, `vstval` is written with 0 when a store address is misaligned and `MISALIGNED_LDST` is false.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT

When true, `vstval` is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, `vstval` is written with 0 when a store causes a **StoreAmoPageFault**.

SV32X4_TRANSLATION

Whether or not Sv32x4 translation mode is supported.

SV39X4_TRANSLATION

Whether or not Sv39x4 translation mode is supported.

SV48X4_TRANSLATION

Whether or not Sv48x4 translation mode is supported.

SV57X4_TRANSLATION

Whether or not Sv57x4 translation mode is supported.

VMID_WIDTH

Number of bits supported in `hgap.VMID` (i.e., the supported width of a virtual machine ID).

VSXLEN

Set of XLENs supported in VS-mode. Can be one of:

- 32: VSXLEN is always 32
- 64: VSXLEN is always 64
- 3264: VSXLEN can be changed (via `hstatus.VSXL`) between 32 and 64

VS_MODE_ENDIANESS

Endianess of data in VS-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field `hstatus.VSBE`

VUXLEN

Set of XLENs supported in VU-mode. Can be one of:

- 32: VUXLEN is always 32
- 64: VUXLEN is always 64
- 3264: VUXLEN can be changed (via `vsstatus.UXL`) between 32 and 64

VU_MODE_ENDIANESS

Endianess of data in VU-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field `vsstatus.UBE`

DRAFT

Appendix A: Extension Specifications

DRAFT

A.1. I Extension

Base integer ISA

Table 2. Status

Profile	v2.1
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.1

Ratification date

2019-06

Changes

ratified RVWMO memory model and exclusion of FENCE.I, counters, and CSR instructions that were in previous base ISA

A.1.1. Synopsys

Base integer instructions – TODO

A.1.2. Instructions

The following instructions are added by this extension:

<code>add</code>	Integer add
<code>addi</code>	Add immediate
<code>addiw</code>	Add immediate word
<code>addw</code>	Add word
<code>and</code>	And
<code>andi</code>	And immediate
<code>auipc</code>	Add upper immediate to pc
<code>beq</code>	Branch if equal
<code>bge</code>	Branch if greater than or equal
<code>bgeu</code>	Branch if greater than or equal unsigned
<code>blt</code>	Branch if less than
<code>bltu</code>	Branch if less than unsigned
<code>bne</code>	Branch if not equal
<code>ebreak</code>	Breakpoint exception
<code>ecall</code>	Environment call
<code>fence</code>	Memory ordering fence
<code>jal</code>	Jump and link

jalr	Jump and link register
lb	Load byte
lbu	Load byte unsigned
ld	Load doubleword
lh	Load halfword
lhu	Load halfword unsigned
lui	Load upper immediate
lw	Load word
lwu	Load word unsigned
mret	Machine Exception Return
or	Or
ori	Or immediate
sb	Store byte
sd	Store doubleword
sh	Store halfword
sll	Shift left logical
slli	Shift left logical immediate
slliw	Shift left logical immediate word
sllw	Shift left logical word
slt	Set on less than
slti	Set on less than immediate
sltiu	Set on less than immediate unsigned
sltu	Set on less than unsigned
sra	Shift right arithmetic
srai	Shift right arithmetic immediate
sraiw	Shift right arithmetic immediate word
sraw	Shift right arithmetic word
srl	Shift right logical
srli	Shift right logical immediate
srliw	Shift right logical immediate word
srlw	Shift right logical word
sub	Subtract
subw	Subtract word
sw	Store word
wfi	Wait for interrupt
xor	Exclusive Or
xori	Exclusive Or immediate

A.1.3. Parameters

This extension has the following implementation options:

ARCH_ID

Vendor-specific architecture ID in [marchid](#)

CONFIG_PTR_ADDRESS

Physical address of the unified discovery configuration data structure this address is reported in the [mconfigptr](#) CSR

COUNTINHIBIT_EN

Indicates which counters can be disabled from [mcountinhibit](#)

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `COUNTINHIBIT_EN[3]` to true.

`COUNTINHIBIT_EN[1]` can never be true, since it corresponds to [mcountinhibit](#), which is always read-only-0.

`COUNTINHIBIT_EN[3:31]` must all be false if `Zihpm` is not implemented.

IMP_ID

Vendor-specific implementation ID in [mimpid](#)

MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE

The maximum granule size, in bytes, that the hart can atomically perform a misaligned load/store/AMO without raising a Misaligned exception. When `MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE` is 0, the hart cannot atomically perform a misaligned load/store/AMO. When a power of two, the hart can atomically load/store/AMO a misaligned access that is fully contained in a `MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE`-aligned region.



Even if the hart is capable of performing a misaligned load/store/AMO atomically, a misaligned exception may still occur if the access does not have the appropriate Misaligned Atomicity Granule PMA set.

MCOUNTENABLE_EN

Indicates which counters can be delegated via [mcounteren](#)

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `MCOUNTENABLE_EN[3]` to true.

`MCOUNTENABLE_EN[0:2]` must all be false if `Zicntr` is not implemented. `MCOUNTENABLE_EN[3:31]` must all be false if `Zihpm` is not implemented.

MISALIGNED_LDST

Whether or not the implementation supports non-atomic misaligned loads and stores in main memory (does **not** affect misaligned support to device memory)

MISALIGNED_LDST_EXCEPTION_PRIORITY

The relative priority of a load/store/AMO exception vs. load/store/AMO page-fault or access-fault exceptions.

May be one of:

high	Misaligned load/store/AMO exceptions are always higher priority than load/store/AMO page-fault and access-fault exceptions.
low	Misaligned load/store/AMO exceptions are always lower priority than load/store/AMO page-fault and access-fault exceptions.

MISALIGNED_LDST_EXCEPTION_PRIORITY cannot be "high" when MAX_MISALIGNED_ATOMICITY_GRANULE_SIZE is non-zero, since the atomicity of an access cannot be determined in that case until after address translation.

MISALIGNED_SPLIT_STRATEGY

when misaligned accesses are supported, this determines the **order** in the implementation appears to process the load/store, which determines how/which exceptions will be reported

Options: * by_byte: The load/store appears to be broken into byte-sized accesses that processed sequentially from smallest address to largest address * custom: Something else. Will result in a call to unpredictable() in the execution

MTVAL_WIDTH

The number of implemented bits in [mtval](#).

Must be greater than or equal to $\max(\text{PHYS_ADDR_WIDTH}, \text{VA_SIZE})$

M_MODE_ENDIANESS

Endianness of data in M-mode. Can be one of:

little	M-mode data is always little endian
big	M-mode data is always big endian
dynamic	M-mode data can be either little or big endian, depending on the CSR field mstatus.MBE

NUM_PMP_ENTRIES

Number of implemented PMP entries. Can be any value between 0-64, inclusive.

The architecture mandates that the number of implemented PMP registers must appear to be 0, 16, or 64.

Therefore, pmp registers will behave as follows according to NUM_PMP_ENTRIES:

NUM_PMP_ENTRIES	pmpaddr<0-15> / pmpcfg<0-3>	pmpaddr<16-63> / pmpcfg<4-15>
0	N	N
1-16	Y	N
17-64	Y	Y

- N = Not implemented; access will cause **IllegalInstruction** if TRAP_ON_UNIMPLEMENTED_CSR is true
- Y = Implemented; access will not cause an exception (from M-mode), but register may be read-

only-zero if NUM_PMP_ENTRIES is less than the corresponding register



pmpcfgN for an odd N never exists when $XLEN == 64$

When NUM_PMP_ENTRIES is not exactly 0, 16, or 64, some extant pmp registers, and associated pmpNcfg, will be read-only zero (but will never cause an exception).

PHYS_ADDR_WIDTH

Number of bits in the physical address space.

PMA_GRANULARITY

\log_2 of the smallest supported PMA region.

Generally, for systems with an MMU, should not be smaller than 12, as that would preclude caching PMP results in the TLB along with virtual memory translations

PMP_GRANULARITY

\log_2 of the smallest supported PMP region.

Generally, for systems with an MMU, should not be smaller than 12, as that would preclude caching PMP results in the TLB along with virtual memory translations

Note that PMP_GRANULARITY is equal to $G+2$ (not G) as described in the privileged architecture.

REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION

When true, **mtval** is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false **mtval** is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_MTVAL_ON_BREAKPOINT

When true, **mtval** is written with the virtual PC of the EBREAK instruction (same information as **mepc**).

When false, **mtval** is written with 0 on an EBREAK instruction.

Regardless, **mtval** is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, **mtval** is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, **mtval** is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED

When true, **mtval** is written with the virtual PC when an instruction fetch is misaligned.

When false, **mtval** is written with 0 when an instruction fetch is misaligned.

Note that when $IALIGN=16$ (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_MTVAl_ON_INSTRUCTION_PAGE_FAULT

When true, **mtval** is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, **mtval** is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_MTVAl_ON_LOAD_ACCESS_FAULT

When true, **mtval** is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, **mtval** is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_MTVAl_ON_LOAD_MISALIGNED

When true, **mtval** is written with the virtual address of a load instruction when the address is misaligned and **MISALIGNED_LDST** is false.

When false, **mtval** is written with 0 when a load address is misaligned and **MISALIGNED_LDST** is false.

REPORT_VA_IN_MTVAl_ON_LOAD_PAGE_FAULT

When true, **mtval** is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, **mtval** is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_MTVAl_ON_STORE_AMO_ACCESS_FAULT

When true, **mtval** is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, **mtval** is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_MTVAl_ON_STORE_AMO_MISALIGNED

When true, **mtval** is written with the virtual address of a store instruction when the address is misaligned and **MISALIGNED_LDST** is false.

When false, **mtval** is written with 0 when a store address is misaligned and **MISALIGNED_LDST** is false.

REPORT_VA_IN_MTVAl_ON_STORE_AMO_PAGE_FAULT

When true, **mtval** is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, **mtval** is written with 0 when a store causes a **StoreAmoPageFault**.

TRAP_ON_ILLEGAL_WLRL

When true, writing an illegal value to a WLRL CSR field raises an **IllegalInstruction** exception.

When false, writing an illegal value to a WLRL CSR field is **unpredictable**.

TRAP_ON_UNIMPLEMENTED_CSR

When true, accessing an unimplemented CSR (via a Zicsr instruction) will cause an **IllegalInstruction** exception.

When false, accessing an unimplemented CSR (via a Zicsr instruction) is **unpredictable**.

TRAP_ON_UNIMPLEMENTED_INSTRUCTION

When true, fetching an unimplemented instruction will cause an **IllegalInstruction** exception.

When false, fetching an unimplemented instruction is **unpredictable**.

VENDOR_ID_BANK

JEDEC Vendor ID bank, for [mvendorid](#)

VENDOR_ID_OFFSET

Vendor JEDEC code offset, for [mvendorid](#)

XLEN

XLEN in M-mode (*i.e.*, *MXLEN*)

DRAFT

A.2. A Extension

Atomic instructions

Table 3. Status

Profile	v2.1
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.1

Ratification date

2019-12

Implies

- Zaamo version 1.0
- Zalrsc version 1.0

A.2.1. Synopsis

The atomic-instruction extension, named A, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model. cite:[Gharachorloo90memoryconsistency]



After much debate, the language community and architecture community appear to have finally settled on release consistency as the standard memory consistency model and so the RISC-V atomic support is built around this model.

The A extension comprises instructions provided by the Zaamo and Zalrsc extensions.

A.2.2. Specifying Ordering of Atomic Instructions

The base RISC-V ISA has a relaxed memory model, with the **FENCE** instruction used to impose additional ordering constraints. The address space is divided by the execution environment into memory and I/O domains, and the **FENCE** instruction provides options to order accesses to one or both of these two address domains.

To provide more efficient support for release consistency cite:[Gharachorloo90memoryconsistency], each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V harts. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a **FENCE** instruction should be used to order across both domains.

If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the *aq* bit is set, the atomic memory operation is treated as an *acquire* access, i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation. If only the *rl* bit is set, the atomic memory operation is treated as a *release* access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the *aq* and *rl* bits are set, the atomic memory operation is *sequentially consistent* and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

A.2.3. Instructions

The following instructions are added by this extension:

amoadd.d	Atomic fetch-and-add doubleword
amoadd.w	Atomic fetch-and-add word
amoand.d	Atomic fetch-and-and doubleword
amoand.w	Atomic fetch-and-and word
amomax.d	Atomic MAX doubleword
amomax.w	Atomic MAX word
amomaxu.d	Atomic MAX unsigned doubleword
amomaxu.w	Atomic MAX unsigned word
amomin.d	Atomic MIN doubleword
amomin.w	Atomic MIN word
amominu.d	Atomic MIN unsigned doubleword
amominu.w	Atomic MIN unsigned word
amoor.d	Atomic fetch-and-or doubleword
amoor.w	Atomic fetch-and-or word
amoswap.d	Atomic SWAP doubleword
amoswap.w	Atomic SWAP word
amoxor.d	Atomic fetch-and-xor doubleword
amoxor.w	Atomic fetch-and-xor word
lr.d	Load reserved doubleword
lr.w	Load reserved word
sc.d	Store conditional doubleword
sc.w	Store conditional word

A.2.4. Parameters

This extension has the following implementation options:

LRSC_FAIL_ON_NON_EXACT_LRSC

Whether or not a Store Conditional fails if its physical address and size do not exactly match the physical address and size of the last Load Reserved in program order (independent of whether or not the SC is in the current reservation set)

LRSC_FAIL_ON_VA_SYNONYM

Whether or not an **sc.l/sc.d** will fail if its VA does not match the VA of the prior **lr.l/lr.d**, even if the physical address of the SC and LR are the same

LRSC_MISALIGNED_BEHAVIOR

What to do when an LR/SC address is misaligned and MISALIGNED_AMO == false.

- 'always raise misaligned exception': self-explanitory
- 'always raise access fault': self-explanitory
- 'custom': Custom behavior; misaligned LR/SC may sometimes raise a misaligned exception and sometimes raise a access fault. Will lead to an 'unpredictable' call on any misaligned LR/SC access

LRSC_RESERVATION_STRATEGY

Strategy used to handle reservation sets.

- "reserve naturally-aligned 64-byte region": Always reserve the 64-byte block containing the LR/SC address
- "reserve naturally-aligned 128-byte region": Always reserve the 128-byte block containing the LR/SC address
- "reserve exactly enough to cover the access": Always reserve exactly the LR/SC access, and no more
- "custom": Custom behavior, leading to an 'unpredictable' call on any LR/SC

MISALIGNED_AMO

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A

When the A extensions is supported, indicates whether or not the extension can be disabled in the misa.A bit.

A.3. C Extension

Compressed instructions

Table 4. Status

Profile	v2.2
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.2

Ratification date
2019-12

A.3.1. Synopsys

The C extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%-60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%-30% code-size reduction.

A.3.2. Overview

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (**x0**), the ABI link register (**x1**), or the ABI stack pointer (**x2**), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions.



Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.

The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 34, a few opcodes are used for different purposes depending on base ISA. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the

appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.



Double-precision loads and stores are a significant fraction of static and dynamic instructions, hence the motivation to include them in the RV32C and RV64C encoding.

Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.

Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.

Although reusing opcodes for different purposes for different base ISAs adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISAs. The compressed floating-point load and store variants use the same instruction format with the same register specifiers as the wider integer loads and stores.

RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E, RV64I/E, or RV128I) or the F and D standard extensions where present. Adopting this constraint has two main benefits:

- Hardware designs can simply expand RVC instructions during decode, simplifying verification and minimizing modifications to existing microarchitectures.
- Compilers can be unaware of the RVC extension and leave code compression to the assembler and linker, although a compression-aware compiler will generally be able to produce better results.



We felt the multiple complexity reductions of a simple one-one mapping between C and base IFD instructions far outweighed the potential gains of a slightly denser encoding that added additional instructions only supported in the C extension, or that allowed encoding of multiple IFD instructions in one C instruction.

It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA.



Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch cite:[stretch], developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture cite:[ibm360] supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 cite:[cdc6600], a precursor to RISC architectures, that introduced a register-rich load-store architecture with

instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25-30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.

Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to significant complexity in documentation, implementations, and software tools support.

Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size. cite:[waterman-ms]

A.3.3. Compressed Instruction Formats

Table 5 shows the nine compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, CA, and CB are limited to just 8 of them. Table 6 lists these popular registers, which correspond to registers **x8** to **x15**. Note that there is a separate version of load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow access to all 32 data

registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.



The RISC-V ABI was changed to make the frequently used registers map to registers 'x8-x15'. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E and RV64E base ISAs, which only have 16 integer registers.

Compressed register-based floating-point loads and stores also use the CL and CS formats respectively, with the eight registers mapping to **f8** to **f15**.



*The standard RISC-V calling convention maps the most frequently used floating-point registers to registers **f8** to **f15**, which allows the same register decompression decoding as for integer register numbers.*

The formats were designed to keep bits for the two register source specifiers in the same place in all instructions, while the destination register field can move. When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding. Where immediates are sign-extended, the sign extension is always from bit 12. Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate muxes required.



The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations.

For many RVC instructions, zero-valued immediates are disallowed and **x0** is not a valid 5-bit register specifier. These restrictions free up encoding space for other instructions requiring fewer operand bits.

Table 5. Compressed 16-bit RVC instruction formats

Format	Meaning	15 14 13 12		11 10 9 8 7			6 5 4 3 2			1 0
CR	Register	funct4		rd/rs1			rs2			op
CI	Immediate	funct3	imm	rd/rs1			imm			op
CSS	Stack-relative Store	funct3	imm			rs2			op	
CIW	Wide Immediate	funct3	imm					rd'	op	
CL	Load	funct3	imm		rs1'	imm	rd'	op		
CS	Store	funct3	imm		rs1'	imm	rs2'	op		
CA	Arithmetic	funct6			rd'/rs1'	funct2	rs2'	op		
CB	Branch/Arithmetic	funct3	offset		rd'/rs1'	offset		op		
CJ	Jump	funct3	jump target						op	

Table 6. Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, CA, and CB formats.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

A.3.4. Parameters

This extension has the following implementation options:

MUTABLE_MISA_C

Indicates whether or not the C extension can be disabled with the misa.C bit.

DRAFT

A.4. D Extension

Double-precision floating-point

Table 7. Status

Profile	v2.2
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.2

Ratification date

2019-12

Changes

Define NaN-boxing scheme, changed definition of FMAX and FMIN

Implies

- F version 2.2

A.4.1. Synopsis

The D extension adds double-precision floating-point computational instructions compliant with the [IEEE 754-2008](#) arithmetic standard. The D extension depends on the base single-precision instruction subset F.

A.4.2. D Register State

The D extension widens the 32 floating-point registers, **f0-f31**, to 64 bits (FLEN=64 in [Table 9](#). The **f** registers can now hold either 32-bit or 64-bit floating-point values as described below in [Section A.4.3](#).



FLEN can be 32, 64, or 128 depending on which of the F, D, and Q extensions are supported. There can be up to four different floating-point precisions supported, including H, F, D, and Q.

A.4.3. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower n -bit types, $n < \text{FLEN}$, are represented in the lower n bits of an FLEN-bit NaN value, in a process termed NaN-boxing. The upper bits of a valid NaN-boxed value must be all 1s. Valid NaN-boxed n -bit values therefore appear as negative quiet NaNs (qNaNs) when viewed as any wider m -bit value, $n < m \leq \text{FLEN}$. Any operation that writes a narrower result to an 'f' register must write all 1s to the uppermost FLEN- n bits to yield a legal NaN-boxed value.



Software might not know the current type of data stored in a floating-point register but

has to be able to save and restore the register values, hence the result of using wider operations to transfer narrower values has to be defined. A common case is for callee-saved registers, but a standard convention is also desirable for features including varargs, user-level threading libraries, virtual machine migration, and debugging.

Floating-point n -bit transfer operations move external values held in IEEE standard formats into and out of the **f** registers, and comprise floating-point loads and stores (FL n /FS n) and floating-point move instructions (FMV. n .X/FMV.X. n). A narrower n -bit transfer, $n < \text{FLEN}$, into the **f** registers will create a valid NaN-boxed value. A narrower n -bit transfer out of the floating-point registers will transfer the lower n bits of the register ignoring the upper FLEN- n bits.

Apart from transfer operations described in the previous paragraph, all other floating-point operations on narrower n -bit operations, $n < \text{FLEN}$, check if the input operands are correctly NaN-boxed, i.e., all upper FLEN- n bits are 1. If so, the n least-significant bits of the input are used as the input value, otherwise the input value is treated as an n -bit canonical NaN.

Earlier versions of this document did not define the behavior of feeding the results of narrower or wider operands into an operation, except to require that wider saves and restores would preserve the value of a narrower operand. The new definition removes this implementation-specific behavior, while still accommodating both non-recoded and recoded implementations of the floating-point unit. The new definition also helps catch software errors by propagating NaNs if values are used incorrectly.

Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in writing all 1s to the upper bits of a result.

Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can be done without adding new state bits by recoding NaNs internally in the exponent field. Small modifications are needed to the pipelines used to transfer values in and out of the recoded format, but the datapath and latency costs are minimal. The recoding process has to handle shifting of input subnormal values for wide operands in any case, and extracting the NaN-boxed value is a similar process to normalization except for skipping over leading-1 bits instead of skipping over leading-0 bits, allowing the datapath muxing to be shared.

A.4.4. Parameters

This extension has the following implementation options:

MUTABLE_MISA_D

Indicates whether or not the D extension can be disabled with the misa.D bit.

A.5. F Extension

Single-precision floating-point

Table 8. Status

Profile	v2.2
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.2

Ratification date

2019-12

Changes

Define NaN-boxing scheme, changed definition of FMAX and FMIN

A.5.1. Synopsys

This chapter describes the standard instruction-set extension for single-precision floating-point, which is named "F" and adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard cite:[ieee754-2008]. The F extension depends on the "Zicsr" extension for control and status register access.

A.5.2. F Register State

The F extension adds 32 floating-point registers, **f0-f31**, each 32 bits wide, and a floating-point control and status register **fcsr**, which contains the operating mode and exception status of the floating-point unit. This additional state is shown in Table 9. We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, and FLEN=32 for the F single-precision floating-point extension. Most floating-point instructions operate on values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.



We considered a unified register file for both integer and floating-point values as this simplifies software register allocation and calling conventions, and reduces total user state. However, a split organization increases the total number of registers accessible with a given instruction width, simplifies provision of enough regfile ports for wide superscalar issue, supports decoupled floating-point-unit architectures, and simplifies use of internal floating-point encoding techniques. Compiler support and calling conventions for split register file architectures are well understood, and using dirty bits on floating-point register file state can reduce context-switch overhead.

Table 9. RISC-V standard F extension single-precision floating-point state

FLEN-1	0
f0	
f1	

FLEN-1	0
f2	
f3	
f4	
f5	
f6	
f7	
f8	
f9	
f10	
f11	
f12	
f13	
f14	
f15	
f16	
f17	
f18	
f19	
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	
FLEN	
31	0
fcsr	
32	

Floating-Point Control and Status Register

The floating-point control and status register, [fcsr](#), is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in [Floating-Point Control and Status Register](#).

Floating-point control and status register

Unresolved directive in rva.adoc - include::images/wavedrom/float-csr.adoc[]

The **fcsr** register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads **fcsr** by copying it into integer register *rd*. FSCSR swaps the value in **fcsr** by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into **fcsr**.

The fields within the **fcsr** can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field **frm** (**fcsr** bits 7–5) and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in **frm** by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into **frm**. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field **fflags** (**fcsr** bits 4–0).

Bits 31–8 of the **fcsr** are reserved for other standard extensions. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in **frm**. Rounding modes are encoded as shown in Table 10. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in **frm**. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111). Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

Table 10. Rounding mode encoding.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		<i>Reserved for future use.</i>
110		<i>Reserved for future use.</i>
111	DYN	In instruction's <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, <i>reserved</i> .



The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

The ratified version of the F spec mandated that an illegal-instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This

has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal-instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 11. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 11. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact



As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

A.5.3. NaN Generation and Propagation

Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN. The canonical NaN has a positive sign and all significand bits clear except the MSB, a.k.a. the quiet bit. For single-precision floating-point, this corresponds to the pattern **0x7fc00000**.



We considered propagating NaN payloads, as is recommended by the standard, but this decision would have increased hardware cost. Moreover, since this feature is optional in the standard, it cannot be used in portable code.

Implementors are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.



We require implementations to return the standard-mandated default values in the case of exceptional conditions, without any further intervention on the part of user-level software (unlike the Alpha ISA floating-point trap barriers). We believe full hardware handling of exceptional cases will become more common, and so wish to avoid complicating the user-level ISA to optimize other approaches. Implementations can always trap to machine-mode software handlers to provide exceptional default values.

A.5.4. Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

In the parlance of the IEEE standard, tininess is detected after rounding.



Detecting tininess after rounding results in fewer spurious underflow signals.

A.5.5. Instructions

The following instructions are added by this extension:

fmv.w.x	Single-precision floating-point move from integer
fmv.h.x	Half-precision floating-point move from integer

A.5.6. Parameters

This extension has the following implementation options:

MUTABLE_MISA_F

Indicates whether or not the F extension can be disabled with the misa.F bit.

DRAFT

A.6. M Extension

Integer multiply and divide instructions

Table 12. Status

Profile	v2.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.0

Ratification date

2019-12

A.6.1. Synopsis

This chapter describes the standard integer multiplication and division instruction extension, which is named M and contains instructions that multiply or divide values held in two integer registers.



We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

A.6.2. Instructions

The following instructions are added by this extension:

div	Signed division
divu	Unsigned division
divuw	Unsigned 32-bit division
divw	Signed 32-bit division
mul	Signed multiply
mulh	Signed multiply high
mulhsu	Signed/unsigned multiply high
mulhu	Unsigned multiply high
mulw	Signed 32-bit multiply
rem	Signed remainder
remu	Unsigned remainder
remuw	Unsigned 32-bit remainder
remw	Signed 32-bit remainder

A.6.3. Parameters

This extension has the following implementation options:

MUTABLE_MISA_M

Indicates whether or not the M extension can be disabled with the misa.M bit.

DRAFT

A.7. U Extension

User-level privilege mode

Table 13. Status

Profile	v1.12
RVA20S64	-
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.12

Ratification date

2019-12

A.7.1. Synopsys

User-level privilege mode

A.7.2. Parameters

This extension has the following implementation options:

MUTABLE_MISA_U

Indicates whether or not the U extension can be disabled with the misa.U bit.

UXLEN

Set of XLENs supported in U-mode. Can be one of:

- 32: SXLEN is always 32
- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via mstatus.UXL) between 32 and 64

U_MODE_ENDIANESS

Endianess of data in U-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field mstatus.UBE

A.8. Zicntr Extension

Architectural performance counters

Table 14. Status

Profile	v2.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

2.0

Ratification date
2019-12

A.8.1. Synopsys

Architectural performance counters

DRAFT

A.9. Ziccif Extension

Main memory fetch requirement for RVA profiles

Table 15. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

==== Synopsys

Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to min(ILEN,XLEN) (i.e., 32 bits for RVA22) are atomic.



This extension was ratified as part of the RVA20 profile.

DRAFT

A.10. Ziccrse Extension

Main memory reservability requirement for RVA profiles

Table 16. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

==== Synopsys

Main memory regions with both the cacheability and coherence PMAs must support RsrvEventual.



This extension was ratified as part of the RVA20 profile.

DRAFT

A.11. Ziccamoa Extension

Main memory atomicity requirement for RVA profiles

Table 17. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

==== Synopsys

Main memory regions with both the cacheability and coherence PMAs must support AMOArithmetic.



This extension was ratified as part of the RVA20 profile.

DRAFT

A.12. Za128rs Extension

Reservation set requirement for RVA profiles

Table 18. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

==== Synopsys

Reservation sets must be contiguous, naturally aligned, and at most 128 bytes in size.



This extension was ratified as part of the RVA20 profile.



The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

A.13. Zicclsm Extension

Main memory misaligned requirement for RVA profiles

Table 19. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	mandatory
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

==== Synopsys

Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.



This extension was ratified as part of the RVA20 profile.



This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

A.14. S Extension

Supervisor mode

Table 20. Status

Profile	v1.12
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	-

1.12

Ratification date

2019-12

A.14.1. Synopsis

This chapter describes the RISC-V supervisor-level architecture, which contains a common core that is used with various supervisor-level address translation and protection schemes.



Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization. In this spirit, certain supervisor-level facilities, including requests for timer and interprocessor interrupts, are provided by implementation-specific mechanisms. In some systems, a supervisor execution environment (SEE) provides these facilities in a manner specified by a supervisor binary interface (SBI). Other systems supply these facilities directly, through some other implementation-defined mechanism.

A.14.2. Instructions

The following instructions are added by this extension:

sfence.vma	Supervisor memory-management fence
sret	Supervisor Exception Return
hinvl.gvma	Invalidate cached address translations
hinvl.vvma	Invalidate cached address translations
sfence.inval.ir	Order implicit page table reads after invalidation
sfence.w.inval	Order writes before sfence
sinval.vma	Invalidate cached address translations

A.14.3. Parameters

This extension has the following implementation options:

ASID_WIDTH

Number of implemented ASID bits. Maximum is 16 for XLEN==64, and 9 for XLEN==32

MUTABLE_MISA_S

Indicates whether or not the S extension can be disabled with the `misa.S` bit.

REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION

When true, `stval` is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false `stval` is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_STVAL_ON_BREAKPOINT

When true, `stval` is written with the virtual PC of the EBREAK instruction (same information as `mepc`).

When false, `stval` is written with 0 on an EBREAK instruction.

Regardless, `stval` is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, `stval` is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, `stval` is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED

When true, `stval` is written with the virtual PC when an instruction fetch is misaligned.

When false, `stval` is written with 0 when an instruction fetch is misaligned.

Note that when `IALIGN=16` (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT

When true, `stval` is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, `stval` is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT

When true, `stval` is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, `stval` is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED

When true, `stval` is written with the virtual address of a load instruction when the address is misaligned and `MISALIGNED_LDST` is false.

When false, `stval` is written with 0 when a load address is misaligned and `MISALIGNED_LDST` is false.

REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT

When true, [stval](#) is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, [stval](#) is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT

When true, [stval](#) is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, [stval](#) is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED

When true, [stval](#) is written with the virtual address of a store instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, [stval](#) is written with 0 when a store address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT

When true, [stval](#) is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, [stval](#) is written with 0 when a store causes a **StoreAmoPageFault**.

SCOUNTENABLE_EN

Indicates which counters can be delegated via [scounteren](#)

An unimplemented counter cannot be specified, i.e., if HPM_COUNTER_EN[3] is false, it would be illegal to set SCOUNTENABLE_EN[3] to true.

SCOUNTENABLE_EN[0:2] must all be false if Zicntr is not implemented.
SCOUNTENABLE_EN[3:31] must all be false if Zihpm is not implemented.

STVAL_WIDTH

The number of implemented bits in [stval](#).

Must be greater than or equal to $\max(\text{PHYS_ADDR_WIDTH}, \text{VA_SIZE})$

STVEC_MODE_DIRECT

Whether or not stvec.MODE supports Direct (0).

STVEC_MODE_VECTORED

Whether or not stvec.MODE supports Vectored (1).

SV_MODE_BARE

Whether or not writing mode=Bare is supported in the [satp](#) register.

SXLEN

Set of XLENs supported in S-mode. Can be one of:

- 32: SXLEN is always 32

- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via mstatus.SXL) between 32 and 64

S_MODE_ENDIANESS

Endianess of data in S-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field mstatus.SBE

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY

For implementations that make [satp.MODE](#) read-only zero (always Bare, *i.e.*, no virtual translation is implemented), attempts to execute an SFENCE.VMA instruction might raise an illegal-instruction exception.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY indicates whether or not that exception occurs.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY has no effect when some virtual translation mode is supported.

DRAFT

A.15. Zifencei Extension

Instruction fence

Table 21. Status

Profile	v2.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

2.0

Ratification date

==== Synopsys

This chapter defines the "Zifencei" extension, which includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart. Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetches.



We considered but did not include a "store instruction word" instruction as in cite:[majc]. JIT compilers may generate a large trace of instructions before a single FENCE.I, and amortize any instruction cache snooping/invalidation overhead by writing translated instructions to memory regions that are known not to reside in the I-cache.



The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, or if the memory system consists of only uncached RAMs, then just the fetch pipeline needs to be flushed at a FENCE.I.

The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.

First, it has been recognized that on some systems, FENCE.I will be expensive to implement and alternate mechanisms are being discussed in the memory model task group. In particular, for designs that have an incoherent instruction cache and an incoherent data cache, or where the instruction cache refill does not snoop a coherent data cache, both caches must be completely flushed when a FENCE.I instruction is encountered. This problem is exacerbated when there are multiple levels of I and D cache in front of a unified cache or outer memory system.

Second, the instruction is not powerful enough to make available at user level in a Unix-like operating system environment. The FENCE.I only synchronizes the local hart, and the OS can reschedule the user hart to a different physical hart after the FENCE.I. This would

require the OS to execute an additional FENCE.I as part of every context migration. For this reason, the standard Linux ABI has removed FENCE.I from user-level and now requires a system call to maintain instruction-fetch coherence, which allows the OS to minimize the number of FENCE.I executions required on current systems and provides forward-compatibility with future improved instruction-fetch coherence mechanisms.

Future approaches to instruction-fetch coherence under discussion include providing more restricted versions of FENCE.I that only target a given address specified in rs1, and/or allowing software to use an ABI that relies on machine-mode cache-maintenance operations.

A.15.1. Instructions

The following instructions are added by this extension:

<code>fence.i</code>	Instruction fence
----------------------	-------------------

DRAFT

A.16. Svbare Extension

Bare virtual addressing

Table 22. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.0

Ratification date

==== Synopsys

This extension mandates that the [satp](#) mode Bare must be supported.



This extension was ratified as part of the RVA22 profile.

DRAFT

A.17. Sv39 Extension

39-bit virtual address translation (3 level)

Table 23. Status

Profile	v1.12
RVA20S64	-
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.12

Ratification date
unknown

Ratification document
github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

A.17.1. Synopsys

39-bit virtual address translation (3 level)

DRAFT

A.18. Svade Extension

Exception on PTE A/D Bits

Table 24. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.0

Ratification date

2023-11

Ratification document

github.com/riscvarchive/riscv-svadu/releases/download/v1.0/riscv-svadu.pdf

A.18.1. Synopsys

The Svade extension indicates that hardware does **not** update the A/D bits of a page table during a page walk. Rather, encountering a PTE with the A bit clear or the D bit clear when an operation is a write will cause a Page Fault.

A.19. Ssccptr Extension

Cacheable and coherent main memory page table reads

Table 25. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.0

Ratification date

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

A.19.1. Synopsys

Main memory regions with both the cacheability and coherence PMAs must support hardware page-table reads.



This extension was ratified with the RVA20 profiles.

A.20. Sstvecd Extension

Direct exception vectoring

Table 26. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.0

Ratification date

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

A.20.1. Synopsys

stvec.MODE must be capable of holding the value 0 (Direct). When stvec.MODE=Direct, stvec.BASE must be capable of holding any valid four-byte-aligned address.

A.21. Sstvala Extension

[stval](#) requirements for RVA profiles

Table 27. Status

Profile	v1.0
RVA20S64	mandatory
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.0

Ratification date

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

A.21.1. Synopsis

[stval](#) must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions other than those caused by execution of the [ebreak](#) or ``c.ebreak` instructions.

For virtual-instruction and illegal-instruction exceptions, [stval](#) must be written with the faulting instruction.



This extension was ratified with the RVA20 profiles.

A.22. Zihpm Extension

Programmable hardware performance counters

Table 28. Status

Profile	v2.0
RVA20S64	optional
RVA20U64	optional
RVA22S64	mandatory
RVA22U64	mandatory

2.0

Ratification date
unknown

A.22.1. Synopsys

Programmable hardware performance counters

A.22.2. Parameters

This extension has the following implementation options:

HPM_COUNTER_EN

List of HPM counters that are enabled. There is one entry for each hpmcounter.

The first three entries **must** be false (as they correspond to CY, IR, TM in, *e.g.* **mhmpcountinhibit**) Index 3 in HPM_COUNTER_EN corresponds to hpmcounter3. Index 31 in HPM_COUNTER_EN corresponds to hpmcounter31.

HPM_EVENTS

List of defined event numbers that can be written into hpmeventN

A.23. Sv48 Extension

48-bit virtual address translation (4 level)

Table 29. Status

Profile	v1.12
RVA20S64	-
RVA20U64	-
RVA22S64	-
RVA22U64	-

1.12

Ratification date

unknown

Ratification document

github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

A.23.1. Synopsys

48-bit virtual address translation (4 level)

A.24. Zihintpause Extension

PAUSE instruction

Table 30. Status

Profile	v2.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

2.0

Ratification date

==== Synopsys

The PAUSE instruction is a HINT that indicates the current hart’s rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero.



Software can use the PAUSE instruction to reduce energy consumption while executing spin-wait code sequences. Multithreaded cores might temporarily relinquish execution resources to other harts when PAUSE is executed. It is recommended that a PAUSE instruction generally be included in the code sequence for a spin-wait loop.

A future extension might add primitives similar to the x86 MONITOR/MWAIT instructions, which provide a more efficient mechanism to wait on writes to a specific memory location. However, these instructions would not supplant PAUSE. PAUSE is more appropriate when polling for non-memory events, when polling for multiple events, or when software does not know precisely what events it is polling for.

The duration of a PAUSE instruction’s effect may vary significantly within and among implementations. In typical implementations this duration should be much less than the time to perform a context switch, probably more on the rough order of an on-chip cache miss latency or a cacheless access to main memory.

A series of PAUSE instructions can be used to create a cumulative delay loosely proportional to the number of PAUSE instructions. In spin-wait loops in portable code, however, only one PAUSE instruction should be used before re-evaluating loop conditions, else the hart might stall longer than optimal on some implementations, degrading system performance.

PAUSE is encoded as a FENCE instruction with $pred=W$, $succ=0$, $fn=0$, $rd=x0$, and $rs1=x0$.



PAUSE is encoded as a hint within the FENCE opcode because some implementations are expected to deliberately stall the PAUSE instruction until outstanding memory transactions have completed. Because the successor set is null, however, PAUSE does not mandate any particular memory ordering—hence, it truly is a HINT.

Like other FENCE instructions, PAUSE cannot be used within LR/SC sequences without voiding the forward-progress guarantee.

The choice of a predecessor set of W is arbitrary, since the successor set is null. Other HINTs similar to PAUSE might be encoded with other predecessor sets.

DRAFT

A.25. Zba Extension

Address generation instructions

Table 31. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

2021-06

A.25.1. Synopsis

The Zba instructions can be used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and XLEN-sized indices: a shifted index is added to a base address.

The shift and add instructions do a left shift of 1, 2, or 3 because these are commonly found in real-world code and because they can be implemented with a minimal amount of additional hardware beyond that of the simple adder. This avoids lengthening the critical path in implementations.

While the shift and add instructions are limited to a maximum left shift of 3, the [slli](#) instruction (from the base ISA) can be used to perform similar shifts for indexing into arrays of wider elements. The [slli.uw](#) — added in this extension — can be used when the index is to be interpreted as an unsigned word.

A.25.2. Instructions

The following instructions are added by this extension:

add.uw	Add unsigned word
sh1add.uw	Shift unsigned word left by 1 and add
sh1add	Shift left by 1 and add
sh2add.uw	Shift unsigned word left by 2 and add
sh2add	Shift left by 2 and add
sh3add.uw	Shift unsigned word left by 3 and add
sh3add	Shift left by 3 and add
slli.uw	Shift left unsigned word (Immediate)

A.26. Zbb Extension

Basic bit manipulation

Table 32. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

2021-06

A.26.1. Synopsys

Basic bit manipulation

A.26.2. Instructions

The following instructions are added by this extension:

andn	AND with inverted operand
clz	Count leading zero bits
clzw	Count leading zero bits in word
cpop	Count set bits
cpopw	Count set bits in word
ctz	Count trailing zero bits
ctzw	Count trailing zero bits in word
max	Maximum
maxu	Unsigned maximum
min	Minimum
minu	Unsigned minumum
orc.b	Bitware OR-combine, byte granule
orn	OR with inverted operand
rev8	Byte-reverse register (RV64 encoding)
rol	Rotate left (Register)
rolw	Rotate left word (Register)
ror	Rotate right (Register)
rori	Rotate right (Immediate)
roriw	Rotate right word (Immediate)
rorw	Rotate right word (Register)

sext.b	Sign-extend byte
sext.h	Sign-extend halfword
xnor	Exclusive NOR
zext.h	Zero-extend halfword

DRAFT

A.27. Zbs Extension

Single-bit instructions

Table 33. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

2021-06

Ratification document

drive.google.com/drive/u/0/folders/1_wqb-rXOVkGa6rqmugN3kwCftWDf1daU

A.27.1. Synopsys

The single-bit instructions provide a mechanism to set, clear, invert, or extract a single bit in a register. The bit is specified by its index

A.27.2. Instructions

The following instructions are added by this extension:

<code>bclr</code>	Single-Bit clear (Register)
<code>bclri</code>	Single-Bit clear (Immediate)
<code>bext</code>	Single-Bit extract (Register)
<code>bexti</code>	Single-Bit extract (Immediate)
<code>binv</code>	Single-Bit invert (Register)
<code>binvi</code>	Single-Bit invert (Immediate)
<code>bset</code>	Single-Bit set (Register)
<code>bseti</code>	Single-Bit set (Immediate)

A.28. Zic64b Extension

64-byte cache blocks

Table 34. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

A.28.1. Synopsys

Cache blocks must be 64 bytes in size, naturally aligned in the address space.



This extension was ratified with the RVA20 profiles.

A.29. Zicbom Extension

Cache block management instructions

Table 35. Status

Profile	v1.0.1-b34ea8a
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0.1-b34ea8a

Ratification date

2022-05

A.29.1. Synopsys

Cache block management instructions

A.29.2. Instructions

The following instructions are added by this extension:

cbo.clean	Cache Block Clean
cbo.flush	Cache Block Flush
cbo.inval	Cache Block Invalidate

A.29.3. Parameters

This extension has the following implementation options:

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

A.30. Zicbop Extension

Cache block prefetch

Table 36. Status

Profile	v1.0.1-b34ea8a
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0.1-b34ea8a

Ratification date
2022-05

A.30.1. Synopsys

Cache block prefetch instruction

A.30.2. Parameters

This extension has the following implementation options:

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

A.31. Zicboz Extension

Cache block zero instruction

Table 37. Status

Profile	v1.0.1-b34ea8a
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0.1-b34ea8a

Ratification date

2022-05

A.31.1. Synopsys

Cache block zero instruction

A.31.2. Instructions

The following instructions are added by this extension:

cbo.zero	Cache Block Zero
--------------------------	------------------

A.31.3. Parameters

This extension has the following implementation options:

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

A.32. Zfhmin Extension

Minimal half-precision Floating-point

Table 38. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	mandatory

1.0

Ratification date

2021-11

A.32.1. Synopsis

Zfhmin provides minimal support for 16-bit half-precision binary floating-point instructions. The Zfhmin extension is a subset of the **Zfh** extension, consisting only of data transfer and conversion instructions. Like **Zfh**, the Zfhmin extension depends on the single-precision floating-point extension, **F**. The expectation is that Zfhmin software primarily uses the half-precision format for storage, performing most computation in higher precision.

The Zfhmin extension includes the following instructions from the **Zfh** extension: [flh](#), [fsh](#), [fmv.x.h](#), [fmv.h.x](#), [fcvt.s.h](#), and [fcvt.h.s](#). If the **D** extension is present, the [fcvt.d.h](#) and [fcvt.h.d](#) instructions are also included. If the **Q** extension is present, the [fcvt.q.h](#) and [fcvt.h.q](#) instructions are additionally included.

*Zfhmin does not include the **fsgnj.h** instruction, because it suffices to instead use the **fsgnj.s** instruction to move half-precision values between floating-point registers.*

Half-precision addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the half-precision operands to single-precision, performing the operation using single-precision arithmetic, then converting back to half-precision. cite:[roux:hal-01091186] Performing half-precision fused multiply-addition using this method incurs a 1-ulp error on some inputs for the RNE and RMM rounding modes.

*Conversion from 8- or 16-bit integers to half-precision can be emulated by first converting to single-precision, then converting to half-precision. Conversion from 32-bit integer can be emulated by first converting to double-precision. If the **D** extension is not present and a 1-ulp error under RNE or RMM is tolerable, 32-bit integers can be first converted to single-precision instead. The same remark applies to conversions from 64-bit integers without the **Q** extension.*

A.32.2. Instructions

The following instructions are added by this extension:

fcvt.h.s	Convert half-precision float to a single-precision float
fcvt.s.h	Convert single-precision float to a half-precision float
flh	Half-precision floating-point load
fmv.x.h	Move half-precision value from floating-point to integer register
fsh	Half-precision floating-point store

DRAFT

A.33. Zkt Extension

Data-independent execution latency

Table 39. Status

Profile	v1.0	v1.0.1
RVA20S64	-	-
RVA20U64	-	-
RVA22S64	mandatory	mandatory
RVA22U64	mandatory	mandatory

1.0

Ratification date
2021-11

1.0.1

Ratification date
Changes

[“Fix typos to show that **c.srli**, **c.srai**, and **c.slli** are Zkt instructions in RV64.”]

A.33.1. Synopsis

The Zkt extension attests that the machine has data-independent execution time for a safe subset of instructions. This property is commonly called "constant-time" although should not be taken with that literal meaning.

All currently proposed cryptographic instructions (scalar **K** extension) are on this list, together with a set of relevant supporting instructions from I, M, C, and B extensions.



Failure to prevent leakage of sensitive parameters via the direct timing channel is considered a serious security vulnerability and will typically result in a CERT CVE security advisory.

Scope and Goal

An "ISA contract" is made between a programmer and the RISC-V implementation that Zkt instructions do not leak information about processed secret data (plaintext, keying information, or other "sensitive security parameters" — FIPS 140-3 term) through differences in execution latency. Zkt does *not* define a set of instructions available in the core; it just restricts the behaviour of certain instructions if those are implemented.

Currently, the scope of this document is within scalar RV32/RV64 processors. Vector cryptography instructions (and appropriate vector support instructions) will be added later, as will other security-related functions that wish to assert leakage-free execution latency properties.

Loads, stores, conditional branches are excluded, along with a set of instructions that are rarely necessary to process secret data. Also excluded are instructions for which workarounds exist in standard cryptographic middleware due to the limitations of other ISA processors.

The stated goal is that OpenSSL, BoringSSL (Android), the Linux Kernel, and similar trusted software will not have directly observable timing side channels when compiled and running on a Zkt-enabled RISC-V target. The Zkt extension explicitly states many of the common latency assumptions made by cryptography developers.

Vendors do not have to implement all of the list's instructions to be Zkt compliant; however, if they claim to have Zkt and implement any of the listed instructions, it must have data-independent latency.

For example, many simple RV32I and RV64I cores (without Multiply, Compressed, Bitmanip, or Cryptographic extensions) are technically compliant with Zkt. A constant-time AES can be implemented on them using "bit-slice" techniques, but it will be excruciatingly slow when compared to implementation with AES instructions. There are no guarantees that even a bit-sliced cipher implementation (largely based on boolean logic instructions) is secure on a core without Zkt attestation.

Out-of-order implementations adhering to Zkt are still free to fuse, crack, change or even ignore sequences of instructions, so long as the optimisations are applied deterministically, and not based on operand data. The guiding principle should be that no information about the data being operated on should be leaked based on the execution latency.



It is left to future extensions or other techniques to tackle the problem of data-independent execution in implementations which advanced out-of-order capabilities which use value prediction, or which are otherwise data-dependent.



Note to software developers

Programming techniques can only mitigate leakage directly caused by arithmetic, caches, and branches. Other ISAs have had micro-architectural issues such as Spectre, Meltdown, Speculative Store Bypass, Rogue System Register Read, Lazy FP State Restore, Bounds Check Bypass Store, TLBleed, and LITF/Foreshadow, etc. See e.g. [NSA Hardware and Firmware Security Guidance](#)

It is not within the remit of this proposal to mitigate these micro-architectural leakages.

Background

- Timing attacks are much more powerful than was realised before the 2010s, which has led to a significant mitigation effort in current cryptographic code-bases.
- Cryptography developers use static and dynamic security testing tools to trace the handling of secret information and detect occasions where it influences a branch or is used for a table lookup.
- Architectural testing for Zkt can be pragmatic and semi-formal; *security by design* against basic timing attacks can usually be achieved via conscious implementation (of relevant iterative multi-cycle instructions or instructions composed of micro-ops) in way that avoids data-dependent latency.
- Laboratory testing may utilize statistical timing attack leakage analysis techniques such as those described in ISO/IEC 17825 cite:[IS16].
- Binary executables should not contain secrets in the instruction encodings (Kerckhoffs's principle), so instruction timing may leak information about immediates, ordering of input registers, etc. There may be an exception to this in systems where a binary loader modifies the executable for purposes of relocation – and it is desirable to keep the execution location (PC)

secret. This is why instructions such as LUI, AUIPC, and ADDI are on the list.

- The rules used by audit tools are relatively simple to understand. Very briefly; we call the plaintext, secret keys, expanded keys, nonces, and other such variables "secrets". A secret variable (arithmetically) modifying any other variable/register turns that into a secret too. If a secret ends up in address calculation affecting a load or store, that is a violation. If a secret affects a branch's condition, that is also a violation. A secret variable location or register becomes a non-secret via specific zeroization/sanitisation or by being declared ciphertext (or otherwise no-longer-secret information). In essence, secrets can only "touch" instructions on the Zkt list while they are secrets.

Specific Instruction Rationale

- HINT instruction forms (typically encodings with **rd=x0**) are excluded from the data-independent time requirement.
- Floating point (F, D, Q, L extensions) are currently excluded from the constant-time requirement as they have very few applications in standardised cryptography. We may consider adding floating point add, sub, multiply as a constant time requirement for some floating point extension in case a specific algorithm (such as the PQC Signature algorithm Falcon) becomes critical.
- Cryptographers typically assume division to be variable-time (while multiplication is constant time) and implement their Montgomery reduction routines with that assumption.
- Zicsr, Zifencei are excluded.
- Some instructions are on the list simply because we see no harm in including them in testing scope.

Programming Information

For background information on secure programming "models", see:

- Thomas Pornin: "Why Constant-Time Crypto?" (A great introduction to timing assumptions.) www.bearssl.org/constanttime.html
- Jean-Philippe Aumasson: "Guidelines for low-level cryptography software." (A list of recommendations.) github.com/veorq/cryptocoding
- Peter Schwabe: "Timing Attacks and Countermeasures." (Lecture slides — nice references.) summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf
- Adam Langley: "ctgrind." (This is from 2010 but is still relevant.) www.imperialviolet.org/2010/04/01/ctgrind.html
- Kris Kwiakowski: "Constant-time code verification with Memory Sanitizer." www.amongbytes.com/post/20210709-testing-constant-time/
- For early examples of timing attack vulnerabilities, see www.kb.cert.org/vuls/id/997481 and related academic papers.

Zkt listings

The following instructions are included in the Zkt subset. They are listed here grouped by their original parent extension.



Note to implementers

You do not need to implement all of these instructions to implement Zkt. Rather, every one of these instructions that the core does implement must adhere to the requirements of Zkt.

RVI (Base Instruction Set)

Only basic arithmetic and **slt*** (for carry computations) are included. The data-independent timing requirement does not apply to HINT instruction encoding forms of these instructions.

RV32	RV64	Mnemonic	Instruction
✓	✓	lui <i>rd, imm</i>	'lui'
✓	✓	auipc <i>rd, imm</i>	'auipc'
✓	✓	addi <i>rd, rs1, imm</i>	'addi'
✓	✓	slti <i>rd, rs1, imm</i>	'slti'
✓	✓	sltiu <i>rd, rs1, imm</i>	'sltiu'
✓	✓	xori <i>rd, rs1, imm</i>	'xori'
✓	✓	ori <i>rd, rs1, imm</i>	'ori'
✓	✓	andi <i>rd, rs1, imm</i>	'andi'
✓	✓	slli <i>rd, rs1, imm</i>	'slli'
✓	✓	srli <i>rd, rs1, imm</i>	'srli'
✓	✓	srai <i>rd, rs1, imm</i>	'srai'
✓	✓	add <i>rd, rs1, rs2</i>	'add'
✓	✓	sub <i>rd, rs1, rs2</i>	'sub'
✓	✓	sll <i>rd, rs1, rs2</i>	'sll'
✓	✓	slt <i>rd, rs1, rs2</i>	'slt'
✓	✓	sltu <i>rd, rs1, rs2</i>	'sltu'
✓	✓	xor <i>rd, rs1, rs2</i>	'xor'
✓	✓	srl <i>rd, rs1, rs2</i>	'srl'
✓	✓	sra <i>rd, rs1, rs2</i>	'sra'
✓	✓	or <i>rd, rs1, rs2</i>	'or'
✓	✓	and <i>rd, rs1, rs2</i>	'and'
	✓	addiw <i>rd, rs1, imm</i>	'addiw'
	✓	slliw <i>rd, rs1, imm</i>	'slliw'
	✓	srliw <i>rd, rs1, imm</i>	'srliw'
	✓	sraiw <i>rd, rs1, imm</i>	'sraiw'
	✓	addw <i>rd, rs1, rs2</i>	'addw'
	✓	subw <i>rd, rs1, rs2</i>	'subw'
	✓	sllw <i>rd, rs1, rs2</i>	'sllw'
	✓	srlw <i>rd, rs1, rs2</i>	'srlw'
	✓	sraw <i>rd, rs1, rs2</i>	'sraw'

RVM (Multiply)

Multiplication is included; division and remaindering excluded.

RV32	RV64	Mnemonic	Instruction
✓	✓	<code>mul rd, rs1, rs2</code>	'mul'
✓	✓	<code>mulh rd, rs1, rs2</code>	'mulh'
✓	✓	<code>mulhsu rd, rs1, rs2</code>	'mulhsu'
✓	✓	<code>mulhu rd, rs1, rs2</code>	'mulhu'
	✓	<code>mulw rd, rs1, rs2</code>	'mulw'

RVC (Compressed)

Same criteria as in RVI. Organised by quadrants.

RV32	RV64	Mnemonic	Instruction
✓	✓	<code>c.nop</code>	'c_nop'
✓	✓	<code>c.addi</code>	'c_addi'
	✓	<code>c.addiw</code>	'c_addiw'
✓	✓	<code>c.lui</code>	'c_lui'
✓	✓	<code>c.srli</code>	'c_srli'
✓	✓	<code>c.srai</code>	'c_srai'
✓	✓	<code>c.andi</code>	'c_andi'
✓	✓	<code>c.sub</code>	'c_sub'
✓	✓	<code>c.xor</code>	'c_xor'
✓	✓	<code>c.or</code>	'c_or'
✓	✓	<code>c.and</code>	'c_and'
	✓	<code>c.subw</code>	'c_subw'
	✓	<code>c.addw</code>	'c_addw'
✓	✓	<code>c.slli</code>	'c_slli'
✓	✓	<code>c.mv</code>	'c_mv'
✓	✓	<code>c.add</code>	'c_add'

RVK (Scalar Cryptography)

All K-specific instructions are included. Additionally, **seed** CSR latency should be independent of **ES16** state output **entropy** bits, as that is a sensitive security parameter. See <<crypto_scalar_appx_es_access'.

RV32	RV64	Mnemonic	Instruction
✓		<code>aes32dsi</code>	'aes32dsi'
✓		<code>aes32dsmi</code>	'aes32dsmi'
✓		<code>aes32esi</code>	'aes32esi'
✓		<code>aes32esmi</code>	'aes32esmi'

RV32	RV64	Mnemonic	Instruction
	✓	aes64ds	'aes64ds'
	✓	aes64dsm	'aes64dsm'
	✓	aes64es	'aes64es'
	✓	aes64esm	'aes64esm'
	✓	aes64im	'aes64im'
	✓	aes64ksli	'aes64ksli'
	✓	aes64ks2	'aes64ks2'
✓	✓	sha256sig0	'sha256sig0'
✓	✓	sha256sig1	'sha256sig1'
✓	✓	sha256sum0	'sha256sum0'
✓	✓	sha256sum1	'sha256sum1'
✓		sha512sig0h	'sha512sig0h'
✓		sha512sig0l	'sha512sig0l'
✓		sha512sig1h	'sha512sig1h'
✓		sha512sig1l	'sha512sig1l'
✓		sha512sum0r	'sha512sum0r'
✓		sha512sum1r	'sha512sum1r'
	✓	sha512sig0	'sha512sig0'
	✓	sha512sig1	'sha512sig1'
	✓	sha512sum0	'sha512sum0'
	✓	sha512sum1	'sha512sum1'
✓	✓	sm3p0	'sm3p0'
✓	✓	sm3p1	'sm3p1'
✓	✓	sm4ed	'sm4ed'
✓	✓	sm4ks	'sm4ks'

RVB (Bitmanip)

The **Zbkb**, **Zbkci** and **Zbkx** extensions are included in their entirety.



Note to implementers

Recall that **rev**, **zip** and **unzip** are pseudoinstructions representing specific instances of **grevi**, **shfli** and **unshfli** respectively.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul	'clmul-sc'
✓	✓	clmulh	'clmulh-sc'
✓	✓	xperm4	'xperm4-sc'
✓	✓	xperm8	'xperm8-sc'
✓	✓	ror	'ror-sc'

RV32	RV64	Mnemonic	Instruction
✓	✓	rol	'rol-sc'
✓	✓	rori	'rori-sc'
	✓	rorw	'rorw-sc'
	✓	rolw	'rolw-sc'
	✓	roriw	'roriw-sc'
✓	✓	andn	'andn-sc'
✓	✓	orn	'orn-sc'
✓	✓	xnor	'xnor-sc'
✓	✓	pack	'pack-sc'
✓	✓	packh	'packh-sc'
	✓	packw	'packw-sc'
✓	✓	brev8	'brev8-sc'
✓	✓	rev8	'rev8-sc'
✓		zip	'zip-sc'
✓		unzip	'unzip-sc'

DRAFT

A.34. Sscouterenw Extension

Supervisor counter enable

Table 40. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	-

1.0

Ratification date

2023-08

Ratification document

drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGlMz6Cbo/view?usp=drive_link

A.34.1. Synopsys

For any hpmcounter that is not read-only zero, the corresponding bit in `scounteren` must be writable.



This extension was ratified with the RVA22 profiles.

A.35. Svpbmt Extension

Page-based memory types

Table 41. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	-

1.0

Ratification date

==== Synopsys

This extension mandates that the [satp](#) mode Bare must be supported.



This extension was ratified as part of the RVA22 profile.

DRAFT

A.36. Svinval Extension

Fine-grained address-translation cache invalidation

Table 42. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	mandatory
RVA22U64	-

1.0

Ratification date

2021-11

A.36.1. Synopsis

The Svinval extension splits [sfence.vma](#), [hfence.vma](#), and [hfence.gvma](#) instructions into finer-grained invalidation and ordering operations that can be more efficiently batched or pipelined on certain classes of high-performance implementation.

The [sinval.vma](#) instruction invalidates any address-translation cache entries that an [sfence.vma](#) instruction with the same values of *rs1* and *rs2* would invalidate. However, unlike [sfence.vma](#), [sinval.vma](#) instructions are only ordered with respect to [sfence.vma](#), [sfence.w.inval](#), and [sfence.inval.ir](#) instructions as defined below.

The [sfence.w.inval](#) instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent [sinval.vma](#) instructions executed by the same hart. The [sfence.inval.ir](#) instruction guarantees that any previous [sinval.vma](#) instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

When executed in order (but not necessarily consecutively) by a single hart, the sequence [sfence.w.inval](#), [sinval.vma](#), and [sfence.inval.ir](#) has the same effect as a hypothetical [sfence.vma](#) instruction in which:

- the values of *rs1* and *rs2* for the [sfence.vma](#) are the same as those used in the [sinval.vma](#),
- reads and writes prior to the [sfence.w.inval](#) are considered to be those prior to the [sfence.vma](#), and
- reads and writes following the [sfence.inval.ir](#) are considered to be those subsequent to the [sfence.vma](#).

If the hypervisor extension is implemented, the Svinval extension also provides two additional instructions: [hinval.vvma](#) and [hinval.gvma](#). These have the same semantics as [sinval.vma](#), except that they combine with [sfence.w.inval](#) and [sfence.inval.ir](#) to replace [hfence.vvma](#) and [hfence.gvma](#), respectively, instead of [sfence.vma](#). In addition, [hinval.gvma](#) uses VMIDs instead of ASIDs.

[sinval.vma](#), [hinval.vvma](#), and [hinval.gvma](#) require the same permissions and raise the same exceptions as [sfence.vma](#), [hfence.vvma](#), and [hfence.gvma](#), respectively. In particular, an attempt to execute any of these instructions in U-mode always raises an `IllegalInstruction` exception, and an attempt to

execute `sinval.vma` or `hinval.gvma` in S-mode or HS-mode when `mstatus.TVM=1` also raises an `IllegalInstruction` exception. An attempt to execute `hinval.vvma` or `hinval.gvma` in VS-mode or VU-mode, or to execute `sinval.vma` in VU-mode, raises a `VirtualInstruction` exception. When `hstatus.VTVM=1`, an attempt to execute `sinval.vma` in VS-mode also raises a `VirtualInstruction` exception.

Attempting to execute `sfence.w.inval` or `sfence.inval.ir` in U-mode raises an `IllegalInstruction` exception. Doing so in VU-mode raises a `VirtualInstruction` exception. `sfence.w.inval` and `sfence.inval.ir` are unaffected by the `mstatus.TVM` and `hstatus.VTVM` fields and hence are always permitted in S-mode and VS-mode.

`sfence.w.inval` and `sfence.inval.ir` instructions do not need to be trapped when `mstatus.TVM=1` or when `hstatus.VTVM=1`, as they only have ordering effects but no visible side effects. Trapping of the `sinval.vma` instruction is sufficient to enable emulation of the intended overall TLB maintenance functionality.

In typical usage, software will invalidate a range of virtual addresses in the address-translation caches by executing an `sfence.w.inval` instruction, executing a series of `sinval.vma`, `hinval.vvma`, or `hinval.gvma` instructions to the addresses (and optionally ASIDs or VMIDs) in question, and then executing an `sfence.inval.ir` instruction.

High-performance implementations will be able to pipeline the address-translation cache invalidation operations, and will defer any pipeline stalls or other memory ordering enforcement until an `sfence.w.inval`, `sfence.inval.ir`, `sfence.vma`, `hfence.gvma`, or `hfence.vvma` instruction is executed.

Simpler implementations may implement `sinval.vma`, `hinval.vvma`, and `hinval.gvma` identically to `sfence.vma`, `hfence.vvma`, and `hfence.gvma`, respectively, while implementing `sfence.w.inval` and `sfence.inval.ir` instructions as no-ops.

A.36.2. Instructions

The following instructions are added by this extension:

<code>hinval.gvma</code>	Invalidate cached address translations
<code>hinval.vvma</code>	Invalidate cached address translations
<code>sfence.inval.ir</code>	Order implicit page table reads after invalidation
<code>sfence.w.inval</code>	Order writes before sfence
<code>sinval.vma</code>	Invalidate cached address translations

A.37. Sv57 Extension

57-bit virtual address translation (5 level)

Table 43. Status

Profile	v1.12
RVA20S64	-
RVA20U64	-
RVA22S64	optional
RVA22U64	-

1.12

Ratification date

unknown

Ratification document

github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

A.37.1. Synopsys

57-bit virtual address translation (5 level)

A.38. Sstc Extension

Supervisor mode timer interrupts

Table 44. Status

Profile	v0.9
RVA20S64	-
RVA20U64	-
RVA22S64	-
RVA22U64	-

0.9

Ratification date

Ratification document

drive.google.com/file/d/1m84Re2yK8m_vbW7TspvevCDR82MOBaSX/view?usp=drive_link

A.38.1. Synopsys

Supervisor mode timer interrupts

DRAFT

A.39. Sscofpmf Extension

Counter Overflow and Privilege Mode Filtering

Table 45. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	optional
RVA22U64	-

1.0

Ratification date

2023-08

Ratification document

drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGlMz6Cbo/view?usp=drive_link

A.39.1. Synopsys

Counter Overflow and Privilege Mode Filtering

DRAFT

A.40. H Extension

Hypervisor

Table 46. Status

Profile	v1.0
RVA20S64	-
RVA20U64	-
RVA22S64	optional
RVA22U64	-

1.0

Ratification date

2019-12

A.40.1. Synopsys

This chapter describes the RISC-V hypervisor extension, which virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor. The hypervisor extension changes supervisor mode into *hypervisor-extended supervisor mode* (HS-mode, or *hypervisor mode* for short), where a hypervisor or a hosting-capable operating system runs. The hypervisor extension also adds another stage of address translation, from *guest physical addresses* to supervisor physical addresses, to virtualize the memory and memory-mapped I/O subsystems for a guest operating system. HS-mode acts the same as S-mode, but with additional instructions and CSRs that control the new stage of address translation and support hosting a guest OS in virtual S-mode (VS-mode). Regular S-mode operating systems can execute without modification either in HS-mode or as VS-mode guests.

In HS-mode, an OS or hypervisor interacts with the machine through the same SBI as an OS normally does from S-mode. An HS-mode hypervisor is expected to implement the SBI for its VS-mode guest.

The hypervisor extension depends on an "I" base integer ISA with 32 **x** registers (RV32I or RV64I), not RV32E or RV64E, which have only 16 **x** registers. CSR [mtval](#) must not be read-only zero, and standard page-based address translation must be supported, either Sv32 for RV32, or a minimum of Sv39 for RV64.

The hypervisor extension is enabled by setting bit 7 in the [misa](#) CSR, which corresponds to the letter H. RISC-V harts that implement the hypervisor extension are encouraged not to hardwire [misa](#)[7], so that the extension may be disabled.



The baseline privileged architecture is designed to simplify the use of classic virtualization techniques, where a guest OS is run at user-level, as the few privileged instructions can be easily detected and trapped. The hypervisor extension improves virtualization performance by reducing the frequency of these traps.

The hypervisor extension has been designed to be efficiently emulable on platforms that do not implement the extension, by running the hypervisor in S-mode and trapping into M-mode for hypervisor CSR accesses and to maintain shadow page tables. The majority of CSR accesses for type-2 hypervisors are valid S-mode accesses so need not be trapped. Hypervisors can support nested virtualization analogously.

A.40.2. Privilege Modes

The current *virtualization mode*, denoted V , indicates whether the hart is currently executing in a guest. When $V=1$, the hart is either in virtual S-mode (VS-mode), or in virtual U-mode (VU-mode) atop a guest OS running in VS-mode. When $V=0$, the hart is either in M-mode, in HS-mode, or in U-mode atop an OS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active ($V=1$) or inactive ($V=0$). [Table 47](#) lists the possible privilege modes of a RISC-V hart with the hypervisor extension.

DRAFT

Table 47. Privilege modes with the hypervisor extension.

Virtualization Mode (V)	Nominal Privilege	Abbreviation	Name	Two-Stage Translation
0	U	U-mode	User mode	Off
0	S	HS-mode	Hypervisor-extended	Off
0	M	M-mode	supervisor mode Machine mode	Off
1	U	VU-mode	Virtual user mode	On
1	S	VS-mode	Virtual supervisor mode	On

For privilege modes U and VU, the *nominal privilege mode* is U, and for privilege modes HS and VS, the nominal privilege mode is S.

HS-mode is more privileged than VS-mode, and VS-mode is more privileged than VU-mode. VS-mode interrupts are globally disabled when executing in U-mode.



This description does not consider the possibility of U-mode or VU-mode interrupts and will be revised if an extension for user-level interrupts is adopted.

A.40.3. Parameters

This extension has the following implementation options:

GSTAGE_MODE_BARE

Whether or not writing mode=Bare is supported in the [hgap](#) register.

HCOUNTENABLE_EN

Indicates which counters can be delegated via [hcounteren](#)

An unimplemented counter cannot be specified, i.e., if `HPM_COUNTER_EN[3]` is false, it would be illegal to set `HCOUNTENABLE_EN[3]` to true.

`HCOUNTENABLE_EN[0:2]` must all be false if `Zicntr` is not implemented.
`HCOUNTENABLE_EN[3:31]` must all be false if `Zihpm` is not implemented.

MUTABLE_MISA_H

Indicates whether or not the H extension can be disabled with the `misa.H` bit.

NUM_EXTERNAL_GUEST_INTERRUPTS

Number of supported virtualized guest interrupts

Corresponds to the **GEILEN** parameter in the RVI specs

REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION

When true, [vstval](#) is written with the encoding of an instruction that causes an **IllegalInstruction** exception.

When false [vstval](#) is written with 0 when an **IllegalInstruction** exception occurs.

REPORT_VA_IN_VSTVAL_ON_BREAKPOINT

When true, **vstval** is written with the virtual PC of the EBREAK instruction (same information as **mepc**).

When false, **vstval** is written with 0 on an EBREAK instruction.

Regardless, **vstval** is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT

When true, **vstval** is written with the virtual PC of an instruction when fetch causes an **InstructionAccessFault**.

When false, **vstval** is written with 0 when an instruction fetch causes an **InstructionAccessFault**.

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED

When true, **vstval** is written with the virtual PC when an instruction fetch is misaligned.

When false, **vstval** is written with 0 when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the **Zc*** extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT

When true, **vstval** is written with the virtual PC of an instruction when fetch causes an **InstructionPageFault**.

When false, **vstval** is written with 0 when an instruction fetch causes an **InstructionPageFault**.

REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT

When true, **vstval** is written with the virtual address of a load when it causes a **LoadAccessFault**.

When false, **vstval** is written with 0 when a load causes a **LoadAccessFault**.

REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED

When true, **vstval** is written with the virtual address of a load instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, **vstval** is written with 0 when a load address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT

When true, **vstval** is written with the virtual address of a load when it causes a **LoadPageFault**.

When false, **vstval** is written with 0 when a load causes a **LoadPageFault**.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT

When true, **vstval** is written with the virtual address of a store when it causes a **StoreAmoAccessFault**.

When false, `vstval` is written with 0 when a store causes a **StoreAmoAccessFault**.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED

When true, `vstval` is written with the virtual address of a store instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, `vstval` is written with 0 when a store address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT

When true, `vstval` is written with the virtual address of a store when it causes a **StoreAmoPageFault**.

When false, `vstval` is written with 0 when a store causes a **StoreAmoPageFault**.

SV32X4_TRANSLATION

Whether or not Sv32x4 translation mode is supported.

SV39X4_TRANSLATION

Whether or not Sv39x4 translation mode is supported.

SV48X4_TRANSLATION

Whether or not Sv48x4 translation mode is supported.

SV57X4_TRANSLATION

Whether or not Sv57x4 translation mode is supported.

VMID_WIDTH

Number of bits supported in hcatp.VMID (i.e., the supported width of a virtual machine ID).

VXLEN

Set of XLENs supported in VS-mode. Can be one of:

- 32: VXLEN is always 32
- 64: VXLEN is always 64
- 3264: VXLEN can be changed (via hstatus.VSXL) between 32 and 64

VS_MODE_ENDIANESS

Endianess of data in VS-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field hstatus.VSBE

VUXLEN

Set of XLENs supported in VU-mode. Can be one of:

- 32: VUXLEN is always 32

- 64: VUXLEN is always 64
- 3264: VUXLEN can be changed (via `vsstatus.UXL`) between 32 and 64

VU_MODE_ENDIANESS

Endianess of data in VU-mode. Can be one of:

- little: M-mode data is always little endian
- big: M-mode data is always big endian
- dynamic: M-mode data can be either little or big endian, depending on the CSR field `vsstatus.UBE`

DRAFT

Appendix B: Instruction Specifications

DRAFT

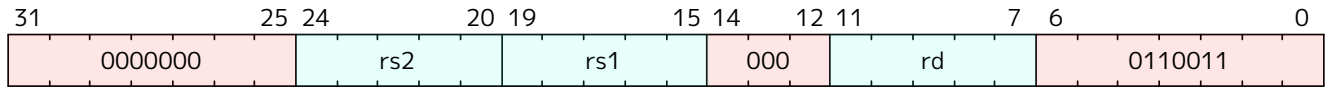
B.1. add

Integer add

This instruction is defined by:

- I, >= 0

B.1.1. Encoding



B.1.2. Synopsis

Add the value in rs1 to rs2, and store the result in rd. Any overflow is thrown away.

B.1.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.1.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.1.5. Execution

```
X[rd] = X[rs1] + X[rs2];
```

B.1.6. Exceptions

DRAFT

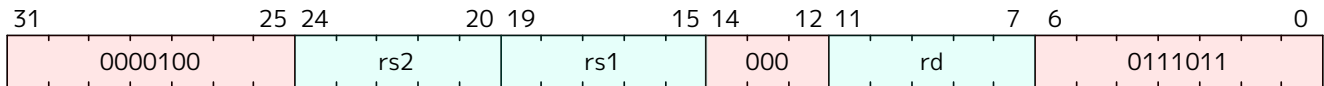
B.2. add.uw

Add unsigned word

This instruction is defined by any of the following:

- B, ≥ 0
- Zba, ≥ 0

B.2.1. Encoding



B.2.2. Synopsis

This instruction performs an XLEN-wide addition between rs2 and the zero-extended least-significant word of rs1.

B.2.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.2.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.2.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + X[rs1][31:0];

```


B.2.6. Exceptions

DRAFT

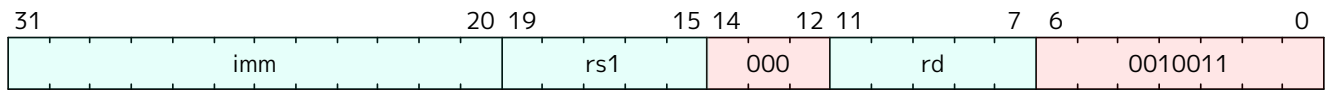
B.3. addi

Add immediate

This instruction is defined by:

- $I, \geq 0$

B.3.1. Encoding



B.3.2. Synopsis

Add an immediate to the value in rs1, and store the result in rd

B.3.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.3.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.3.5. Execution

```
X[rd] = X[rs1] + imm;
```

B.3.6. Exceptions

DRAFT

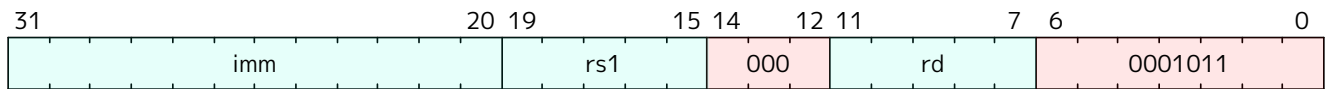
B.4. addiw

Add immediate word

This instruction is defined by:

- $I, \geq 0$

B.4.1. Encoding



B.4.2. Synopsis

Add an immediate to the 32-bit value in rs1, and store the sign extended result in rd

B.4.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.4.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.4.5. Execution

```

XReg operand = sext(X[rs1], 31);
X[rd] = sext(operand + imm, 31);

```

B.4.6. Exceptions

DRAFT

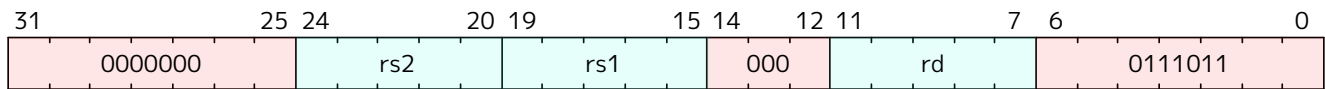
B.5. addw

Add word

This instruction is defined by:

- $I, \geq 0$

B.5.1. Encoding



B.5.2. Synopsis

Add the 32-bit values in rs1 to rs2, and store the sign-extended result in rd. Any overflow is thrown away.

B.5.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.5.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.5.5. Execution

```
XReg operand1 = sext(X[rs1], 31);
XReg operand2 = sext(X[rs2], 31);
X[rd] = sext(operand1 + operand2, 31);
```

B.5.6. Exceptions

DRAFT

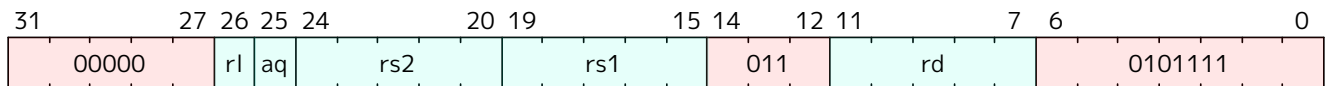
B.6. amoadd.d

Atomic fetch-and-add doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.6.1. Encoding



B.6.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Add the value of register *rs2* to the loaded value
- Write the sum to the address in *rs1*

B.6.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.6.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.6.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Add, aq, rl);

```


DRAFT

B.6.6. Exceptions

DRAFT

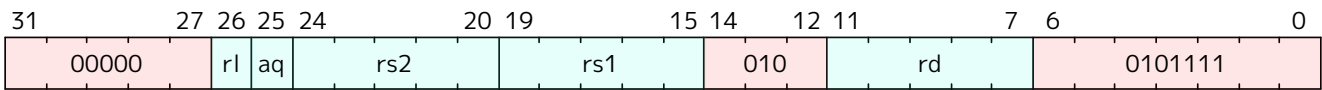
B.7. amoadd.w

Atomic fetch-and-add word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.7.1. Encoding



B.7.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Add the least-significant word of register *rs2* to the loaded value
- Write the sum to the address in *rs1*

B.7.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.7.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.7.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Add, aq,
```

```
rl);
```

DRAFT

B.7.6. Exceptions

DRAFT

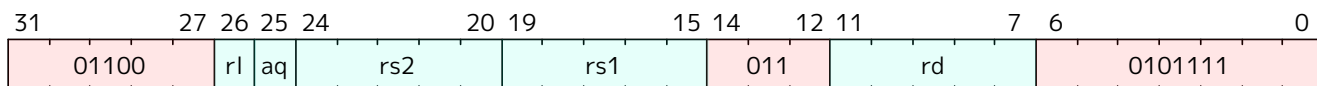
B.8. amoand.d

Atomic fetch-and-and doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.8.1. Encoding



B.8.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- AND the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.8.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.8.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.8.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::And, aq, rl);

```

DRAFT

B.8.6. Exceptions

DRAFT

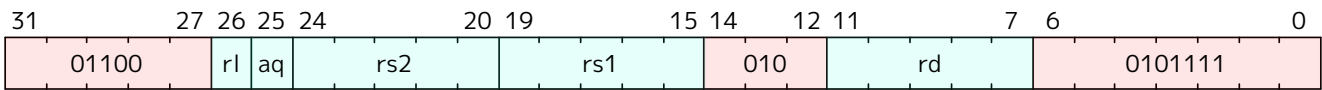
B.9. amoand.w

Atomic fetch-and-and word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.9.1. Encoding



B.9.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- AND the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.9.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.9.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.9.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::And, aq,
```

```
rl);
```

DRAFT

B.9.6. Exceptions

DRAFT

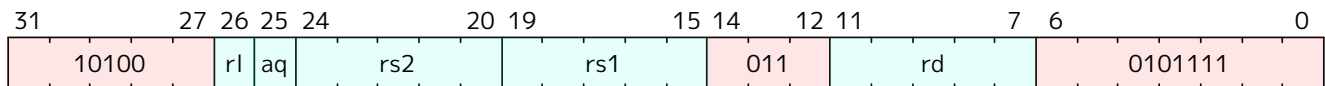
B.10. amomax.d

Atomic MAX doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.10.1. Encoding



B.10.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Signed compare the value of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

B.10.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.10.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.10.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Max, aq, rl);

```

DRAFT

B.10.6. Exceptions

DRAFT

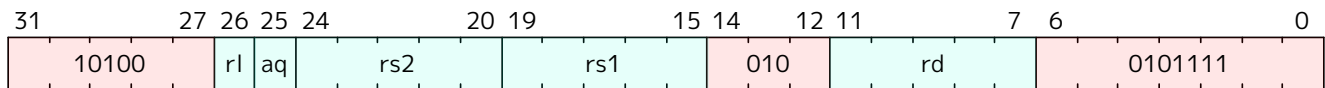
B.11. amomax.w

Atomic MAX word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.11.1. Encoding



B.11.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Signed compare the least-significant word of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

B.11.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.11.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.11.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
```

```
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Max, aq,  
rl);
```

DRAFT

B.11.6. Exceptions

DRAFT

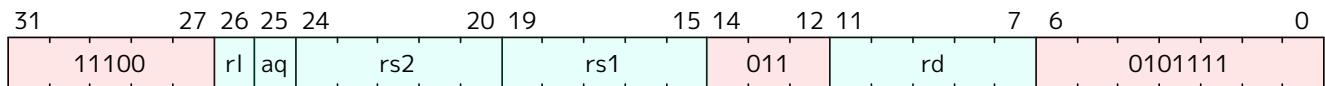
B.12. amomaxu.d

Atomic MAX unsigned doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.12.1. Encoding



B.12.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Unsigned compare the value of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

B.12.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.12.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.12.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Maxu, aq, rl);

```

DRAFT

B.12.6. Exceptions

DRAFT

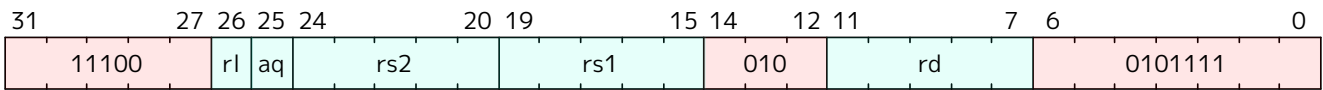
B.13. amomaxu.w

Atomic MAX unsigned word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.13.1. Encoding



B.13.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Unsigned compare the least-significant word of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

B.13.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.13.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.13.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
```

```
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Maxu, aq,  
rl);
```

DRAFT

B.13.6. Exceptions

DRAFT

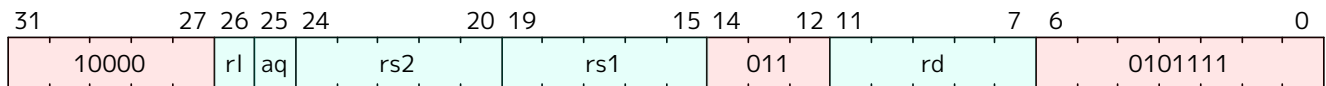
B.14. amomin.d

Atomic MIN doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.14.1. Encoding



B.14.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Signed compare the value of register *rs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *rs1*

B.14.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.14.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.14.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Min, aq, rl);

```


DRAFT

B.14.6. Exceptions

DRAFT

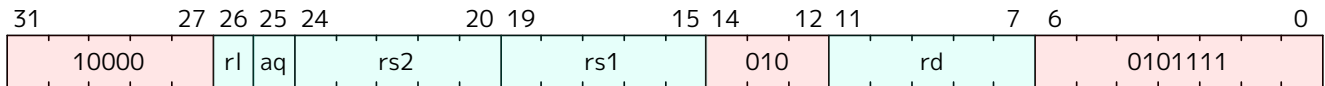
B.15. amomin.w

Atomic MIN word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.15.1. Encoding



B.15.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Signed compare the least-significant word of register *rs2* to the loaded value, and select the minimum value
- Write the result to the address in *rs1*

B.15.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.15.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.15.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];

```

```
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Min, aq,  
rl);
```

DRAFT

B.15.6. Exceptions

DRAFT

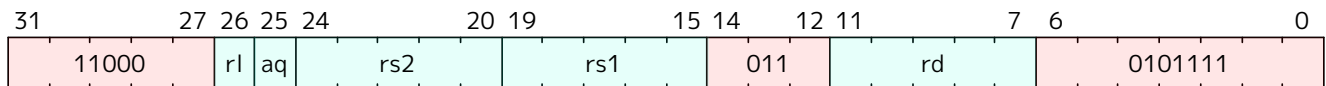
B.16. amominu.d

Atomic MIN unsigned doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.16.1. Encoding



B.16.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Unsigned compare the value of register *rs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *rs1*

B.16.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.16.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.16.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Minu, aq, rl);

```

DRAFT

B.16.6. Exceptions

DRAFT

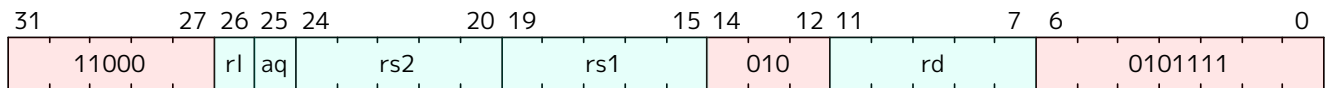
B.17. amominu.w

Atomic MIN unsigned word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.17.1. Encoding



B.17.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Unsigned compare the least-significant word of register *rs2* to the loaded word, and select the minimum value
- Write the result to the address in *rs1*

B.17.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.17.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.17.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
```

```
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Minu, aq,  
rl);
```

DRAFT

B.17.6. Exceptions

DRAFT

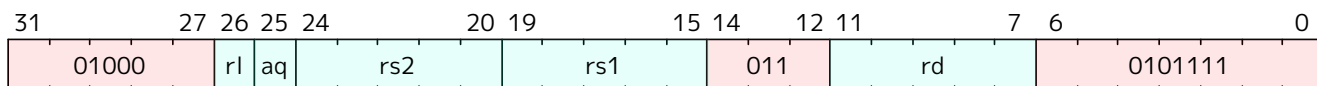
B.18. amoor.d

Atomic fetch-and-or doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.18.1. Encoding



B.18.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- OR the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.18.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.18.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.18.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Or, aq, rl);

```

DRAFT

B.18.6. Exceptions

DRAFT

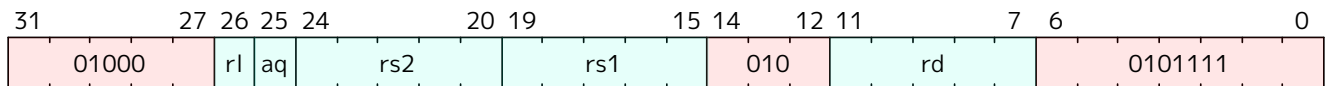
B.19. amoor.w

Atomic fetch-and-or word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.19.1. Encoding



B.19.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- OR the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.19.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.19.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.19.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Or, aq,
```

```
rl);
```

DRAFT

B.19.6. Exceptions

DRAFT

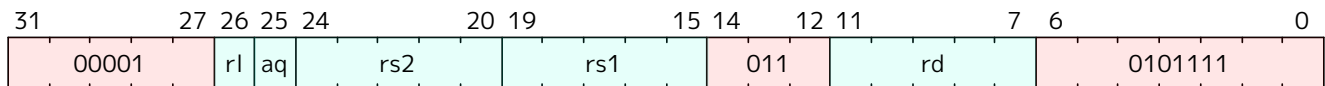
B.20. amoswap.d

Atomic SWAP doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.20.1. Encoding



B.20.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the value into *rd*
- Store the value of register *rs2* to the address in *rs1*

B.20.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.20.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.20.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Swap, aq, rl);

```

B.20.6. Exceptions

DRAFT

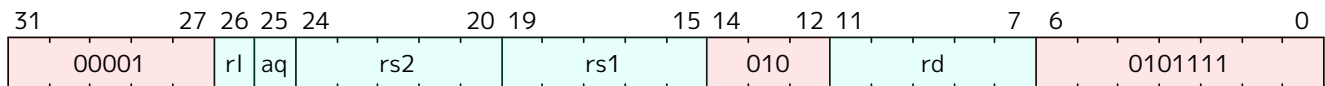
B.21. amoswap.w

Atomic SWAP word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.21.1. Encoding



B.21.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Store the least-significant word of register *rs2* to the address in *rs1*

B.21.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.21.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.21.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Swap, aq,
rl);

```

B.21.6. Exceptions

DRAFT

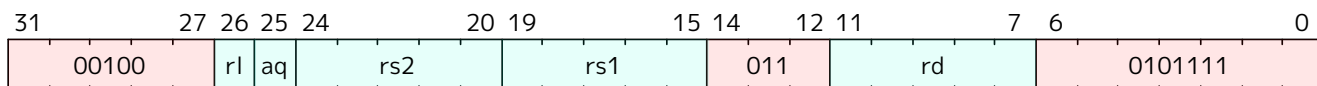
B.22. amoxor.d

Atomic fetch-and-xor doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.22.1. Encoding



B.22.2. Synopsis

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- XOR the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.22.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.22.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.22.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Xor, aq, rl);

```

DRAFT

B.22.6. Exceptions

DRAFT

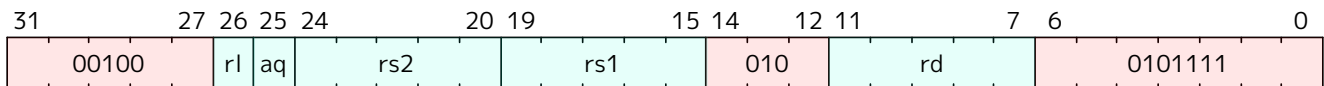
B.23. amoxor.w

Atomic fetch-and-xor word

This instruction is defined by any of the following:

- A, >= 0
- Zaamo, >= 0

B.23.1. Encoding



B.23.2. Synopsis

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- XOR the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

B.23.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.23.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.23.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Xor, aq,

```

```
rl);
```

DRAFT

B.23.6. Exceptions

DRAFT

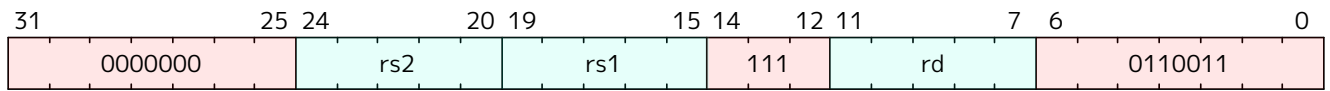
B.24. and

And

This instruction is defined by:

- $I, \geq 0$

B.24.1. Encoding



B.24.2. Synopsis

And rs1 with rs2, and store the result in rd

B.24.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.24.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.24.5. Execution

```
X[rd] = X[rs1] & X[rs2];
```

B.24.6. Exceptions

DRAFT

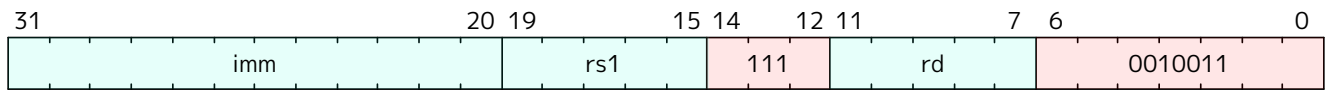
B.25. andi

And immediate

This instruction is defined by:

- $I, \geq 0$

B.25.1. Encoding



B.25.2. Synopsis

And an immediate to the value in rs1, and store the result in rd

B.25.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.25.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.25.5. Execution

```
X[rd] = X[rs1] & imm;
```

B.25.6. Exceptions

DRAFT

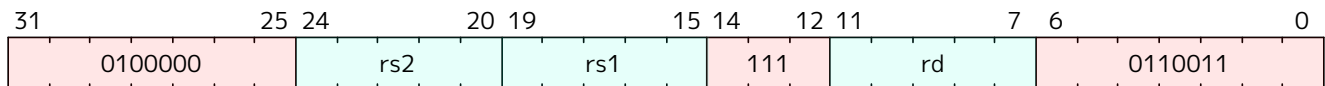
B.26. andn

AND with inverted operand

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.26.1. Encoding



B.26.2. Synopsis

This instruction performs the bitwise logical AND operation between **rs1** and the bitwise inversion of **rs2**.

B.26.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.26.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.26.5. Execution

```

raise(ExceptionCode::IllegalInstruction, $encoding) if CSR[misa].B ==
1'b0;
X[rd] = X[rs2] & ~X[rs1];

```


B.26.6. Exceptions

DRAFT

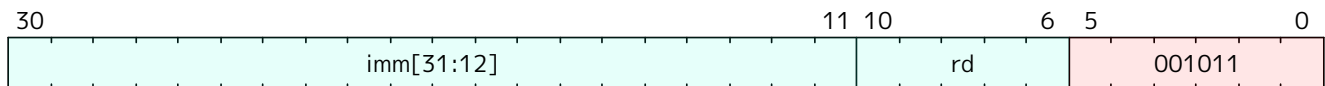
B.27. auipc

Add upper immediate to pc

This instruction is defined by:

- $I, \geq 0$

B.27.1. Encoding



B.27.2. Synopsis

Add an immediate to the current PC.

B.27.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.27.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.27.5. Execution

```
X[rd] = PC + imm;
```

B.27.6. Exceptions

DRAFT

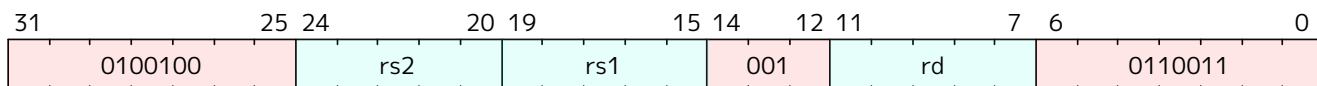
B.28. bclr

Single-Bit clear (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.28.1. Encoding



B.28.2. Synopsis

This instruction returns rs1 with a single bit cleared at the index specified in rs2. The index is read from the lower $\log_2(\text{XLEN})$ bits of rs2.

B.28.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.28.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.28.5. Execution

```

raise(ExceptionCode::IllegalInstruction, $encoding) if CSR[misa].B ==
1'b0;
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] & ~(1 << index);

```

B.28.6. Exceptions

DRAFT

B.29. bclri

Single-Bit clear (Immediate)

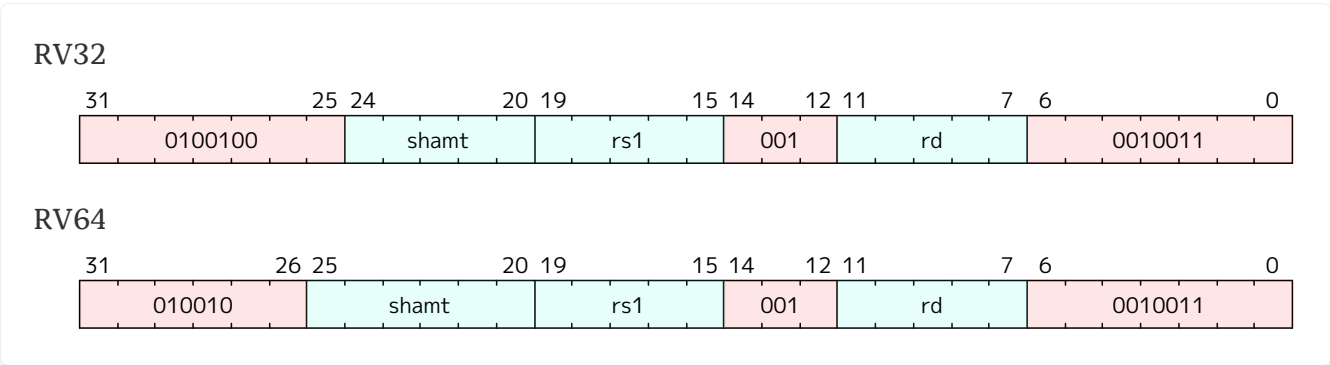
This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.29.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.29.2. Synopsis

This instruction returns rs1 with a single bit cleared at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.29.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.29.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
```

```
Bits<5> rd = $encoding[11:7];
```

B.29.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg index = shamt & (xlen() - 1);  
X[rd] = X[rs1] & ~(1 << index);
```

DRAFT

B.29.6. Exceptions

DRAFT

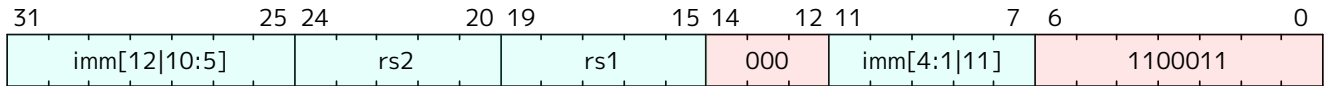
B.30. beq

Branch if equal

This instruction is defined by:

- $I, \geq 0$

B.30.1. Encoding



B.30.2. Synopsis

Branch to PC + imm if the value in register rs1 is equal to the value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.30.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.30.4. Decode Variables

```

Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],
$encoding[11:8], 1'd0 };
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.30.5. Execution

```

XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs == rhs) {
    jump_halfword(PC + imm);
}

```

B.30.6. Exceptions

DRAFT

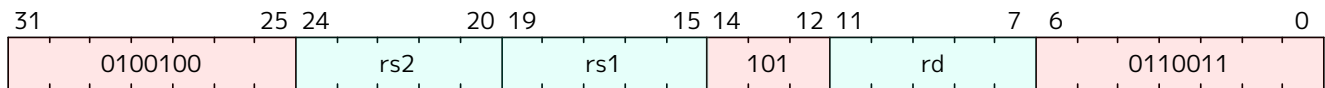
B.31. bext

Single-Bit extract (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.31.1. Encoding



B.31.2. Synopsis

This instruction returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower $\log_2(\text{XLEN})$ bits of rs2.

B.31.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.31.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.31.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = (X[rs1] >> index) & 1;
```

B.31.6. Exceptions

DRAFT

B.32. bexti

Single-Bit extract (Immediate)

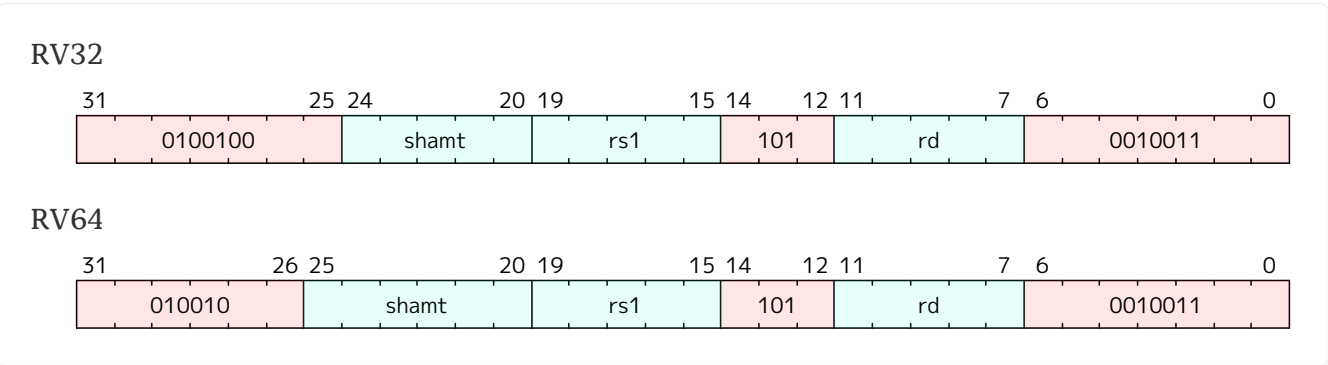
This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.32.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.32.2. Synopsis

This instruction returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.32.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.32.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
```

```
Bits<5> rd = $encoding[11:7];
```

B.32.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg index = shamt & (xlen() - 1);  
X[rd] = (X[rs1] >> index) & 1;
```

DRAFT

B.32.6. Exceptions

DRAFT

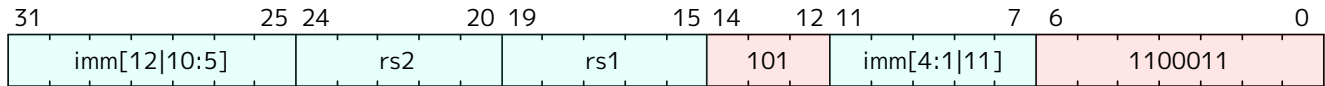
B.33. bge

Branch if greater than or equal

This instruction is defined by:

- $I, \geq 0$

B.33.1. Encoding



B.33.2. Synopsis

Branch to PC + imm if the signed value in register rs1 is greater than or equal to the signed value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.33.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.33.4. Decode Variables

```

Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],
$encoding[11:8], 1'd0 };
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.33.5. Execution

```

XReg lhs = X[rs1];
XReg rhs = X[rs2];
if ($signed(lhs) >= $signed(rhs)) {
    jump_halfword(PC + imm);
}

```


B.33.6. Exceptions

DRAFT

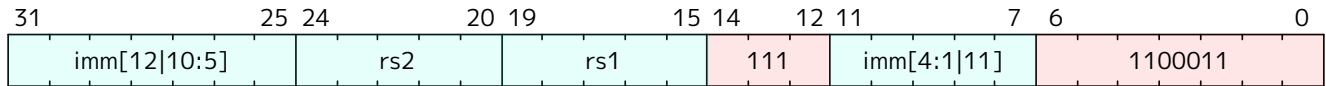
B.34. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

- $I, \geq 0$

B.34.1. Encoding



B.34.2. Synopsis

Branch to PC + imm if the unsigned value in register rs1 is greater than or equal to the unsigned value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.34.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.34.4. Decode Variables

```

Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],
$encoding[11:8], 1'd0 };
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.34.5. Execution

```

XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs >= rhs) {
    jump_halfword(PC + imm);
}

```

B.34.6. Exceptions

DRAFT

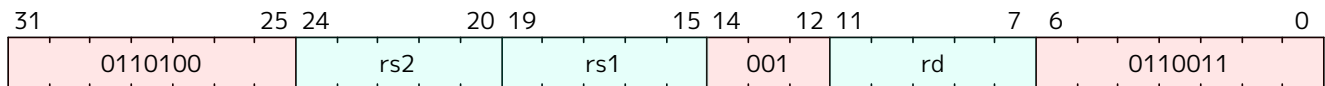
B.35. binv

Single-Bit invert (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.35.1. Encoding



B.35.2. Synopsis

This instruction returns rs1 with a single bit inverted at the index specified in rs2. The index is read from the lower $\log_2(\text{XLEN})$ bits of rs2.

B.35.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.35.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.35.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] ^ (1 << index);

```

B.35.6. Exceptions

DRAFT

B.36. binvi

Single-Bit invert (Immediate)

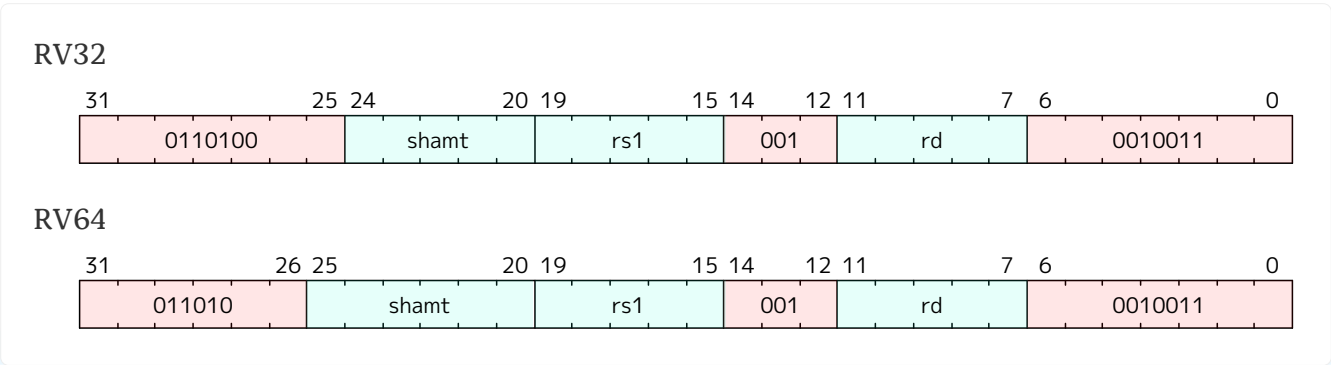
This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.36.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.36.2. Synopsis

This instruction returns rs1 with a single bit inverted at the index specified in shamt. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.36.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.36.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
```

```
Bits<5> rd = $encoding[11:7];
```

B.36.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg index = shamt & (xlen() - 1);  
X[rd] = X[rs1] ^ (1 << index);
```

DRAFT

B.36.6. Exceptions

DRAFT

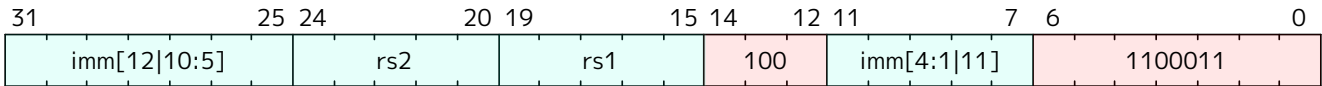
B.37. blt

Branch if less than

This instruction is defined by:

- I, >= 0

B.37.1. Encoding



B.37.2. Synopsis

Branch to PC + imm if the signed value in register rs1 is less than the signed value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.37.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.37.4. Decode Variables

```
Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],  
$encoding[11:8], 1'd0};  
Bits<5> rs2 = $encoding[24:20];  
Bits<5> rs1 = $encoding[19:15];
```

B.37.5. Execution

```
XReg lhs = X[rs1];  
XReg rhs = X[rs2];  
if ($signed(lhs) < $signed(rhs)) {  
  jump_halfword(PC + imm);  
}
```

B.37.6. Exceptions

DRAFT

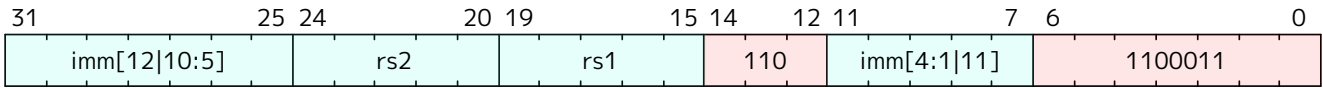
B.38. bltu

Branch if less than unsigned

This instruction is defined by:

- $I, \geq 0$

B.38.1. Encoding



B.38.2. Synopsis

Branch to PC + imm if the unsigned value in register rs1 is less than the unsigned value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.38.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.38.4. Decode Variables

```
Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],  
$encoding[11:8], 1'd0};  
Bits<5> rs2 = $encoding[24:20];  
Bits<5> rs1 = $encoding[19:15];
```

B.38.5. Execution

```
XReg lhs = X[rs1];  
XReg rhs = X[rs2];  
if (lhs < rhs) {  
    jump_halfword(PC + imm);  
}
```

B.38.6. Exceptions

DRAFT

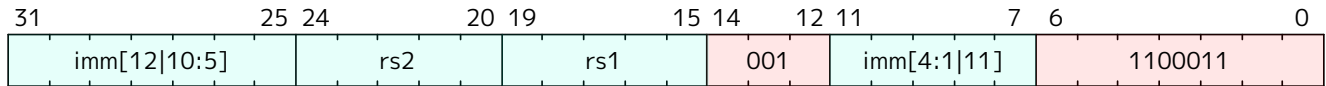
B.39. bne

Branch if not equal

This instruction is defined by:

- $I, \geq 0$

B.39.1. Encoding



B.39.2. Synopsis

Branch to PC + imm if the value in register rs1 is not equal to the value in register rs2.

Raise a **MisalignedAddress** exception if PC + imm is misaligned.

B.39.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.39.4. Decode Variables

```

Bits<13> imm = { $encoding[31], $encoding[7], $encoding[30:25],
$encoding[11:8], 1'd0 };
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.39.5. Execution

```

XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs != rhs) {
    jump_halfword(PC + imm);
}

```

B.39.6. Exceptions

DRAFT

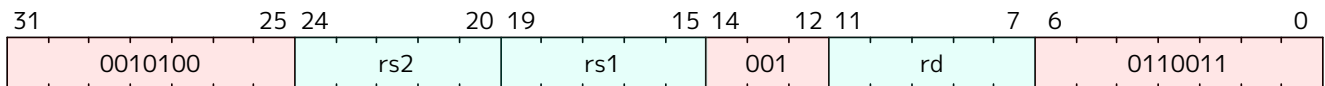
B.40. bset

Single-Bit set (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.40.1. Encoding



B.40.2. Synopsis

This instruction returns rs1 with a single bit set at the index specified in rs2. The index is read from the lower $\log_2(\text{XLEN})$ bits of rs2.

B.40.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.40.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.40.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] | (1 << index);

```

B.40.6. Exceptions

DRAFT

B.41. bseti

Single-Bit set (Immediate)

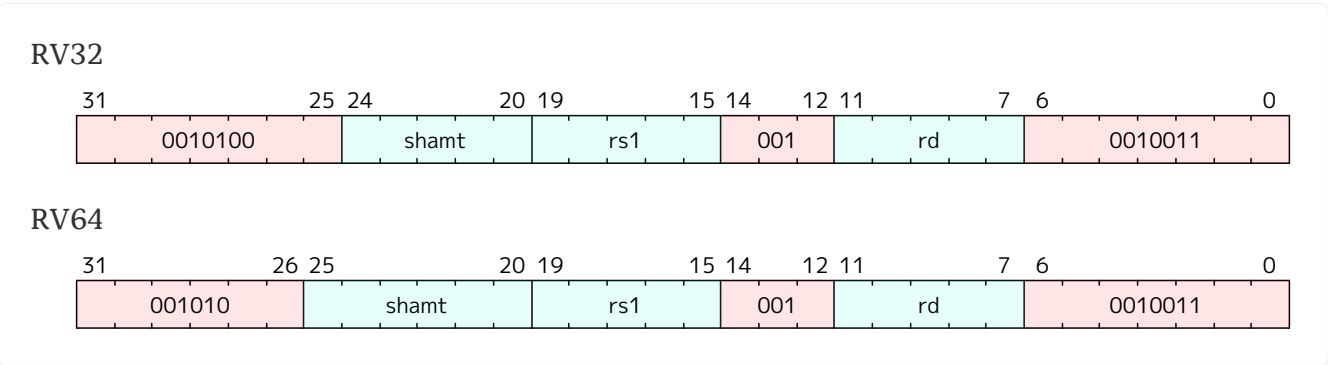
This instruction is defined by any of the following:

- B, >= 0
- Zbs, >= 0

B.41.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.41.2. Synopsis

This instruction returns rs1 with a single bit set at the index specified in shamt. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to $\text{shamt}[5]=1$ are reserved.

B.41.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.41.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
```

```
Bits<5> rd = $encoding[11:7];
```

B.41.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg index = shamt & (xlen() - 1);  
X[rd] = X[rs1] | (1 << index);
```

DRAFT

B.41.6. Exceptions

DRAFT

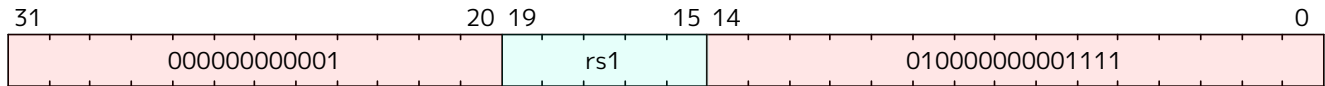
B.42. cbo.clean

Cache Block Clean

This instruction is defined by:

- Zicbom, ≥ 0

B.42.1. Encoding



B.42.2. Synopsis

Cleans an entire cache block globally throughout the system.

Exactly what happens is coherence protocol-dependent, but in general it is expected that after this operation():

- The cache block will be in the clean (not dirty) state in any coherent cache holding a valid copy of the line.
- The data will be cleaned to a point such that an incoherent load can observe the cleaned data.

`cbo.clean` is ordered by **FENCE** instructions but not **FENCE.I** or **SFENCE.VMA**.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Both PMP and PMA access control must be the same for all bytes in the block; otherwise, `cbo.clean` has UNSPECIFIED behavior. <%- end -%>

Clean operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AM0 Guest-Page Fault` if virtual memory translation fails
during G-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if
ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AM0 Access Fault` if a PMP or PMA access check fails
```

<%- if CACHE_BLOCK_SIZE.bit_length ≤ [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

B.42.3. Access

M	HS	U	VS	VU
Always	Sometimes	Sometimes	Sometimes	Sometimes

Access is controled through `menvcfg.CBZE`, `senvcfg.CBZE`, and `henvcfg.CBZE`. When access is denied, the instruction either raises an **Illegal Instruction** or **Virtual Instruction** exception according to the table below.

menvcfg.CBCFE	senvcfg.CBCFE	henvcfg.CBCFE	cbo.clean Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	0	0	executes	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	1	0	executes	executes	Virtual Instruction	Virtual Instruction
1	0	1	executes	Illegal Instruction	executes	Virtual Instruction
1	1	1	executes	executes	executes	executes

B.42.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
```

B.42.5. Execution

B.42.6. Exceptions

DRAFT

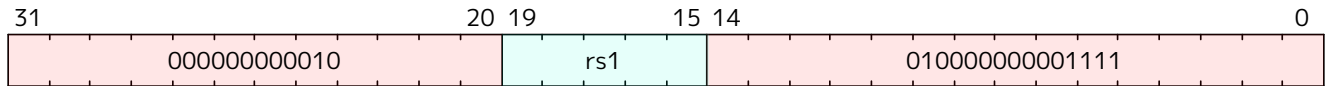
B.43. cbo.flush

Cache Block Flush

This instruction is defined by:

- Zicbom, ≥ 0

B.43.1. Encoding



B.43.2. Synopsis

Flushes an entire cache block by cleaning it and then invalidating it in all caches.

`cbo.flush` is ordered by **FENCE** instructions but not **FENCE.I** or **SFENCE.VMA**.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Both PMP and PMA access control must be the same for all bytes in the block; otherwise, `cbo.flush` has UNSPECIFIED behavior. <%- end -%>

Flush operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AM0 Guest-Page Fault` if virtual memory translation fails
during G-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if
ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AM0 Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ≤ [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

B.43.3. Access

M	HS	U	VS	VU
Always	Sometimes	Sometimes	Sometimes	Sometimes

Access is controled through `menvcfg.CBZE`, `senvcfg.CBZE`, and `henvcfg.CBZE`. When access is denied, the instruction either raises an **Illegal Instruction** or **Virtual Instruction** exception according to the table below.

menvcfg.C BCFE	senvcfg.C BCFE	henvcfg. CBCFE	cbo.flush Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	0	0	executes	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	1	0	executes	executes	Virtual Instruction	Virtual Instruction
1	0	1	executes	Illegal Instruction	executes	Virtual Instruction
1	1	1	executes	executes	executes	executes

B.43.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
```

B.43.5. Execution

DRAFT

B.43.6. Exceptions

DRAFT

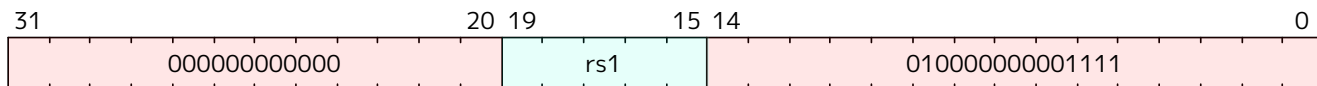
B.44. cbo.inval

Cache Block Invalidate

This instruction is defined by:

- Zicbom, ≥ 0

B.44.1. Encoding



B.44.2. Synopsis

Either invalidates or flushes (clean + invalidate) a cache block, depending on the current mode and value of `menvcfg.CBIE`, `senvcfg.CBIE`, and/or **`henvcfg.CBIE`**.

The instruction is an invalidate (without a clean) when:

- In M-mode
- In (H)S-mode and `menvcfg.CBIE == 11`
- In U-mode and `menvcfg.CBIE == 11` and `senvcfg.CBIE == 11`
- In VS-mode and `menvcfg.CBIE == 11` and **`henvcfg.CBIE == 11`**
- In VU-mode and `menvcfg.CBIE == 11` and **`henvcfg.CBIE == 11`** and `senvcfg.CBIE == 11`

Otherwise, if the instruction does not trap (see Access section), the operation is a flush. The table below summarizes the options.

menvcfg. CBIE	senvcfg. CBIE	henvcfg .CBIE	cbe.inval Operation				
			M-mode	S-mode	U-mode	VS-mode	VU-mode
00	-	-	Invalidate	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction
01	00	00	Invalidate	Flush	Illegal Instruction	Virtual Instruction	Virtual Instruction
01	00	01	Invalidate	Flush	Illegal Instruction	Flush	Virtual Instruction
01	00	11	Invalidate	Flush	Illegal Instruction	Flush	Virtual Instruction
01	01	00	Invalidate	Flush	Flush	Virtual Instruction	Virtual Instruction
01	01	01	Invalidate	Flush	Flush	Flush	Flush
01	01	11	Invalidate	Flush	Flush	Flush	Flush

01	11	00	Invalid ate	Flush	Flush	Virtual Instruction	Virtual Instruction
01	11	01	Invalid ate	Flush	Flush	Flush	Flush
01	11	11	Invalid ate	Flush	Flush	Flush	Flush
11	00	00	Invalid ate	Invalidate	Illegal Instruction	Virtual Instruction	Virtual Instruction
11	00	01	Invalid ate	Invalidate	Illegal Instruction	Flush	Virtual Instruction
11	00	11	Invalid ate	Invalidate	Illegal Instruction	Invalidate	Virtual Instruction
11	01	00	Invalid ate	Invalidate	Flush	Virtual Instruction	Virtual Instruction
11	01	01	Invalid ate	Invalidate	Flush	Flush	Flush
11	01	11	Invalid ate	Invalidate	Flush	Invalidate	Flush
11	11	00	Invalid ate	Invalidate	Invalidate	Virtual Instruction	Virtual Instruction
11	11	01	Invalid ate	Invalidate	Invalidate	Flush	Flush
11	11	11	Invalid ate	Invalidate	Invalidate	Invalidate	Invalidate

`cbo.inval` is ordered by **FENCE** instructions but not **FENCE.I** or **SFENCE.VMA**.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Both PMP and PMA access control must be the same for all bytes in the block; otherwise, `cbo.zero` has UNSPECIFIED behavior. <%- end -%>

Invalidate operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AM0 Guest-Page Fault` if virtual memory translation fails
during G-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if
ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AM0 Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ≤ [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

B.44.3. Access

M	HS	U	VS	VU
Always	Sometimes	Sometimes	Sometimes	Sometimes

Access is controled through `menvcfg.CBIE`, `senvcfg.CBIE`, and `henvcfg.CBIE`. When access is denied, the instruction either raises an **Illegal Instruction** or **Virtual Instruction** exception according to the table below.



`xenvcfg.CBIE == 10` is reserved, and cannot be written by software. As such, that pattern is excluded from the table below.

menvcfg.CBIE	senvcfg.CBIE	henvcfg.CBIE	cbo.inval Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
00	-	-	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction
01/11	00	00	executes	Illegal Instruction	Virtual Instruction	Virtual Instruction
01/11	01/11	00	executes	executes	Virtual Instruction	Virtual Instruction
01/11	00	01/11	executes	Illegal Instruction	executes	Virtual Instruction
01/11	01/11	01/11	executes	executes	executes	executes

B.44.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
```

B.44.5. Execution

B.44.6. Exceptions

DRAFT

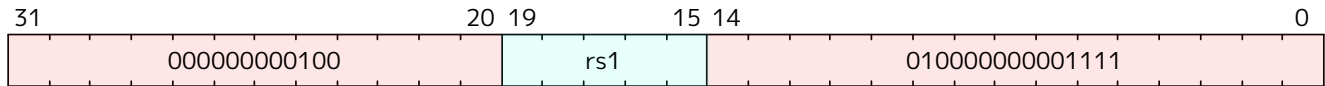
B.45. cbo.zero

Cache Block Zero

This instruction is defined by:

- Zicboz, ≥ 0

B.45.1. Encoding



B.45.2. Synopsis

Zeros an entire cache block

The block zeroing does not need to be atomic.

`cbo.zero` is ordered by **FENCE** instructions but not **FENCE . I** or **SFENCE . VMA**.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Both PMP and PMA access control must be the same for all bytes in the block; otherwise, `cbo.zero` has UNSPECIFIED behavior. <%- end -%>

Clean operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AM0 Guest-Page Fault` if virtual memory translation fails
during G-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if
ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AM0 Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ≤ [PMP_GRANULARITY, PMA_GRANULARITY].min -%>
Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

B.45.3. Access

M	HS	U	VS	VU
Always	Sometimes	Sometimes	Sometimes	Sometimes

Access is controled through `menvcfg.CBZE`, `senvcfg.CBZE`, and `henvcfg.CBZE`. When access is

denied, the instruction either raises an **Illegal Instruction** or **Virtual Instruction** exception according to the table below.

menvcfg. CBZE	senvcfg.C BZE	henvcfg. CBZE	cbo.zero Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	0	0	executes	Illegal Instruction	Virtual Instruction	Virtual Instruction
1	1	0	executes	executes	Virtual Instruction	Virtual Instruction
1	0	1	executes	Illegal Instruction	executes	Virtual Instruction
1	1	1	executes	executes	executes	executes

B.45.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
```

B.45.5. Execution

B.45.6. Exceptions

DRAFT

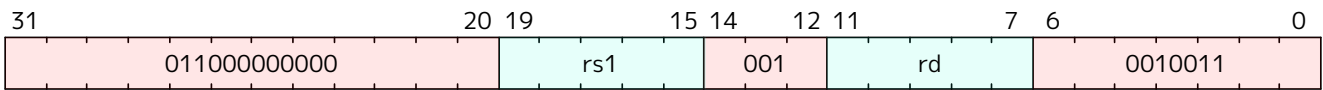
B.46. clz

Count leading zero bits

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.46.1. Encoding



B.46.2. Synopsis

This instruction counts the number of 0's before the first 1, starting at the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN, and if the most-significant bit of the input is a 1, the output is 0.

B.46.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.46.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.46.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = (xlen() - 1) - $signed(highest_set_bit(X[rs1]));
```

B.46.6. Exceptions

DRAFT

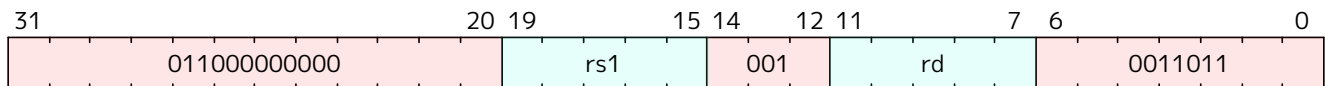
B.47. clzw

Count leading zero bits in word

This instruction is defined by any of the following:

- B, ≥ 0
- Zbb, ≥ 0

B.47.1. Encoding



B.47.2. Synopsis

This instruction counts the number of 0's before the first 1 starting at bit 31 and progressing to bit 0. Accordingly, if the least-significant word is 0, the output is 32, and if the most-significant bit of the word (*i.e.*, bit 31) is a 1, the output is 0.

B.47.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.47.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.47.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = 31 - $signed(highest_set_bit(X[rs1][31:0]));
```

B.47.6. Exceptions

DRAFT

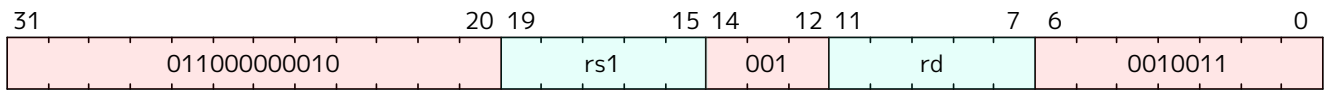
B.48. cpop

Count set bits

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.48.1. Encoding



B.48.2. Synopsis

This instructions counts the number of 1’s (i.e., set bits) in the source register.

This operations is known as population count, popcount, sideways sum, bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is implemented by cpop on RV32 and by cpopw on RV64. The GCC builtin function `__builtin_popcountl (unsigned long x)` for LP64 is implemented by cpop on RV64.

Listing 1. Software Hint

B.48.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.48.4. Decode Variables

Bits<5> rs1 = \$encoding[19:15];
Bits<5> rd = \$encoding[11:7];

B.48.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg bitcount = 0;
```

```
XReg rs1_val = X[rs1];
for (U32 i = 0; i < xlen(); i++) {
    if (rs1_val[i] == 1'b1) {
        bitcount = bitcount + 1;
    }
}
X[rd] = bitcount;
```

DRAFT

B.48.6. Exceptions

DRAFT

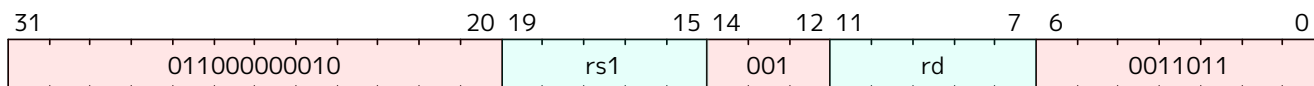
B.49. cpopw

Count set bits in word

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.49.1. Encoding



B.49.2. Synopsis

This instructions counts the number of 1's (i.e., set bits) in the least-significant word of the source register.

This operations is known as population count, popcount, sideways sum, bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is implemented by cpop on RV32 and by cpopw on RV64. The GCC builtin function `__builtin_popcountl (unsigned long x)` for LP64 is implemented by cpop on RV64.

Listing 2. Software Hint

B.49.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.49.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.49.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
```



```
XReg bitcount = 0;
XReg rs1_val = X[rs1];
for (U32 i = 0; i < 32; i++) {
    if (rs1_val[i] == 1'b1) {
        bitcount = bitcount + 1;
    }
}
X[rd] = bitcount;
```

DRAFT

B.49.6. Exceptions

DRAFT

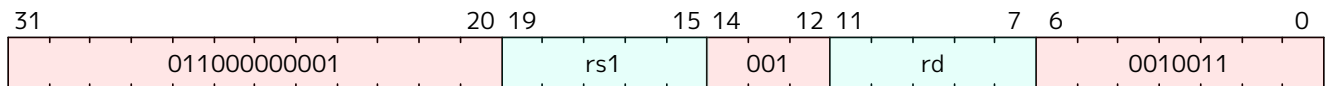
B.50. ctz

Count trailing zero bits

This instruction is defined by any of the following:

- B, ≥ 0
- Zbb, ≥ 0

B.50.1. Encoding



B.50.2. Synopsis

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN, and if the least-significant bit of the input is a 1, the output is 0.

B.50.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.50.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.50.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = (xlen() - 1) - $signed(lowest_set_bit(X[rs1]));
```

B.50.6. Exceptions

DRAFT

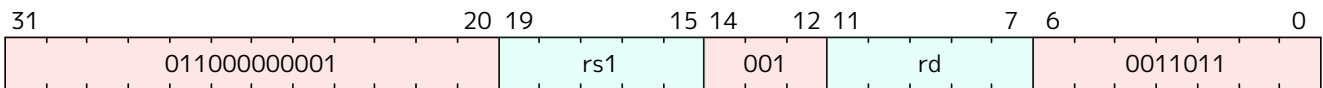
B.51. ctzw

Count trailing zero bits in word

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.51.1. Encoding



B.51.2. Synopsis

This instruction counts the number of 0’s before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit of the least-significant word (i.e., 31). Accordingly, if the least-significant word is 0, the output is 32, and if the least-significant bit of the input is a 1, the output is 0.

B.51.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.51.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.51.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = (xlen() - 1) - $signed(lowest_set_bit(X[rs1][31:0]));
```

B.51.6. Exceptions

DRAFT

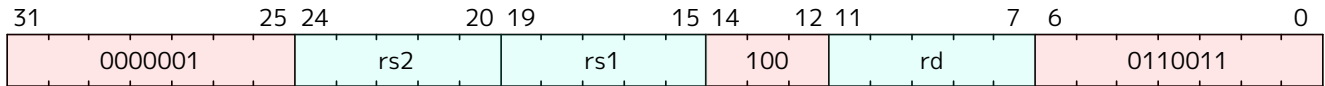
B.52. div

Signed division

This instruction is defined by:

- M, >= 0

B.52.1. Encoding



B.52.2. Synopsis

Divide rs1 by rs2, and store the result in rd. The remainder is discarded.

Division by zero will put -1 into rd.

Division resulting in signed overflow (when most negative number is divided by -1) will put the most negative number into rd;

B.52.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.52.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.52.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = {XLEN{1'b1}};
} else if ((src1 == {1'b1, {XLEN - 1{1'b0}}}) && (src2 == {XLEN{1'b1}})) {
  {
    X[rd] = {1'b1, {XLEN - 1{1'b0}}};
  }
}

```

```
} else {  
    X[rd] = $signed(src1) / $signed(src2);  
}
```

DRAFT

B.52.6. Exceptions

DRAFT

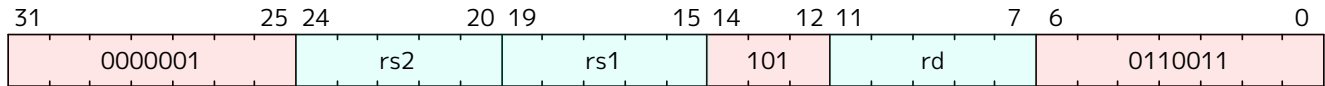
B.53. divu

Unsigned division

This instruction is defined by:

- $M, \geq 0$

B.53.1. Encoding



B.53.2. Synopsis

Divide unsigned values in rs1 by rs2, and store the result in rd.

The remainder is discarded.

If the value in rs2 is zero, rd gets the largest unsigned value.

B.53.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.53.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.53.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = {XLEN{1'b1}};
} else {
  X[rd] = src1 / src2;
}

```

B.53.6. Exceptions

DRAFT

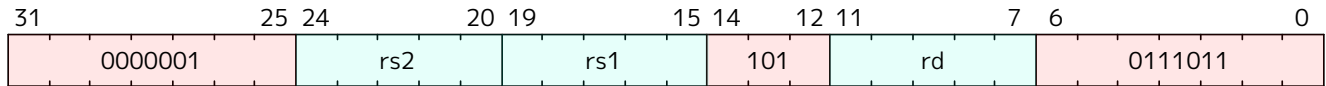
B.54. divuw

Unsigned 32-bit division

This instruction is defined by:

- $M, \geq 0$

B.54.1. Encoding



B.54.2. Synopsis

Divide the unsigned 32-bit values in rs1 and rs2, and store the sign-extended result in rd.

The remainder is discarded.

If the value in rs2 is zero, rd is written with all 1s.

B.54.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.54.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.54.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  X[rd] = {64{1'b1}};
} else {
  Bits<32> result = src1 / src2;
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}

```

}

DRAFT

B.54.6. Exceptions

DRAFT

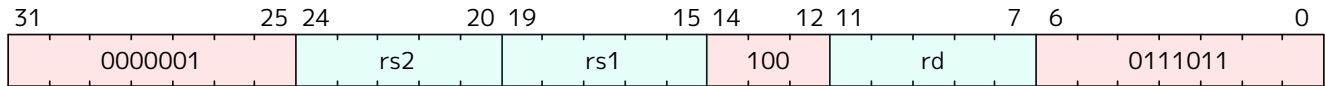
B.55. divw

Signed 32-bit division

This instruction is defined by:

- $M, \geq 0$

B.55.1. Encoding



B.55.2. Synopsis

Divide the lower 32-bits of register rs1 by the lower 32-bits of register rs2, and store the sign-extended result in rd.

The remainder is discarded.

Division by zero will put -1 into rd.

Division resulting in signed overflow (when most negative number is divided by -1) will put the most negative number into rd;

B.55.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.55.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.55.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
    X[rd] = {XLEN{1'b1}};
```

```
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {  
    X[rd] = {33'b1, 31'b0};  
} else {  
    Bits<32> result = $signed(src1) / $signed(src2);  
    Bits<1> sign_bit = result[31];  
    X[rd] = {{32{sign_bit}}, result};  
}
```

DRAFT

B.55.6. Exceptions

DRAFT

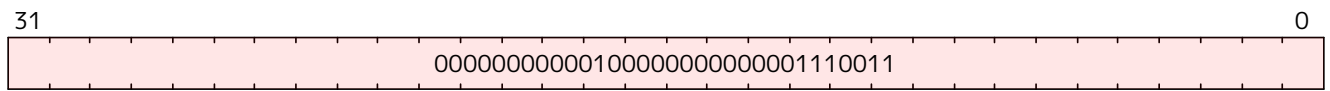
B.56. ebreak

Breakpoint exception

This instruction is defined by:

- I, >= 0

B.56.1. Encoding



B.56.2. Synopsis

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, EBREAK raises a breakpoint exception and performs no other operation.



As described in the C Standard Extension for Compressed Instructions, the **c.ebreak** instruction performs the same operation as the EBREAK instruction.

EBREAK causes the receiving privilege mode’s epc register to be set to the address of the EBREAK instruction itself, not the address of the following instruction. As EBREAK causes a synchronous exception, it is not considered to retire, and should not increment the minstret CSR.

B.56.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.56.4. Decode Variables

B.56.5. Execution

```
raise(ExceptionCode::Breakpoint, $pc);
```

B.56.6. Exceptions

DRAFT

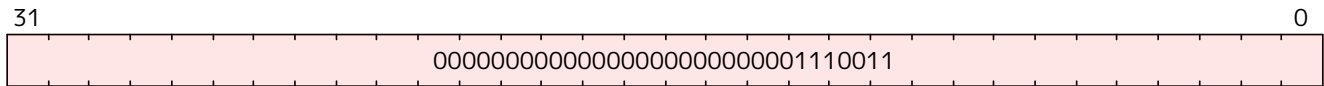
B.57. ecall

Environment call

This instruction is defined by:

- $I, \geq 0$

B.57.1. Encoding



B.57.2. Synopsis

The ECALL instruction is used to make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.



ECALL generates a different exception for each originating privilege mode so that environment call exceptions can be selectively delegated. A typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception but not the others.

ECALL causes the receiving privilege mode's epc register to be set to the address of the ECALL instruction itself, not the address of the following instruction. As ECALL causes a synchronous exception, it is not considered to retire, and should not increment the [minstret](#) CSR.

B.57.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.57.4. Decode Variables

B.57.5. Execution

```
if (mode() == PrivilegeMode::M) {
    raise(ExceptionCode::Mcall, 0);
} else if (mode() == PrivilegeMode::S) {
    raise(ExceptionCode::Scall, 0);
} else if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::Ucall, 0);
}
```

}

DRAFT

B.57.6. Exceptions

DRAFT

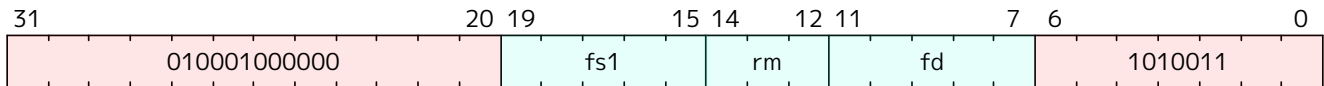
B.58. fcvt.h.s

Convert half-precision float to a single-precision float

This instruction is defined by any of the following:

- Zfh, >= 0
- Zfhmin, >= 0

B.58.1. Encoding



B.58.2. Synopsis

Converts a half-precision number in floating-point register *fs1* into a single-precision floating-point number in floating-point register *fd*.

[fcvt.h.s](#) rounds according to the *rm* field.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

B.58.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.58.4. Decode Variables

```

Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];

```

B.58.5. Execution

```

check_f_ok();
Bits<16> hp_value = f[fs1][15:0];
Bits<1> sign = hp_value[15];
Bits<5> exp = hp_value[14:10];
Bits<10> frac = hp_value[9:0];
if (exp == 0x1F) {
    if (frac != 0) {
        if ((hp_value & 0x0200) != 0) {

```

```

        set_fp_flag(FpFlag::NV);
    }
    f[fd] = HP_CANONICAL_NAN;
} else {
    f[fd] = packToF32UI(sign, 0xFF, 0);
}
} else {
    if (exp != 0) {
        if (frac != 0) {
            f[fd] = packToF32UI(sign, 0, 0);
        } else {
            Bits<6> norm_exp;
            (norm_exp, frac = soffloat_normSubnormalF16Sig(frac));
            exp = norm_exp - 1;
            f[fd] = packToF3UI(sign, exp + 0x70, frac << 13);
        }
    } else {
        f[fd] = packToF3UI(sign, exp + 0x70, frac << 13);
    }
}
}
mark_f_state_dirty();

```


B.58.6. Exceptions

DRAFT

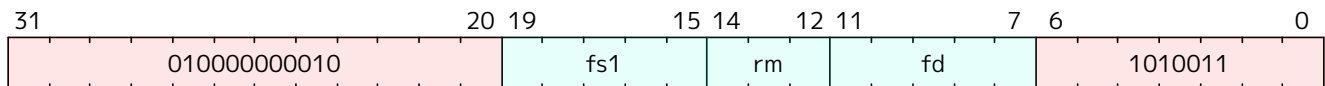
B.59. fcvt.s.h

Convert single-precision float to a half-precision float

This instruction is defined by any of the following:

- Zfh, >= 0
- Zfhmin, >= 0

B.59.1. Encoding



B.59.2. Synopsis

Converts a single-precision number in floating-point register *fs1* into a half-precision floating-point number in floating-point register *fd*.

[fcvt.s.h](#) will never round, and so the 'rm' field is effectively ignored.

B.59.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.59.4. Decode Variables

```

Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];

```

B.59.5. Execution

```

check_f_ok();
Bits<32> sp_value = f[fs1][31:0];
Bits<1> sign = sp_value[31];
Bits<8> exp = sp_value[30:23];
Bits<23> frac = sp_value[22:0];
if (exp == 0xFF) {
    if (frac != 0) {
        if ((sp_value & 0x00400000) != 0) {
            set_fp_flag(FpFlag::NV);
        }
    }
}

```

```

    f[fd] = nan_box<16, FLEN>(HP_CANONICAL_NAN);
} else {
    f[fd] = nan_box<16, FLEN>(packToF16UI(sign, 0x1F, 0));
}
} else {
    Bits<16> frac16 = (frac >> 9) | frac & 0x1ff) != 0 ? 1 : 0);    if
((exp | frac16) == 0) {      f[fd] = nan_box<16, FLEN>(packToF16UI(sign,
0, 0;
    } else {
        f[fd] = soffloat_roundPackToF16(sign, exp - 0x71, frac16 | 0x4000);
    }
}
mark_f_state_dirty();

```

DRAFT

B.59.6. Exceptions

DRAFT

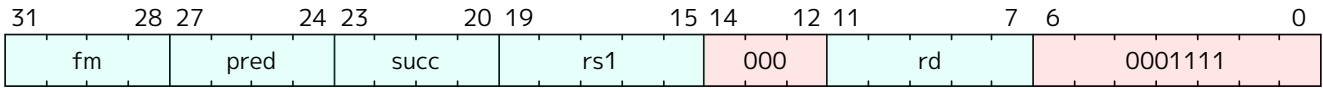
B.60. fence

Memory ordering fence

This instruction is defined by:

- I, >= 0

B.60.1. Encoding



B.60.2. Synopsis

Orders memory operations.

The [fence](#) instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a [fence](#) before any operation in the *predecessor* set preceding the [fence](#).

The predecessor and successor fields have the same format to specify operation types:

pred				succ			
27	26	25	24	23	22	21	20
PI	PO	PR	PW	SI	SO	SR	SW

Table 48. Fence mode encoding

fm field	Mnemonic	Meaning
0000	none	Normal Fence
1000	TSO	With FENCE RW, RW : exclude write-to-read ordering; otherwise: <i>Reserved for future use</i> .
other		<i>Reserved for future use</i> .

When the mode field *fm* is **0001** and both the predecessor and successor sets are 'RW', then the instruction acts as a special-case **fence.tso**. **fence.tso** orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the 'fence.tso's predecessor set unordered with non-AMO loads in its successor set.

When mode field *fm* is not **0001**, or when mode field *fm* is **0001** but the *pred* and *succ* fields are not both 'RW' (0x3), then the fence acts as a baseline fence (e.g., *fm* is effectively **0000**). This is unaffected by the FIOM bits, described below (implicit promotion does not change how **fence.tso** is decoded).

The **rs1** and **rd** fields are unused and ignored.

In modes other than M-mode, [fence](#) is further affected by `menvcfg.FIOM`, `senvcfg.FIOM`<% if ext?:(H)

%>, and/or **henvcfg.FIOM**<% end %> as follows:

Table 49. Effective PR/PW/SR/SW in (H)S-mode

menvcfg.FIOM	pred.PI pred.P0 succ.SI succ.S0	→ → → →	effective PR effective PW effective SR effective SW
0	-		from encoding
1	0		from encoding
1	1		1

Table 50. Effective PR/PW/SR/SW in U-mode

menvcfg.FIOM	senvcfg.FIOM	pred.PI pred.P0 succ.SI succ.S0	→ → → →	effective PR effective PW effective SR effective SW
0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding
1	-	1		1

<%- if ext?(H) -%> .Effective PR/PW/SR/SW in VS-mode and VU-mode

menvcfg.FIOM	henvcfg.FIOM	pred.PI pred.P0 succ.SI succ.S0	→ → → →	effective PR effective PW effective SR effective SW
0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding
1	-	1		1

<%- end -%>

B.60.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.60.4. Decode Variables

```
Bits<4> fm = $encoding[31:28];
```

```

Bits<4> pred = $encoding[27:24];
Bits<4> succ = $encoding[23:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.60.5. Execution

```

Boolean is_fence_tso;
Boolean is_pause;
if (fm == 1) {
    if (pred == 0x3 && succ == 0x3) {
        is_fence_tso = true;
    }
}
if (implemented?(ExtensionName::Zihintpause)) {
    if ((pred == 1) && (succ == 0) && (fm == 0) && (rd == 0) && (rs1 ==
0)) {
        is_pause = true;
    }
}
Boolean pred_i = pred[3];
Boolean pred_o = pred[2];
Boolean pred_r = pred[1];
Boolean pred_w = pred[0];
Boolean succ_i = succ[3];
Boolean succ_o = succ[2];
Boolean succ_r = succ[1];
Boolean succ_w = succ[0];
if (is_fence_tso) {
    fence_tso();
} else if (is_pause) {
    pause();
} else {
    if (mode() == PrivilegeMode::S) {
        if (CSR[menvcfg].FIOM) {
            if (pred_i) {
                pred_r = true;
            }
            if (pred_o) {
                pred_w = true;
            }
            if (succ_i) {
                succ_r = true;
            }
            if (succ_o) {

```

```

        succ_w = true;
    }
}
} else if (mode() == PrivilegeMode::U) {
    if (CSR[menvcfg].FIOM || CSR[senvcfg].FIOM) {
        if (pred_i) {
            pred_r = true;
        }
        if (pred_o) {
            pred_w = true;
        }
        if (succ_i) {
            succ_r = true;
        }
        if (succ_o) {
            succ_w = true;
        }
    }
} else if (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU)
{
    if (CSR[menvcfg].FIOM || CSR[henvcfg].FIOM) {
        if (pred_i) {
            pred_r = true;
        }
        if (pred_o) {
            pred_w = true;
        }
        if (succ_i) {
            succ_r = true;
        }
        if (succ_o) {
            succ_w = true;
        }
    }
}
}
fence(pred_i, pred_o, pred_r, pred_w, succ_i, succ_o, succ_r, succ_w);
}

```


B.60.6. Exceptions

DRAFT

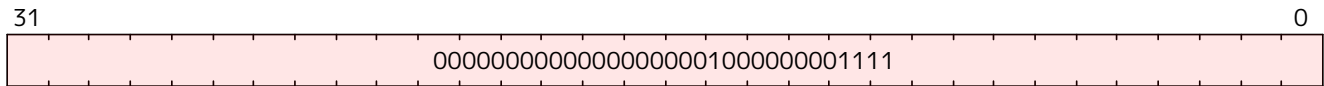
B.61. fence.i

Instruction fence

This instruction is defined by:

- Zifencei, ≥ 0

B.61.1. Encoding



B.61.2. Synopsis

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction. A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does *not* ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart also has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.

The unused fields in the FENCE.I instruction, *imm*[11:0], *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.



Because FENCE.I only orders stores with a hart's own instruction fetches, application code should only rely upon FENCE.I if the application thread will not be migrated to a different hart. The EEI can provide mechanisms for efficient multiprocessor instruction-stream synchronization.

B.61.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.61.4. Decode Variables

B.61.5. Execution

```
ifence();
```

B.61.6. Exceptions

DRAFT

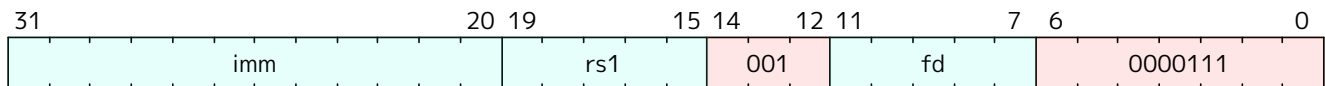
B.62. flh

Half-precision floating-point load

This instruction is defined by any of the following:

- Zfh, >= 0
- Zfhmin, >= 0

B.62.1. Encoding



B.62.2. Synopsis

The `flh` instruction loads a single-precision floating-point value from memory at address $rs1 + imm$ into floating-point register rd .

`flh` does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

`flh` is only guaranteed to execute atomically if the effective address is naturally aligned.

B.62.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.62.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];

```

B.62.5. Execution

```

check_f_ok();
XReg virtual_address = X[rs1] + $signed(imm);
Bits<16> hp_value = read_memory<16>(virtual_address);
f[fd] = nan_box<16, FLEN>(hp_value);
mark_f_state_dirty();

```

B.62.6. Exceptions

DRAFT

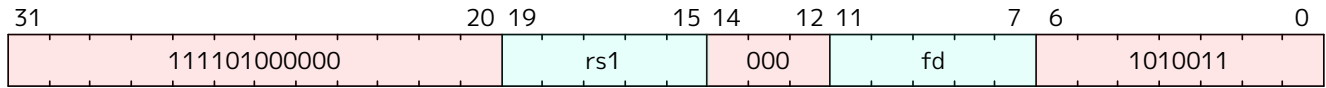
B.63. fmv.h.x

Half-precision floating-point move from integer

This instruction is defined by:

- $F, \geq 0$

B.63.1. Encoding



B.63.2. Synopsis

Moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register **rs1** to the floating-point register **fd**. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

B.63.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.63.4. Decode Variables

```

Bits<5> rs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];

```

B.63.5. Execution

```

check_f_ok();
Bits<16> hp_value = X[rs1][15:0];
f[fd] = nan_box<16, FLEN

```

B.63.6. Exceptions

DRAFT

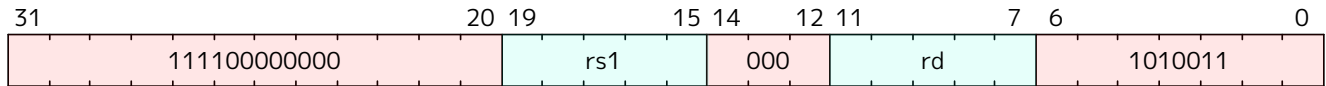
B.64. fmv.w.x

Single-precision floating-point move from integer

This instruction is defined by:

- $F, \geq 0$

B.64.1. Encoding



B.64.2. Synopsis

Moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register **rs1** to the floating-point register **rd**. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

B.64.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.64.4. Decode Variables

```

Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.64.5. Execution

B.64.6. Exceptions

DRAFT

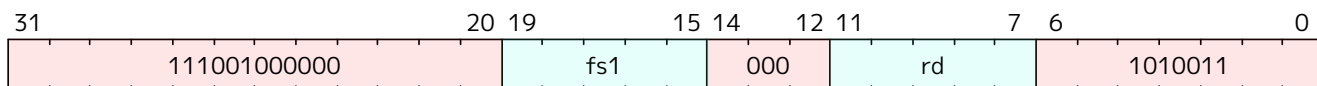
B.65. fmv.x.h

Move half-precision value from floating-point to integer register

This instruction is defined by any of the following:

- Zfh, >= 0
- Zfhmin, >= 0

B.65.1. Encoding



B.65.2. Synopsis

Moves the half-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 16 bits of integer register rd.

The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

The highest XLEN-16 bits of the destination register are filled with copies of the floating-point number's sign bit.

B.65.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.65.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.65.5. Execution

```
check_f_ok();
X[rd] = sext(f[fs1][15:0], 16);
```

B.65.6. Exceptions

DRAFT

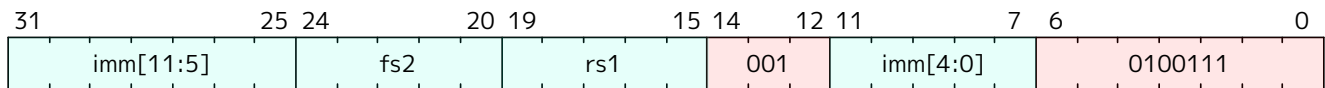
B.66. fsh

Half-precision floating-point store

This instruction is defined by any of the following:

- Zfh, >= 0
- Zfhmin, >= 0

B.66.1. Encoding



B.66.2. Synopsis

The `fsh` instruction stores a half-precision floating-point value from register `rd` to memory at address `rs1 + imm`.

`fsh` does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

`fsh` ignores all but the lower 16 bits in `rs2`.

`fsh` is only guaranteed to execute atomically if the effective address is naturally aligned.

B.66.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.66.4. Decode Variables

```

Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs1 = $encoding[19:15];
Bits<5> fs2 = $encoding[24:20];

```

B.66.5. Execution

```

check_f_ok();
XReg virtual_address = X[rs1] + $signed(imm);
Bits<16> hp_value = f[rs2][15:0];
write_memory<16>(virtual_address, hp_value);

```

B.66.6. Exceptions

DRAFT

B.67. hINVAL.GVMA

Invalidate cached address translations

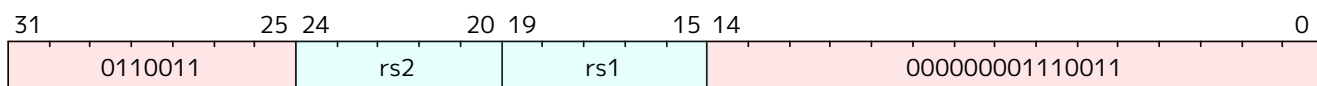
This instruction is defined by:

- $S_{\text{INVAL}}, \geq 0$

Additionally, this instruction is only defined if the following extension is also present and active:

- $H, \geq 0$

B.67.1. Encoding



B.67.2. Synopsis

`hINVAL.GVMA` has the same semantics as `SINVAL.VMA` except that it combines with `SFENCE.W.INVAL` and `SFENCE.INVAL.IR` to replace `HFENCE.GVMA` and uses VMID instead of ASID.

B.67.3. Access

M	HS	U	VS	VU
Always	Sometimes	Never	Never	Never

B.67.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.67.5. Execution

```
XReg gaddr = X[rs1];
Bits<VMID_WIDTH> vmid = X[rs2][VMID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if ((mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU)) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
```

```
VmaOrderType vma_type;
vma_type.gstage = true;
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_vmid = true;
    vma_type.vmid = vmid;
    invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_vaddr = true;
        vma_type.vaddr = gaddr;
        invalidate_translations(vma_type);
    }
} else {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_vmid = true;
        vma_type.vmid = vmid;
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        invalidate_translations(vma_type);
    }
}
```

B.67.6. Exceptions

DRAFT

B.68. hinval.vvma

Invalidate cached address translations

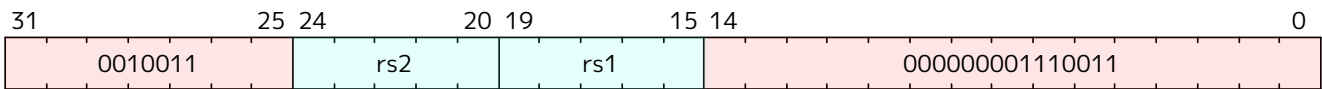
This instruction is defined by:

- Svinval, ≥ 0

Additionally, this instruction is only defined if the following extension is also present and active:

- H, ≥ 0

B.68.1. Encoding



B.68.2. Synopsis

`hinval.vvma` has the same semantics as `sINVAL.vma` except that it combines with `sfence.w.inval` and `sfence.inval.ir` to replace `hfence.vvma`.

B.68.3. Access

M	HS	U	VS	VU
Always	Always	Never	Never	Never

B.68.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.68.5. Execution

```
XReg vaddr = X[rs1];
Bits<ASID_WIDTH> asid = X[rs2][ASID_WIDTH - 1:0];
Bits<VMID_WIDTH> vmid = CSR[hgatp].VMID;
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if ((CSR[misa].H == 1) && (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU)) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
VmaOrderType vma_type;
```

```

vma_type.vsmode = true;
vma_type.single_vmid = true;
vma_type.vmid = vmid;
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        invalidate_translations(vma_type);
    }
} else {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_asid = true;
        vma_type.asid = asid;
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        invalidate_translations(vma_type);
    }
}
}

```

B.68.6. Exceptions

DRAFT

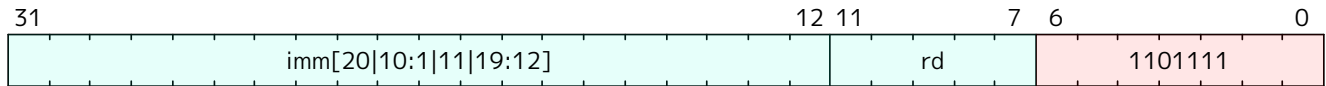
B.69. jal

Jump and link

This instruction is defined by:

- $I, \geq 0$

B.69.1. Encoding



B.69.2. Synopsis

Jump to a PC-relative offset and store the return address in rd.

B.69.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.69.4. Decode Variables

```
signed Bits<21> imm = sext({$encoding[31], $encoding[19:12], $encoding
[20], $encoding[30:21], 1'd0});
Bits<5> rd = $encoding[11:7];
```

B.69.5. Execution

```
XReg retrun_addr = PC + 4;
jump_halfword(PC + imm);
X[rd] = retrun_addr;
```

B.69.6. Exceptions

DRAFT

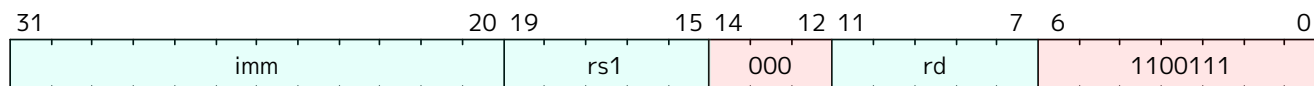
B.70. jalr

Jump and link register

This instruction is defined by:

- $I, \geq 0$

B.70.1. Encoding



B.70.2. Synopsis

Jump to an address formed by adding rs1 to a signed offset, and store the return address in rd.

B.70.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.70.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.70.5. Execution

```

XReg returnaddr;
returnaddr = PC + 4;
jump(X[rs1] + imm);
X[rd] = returnaddr;

```

B.70.6. Exceptions

DRAFT

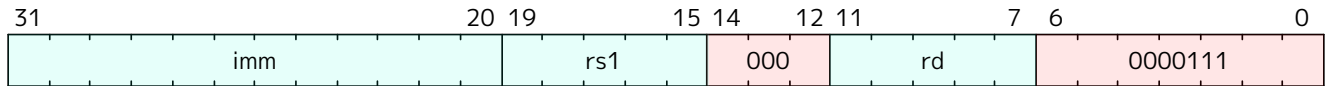
B.71. lb

Load byte

This instruction is defined by:

- $I, \geq 0$

B.71.1. Encoding



B.71.2. Synopsis

Load 8 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Sign extend the result.

B.71.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.71.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.71.5. Execution

```

XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<8>(virtual_address), 8);

```


B.71.6. Exceptions

DRAFT

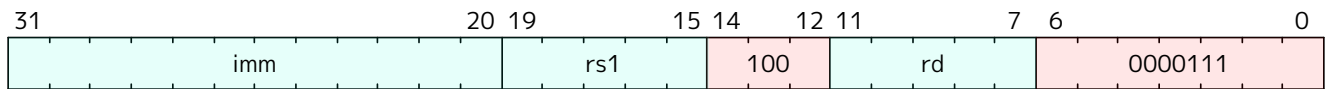
B.72. lbu

Load byte unsigned

This instruction is defined by:

- I, >= 0

B.72.1. Encoding



B.72.2. Synopsis

Load 8 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Zero extend the result.

B.72.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.72.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.72.5. Execution

```

XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<8>(virtual_address);

```

B.72.6. Exceptions

DRAFT

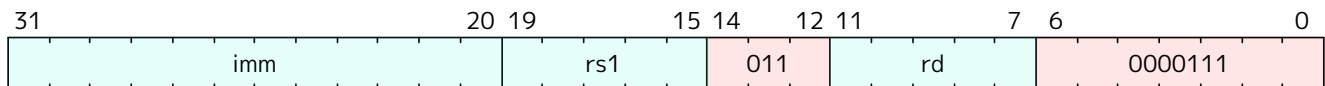
B.73. ld

Load doubleword

This instruction is defined by:

- $I, \geq 0$

B.73.1. Encoding



B.73.2. Synopsis

Load 64 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset.

B.73.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.73.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.73.5. Execution

```

XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<64>(virtual_address);

```

B.73.6. Exceptions

DRAFT

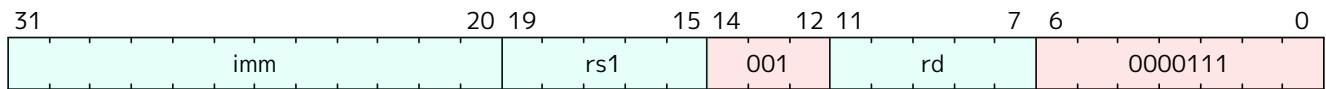
B.74. lh

Load halfword

This instruction is defined by:

- $I, \geq 0$

B.74.1. Encoding



B.74.2. Synopsis

Load 16 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Sign extend the result.

B.74.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.74.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.74.5. Execution

```

XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<16>(virtual_address), 16);

```

B.74.6. Exceptions

DRAFT

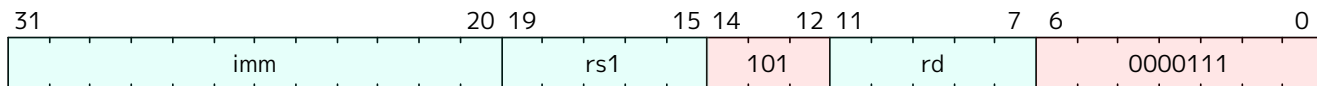
B.75. lhu

Load halfword unsigned

This instruction is defined by:

- $I, \geq 0$

B.75.1. Encoding



B.75.2. Synopsis

Load 16 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Zero extend the result.

B.75.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.75.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.75.5. Execution

```

XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<16>(virtual_address);

```


B.75.6. Exceptions

DRAFT

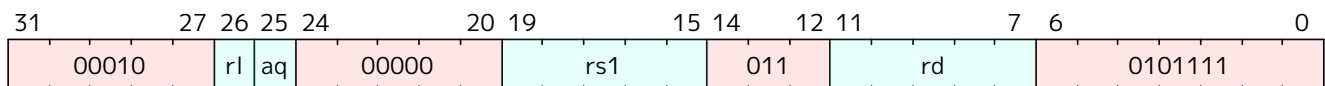
B.76. lr.d

Load reserved doubleword

This instruction is defined by any of the following:

- A, >= 0
- Zalrsc, >= 0

B.76.1. Encoding



B.76.2. Synopsis

Loads a word from the address in rs1, places the value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

The address in rs1 must be 8-byte aligned.

If the address is not naturally aligned, a **LoadAddressMisaligned** exception or an **LoadAccessFault** exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR.

These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.76.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.76.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.76.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<XLEN, 64>(virtual_address)) {
    if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception")
    {
        raise(ExceptionCode::LoadAddressMisaligned, virtual_address);
    } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault")
    {
        raise(ExceptionCode::LoadAccessFault, virtual_address);
    } else {
        unpredictable("Implementations may raise either a
LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
    }
}
X[rd] = load_reserved<32>(virtual_address, aq, rl);
```

B.76.6. Exceptions

DRAFT

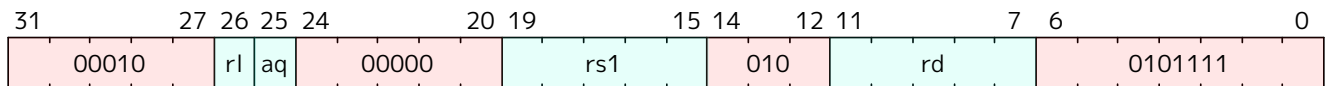
B.77. lr.w

Load reserved word

This instruction is defined by:

- A, ≥ 0

B.77.1. Encoding



B.77.2. Synopsis

Loads a word from the address in rs1, places the sign-extended value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

<%- if XLEN == 64 -%> The 32-bit load result is sign-extended to 64-bits. <%- end -%>

The address in rs1 must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words).

If the address is not naturally aligned, a **LoadAddressMisaligned** exception or an **LoadAccessFault** exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR.

These writes are not required to invalidate the reservation when they access other bytes in

the reservation set.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.77.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.77.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.77.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<XLEN, 32>(virtual_address)) {
    if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception")
    {
        raise(ExceptionCode::LoadAddressMisaligned, virtual_address);
    } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault")
    {
        raise(ExceptionCode::LoadAccessFault, virtual_address);
    } else {
        unpredictable("Implementations may raise either a
LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
    }
}
XReg load_value = load_reserved<32>(virtual_address, aq, rl);
if (xlen() == 64) {
    X[rd] = load_value;
} else {
    X[rd] = sext(load_value[31:0], 32);
}

```

}

DRAFT

B.77.6. Exceptions

DRAFT

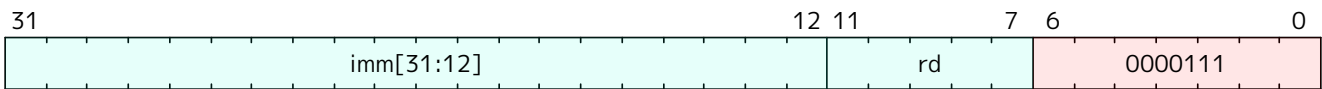
B.78. lui

Load upper immediate

This instruction is defined by:

- I, >= 0

B.78.1. Encoding



B.78.2. Synopsis

Load the zero-extended imm into rd.

B.78.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.78.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.78.5. Execution

```
X[rd] = imm;
```

B.78.6. Exceptions

DRAFT

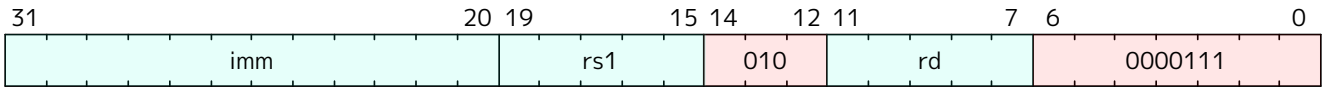
B.79. lw

Load word

This instruction is defined by:

- I, >= 0

B.79.1. Encoding



B.79.2. Synopsis

Load 32 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Sign extend the result.

B.79.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.79.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.79.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address);
```

B.79.6. Exceptions

DRAFT

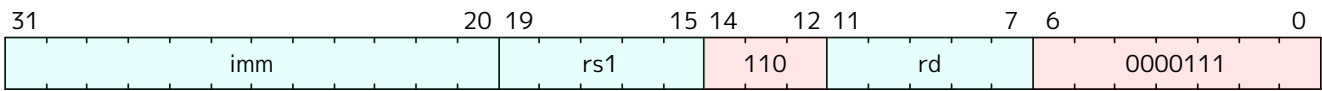
B.80. lwu

Load word unsigned

This instruction is defined by:

- $I, \geq 0$

B.80.1. Encoding



B.80.2. Synopsis

Load 64 bits of data into register **rd** from an address formed by adding **rs1** to a signed offset. Zero extend the result.

B.80.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.80.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.80.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address);
```

B.80.6. Exceptions

DRAFT

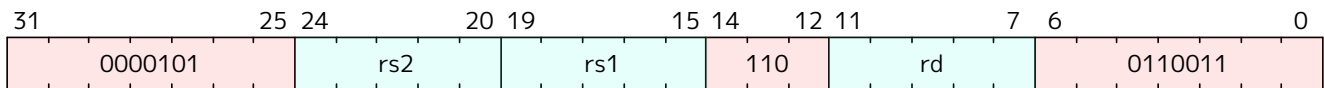
B.81. max

Maximum

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.81.1. Encoding



B.81.2. Synopsis

This instruction returns the larger of two signed integers.

Software Hint

Calculating the absolute value of a signed integer can be performed using the following sequence: **neg rD,rS** followed by **max rD,rS,rD**. When using this common sequence, it is suggested that they are scheduled with no intervening instructions so that implementations that are so optimized can fuse them together.

B.81.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.81.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.81.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = ($signed(X[rs1]) > $signed(X[rs2])) ? X[rs1] : X[rs2];
```

B.81.6. Exceptions

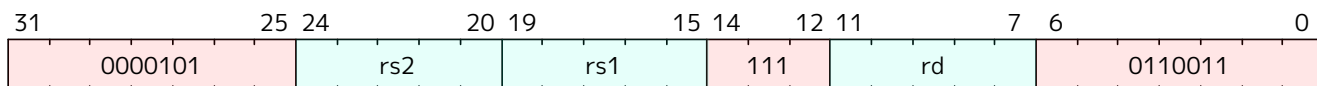
B.82. maxu

Unsigned maximum

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.82.1. Encoding



B.82.2. Synopsis

This instruction returns the larger of two unsigned integers.

B.82.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.82.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.82.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = (X[rs1] > X[rs2]) ? X[rs1] : X[rs2];

```


B.82.6. Exceptions

DRAFT

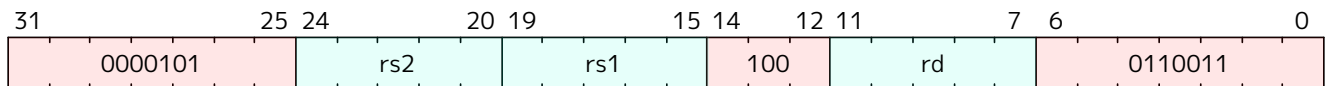
B.83. min

Minimum

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.83.1. Encoding



B.83.2. Synopsis

This instruction returns the smaller of two signed integers.

B.83.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.83.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.83.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = ($signed(X[rs1]) < $signed(X[rs2])) ? X[rs1] : X[rs2];

```

B.83.6. Exceptions

DRAFT

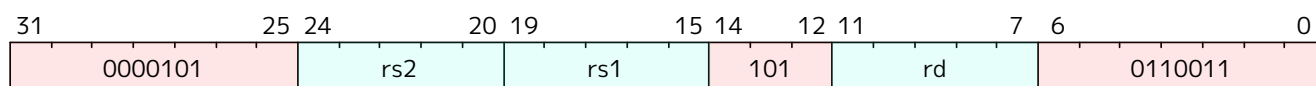
B.84. minu

Unsigned minimum

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.84.1. Encoding



B.84.2. Synopsis

This instruction returns the smaller of two unsigned integers.

B.84.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.84.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.84.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = (X[rs1] < X[rs2]) ? X[rs1] : X[rs2];

```

B.84.6. Exceptions

DRAFT

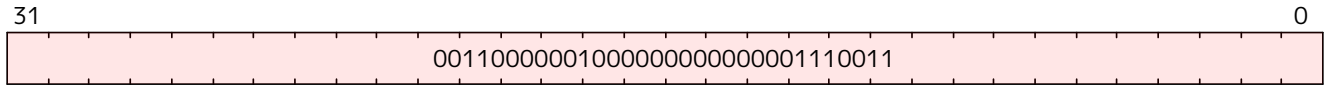
B.85. mret

Machine Exception Return

This instruction is defined by:

- $I, \geq 0$

B.85.1. Encoding



B.85.2. Synopsis

Returns from an exception in M-mode.

B.85.3. Access

M	HS	U	VS	VU
Always	Never	Never	Never	Never

B.85.4. Decode Variables

B.85.5. Execution

```

if (implemented?(ExtensionName::S) && CSR[mstatus].MPP != 2'b11) {
    CSR[mstatus].MPRV = 0;
}
CSR[mstatus].MIE = CSR[mstatus].MPIE;
CSR[mstatus].MPIE = 1;
if (CSR[mstatus].MPP == 2'b00) {
    set_mode(PrivilegeMode::U);
} else if (CSR[mstatus].MPP == 2'b01) {
    set_mode(PrivilegeMode::S);
} else if (CSR[mstatus].MPP == 2'b11) {
    set_mode(PrivilegeMode::M);
}
CSR[mstatus].MPP = implemented?(ExtensionName::U) ? 2'b00 : 2'b11;
PC = $bits(CSR[mepc]);

```

B.85.6. Exceptions

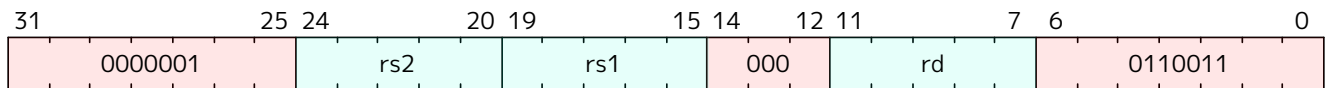
B.86. mul

Signed multiply

This instruction is defined by any of the following:

- M, >= 0
- Zmmul, >= 0

B.86.1. Encoding



B.86.2. Synopsis

MUL performs an XLEN-bitxXLEN-bit multiplication of **rs1** by **rs2** and places the lower XLEN bits in the destination register. Any overflow is thrown away.



If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

B.86.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.86.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.86.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
X[rd] = (src1 * src2)[XLEN - 1:0];
```

B.86.6. Exceptions

DRAFT

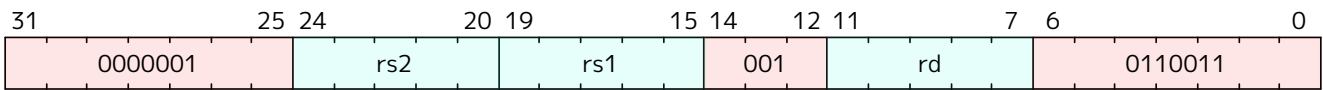
B.87. mulh

Signed multiply high

This instruction is defined by any of the following:

- M, >= 0
- Zmmul, >= 0

B.87.1. Encoding



B.87.2. Synopsis

Multiply the signed values in rs1 to rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulh rdh, rs1, rs2
mul  rdl, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.87.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.87.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd  = $encoding[11:7];
```

B.87.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
```

```
}  
Bits<1> rs1_sign_bit = X[rs1][XLEN - 1];  
Bits<XLEN << 1> src1 = {{XLEN{rs1_sign_bit}}, X[rs1]};  
Bits<1> rs2_sign_bit = X[rs2][XLEN - 1];  
Bits<XLEN << 1> src2 = {{XLEN{rs2_sign_bit}}, X[rs2]};  
X[rd] = (src1 * src2)[(XLEN * 8'd2) - 1:XLEN];
```

DRAFT

B.87.6. Exceptions

DRAFT

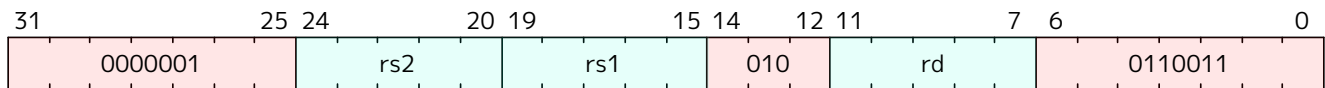
B.88. mulhsu

Signed/unsigned multiply high

This instruction is defined by any of the following:

- M, >= 0
- Zmmul, >= 0

B.88.1. Encoding



B.88.2. Synopsis

Multiply the signed value in rs1 by the unsigned value in rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulhsu rdh, rs1, rs2
mul    rdL, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.88.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.88.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd  = $encoding[11:7];
```

B.88.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
```

```
}  
Bits<1> rs1_sign_bit = X[rs1][XLEN - 1];  
Bits<XLEN * 8'd2> src1 = {{XLEN{rs1_sign_bit}}, X[rs1]};  
Bits<XLEN * 8'd2> src2 = {{XLEN{1'b0}}, X[rs2]};  
X[rd] = (src1 * src2)[(XLEN * 8'd2) - 1:XLEN];
```

DRAFT

B.88.6. Exceptions

DRAFT

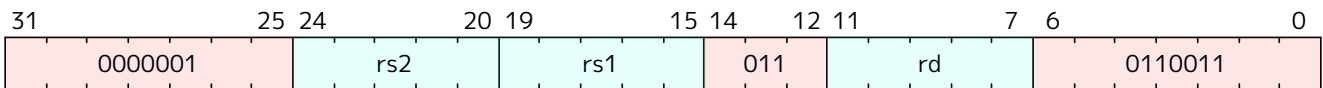
B.89. mulhu

Unsigned multiply high

This instruction is defined by any of the following:

- M, >= 0
- Zmmul, >= 0

B.89.1. Encoding



B.89.2. Synopsis

Multiply the unsigned values in rs1 to rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulhu rdh, rs1, rs2
mul    rdl, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.89.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.89.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd  = $encoding[11:7];
```

B.89.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
```

```
}  
Bits<XLEN * 8'd2> src1 = {{XLEN{1'b0}}, X[rs1]};  
Bits<XLEN * 8'd2> src2 = {{XLEN{1'b0}}, X[rs2]};  
X[rd] = (src1 * src2)[(XLEN * 8'd2) - 1:XLEN];
```

DRAFT

B.89.6. Exceptions

DRAFT

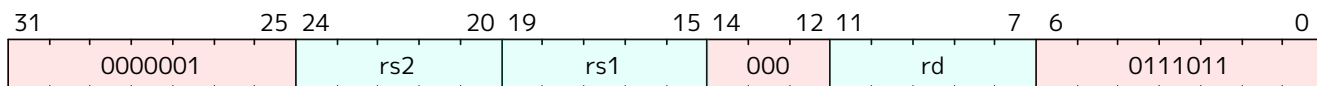
B.90. mulw

Signed 32-bit multiply

This instruction is defined by any of the following:

- M, >= 0
- Zmmul, >= 0

B.90.1. Encoding



B.90.2. Synopsis

Multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register.

Any overflow is thrown away.



In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].

B.90.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.90.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.90.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
Bits<32> result = src1 * src2;

```

```
Bits<1> sign_bit = result[31];  
X[rd] = {{32{sign_bit}}, result};
```

DRAFT

B.90.6. Exceptions

DRAFT

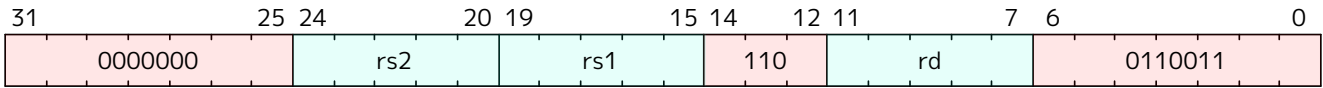
B.91. or

Or

This instruction is defined by:

- $I, \geq 0$

B.91.1. Encoding



B.91.2. Synopsis

Or rs1 with rs2, and store the result in rd

B.91.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.91.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.91.5. Execution

```
X[rd] = X[rs1] | X[rs2];
```

B.91.6. Exceptions

DRAFT

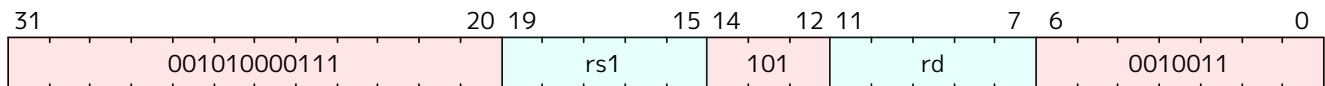
B.92. orc.b

Bitware OR-combine, byte granule

This instruction is defined by any of the following:

- B, ≥ 0
- Zbb, ≥ 0

B.92.1. Encoding



B.92.2. Synopsis

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result rd to all zeros if no bit within the respective byte of rs is set, or to all ones if any bit within the respective byte of rs is set.

B.92.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.92.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.92.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg input = X[rs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
  output[(i + 7):i] = (input[(i + 7):i] == 0) ? 8'd0 : ~8'd0;
}
X[rd] = output;
```

B.92.6. Exceptions

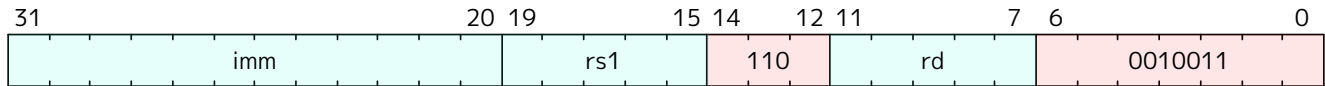
B.93. ori

Or immediate

This instruction is defined by:

- $I, \geq 0$

B.93.1. Encoding



B.93.2. Synopsis

Or an immediate to the value in rs1, and store the result in rd

B.93.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.93.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.93.5. Execution

```

if (implemented?(ExtensionName::Zicbop)) {
  if (rd == 0) {
    if (imm[4:0] == 0) {
      Bits<12> offset = {imm[31:25], imm[11:7]};
      prefetch_instruction(offset);
    } else if (imm[4:0] == 1) {
      Bits<12> offset = {imm[31:25], imm[11:7]};
      prefetch_read(offset);
    } else if (imm[4:0] == 3) {
      Bits<12> offset = {imm[31:25], imm[11:7]};
      prefetch_write(offset);
    }
  }
}

```



```
X[rd] = X[rs1] | imm;
```

DRAFT

B.93.6. Exceptions

DRAFT

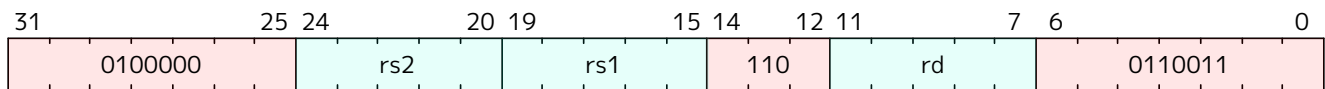
B.94. orn

OR with inverted operand

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.94.1. Encoding



B.94.2. Synopsis

This instruction performs the bitwise logical OR operation between rs1 and the bitwise inversion of rs2.

B.94.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.94.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.94.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs1] | ~X[rs2];

```

B.94.6. Exceptions

DRAFT

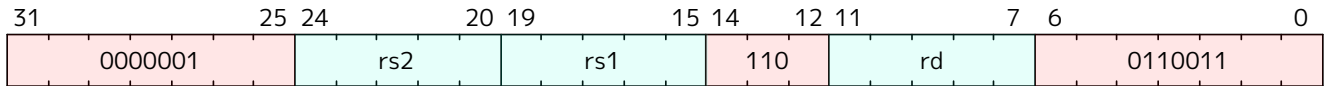
B.95. rem

Signed remainder

This instruction is defined by:

- $M, \geq 0$

B.95.1. Encoding



B.95.2. Synopsis

Calculate the remainder of signed division of rs1 by rs2, and store the result in rd.

If the value in register rs2 is zero, write the value in rs1 into rd;

If the result of the division overflows, write zero into rd;

B.95.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.95.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.95.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = src1;
} else if ((src1 == {1'b1, {XLEN - 1{1'b0}}}) && (src2 == {XLEN{1'b1}})) {
  {
    X[rd] = 0;
  } else {
```

```
X[rd] = $signed(src1) % $signed(src2);  
}
```

DRAFT

B.95.6. Exceptions

DRAFT

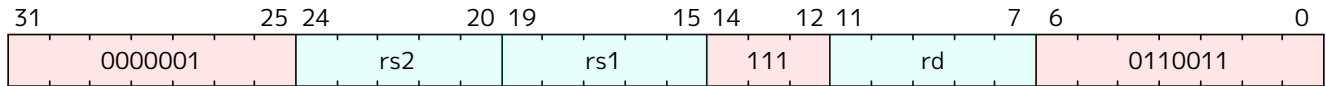
B.96. remu

Unsigned remainder

This instruction is defined by:

- $M, \geq 0$

B.96.1. Encoding



B.96.2. Synopsis

Calculate the remainder of unsigned division of rs1 by rs2, and store the result in rd.

B.96.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.96.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.96.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = src1;
} else {
  X[rd] = src1 % src2;
}

```


B.96.6. Exceptions

DRAFT

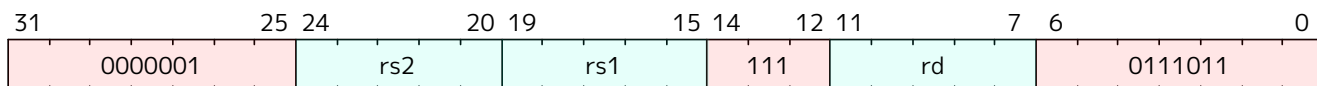
B.97. remuw

Unsigned 32-bit remainder

This instruction is defined by:

- $M, \geq 0$

B.97.1. Encoding



B.97.2. Synopsis

Calculate the remainder of unsigned division of the 32-bit values in rs1 by rs2, and store the sign-extended result in rd.

If the value in rs2 is zero, rd gets the sign-extended value in rs1.

B.97.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.97.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.97.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[rd] = {{32{sign_bit}}, src1};
} else {
  Bits<32> result = src1 % src2;
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}

```

}

DRAFT

B.97.6. Exceptions

DRAFT

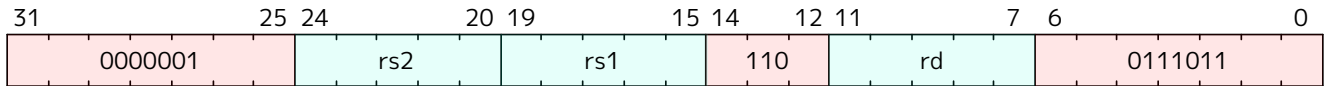
B.98. remw

Signed 32-bit remainder

This instruction is defined by:

- $M, \geq 0$

B.98.1. Encoding



B.98.2. Synopsis

Calculate the remainder of signed division of the 32-bit values rs1 by rs2, and store the sign-extended result in rd.

If the value in register rs2 is zero, write the sign-extended 32-bit value in rs1 into rd;

If the result of the division overflows, write zero into rd;

B.98.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.98.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.98.5. Execution

```

if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[rd] = {{32{sign_bit}}, src1};
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {
  X[rd] = 0;
}

```

```
} else {  
  Bits<32> result = $signed(src1) % $signed(src2);  
  Bits<1> sign_bit = result[31];  
  X[rd] = {{32{sign_bit}}, result};  
}
```

DRAFT

B.98.6. Exceptions

DRAFT

B.99. rev8

Byte-reverse register (RV64 encoding)

This instruction is defined by any of the following:

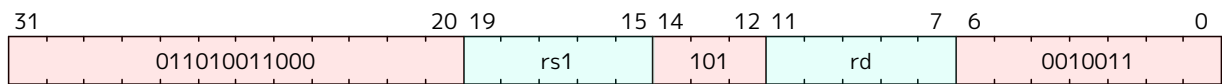
- B, ≥ 0
- Zbb, ≥ 0
- Zbkb, ≥ 0

B.99.1. Encoding

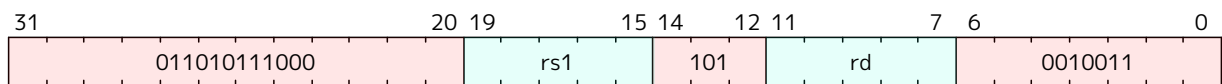


This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.99.2. Synopsis

This instruction reverses the order of the bytes in rs1.



The rev8 mnemonic corresponds to different instruction encodings in RV32 and RV64.



*The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a **rev8 rd,rs** followed by a **srai rd,rd,K**, where K is XLEN-32 and XLEN-16, respectively.*

B.99.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.99.4. Decode Variables

RV32

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```


RV64

```

Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.99.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg input = X[rs1];
XReg output = 0;
XReg j = xlen() - 1;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
    output[(i + 7):i] = input[j:(j - 7)];
    j = j - 8;
}
X[rd] = output;

```

B.99.6. Exceptions

DRAFT

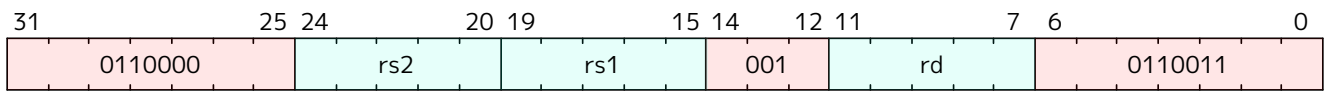
B.100. rol

Rotate left (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.100.1. Encoding



B.100.2. Synopsis

This instruction performs a rotate left of rs1 by the amount in least-significant $\log_2(\text{XLEN})$ bits of rs2.

B.100.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.100.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.100.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg shamt = (xlen() == 32) ? X[rs2][4:0] : X[rs2][5:0];
X[rd] = (X[rs1] << shamt) | (X[rs1] >> (xlen() - shamt));
```

B.100.6. Exceptions

DRAFT

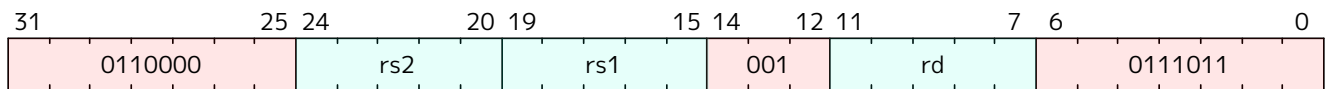
B.101. rolw

Rotate left word (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.101.1. Encoding



B.101.2. Synopsis

This instruction performs a rotate left of the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

B.101.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.101.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.101.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg shamt = X[rs2][4:0];
XReg unextended_result = (rs1_word << shamt) | (rs1_word >> (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};

```

B.101.6. Exceptions

DRAFT

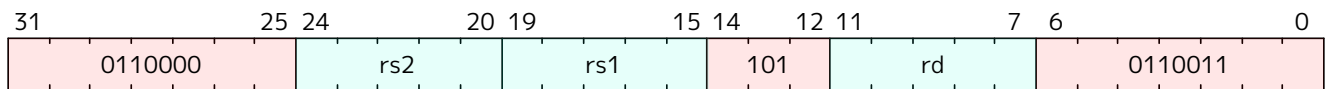
B.102. ror

Rotate right (Register)

This instruction is defined by any of the following:

- B, ≥ 0
- Zbb, ≥ 0
- Zbkb, ≥ 0

B.102.1. Encoding



B.102.2. Synopsis

This instruction performs a rotate right of rs1 by the amount in least-significant $\log_2(\text{XLEN})$ bits of rs2.

B.102.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.102.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.102.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg shamt = (xlen() == 32) ? X[rs2][4:0] : X[rs2][5:0];
X[rd] = (X[rs1] >> shamt) | (X[rs1] << (xlen() - shamt));

```

B.102.6. Exceptions

DRAFT

B.103. rori

Rotate right (Immediate)

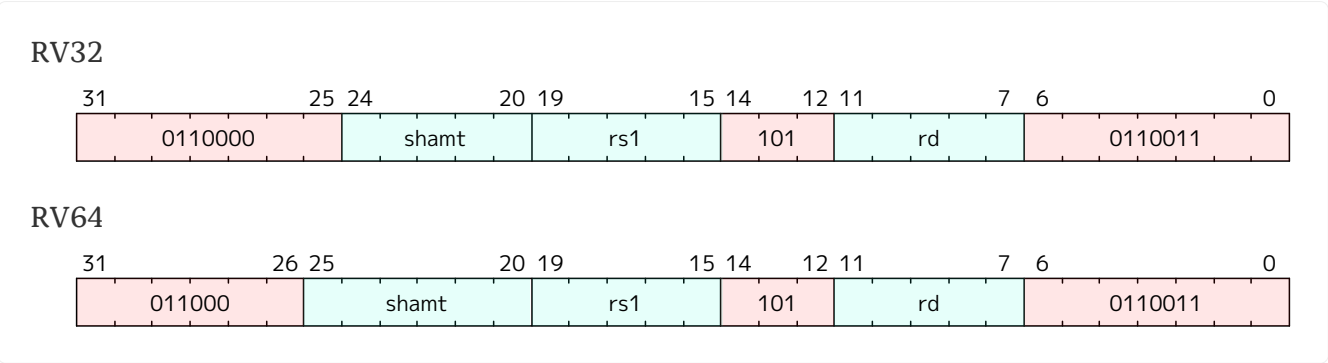
This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.103.1. Encoding



This instruction has different encodings in RV32 and RV64.



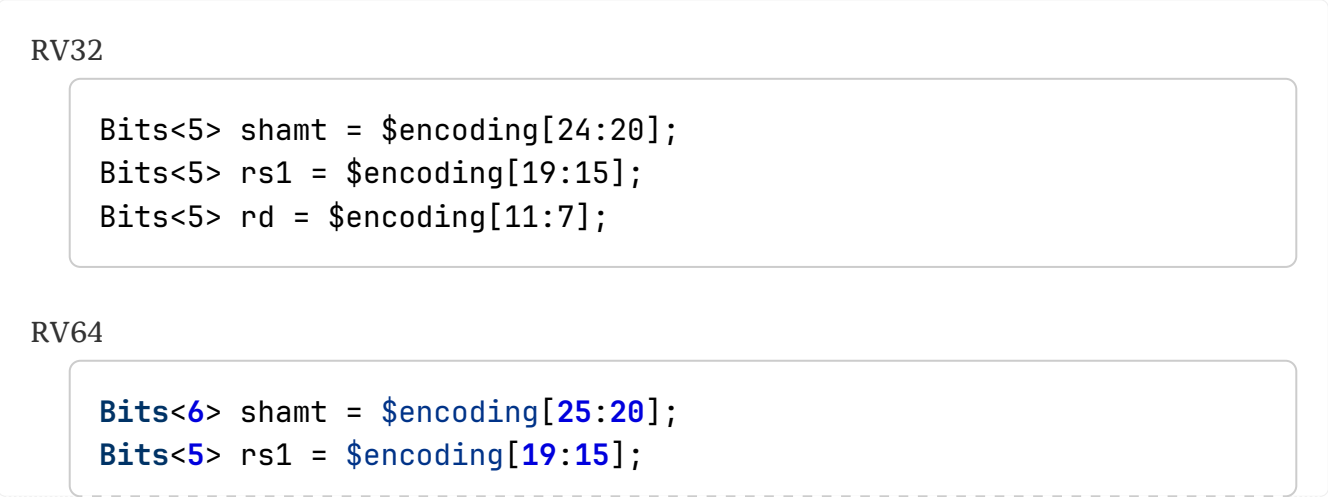
B.103.2. Synopsis

This instruction performs a rotate right of rs1 by the amount in the least-significant log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.103.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.103.4. Decode Variables



```
Bits<5> rd = $encoding[11:7];
```

B.103.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
XReg shamt = (xlen() == 32) ? shamt[4:0] : shamt[5:0];  
X[rd] = (X[rs1] >> shamt) | (X[rs1] << (xlen() - shamt));
```

DRAFT

B.103.6. Exceptions

DRAFT

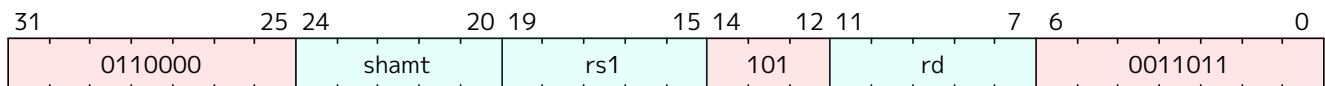
B.104. roriw

Rotate right word (Immediate)

This instruction is defined by any of the following:

- B, ≥ 0
- Zbb, ≥ 0
- Zbkb, ≥ 0

B.104.1. Encoding



B.104.2. Synopsis

This instruction performs a rotate right on the least-significant word of rs1 by the amount in the least-significant $\log_2(\text{XLEN})$ bits of shamt. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

B.104.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.104.4. Decode Variables

```

Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.104.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg unextended_result = (X[rs1] >> shamt) | (X[rs1] << (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};

```

B.104.6. Exceptions

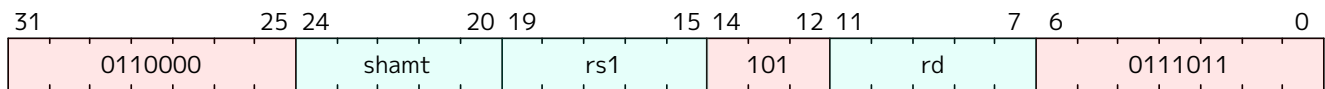
B.105. rorw

Rotate right word (Register)

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.105.1. Encoding



B.105.2. Synopsis

This instruction performs a rotate right on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

B.105.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.105.4. Decode Variables

```

Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.105.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg shamt = X[rs1][4:0];
XReg unextended_result = (X[rs1] >> shamt) | (X[rs1] << (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};

```

B.105.6. Exceptions

DRAFT

B.106. sb

Store byte

This instruction is defined by:

- I, >= 0

B.106.1. Encoding



B.106.2. Synopsis

Store 8 bits of data from register **rs2** to an address formed by adding **rs1** to a signed offset.

B.106.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.106.4. Decode Variables

```
Bits<12> imm = { $encoding[31:25], $encoding[11:7] };
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.106.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<8>(virtual_address, X[rs2][7:0]);
```

B.106.6. Exceptions

DRAFT

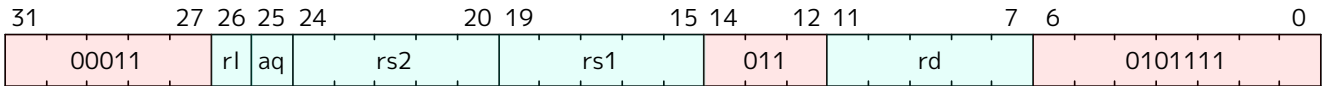
B.107. sc.d

Store conditional doubleword

This instruction is defined by:

- A, >= 0

B.107.1. Encoding



B.107.2. Synopsis

sc.d conditionally writes a doubleword in rs2 to the address in rs1: the sc.d succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.d succeeds, the instruction writes the doubleword in rs2 to memory, and it writes zero to rd. If the sc.d fails, the instruction does not write to memory, and it writes a nonzero value to rd. For the purposes of memory protection, a failed sc.d may be treated like a store. Regardless of success or failure, executing an sc.d instruction invalidates any reservation held by this hart.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in rs1 must be naturally aligned to the size of the operand (i.e., eight-byte aligned). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.



Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- *during a preemptive context switch, and*
- *if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction

operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.107.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.107.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.107.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<XLEN, 64>(virtual_address)) {
    if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception")
    {
        raise(ExceptionCode::LoadAddressMisaligned, virtual_address);
    } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault")
    {
        raise(ExceptionCode::LoadAccessFault, virtual_address);
    } else {
        unpredictable("Implementations may raise either a
LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
    }
}
Boolean success = store_conditional<64>(virtual_address, value, aq, rl);
X[rd] = success ? 0 : 1;

```

B.107.6. Exceptions

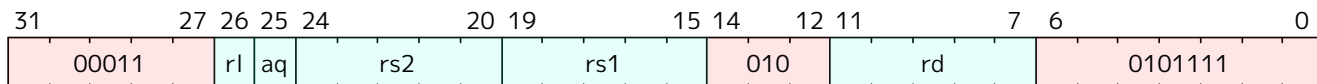
B.108. sc.w

Store conditional word

This instruction is defined by:

- A, >= 0

B.108.1. Encoding



B.108.2. Synopsis

`sc.w` conditionally writes a word in `rs2` to the address in `rs1`: the `sc.w` succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the `sc.w` succeeds, the instruction writes the word in `rs2` to memory, and it writes zero to `rd`. If the `sc.w` fails, the instruction does not write to memory, and it writes a nonzero value to `rd`. For the purposes of memory protection, a failed `sc.w` may be treated like a store. Regardless of success or failure, executing an `sc.w` instruction invalidates any reservation held by this hart.

<%- if XLEN == 64 -%>



If a value other than 0 or 1 is defined as a result for `sc.w`, the value will before sign-extended into `rd`. <%- end -%>

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in `rs1` must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.



Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is

allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:



- during a preemptive context switch, and
- if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.



The LR/SC sequence can be given acquire semantics by setting the `aq` bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the `rl` bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the `aq` bit on the LR instruction, and setting the `rl` bit on the SC instruction makes the LR/SC sequence sequentially consistent in the `C` memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other `C` atomic operations, or stronger

notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.108.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.108.4. Decode Variables

```

Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.108.5. Execution

```

if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<XLEN, 32>(virtual_address)) {
    if (LRSC_MISALIGNED_BEHAVIOR == ""always raise misaligned exception")
    {
        raise(ExceptionCode::LoadAddressMisaligned, virtual_address);
    } else if (LRSC_MISALIGNED_BEHAVIOR == ""always raise access fault")
    {
        raise(ExceptionCode::LoadAccessFault, virtual_address);
    } else {
        unpredictable(""Implementations may raise either a
LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
    }
}
}

```

```
Boolean success = store_conditional<32>(virtual_address, value, aq, rl);  
X[rd] = success ? 0 : 1;
```

DRAFT

B.108.6. Exceptions

DRAFT

B.109. sd

Store doubleword

This instruction is defined by:

- I, >= 0

B.109.1. Encoding



B.109.2. Synopsis

Store 64 bits of data from register **rs2** to an address formed by adding **rs1** to a signed offset.

B.109.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.109.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[31:25], $encoding[11:7]});
Bits<5> rs1 = $encoding[19:15];
Bits<5> rs2 = $encoding[24:20];
```

B.109.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<64>(virtual_address, X[rs2]);
```

B.109.6. Exceptions

DRAFT

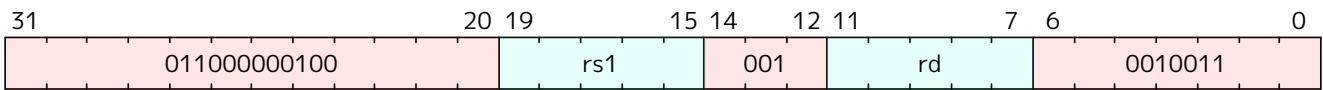
B.110. sext.b

Sign-extend byte

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.110.1. Encoding



B.110.2. Synopsis

This instruction sign-extends the least-significant byte in the source to XLEN by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

B.110.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.110.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.110.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (xlen() == 32) {
  X[rd] = {{24{X[rs1][7]}}, X[rs1][7:0]};
} else if (xlen() == 64) {
  X[rd] = {{56{X[rs1][7]}}, X[rs1][7:0]};
}
```

B.110.6. Exceptions

DRAFT

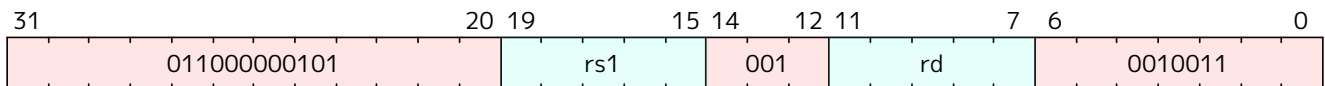
B.111. sext.h

Sign-extend halfword

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.111.1. Encoding



B.111.2. Synopsis

This instruction sign-extends the least-significant halfword in the source to XLEN by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

B.111.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.111.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.111.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (xlen() == 32) {
  X[rd] = {{16{X[rs1][15]}}, X[rs1][15:0]};
} else if (xlen() == 64) {
  X[rd] = {{48{X[rs1][15]}}, X[rs1][15:0]};
}
```

B.111.6. Exceptions

DRAFT

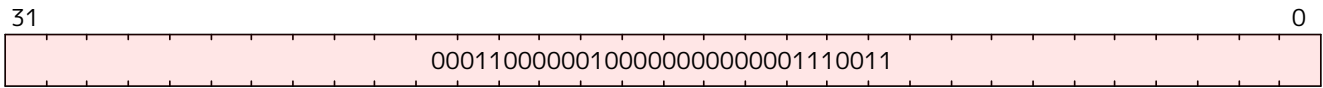
B.112. sfence.inval.ir

Order implicit page table reads after invalidation

This instruction is defined by:

- Svinval, >= 0

B.112.1. Encoding



B.112.2. Synopsis

The [sfence.inval.ir](#) instruction guarantees that any previous [sINVAL.vma](#) instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

B.112.3. Access

M	HS	U	VS	VU
Always	Sometimes	Never	Sometimes	Never

B.112.4. Decode Variables

B.112.5. Execution

```
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (CSR[misa].H == 1) {
    vma_type.vsmode = true;
    vma_type.gstage = true;
}
order_pgtbl_reads_after_vmafence(vma_type);
```

B.112.6. Exceptions

DRAFT

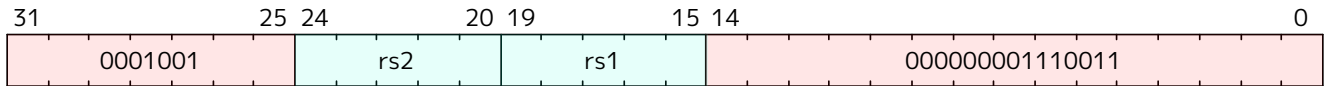
B.113. sfence.vma

Supervisor memory-management fence

This instruction is defined by:

- $S, \geq 0$

B.113.1. Encoding



B.113.2. Synopsis

The supervisor memory-management fence instruction **SFENCE.VMA** is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *rs1* and *rs2*, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see [\[sv32algorithm\]](#)). Further details on the behavior of this instruction are described in [\[virt-control\]](#) and [\[pmp-vmem\]](#).



The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

SFENCE.VMA orders only the local hart's implicit references to the memory-management data structures.



Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VMA in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shutdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *rs2* can specify the address space. The behavior of SFENCE.VMA depends on *rs1* and *rs2* as follows:

- If $rs1 = x0$ and $rs2 = x0$, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address

spaces.

- If $rs1=x0$ and $rs2\neq x0$, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register $rs2$. Accesses to *global* mappings (see [\[translation\]](#)) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register $rs2$, except for entries containing global mappings.
- If $rs1\neq x0$ and $rs2=x0$, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in $rs1$, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in $rs1$, for all address spaces.
- If $rs1\neq x0$ and $rs2\neq x0$, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in $rs1$, for the address space identified by integer register $rs2$. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in $rs1$ and that match the address space identified by integer register $rs2$, except for entries containing global mappings.

If the value held in $rs1$ is not a valid virtual address, then the SFENCE.VMA instruction has no effect. No exception is raised in this case.

When $rs2\neq x0$, bits SXLEN-1:ASIDMAX of the value held in $rs2$ are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLLEN<ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLLEN of the value held in $rs2$.



It is always legal to over-fence, e.g., by fencing only based on a subset of the bits in $rs1$ and/or $rs2$, and/or by simply treating all SFENCE.VMA instructions as having $rs1=x0$ and/or $rs2=x0$. For example, simpler implementations can ignore the virtual address in $rs1$ and the ASID value in $rs2$ and always perform a global fence. The choice not to raise an exception when an invalid virtual address is held in $rs1$ facilitates this type of simplification.

An implicit read of the memory-management data structures may return any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. The ordering implied by SFENCE.VMA does not place implicit reads and writes to the memory-management data structures into the global memory order in a way that interacts cleanly with the standard RVWMO ordering rules. In particular, even though an SFENCE.VMA orders prior explicit accesses before subsequent implicit accesses, and those implicit accesses are ordered before their associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.



A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-

leaf PTE's valid bit and executing an `SFENCE.VMA` with `rs1=x0`. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the `satp` register or a subsequent valid (V=1) translation data structure entry, and must only raise exceptions for implicit accesses that are generated as a result of instruction execution, not those that are performed speculatively.

Changes to the `sstatus` fields SUM and MXR take effect immediately, without the need to execute an `SFENCE.VMA` instruction. Changing `satp.MODE` from Bare to other modes and vice versa also takes effect immediately, without the need to execute an `SFENCE.VMA` instruction. Likewise, changes to `satp.ASID` take effect immediately.

The following common situations typically require executing an `SFENCE.VMA` instruction:

- When software recycles an ASID (i.e., reassociates it with a different page table), it should first change `satp` to point to the new page table using the recycled ASID, then execute `SFENCE.VMA` with `rs1=x0` and `rs2` set to the recycled ASID. Alternatively, software can execute the same `SFENCE.VMA` instruction while a different ASID is loaded into `satp`, provided the next time `satp` is loaded with the recycled ASID, it is simultaneously loaded with the new page table.
- If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every `satp` write, software should execute `SFENCE.VMA` with `rs1=x0`. In the common case that no global translations have been modified, `rs2` should be set to a register other than `x0` but which contains the value zero, so that global translations are not flushed.
- If software modifies a non-leaf PTE, it should execute `SFENCE.VMA` with `rs1=x0`. If any PTE along the traversal path had its G bit set, `rs2` must be `x0`; otherwise, `rs2` should be set to the ASID for which the translation is being modified.
- If software modifies a leaf PTE, it should execute `SFENCE.VMA` with `rs1` set to a virtual address within the page. If any PTE along the traversal path had its G bit set, `rs2` must be `x0`; otherwise, `rs2` should be set to the ASID for which the translation is being modified.
- For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the `SFENCE.VMA` lazily. After modifying the PTE but before executing `SFENCE.VMA`, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute `SFENCE.VMA` in accordance with the previous



bullet point.

If a hart employs an address-translation cache, that cache must appear to be private to that hart. In particular, the meaning of an ASID is local to a hart; software may choose to use the same ASID to refer to different address spaces on different harts.



A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shutdown. However, as Oses have evolved to significantly reduce the scope of TLB shutdowns using novel ASID-management techniques, we expect the local-ASID scheme to remain attractive for its simplicity and possibly better scalability.

For implementations that make `satp.MODE` read-only zero (always Bare), attempts to execute an `SFENCE.VMA` instruction might raise an illegal-instruction exception.

B.113.3. Access

M	HS	U	VS	VU
Always	Always	Never	Always	Never

B.113.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.113.5. Execution

```
XReg vaddr = X[rs1];
Bits<ASID_WIDTH> asid = X[rs2][ASID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) || (mode() ==
PrivilegeMode::VS) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[misa].H == 1 && CSR[hstatus].VTVM == 1 && mode() ==
PrivilegeMode::VS) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
if (!implemented?(ExtensionName::Sv32) && !implemented?(ExtensionName
::Sv39) && !implemented?(ExtensionName::Sv48) && !implemented?
```

```

(ExtensionName::Sv57)) {
    if (TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY) {
        raise(ExceptionCode::IllegalInstruction, $encoding);
    }
}
VmaInvalType vma_type;
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VS) {
    vma_type.vsmode = true;
    vma_type.single_vmid = true;
    vma_type.vmid = CSR[hgatp].VMID;
} else {
    vma_type.smode = true;
}
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        order_pgtbl_writes_before_vmafence(vma_type);
        invalidate_translations(vma_type);
        order_pgtbl_reads_after_vmafence(vma_type);
    }
} else {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_asid = true;
        vma_type.asid = asid;
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        order_pgtbl_writes_before_vmafence(vma_type);
        invalidate_translations(vma_type);
        order_pgtbl_reads_after_vmafence(vma_type);
    }
}
}

```

B.113.6. Exceptions

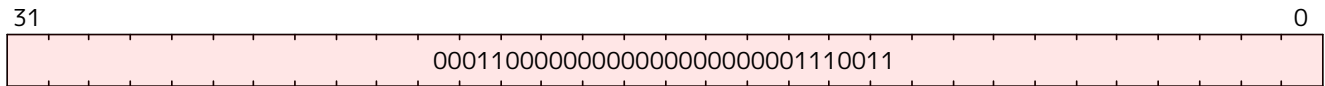
B.114. sfence.w.inval

Order writes before sfence

This instruction is defined by:

- Svinval, ≥ 0

B.114.1. Encoding



B.114.2. Synopsis

The [sfence.w.inval](#) instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent [sINVAL.vma](#) instructions executed by the same hart.

B.114.3. Access

M	HS	U	VS	VU
Always	Sometimes	Never	Sometimes	Never

B.114.4. Decode Variables

B.114.5. Execution

```

if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (CSR[misa].H == 1) {
    vma_type.vsmode = true;
    vma_type.gstage = true;
}
order_pgtbl_writes_before_vma_fence(vma_type);

```

B.114.6. Exceptions

DRAFT

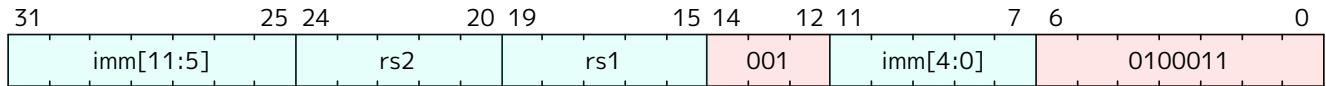
B.115. sh

Store halfword

This instruction is defined by:

- $I, \geq 0$

B.115.1. Encoding



B.115.2. Synopsis

Store 16 bits of data from register **rs2** to an address formed by adding **rs1** to a signed offset.

B.115.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.115.4. Decode Variables

```

Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.115.5. Execution

```

XReg virtual_address = X[rs1] + imm;
write_memory<16>(virtual_address, X[rs2][15:0]);

```


B.115.6. Exceptions

DRAFT

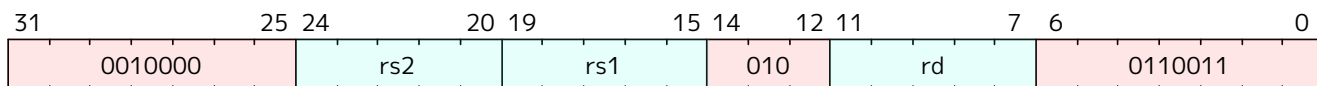
B.116. sh1add

Shift left by 1 and add

This instruction is defined by any of the following:

- B, >= 0
- Zba, >= 0

B.116.1. Encoding



B.116.2. Synopsis

This instruction shifts **rs1** to the left by 1 bit and adds it to **rs2**.

B.116.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.116.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.116.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 1);

```

B.116.6. Exceptions

DRAFT

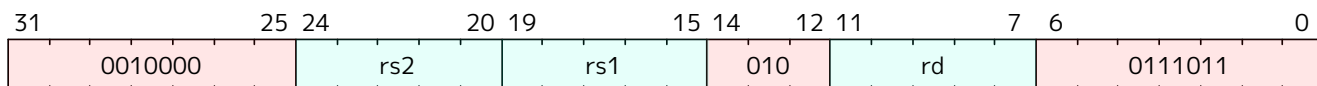
B.117. sh1add.uw

Shift unsigend word left by 1 and add

This instruction is defined by any of the following:

- B, >= 0
- Zba, >= 0

B.117.1. Encoding



B.117.2. Synopsis

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 place.

B.117.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.117.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.117.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 1);
```

B.117.6. Exceptions

DRAFT

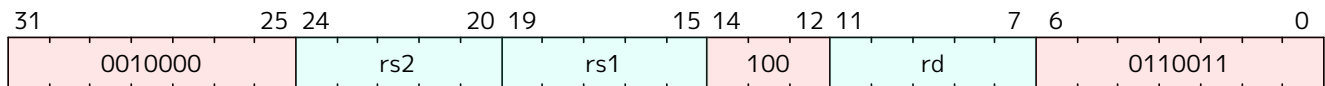
B.118. sh2add

Shift left by 2 and add

This instruction is defined by any of the following:

- B, >= 0
- Zba, >= 0

B.118.1. Encoding



B.118.2. Synopsis

This instruction shifts **rs1** to the left by 2 places and adds it to **rs2**.

B.118.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.118.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.118.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 2);

```

B.118.6. Exceptions

DRAFT

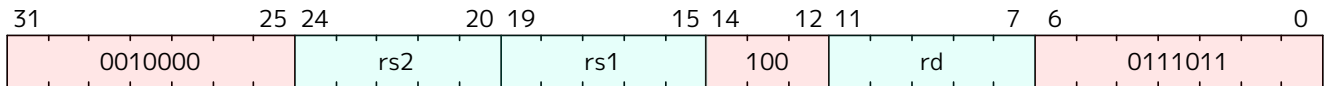
B.119. sh2add.uw

Shift unsigned word left by 2 and add

This instruction is defined by any of the following:

- B, ≥ 0
- Zba, ≥ 0

B.119.1. Encoding



B.119.2. Synopsis

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 2 places.

B.119.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.119.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.119.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 2);

```


B.119.6. Exceptions

DRAFT

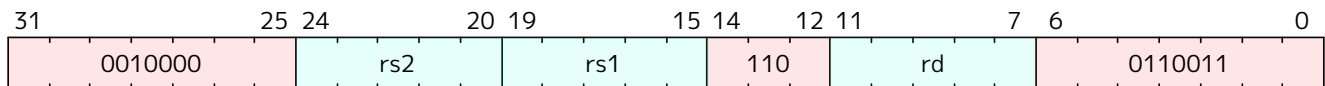
B.120. sh3add

Shift left by 3 and add

This instruction is defined by any of the following:

- B, >= 0
- Zba, >= 0

B.120.1. Encoding



B.120.2. Synopsis

This instruction shifts **rs1** to the left by 3 places and adds it to **rs2**.

B.120.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.120.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.120.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 3);

```

B.120.6. Exceptions

DRAFT

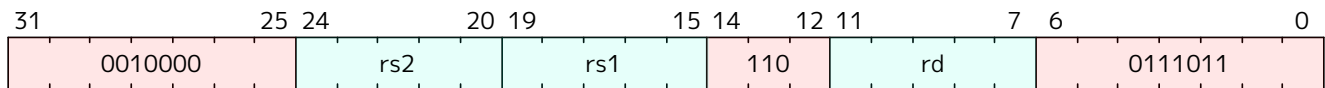
B.121. sh3add.uw

Shift unsigned word left by 3 and add

This instruction is defined by any of the following:

- B, ≥ 0
- Zba, ≥ 0

B.121.1. Encoding



B.121.2. Synopsis

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 places.

B.121.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.121.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.121.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 3);

```

B.121.6. Exceptions

DRAFT

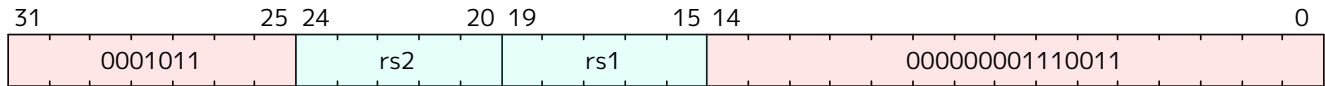
B.122. sinval.vma

Invalidate cached address translations

This instruction is defined by:

- Svinval, ≥ 0

B.122.1. Encoding



B.122.2. Synopsis

The [sinval.vma](#) instruction invalidates any address-translation cache entries that an [sfence.vma](#) instruction with the same values of rs1 and rs2 would invalidate. However, unlike [sfence.vma](#), [sinval.vma](#) instructions are only ordered with respect to [sfence.vma](#), [sfence.w.inval](#), and [sfence.inval.ir](#) instructions as defined below.

B.122.3. Access

M	HS	U	VS	VU
Always	Sometimes	Never	Sometimes	Never

B.122.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.122.5. Execution

```
XReg vaddr = X[rs1];
Bits<ASID_WIDTH> asid = X[rs2][ASID_WIDTH - 1:0];
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) || (mode() ==
PrivilegeMode::VS) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if (CSR[misa].H == 1 && CSR[hstatus].VTVM == 1 && mode() ==
PrivilegeMode::VS) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
```

```

}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, $encoding);
}
VmaOrderType vma_type;
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VS) {
    vma_type.vsmode = true;
    vma_type.single_vmid = true;
    vma_type.vmid = CSR[hgatp].VMID;
} else {
    vma_type.smode = true;
}
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        invalidate_translations(vma_type);
    }
} else {
    if (canonical_vaddr?(vaddr)) {
        vma_type.single_asid = true;
        vma_type.asid = asid;
        vma_type.single_vaddr = true;
        vma_type.vaddr = vaddr;
        invalidate_translations(vma_type);
    }
}
}

```

B.122.6. Exceptions

DRAFT

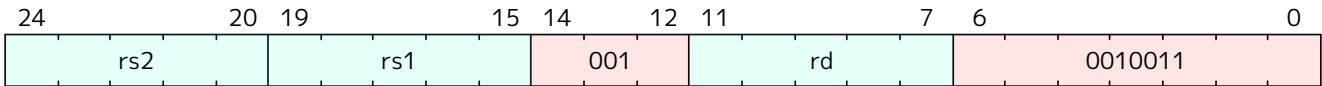
B.123. sll

Shift left logical

This instruction is defined by:

- I, >= 0

B.123.1. Encoding



B.123.2. Synopsis

Shift the value in **rs1** left by the value in the lower 6 bits of **rs2**, and store the result in **rd**.

B.123.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.123.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.123.5. Execution

```
if (xlen() == 64) {
    X[rd] = X[rs1] << X[rs2][5:0];
} else {
    X[rd] = X[rs1] << X[rs2][4:0];
}
```

B.123.6. Exceptions

DRAFT

B.124. slli

Shift left logical immediate

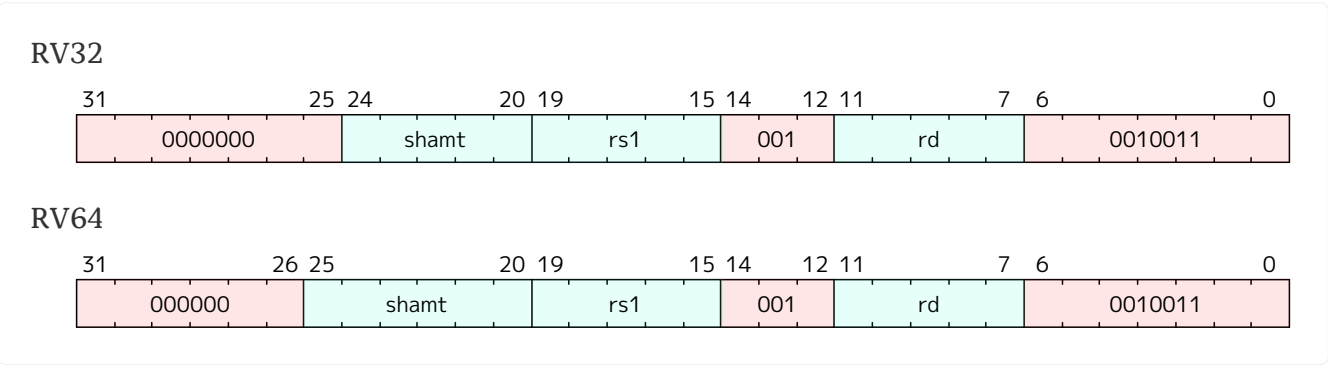
This instruction is defined by:

- $I, \geq 0$

B.124.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.124.2. Synopsis

Shift the value in rs1 left by shamt, and store the result in rd

B.124.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.124.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.124.5. Execution

```
X[rd] = X[rs1] << shamt;
```

DRAFT

B.124.6. Exceptions

DRAFT

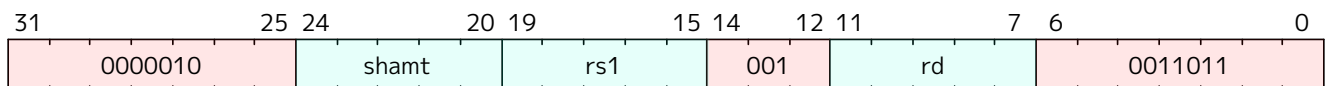
B.125. slli.uw

Shift left unsigned word (Immediate)

This instruction is defined by any of the following:

- B, >= 0
- Zba, >= 0

B.125.1. Encoding



B.125.2. Synopsis

This instruction takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.



*This instruction is the same as [slli](#) with **zext.w** performed on rs1 before shifting.*

B.125.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.125.4. Decode Variables

```

Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.125.5. Execution

```

if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = X[rs1][31:0] << shamt;

```

B.125.6. Exceptions

DRAFT

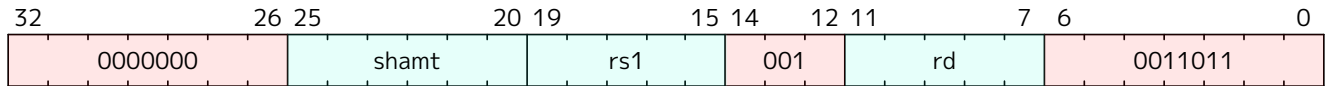
B.126. slliw

Shift left logical immediate word

This instruction is defined by:

- $I, \geq 0$

B.126.1. Encoding



B.126.2. Synopsis

Shift the 32-bit value in rs1 left by shamt, and store the sign-extended result in rd

B.126.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.126.4. Decode Variables

```

Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.126.5. Execution

```

X[rd] = sext(X[rs1] << shamt, 31);

```


B.126.6. Exceptions

DRAFT

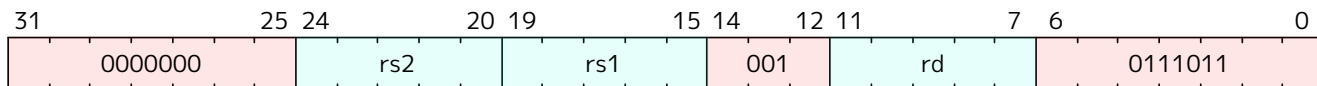
B.127. sllw

Shift left logical word

This instruction is defined by:

- $I, \geq 0$

B.127.1. Encoding



B.127.2. Synopsis

Shift the 32-bit value in **rs1** left by the value in the lower 5 bits of **rs2**, and store the sign-extended result in **rd**.

B.127.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.127.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.127.5. Execution

```

X[rd] = sext(X[rs1] << X[rs2][4:0], 31);

```

B.127.6. Exceptions

DRAFT

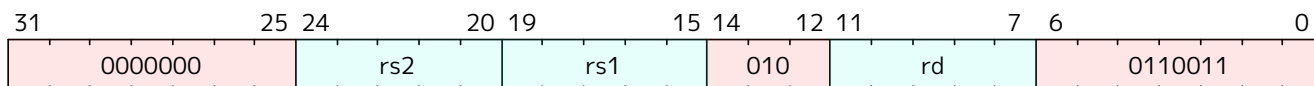
B.128. slt

Set on less than

This instruction is defined by:

- $I, \geq 0$

B.128.1. Encoding



B.128.2. Synopsis

Places the value 1 in register **rd** if register **rs1** is less than the value in register **rs2**, where both sources are treated as signed numbers, else 0 is written to **rd**.

B.128.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.128.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.128.5. Execution

```

XReg src1 = X[rs1];
XReg src2 = X[rs2];
X[rd] = ($signed(src1) < $signed(src2)) ? '1' : '0';

```

B.128.6. Exceptions

DRAFT

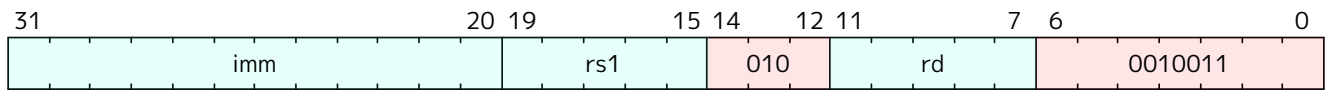
B.129. slti

Set on less than immediate

This instruction is defined by:

- I, >= 0

B.129.1. Encoding



B.129.2. Synopsis

Places the value 1 in register **rd** if register **rs1** is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to **rd**.

B.129.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.129.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.129.5. Execution

```
X[rd] = ($signed(X[rs1]) < $signed(imm)) ? '1' : '0';
```

B.129.6. Exceptions

DRAFT

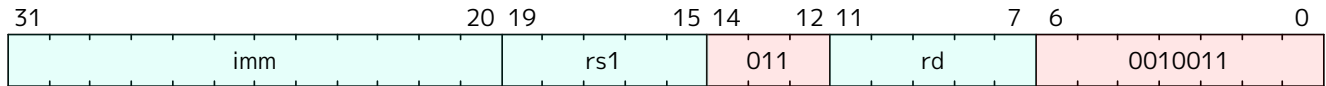
B.130. sltiu

Set on less than immediate unsigned

This instruction is defined by:

- $I, \geq 0$

B.130.1. Encoding



B.130.2. Synopsis

Places the value 1 in register **rd** if register **rs1** is less than the sign-extended immediate when both are treated as unsigned numbers (*i.e.*, the immediate is first sign-extended to XLEN bits then treated as an unsigned number), else 0 is written to **rd**.



sltiu rd, rs1, 1 sets **rd** to 1 if **rs1** equals zero, otherwise sets **rd** to 0 (assembler pseudoinstruction **SEQZ rd, rs**).

B.130.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.130.4. Decode Variables

```

Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.130.5. Execution

```

X[rd] = (X[rs1] < imm) ? 1 : 0;

```


B.130.6. Exceptions

DRAFT

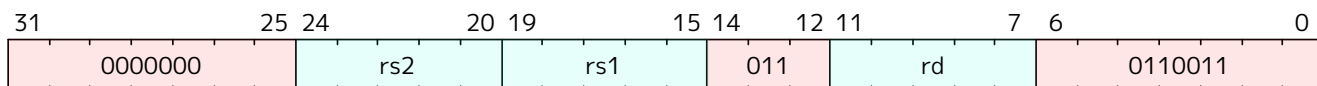
B.131. sltu

Set on less than unsigned

This instruction is defined by:

- $I, \geq 0$

B.131.1. Encoding



B.131.2. Synopsis

Places the value 1 in register **rd** if register **rs1** is less than the value in register **rs2**, where both sources are treated as unsigned numbers, else 0 is written to **rd**.

B.131.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.131.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.131.5. Execution

```

X[rd] = (X[rs1] < X[rs2]) ? 1 : 0;

```

B.131.6. Exceptions

DRAFT

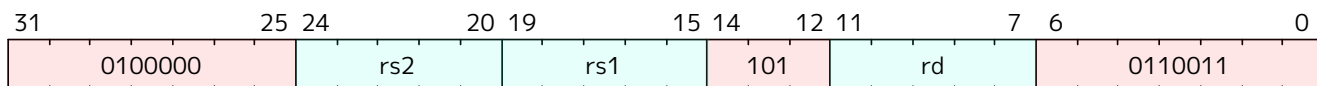
B.132. sra

Shift right arithmetic

This instruction is defined by:

- $I, \geq 0$

B.132.1. Encoding



B.132.2. Synopsis

Arithmetic shift the value in **rs1** right by the value in the lower 5 bits of **rs2**, and store the result in **rd**.

B.132.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.132.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.132.5. Execution

```
if (xlen() == 64) {
    X[rd] = X[rs1] >>> X[rs2][5:0];
} else {
    X[rd] = X[rs1] >>> X[rs2][4:0];
}
```

B.132.6. Exceptions

DRAFT

B.133. srai

Shift right arithmetic immediate

This instruction is defined by:

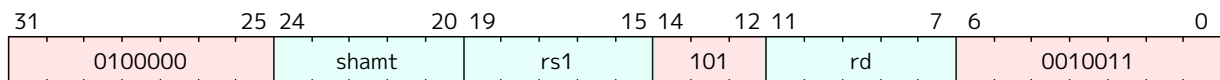
- $I, \geq 0$

B.133.1. Encoding

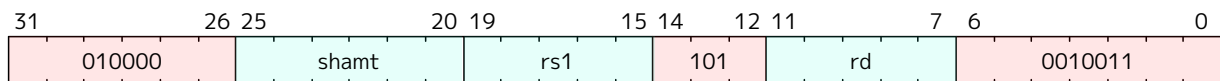


This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.133.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the value in rs1 right by shamt, and store the result in rd.

B.133.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.133.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.133.5. Execution

```
X[rd] = X[rs1] >>> shamt;
```

DRAFT

B.133.6. Exceptions

DRAFT

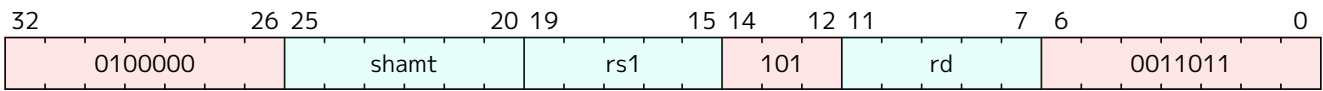
B.134. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

- I, >= 0

B.134.1. Encoding



B.134.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the 32-bit value in rs1 right by shamt, and store the sign-extended result in rd.

B.134.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.134.4. Decode Variables

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.134.5. Execution

```
XReg operand = sext(X[rs1], 31);
X[rd] = sext(operand >>> shamt, 31);
```

B.134.6. Exceptions

DRAFT

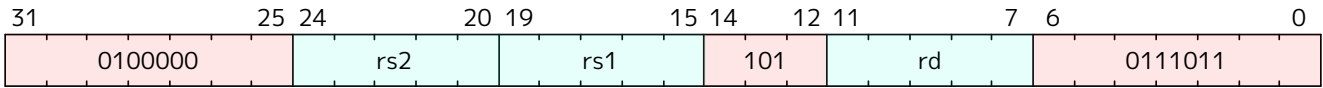
B.135. sraw

Shift right arithmetic word

This instruction is defined by:

- I, >= 0

B.135.1. Encoding



B.135.2. Synopsis

Arithmetic shift the 32-bit value in **rs1** right by the value in the lower 5 bits of **rs2**, and store the sign-extended result in **rd**.

B.135.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.135.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.135.5. Execution

```
XReg operand1 = sext(X[rs1], 31);
X[rd] = sext(operand1 >>> X[rs2][4:0], 31);
```

B.135.6. Exceptions

DRAFT

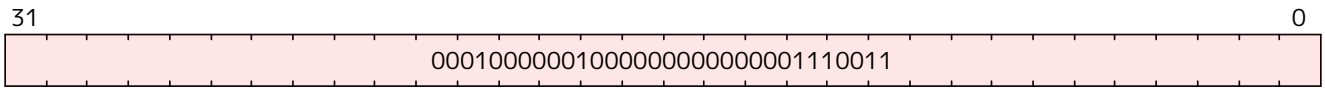
B.136. sret

Supervisor Exception Return

This instruction is defined by:

- S, >= 0

B.136.1. Encoding



B.136.2. Synopsis

Returns from an exception.

When [sret](#) is allowed to execute, its behavior depends on whether or not the current privilege mode is virtualized.

When the current privilege mode is (H)S-mode or M-mode

[sret](#) sets [hstatus](#) = 0, [mstatus.SPP](#) = 0, [mstatus.SIE](#) = [mstatus.SPIE](#), and [mstatus.SPIE](#) = 1, changes the privilege mode according to the table below, and then jumps to the address in [sepc](#).

Table 51. Next privilege mode following an [sret](#) in (H)S-mode or M-mode

mstatus.SPP	hstatus.SPV	Mode after sret
0	0	U-mode
0	1	VU-mode
1	0	(H)S-mode
1	1	VS-mode

When the current privilege mode is VS-mode

[sret](#) sets [vsstatus.SPP](#) = 0, [vsstatus.SIE](#) = **[vsstatus.SPIE](#)**, and [vsstatus.SPIE](#) = 1, changes the privilege mode according to the table below, and then jumps to the address in [vsepc](#).

Table 52. Next privilege mode following an [sret](#) in (H)S-mode or M-mode

vsstatus.SPP	Mode after sret
0	VU-mode
1	VS-mode

B.136.3. Access

M	HS	U	VS	VU
Always	Sometimes	Never	Sometimes	Never

Access is determined as follows:

mstatus.TSR	hstatus.VTSR	Behavior when executed from:				
		M-mode	U-mode	(H)S-mode	VU-mode	VS-mode
0	0	executes	Illegal Instruction	executes	Virtual Instruction	executes
0	1	executes	Illegal Instruction	executes	Virtual Instruction	Virtual Instruction
1	0	executes	Illegal Instruction	Illegal Instruction	Virtual Instruction	executes
1	1	executes	Illegal Instruction	Illegal Instruction	Virtual Instruction	Virtual Instruction

B.136.4. Decode Variables

B.136.5. Execution

```

if (implemented?(ExtensionName::H)) {
  if (CSR[mstatus].TSR == 1'b0 && CSR[hstatus].VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, $encoding);
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b0 && CSR[hstatus].VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode
::VS) {
      raise(ExceptionCode::VirtualInstruction, $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b1 && CSR[hstatus].VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
      raise(ExceptionCode::IllegalInstruction, $encoding);
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b1 && CSR[hstatus].VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
      raise(ExceptionCode::IllegalInstruction, $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode
::VS) {
      raise(ExceptionCode::VirtualInstruction, $encoding);
    }
  }
}

```

```

    }
  }
} else {
  if (mode() != PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
  }
}
if (!virtual_mode?()) {
  if (implemented?(ExtensionName::H)) {
    if (CSR[hstatus].SPV == 1'b1) {
      if (CSR[mstatus].SPP == 2'b01) {
        set_mode(PrivilegeMode::VS);
      } else if (CSR[mstatus].SPP == 2'b00) {
        set_mode(PrivilegeMode::VU);
      }
    }
    CSR[hstatus].SPV = 0;
  }
  CSR[mstatus].SIE = CSR[mstatus].SPIE;
  CSR[mstatus].SPIE = 1;
  CSR[mstatus].SPP = 2'b00;
  PC = $bits(CSR[sepc]);
} else {
  if (CSR[mstatus].TSR == 1'b1) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
  }
  CSR[vsstatus].SPP = 0;
  CSR[vsstatus].SIE = CSR[vsstatus].SPIE;
  CSR[vsstatus].SPIE = 1;
  PC = $bits(CSR[vsepc]);
}
}

```

B.136.6. Exceptions

DRAFT

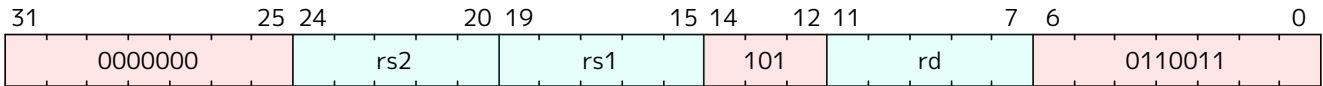
B.137. srl

Shift right logical

This instruction is defined by:

- I, >= 0

B.137.1. Encoding



B.137.2. Synopsis

Logical shift the value in **rs1** right by the value in the lower bits of **rs2**, and store the result in **rd**.

B.137.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.137.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.137.5. Execution

```
if (xlen() == 64) {
    X[rd] = X[rs1] >> X[rs2][5:0];
} else {
    X[rd] = X[rs1] >> X[rs2][4:0];
}
```

B.137.6. Exceptions

DRAFT

B.138. srli

Shift right logical immediate

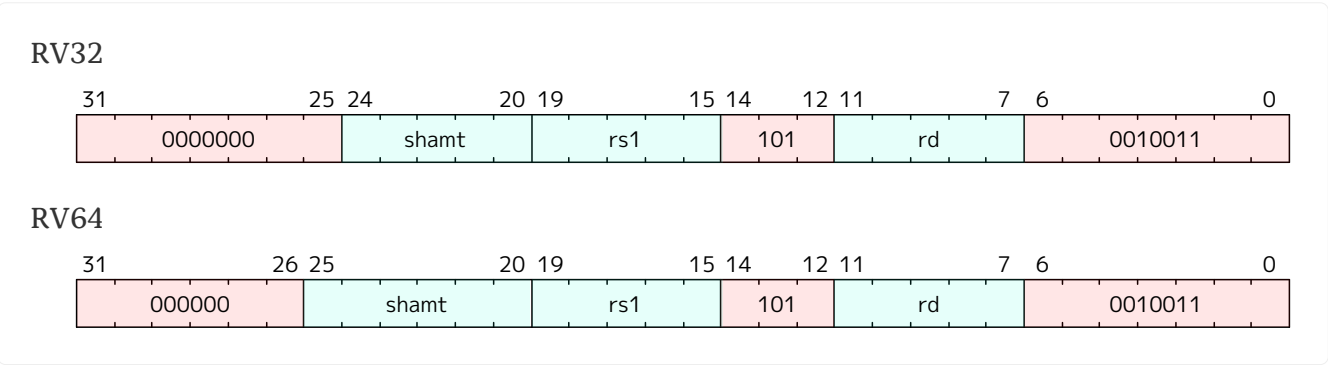
This instruction is defined by:

- $I, \geq 0$

B.138.1. Encoding



This instruction has different encodings in RV32 and RV64.



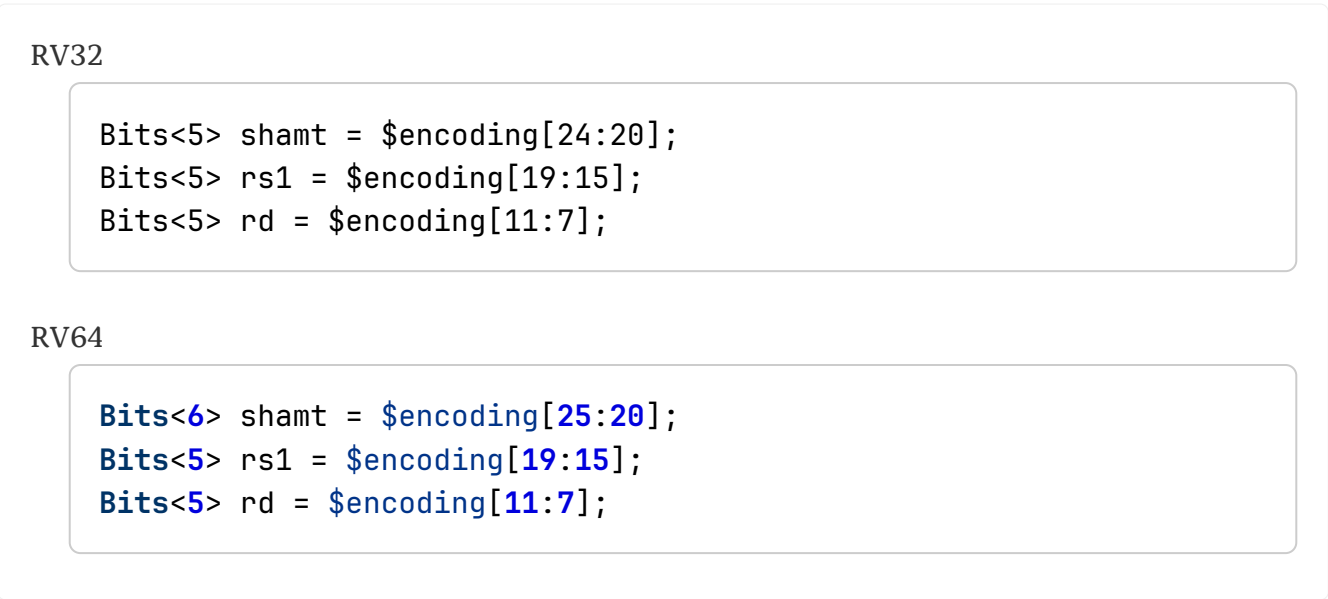
B.138.2. Synopsis

Shift the value in rs1 right by shamt, and store the result in rd

B.138.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.138.4. Decode Variables



B.138.5. Execution

```
X[rd] = X[rs1] >> shamt;
```

DRAFT

B.138.6. Exceptions

DRAFT

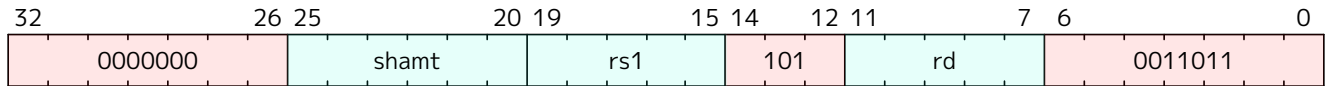
B.139. srlw

Shift right logical immediate word

This instruction is defined by:

- $I, \geq 0$

B.139.1. Encoding



B.139.2. Synopsis

Shift the 32-bit value in rs1 right by shamt, and store the sign-extended result in rd

B.139.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.139.4. Decode Variables

```

Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.139.5. Execution

```

XReg operand = X[rs1][31:0];
X[rd] = sext(operand >> shamt, 31);

```

B.139.6. Exceptions

DRAFT

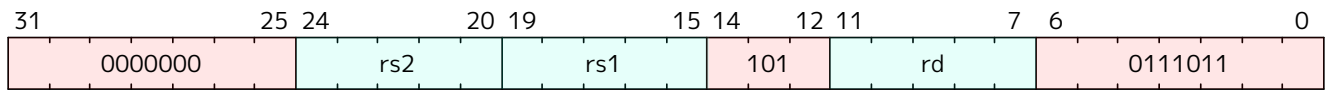
B.140. srlw

Shift right logical word

This instruction is defined by:

- I, >= 0

B.140.1. Encoding



B.140.2. Synopsis

Logical shift the 32-bit value in **rs1** right by the value in the lower 5 bits of **rs2**, and store the sign-extended result in **rd**.

B.140.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.140.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.140.5. Execution

```
X[rd] = sext(X[rs1][31:0] >> X[rs2][4:0], 31);
```


B.140.6. Exceptions

DRAFT

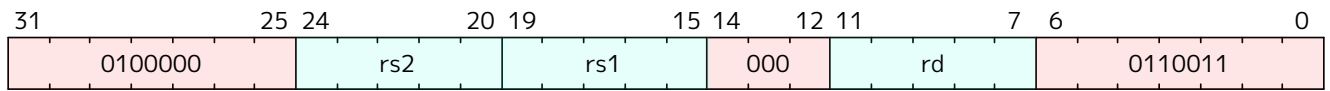
B.141. sub

Subtract

This instruction is defined by:

- I, >= 0

B.141.1. Encoding



B.141.2. Synopsis

Subtract the value in rs2 from rs1, and store the result in rd

B.141.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.141.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.141.5. Execution

```
XReg t0 = X[rs1];
XReg t1 = X[rs2];
X[rd] = t0 - t1;
```

B.141.6. Exceptions

DRAFT

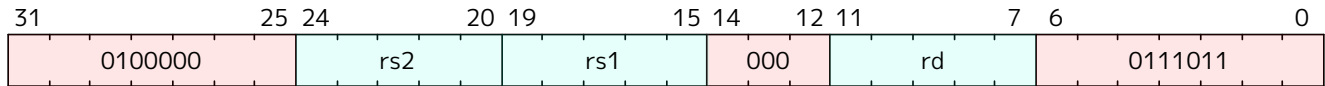
B.142. subw

Subtract word

This instruction is defined by:

- I, ≥ 0

B.142.1. Encoding



B.142.2. Synopsis

Subtract the 32-bit values in rs2 from rs1, and store the sign-extended result in rd

B.142.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.142.4. Decode Variables

```

Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];

```

B.142.5. Execution

```

Bits<32> t0 = X[rs1][31:0];
Bits<32> t1 = X[rs2][31:0];
X[rd] = sext(t0 - t1, 31);

```

B.142.6. Exceptions

DRAFT

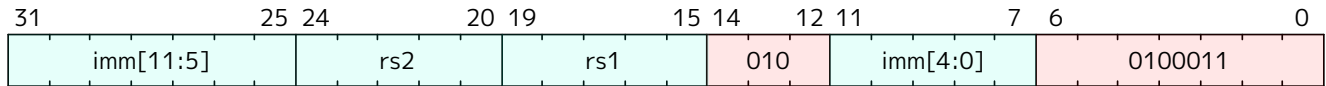
B.143. sw

Store word

This instruction is defined by:

- $I, \geq 0$

B.143.1. Encoding



B.143.2. Synopsis

Store 32 bits of data from register **rs2** to an address formed by adding **rs1** to a signed offset.

B.143.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.143.4. Decode Variables

```

Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];

```

B.143.5. Execution

```

XReg virtual_address = X[rs1] + imm;
write_memory<32>(virtual_address, X[rs2][31:0]);

```

B.143.6. Exceptions

DRAFT

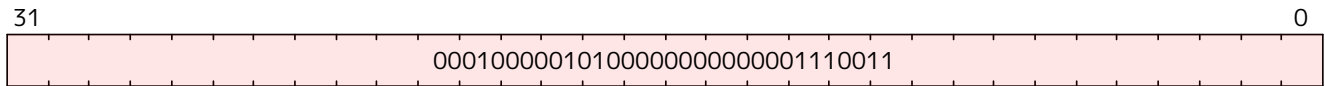
B.144. wfi

Wait for interrupt

This instruction is defined by:

- I, >= 0

B.144.1. Encoding



B.144.2. Synopsis

Can causes the processor to enter a low-power state until the next interrupt occurs.

<%- if ext?(:H) -%> The behavior of **wfi** is affected by the mstatus.TW and hstatus.VTW bits, as summarized below.

mstatus.TW	hstatus.VTW	wfi behavior			
		HS-mode	U-mode	VS-mode	in VU-mode
0	0	Wait	Trap (I)	Wait	Trap (V)
0	1	Wait	Trap (I)	Trap (V)	Trap (V)
1	-	Trap (I)	Trap (I)	Trap (I)	Trap (I)
Trap (I) - Trap with Illegal Instruction code					
Trap (V) - Trap with Virtual Instruction code					

<%- else -%> The **wfi** instruction is also affected by mstatus.TW, as shown below:

mstatus.TW	wfi behavior			
	S-mode	U-mode	O	Wait

<%- end -%>

When **wfi** is marked as causing a trap above, the implementation is allowed to wait for an unspecified period of time to see if an interrupt occurs before raising the trap. That period of time can be zero (*i.e.*, **wfi** always causes a trap in the cases identified above).

B.144.3. Access

M	HS	U	VS	VU
Always	Sometimes	Sometimes	Sometimes	Sometimes

<%- if ext?(:H) -%> The behavior of **wfi** is affected by the mstatus.TW and hstatus.VTW bits, as summarized below.

mstatus.TW	hstatus.VTW	wfi behavior			
		HS-mode	U-mode	VS-mode	in VU-mode
0	0	Wait	Trap (I)	Wait	Trap (V)
0	1	Wait	Trap (I)	Trap (V)	Trap (V)
1	-	Trap (I)	Trap (I)	Trap (I)	Trap (I)
Trap (I) - Trap with Illegal Instruction code					
Trap (V) - Trap with Virtual Instruction code					

<%- else -%> The **wfi** instruction is also affected by mstatus.TW, as shown below:

mstatus.TW	wfi behavior			
	S-mode	U-mode	O	Wait

<%- end -%>

B.144.4. Decode Variables

B.144.5. Execution

```
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, $encoding);
}
if ((CSR[misa].S == 1) && (CSR[mstatus].TW == 1'b1)) {
    if (mode() != PrivilegeMode::M) {
        raise(ExceptionCode::IllegalInstruction, $encoding);
    }
}
if (CSR[misa].H == 1) {
    if (CSR[hstatus].VTW == 1'b0) {
        if (mode() == PrivilegeMode::VU) {
            raise(ExceptionCode::VirtualInstruction, $encoding);
        }
    } else if (CSR[hstatus].VTW == 1'b1) {
        if ((mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU))
        {
            raise(ExceptionCode::VirtualInstruction, $encoding);
        }
    }
}
wfi();
```

B.144.6. Exceptions

DRAFT

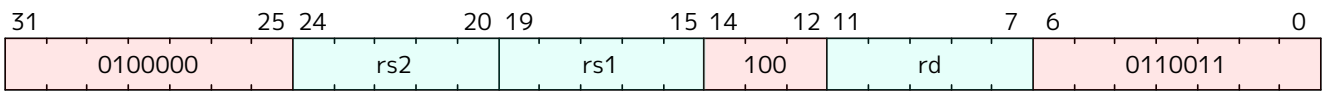
B.145. xnor

Exclusive NOR

This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0
- Zbkb, >= 0

B.145.1. Encoding



B.145.2. Synopsis

This instruction performs the bit-wise exclusive-NOR operation on rs1 and rs2.

B.145.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.145.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.145.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, $encoding);
}
X[rd] = ~(X[rs1] ^ X[rs2]);
```

B.145.6. Exceptions

DRAFT

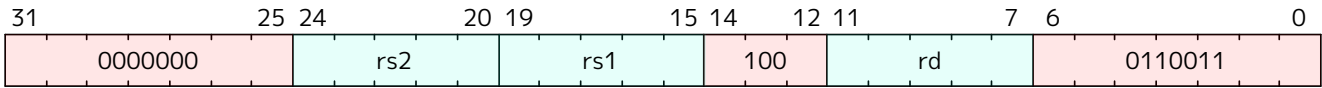
B.146. xor

Exclusive Or

This instruction is defined by:

- I, >= 0

B.146.1. Encoding



B.146.2. Synopsis

Exclusive or rs1 with rs2, and store the result in rd

B.146.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.146.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.146.5. Execution

```
X[rd] = X[rs1] ^ X[rs2];
```

B.146.6. Exceptions

DRAFT

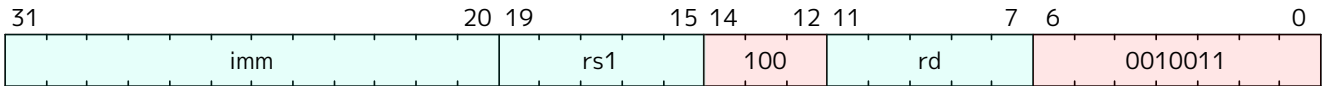
B.147. xori

Exclusive Or immediate

This instruction is defined by:

- I, >= 0

B.147.1. Encoding



B.147.2. Synopsis

Exclusive or an immediate to the value in rs1, and store the result in rd

B.147.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.147.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.147.5. Execution

```
X[rd] = X[rs1] ^ imm;
```

B.147.6. Exceptions

DRAFT

B.148. zext.h

Zero-extend halfword

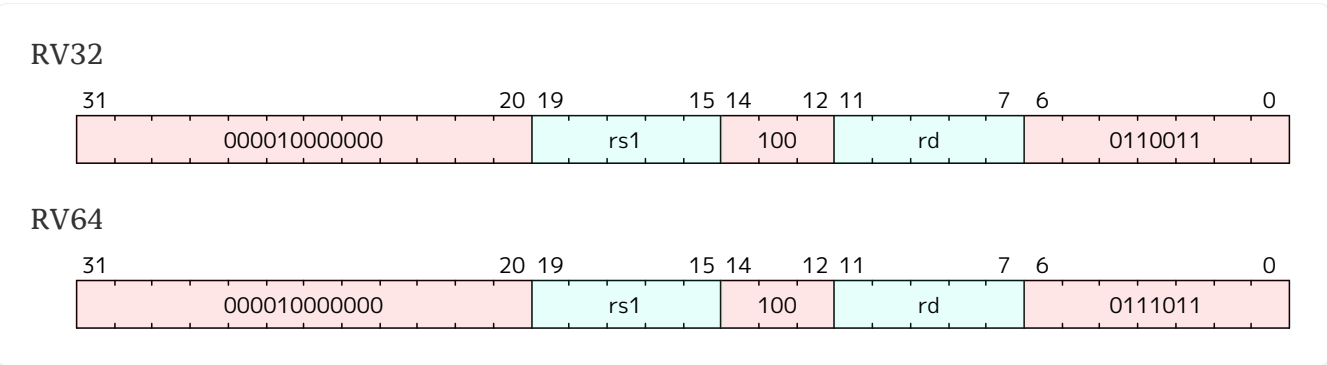
This instruction is defined by any of the following:

- B, >= 0
- Zbb, >= 0

B.148.1. Encoding



This instruction has different encodings in RV32 and RV64.



B.148.2. Synopsis

This instruction zero-extends the least-significant halfword of the source to XLEN by inserting 0's into all of the bits more significant than 15.



*The **zext.h** instruction is a pseudo-op for **pack** when **Zbkb** is implemented and $XLEN == 32$.*



*The **zext.h** instruction is a pseudo-op for **packw** when **Zbkb** is implemented and $XLEN == 64$.*

B.148.3. Access

M	HS	U	VS	VU
Always	Always	Always	Always	Always

B.148.4. Decode Variables

RV32

```
Bits<5> rs1 = $encoding[19:15];  
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<5> rs1 = $encoding[19:15];  
Bits<5> rd = $encoding[11:7];
```

B.148.5. Execution

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {  
    raise(ExceptionCode::IllegalInstruction, $encoding);  
}  
X[rd] = X[rs1][15:0];
```

B.148.6. Exceptions

DRAFT

Appendix C: CSR Specifications

DRAFT

C.1. cycle

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mcycle](#).

Privilege mode access is controlled with mcounteren.CY, scounteren.CY, and hcounteren.CY as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.1.1. Attributes

CSR Address	0xc00
Defining extension	Zicntr >= 0
Length	64-bit
Privilege Mode	U

C.1.2. Format

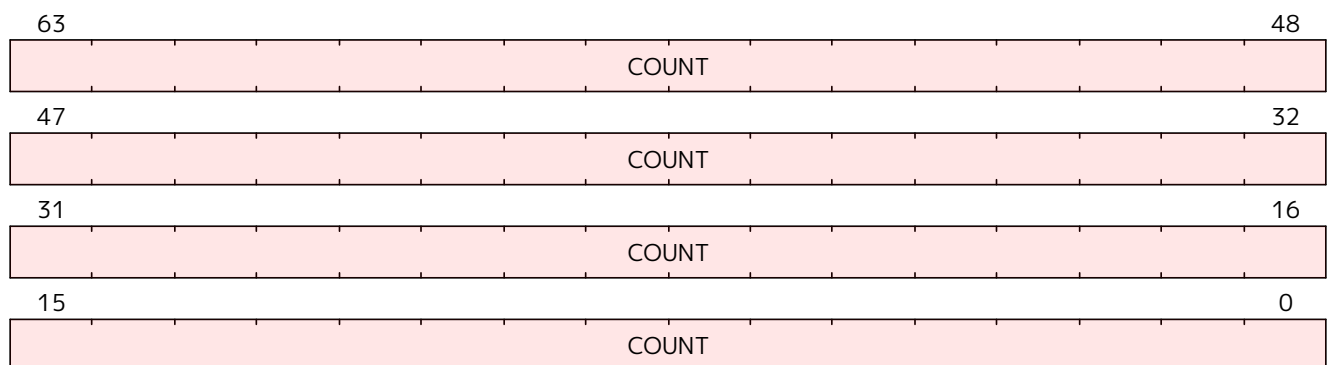


Figure 1. cycle format

C.2. cycleh

High-half cycle counter for RDCYCLE Instruction

 *cycleh* is only defined in RV32.

Alias for M-mode CSR *mcycleh*.

Privilege mode access is controlled with *mcOUNTERen.CY*, *scOUNTERen.CY*, and *hcOUNTERen.CY* as follows:

mcOUNTERen.CY	scOUNTERen.CY	hcOUNTERen.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.2.1. Attributes

CSR Address	0xc80
Defining extension	Zicntr >= 0
Length	32-bit
Privilege Mode	U

C.2.2. Format

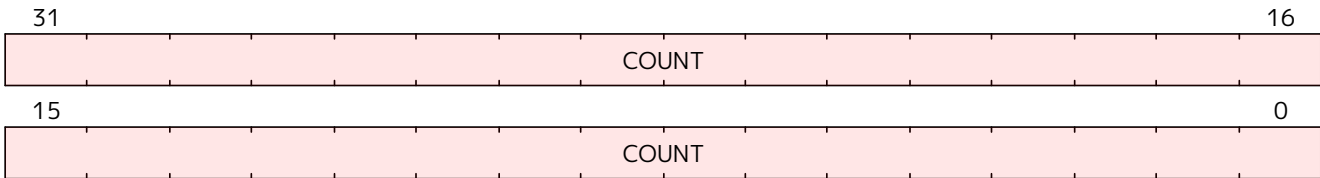


Figure 2. cycleh format

C.3. fcsr

Floating-point control and status register (**f_{rm}** + **f_{flags}**)

The floating-point control and status register, **fcsr**, is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in [Floating-Point Control and Status Register](#).

Floating-point control and status register

Unresolved directive in rva.adoc - include::images/wavedrom/float-csr.adoc[]

The **fcsr** register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads **fcsr** by copying it into integer register *rd*. FSCSR swaps the value in **fcsr** by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into **fcsr**.

The fields within the **fcsr** can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field **f_{rm}** (**fcsr** bits 7–5) and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in **f_{rm}** by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into **f_{rm}**. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field **f_{flags}** (**fcsr** bits 4–0).

Bits 31–8 of the **fcsr** are reserved for other standard extensions. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in **f_{rm}**. Rounding modes are encoded as shown in [Table 10](#). A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in **f_{rm}**. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111). Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.



The ratified version of the F spec mandated that an illegal-instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal-instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 11. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 53. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact



As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

C.3.1. Attributes

CSR Address	0x3
Defining extension	F >= 0
Length	32-bit
Privilege Mode	U

C.3.2. Format



Figure 3. fcsr format

C.4. hcounteren

Hypervisor Counter Enable

Together with [scounteren](#), delegates control of the hardware performance-monitoring counters to VS/VU-mode

See [cycle](#) for a table of how exceptions occur across all modes.

C.4.1. Attributes

CSR Address	0x606
Defining extension	H >= 0
Length	32-bit
Privilege Mode	S

C.4.2. Format

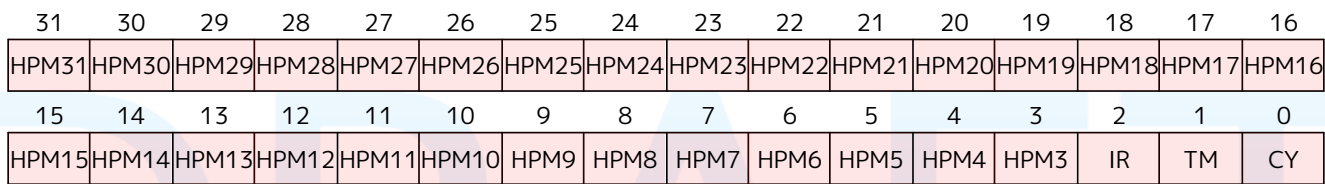


Figure 4. hcounteren format

C.5. hedeleg

Hypervisor Exception Delegation

Controls exception delegation from HS-mode to VS-mode.

By default, all traps at any privilege level are handled in M-mode, though M-mode usually uses the [medeleg](#) and [mideleg](#) CSRs to delegate some traps to HS-mode. The [hedeleg](#) and **hideleg** CSRs allow these traps to be further delegated to a VS-mode guest; their layout is the same as [medeleg](#) and [mideleg](#).

A synchronous trap that has been delegated to HS-mode (using [medeleg](#)) is further delegated to VS-mode if V=1 before the trap and the corresponding [hedeleg](#) bit is set. Each bit of [hedeleg](#) shall be either writable or read-only zero. Many bits of [hedeleg](#) are required specifically to be writable or zero. Bit 0, corresponding to instruction address misaligned exceptions, must be writable if IALIGN=32.



Requiring that certain bits of [hedeleg](#) be writable reduces some of the burden on a hypervisor to handle variations of implementation.

When XLEN=32, [hedelegh](#) is a 32-bit read/write register that aliases bits 63:32 of [hedeleg](#). Register [hedelegh](#) does not exist when XLEN=64.

C.5.1. Attributes

CSR Address	0x602
Defining extension	H >= 0
Length	64-bit
Privilege Mode	S

C.5.2. Format

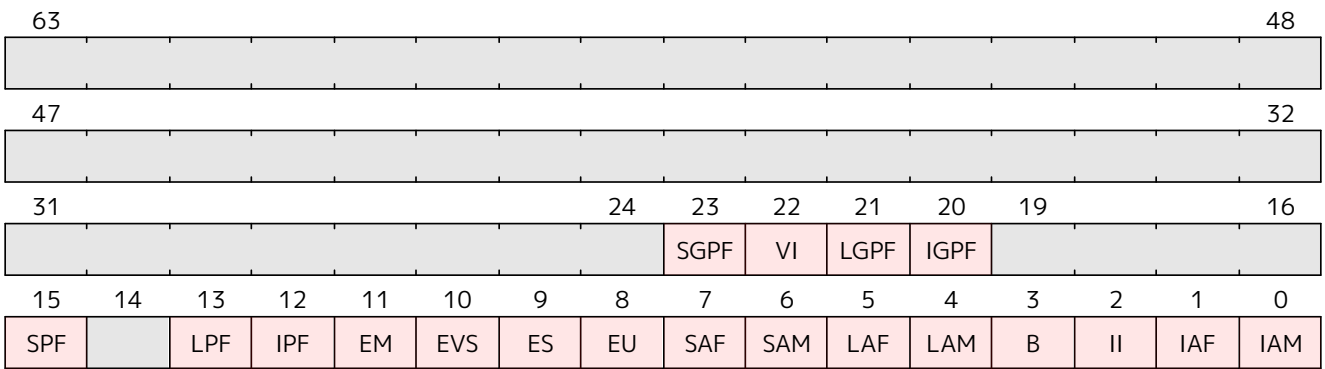


Figure 5. hedeleg format

C.6. hedelegH

Hypervisor Exception Delegation High



hedelegH is only defined in RV32.

Controls exception delegation from HS-mode to VS-mode.

Alias of upper bits of *hedeleg*[63:32].

C.6.1. Attributes

CSR Address	0x612
Defining extension	H >= 0
Length	32-bit
Privilege Mode	S

C.6.2. Format

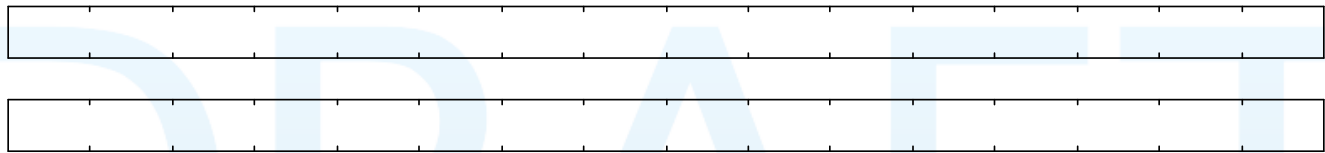


Figure 6. *hedelegH* format

C.7. hgatp

Hypervisor guest address translation and protection

The hgatp register is an HSXLEN-bit read/write register which controls G-stage address translation and protection, the second stage of two-stage translation for guest virtual addresses. Similar to CSR [satp](#), this register holds the physical page number (PPN) of the guest-physical root page table; a virtual machine identifier (VMID), which facilitates address-translation fences on a per-virtual-machine basis; and the MODE field, which selects the address-translation scheme for guest physical addresses. When mstatus.TVM=1, attempts to read or write [hgatp](#) while executing in HS-mode will raise an **IllegalInstruction** exception.

[Table 54](#) shows the encodings of the MODE field when HSXLEN=32 and HSXLEN=64. When MODE=Bare, guest physical addresses are equal to supervisor physical addresses, and there is no further memory protection for a guest virtual machine beyond the physical memory protection scheme described in [\[pmp\]](#). In this case, the remaining fields in [hgatp](#) must be set to zeros.

Table 54. Encoding of [hgatp](#) MODE field.

HSXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32x4	Page-based 34-bit virtual addressing (2-bit extension of Sv32).
HSXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	—	Reserved
8	Sv39x4	Page-based 41-bit virtual addressing (2-bit extension of Sv39).
9	Sv48x4	Page-based 50-bit virtual addressing (2-bit extension of Sv48).
10	Sv57x4	Page-based 59-bit virtual addressing (2-bit extension of Sv57).
11-15	—	Reserved

Implementations are not required to support all defined MODE settings when HSXLEN=64.

A write to [hgatp](#) with an unsupported MODE value is not ignored as it is for [satp](#). Instead, the fields of [hgatp](#) are WARL in the normal way, when so indicated.

As explained in [\[guest-addr-translation\]](#), for the paged virtual-memory schemes (Sv32x4, Sv39x4, Sv48x4, and Sv57x4), the root page table is 16 KiB and must be aligned to a 16-KiB boundary. In these modes, the lowest two bits of the physical page number (PPN) in [hgatp](#) always read as zeros. An implementation that supports only the defined paged virtual-memory schemes and/or Bare may make PPN[1:0] read-only zero.

The number of VMID bits is UNSPECIFIED and may be zero. The number of implemented VMID bits, termed *VMIDLEN*, may be determined by writing one to every bit position in the VMID field, then reading back the value in [hgatp](#) to see which bit positions in the VMID field hold a one. The least-significant bits of VMID are implemented first: that is, if *VMIDLEN* > 0, VMID[VMIDLEN-1:0] is writable. The maximal value of VMIDLEN, termed *VMIDMAX*, is 7 for Sv32x4 or 14 for Sv39x4, Sv48x4, and Sv57x4.

The [hgatp](#) register is considered *active* for the purposes of the address-translation algorithm *unless* the effective privilege mode is U and [hstatus.HU](#)=0.



This definition simplifies the implementation of speculative execution of `hlv`, `hlvx`, and `hsv` instructions.

Note that writing `hgatp` does not imply any ordering constraints between page-table updates and subsequent G-stage address translations. If the new virtual machine’s guest physical page tables have been modified, or if a VMID is reused, it may be necessary to execute an `HFENCE.GVMA` instruction (see [\[hfence.vma\]](#)) before or after writing `hgatp`.

C.7.1. Attributes

CSR Address	0x680
Defining extension	H >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.7.2. Format

This CSR format changes dynamically with XLEN.

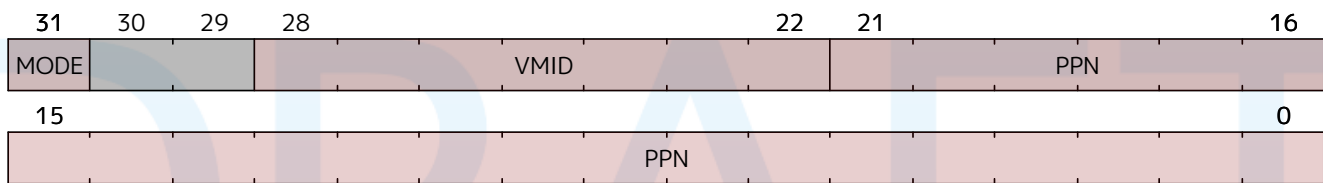


Figure 7. `hgatp` Format when `CSR[mstatus].SXL == 0`

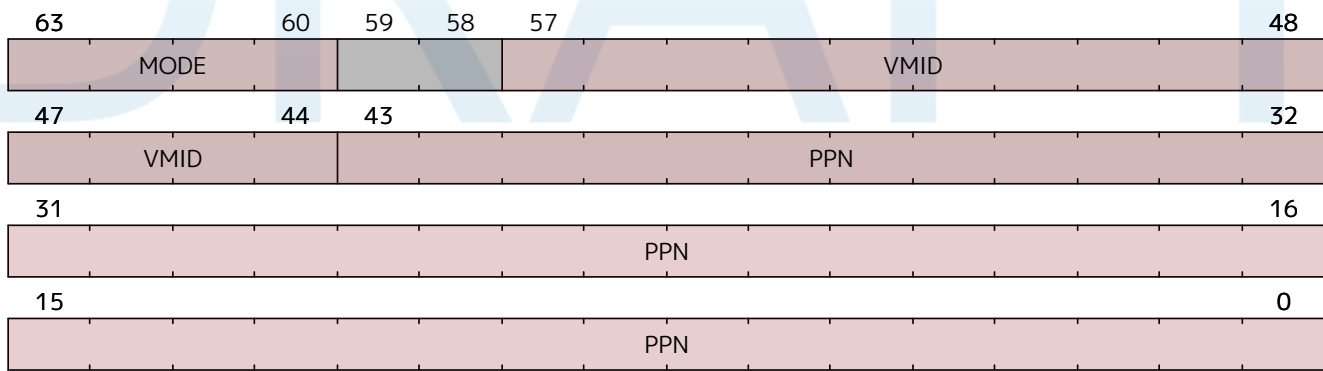


Figure 8. `hgatp` Format when `CSR[mstatus].SXL == 1`

C.8. hpmcounter10

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter10](#).

Privilege mode access is controlled with mcounteren.HPM10 <%- if ext?(:S) -%> , scounteren.HPM10 <%- if ext?(:H) -%> , and hcounteren.HPM10 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM10	scounteren.HPM10	hcounteren.HPM10	hpmcounter10 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM10	scounteren.HPM10	hpmcounter10 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM10	hpmcounter10 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.8.1. Attributes

CSR Address	0xc0a
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.8.2. Format

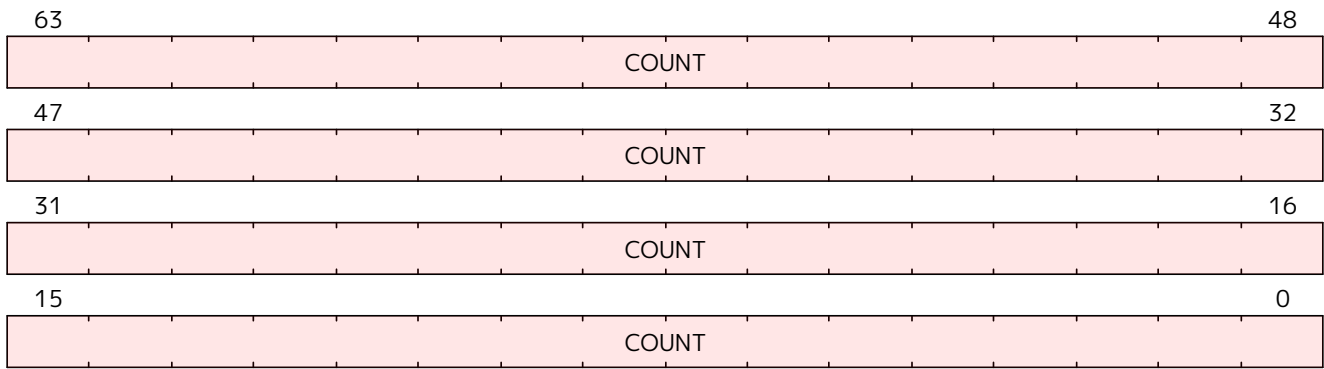


Figure 9. hpmcounter10 format

DRAFT

C.9. hpmcounter10h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter10h](#).

Privilege mode access is controlled with mcounteren.HPM10, scounteren.HPM10, and hcounteren.HPM10 as follows:

mcounteren. HPM10	scounteren. HPM10	hcounteren. HPM10	hpmcounter10h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.9.1. Attributes

CSR Address	0xc8a
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.9.2. Format

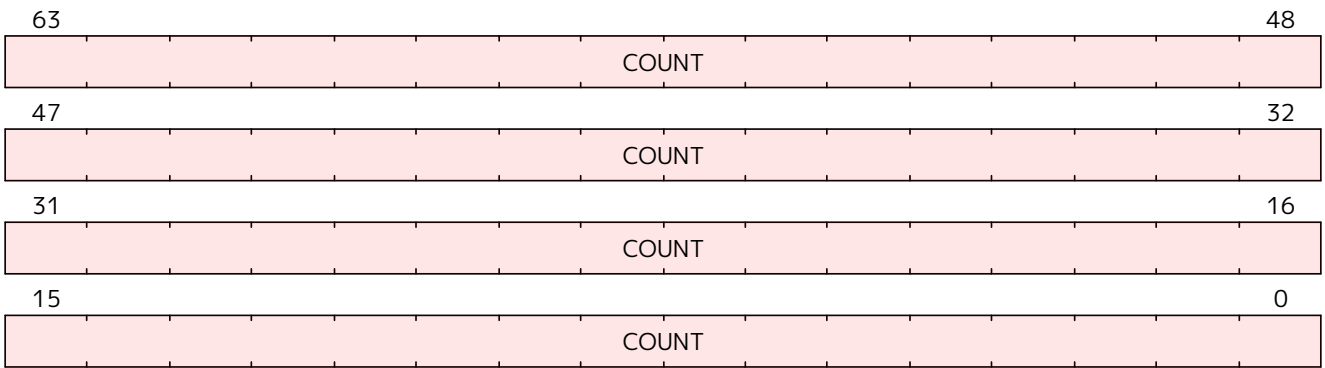


Figure 10. hpmcounter10h format

C.10. hpmcounter11

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter11](#).

Privilege mode access is controlled with mcounteren.HPM11 <%- if ext?(:S) -%> , scounteren.HPM11 <%- if ext?(:H) -%> , and hcounteren.HPM11 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM11	scounteren. HPM11	hcounteren. HPM11	hpmcounter11 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM11	scounteren.HPM11	hpmcounter11 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM11	hpmcounter11 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.10.1. Attributes

CSR Address	0xc0b
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.10.2. Format

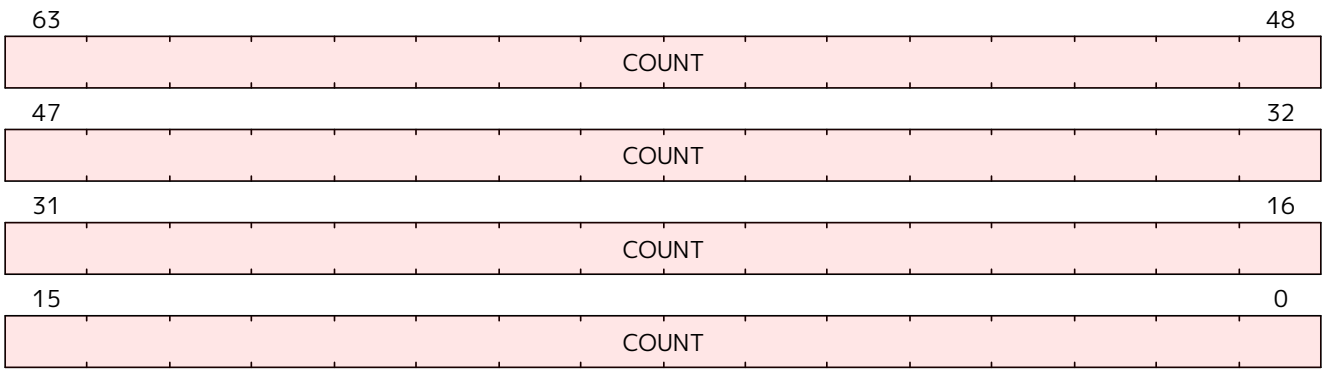


Figure 11. hpmcounter11 format

DRAFT

C.11. hpmcounter11h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter11h](#).

Privilege mode access is controlled with mcounteren.HPM11, scounteren.HPM11, and hcounteren.HPM11 as follows:

mcounteren. HPM11	scounteren. HPM11	hcounteren. HPM11	hpmcounter11h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.11.1. Attributes

CSR Address	0xc8b
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.11.2. Format

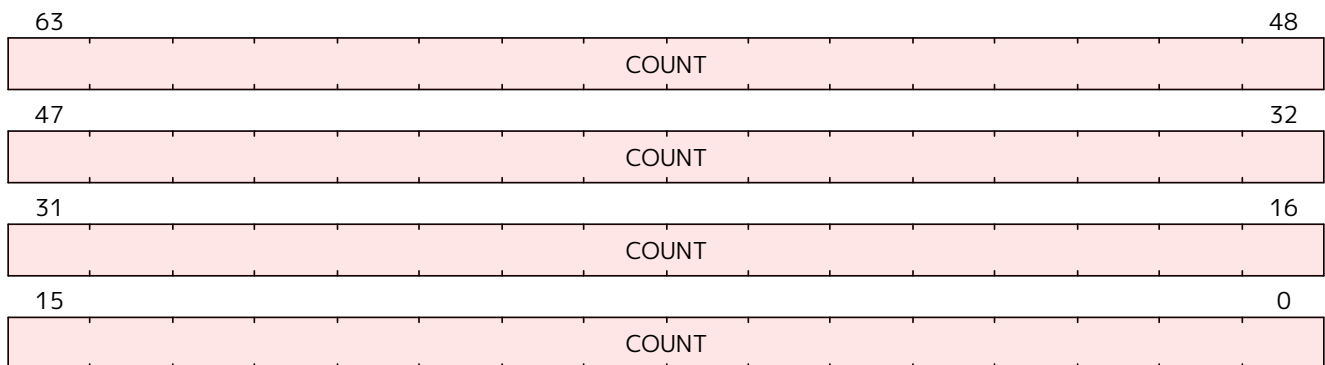


Figure 12. hpmcounter11h format

C.12. hpmcounter12

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter12](#).

Privilege mode access is controlled with mcounteren.HPM12 <%- if ext?(:S) -%> , scounteren.HPM12 <%- if ext?(:H) -%> , and hcounteren.HPM12 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM12	scounteren. HPM12	hcounteren. HPM12	hpmcounter12 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM12	scounteren.HPM12	hpmcounter12 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM12	hpmcounter12 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.12.1. Attributes

CSR Address	0xc0c
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.12.2. Format

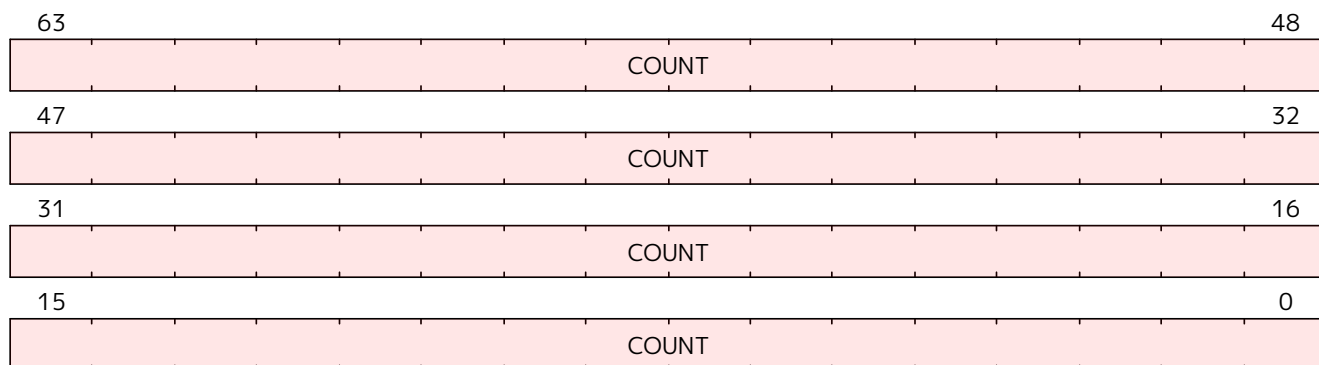


Figure 13. hpmcounter12 format

DRAFT

C.13. hpmcounter12h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter12h](#).

Privilege mode access is controlled with mcounteren.HPM12, scounteren.HPM12, and hcounteren.HPM12 as follows:

mcounteren. HPM12	scounteren. HPM12	hcounteren. HPM12	hpmcounter12h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.13.1. Attributes

CSR Address	0xc8c
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.13.2. Format

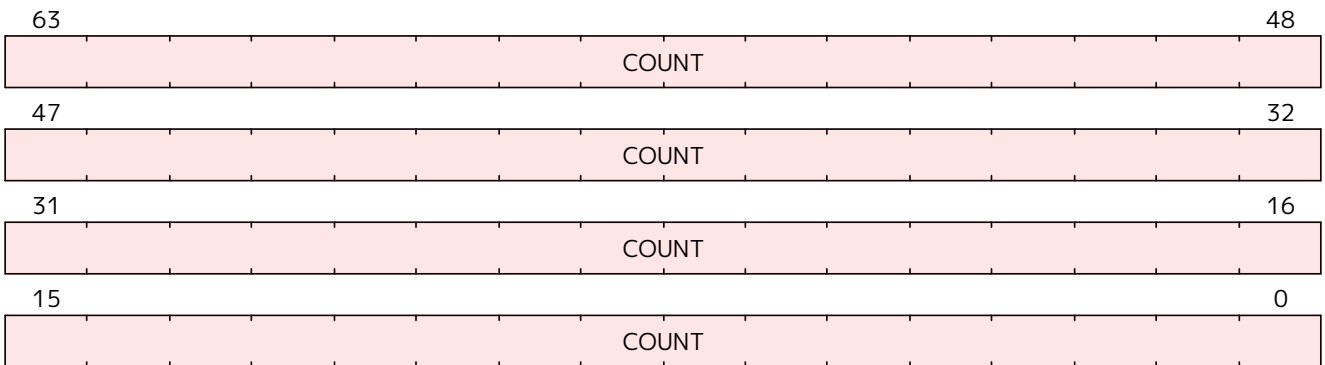


Figure 14. hpmcounter12h format

C.14. hpmcounter13

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter13](#).

Privilege mode access is controlled with mcounteren.HPM13 <%- if ext?(:S) -%> , scounteren.HPM13 <%- if ext?(:H) -%> , and hcounteren.HPM13 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM13	scounteren. HPM13	hcounteren. HPM13	hpmcounter13 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM13	scounteren.HPM13	hpmcounter13 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM13	hpmcounter13 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.14.1. Attributes

CSR Address	0xc0d
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.14.2. Format

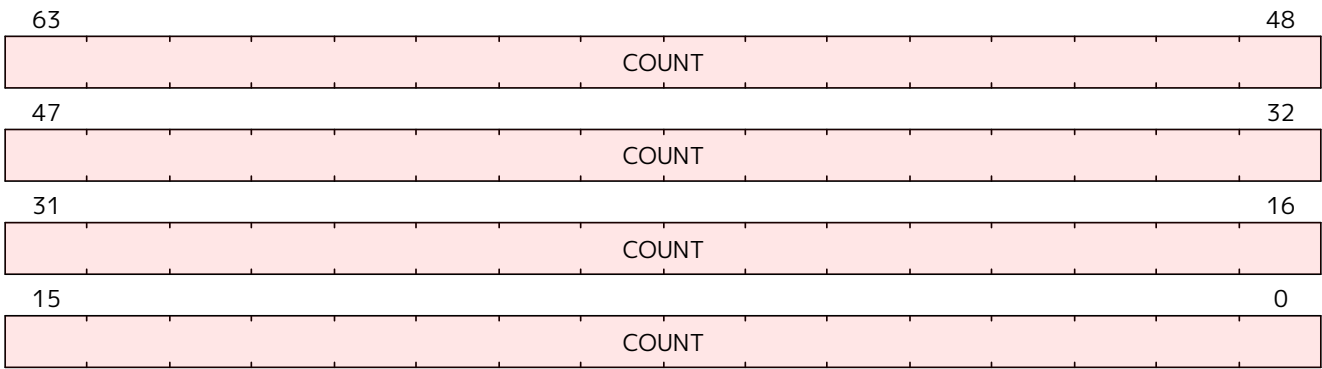


Figure 15. hpmcounter13 format

DRAFT

C.15. hpmcounter13h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter13h](#).

Privilege mode access is controlled with mcounteren.HPM13, scounteren.HPM13, and hcounteren.HPM13 as follows:

mcounteren. HPM13	scounteren. HPM13	hcounteren. HPM13	hpmcounter13h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.15.1. Attributes

CSR Address	0xc8d
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.15.2. Format

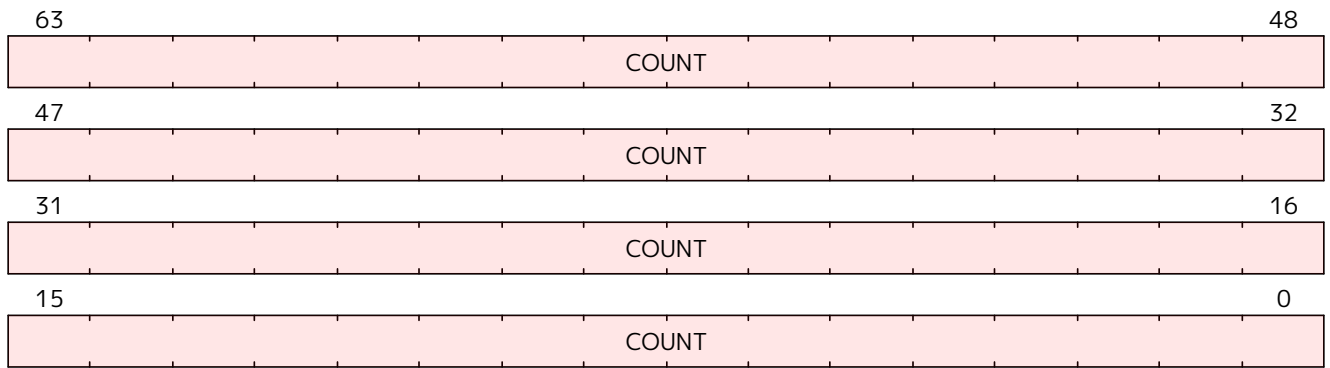


Figure 16. hpmcounter13h format

C.16. hpmcounter14

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter14](#).

Privilege mode access is controlled with mcounteren.HPM14 <%- if ext?(:S) -%> , scounteren.HPM14 <%- if ext?(:H) -%> , and hcounteren.HPM14 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM14	scounteren. HPM14	hcounteren. HPM14	hpmcounter14 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM14	scounteren.HPM14	hpmcounter14 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM14	hpmcounter14 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.16.1. Attributes

CSR Address	0xc0e
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.16.2. Format

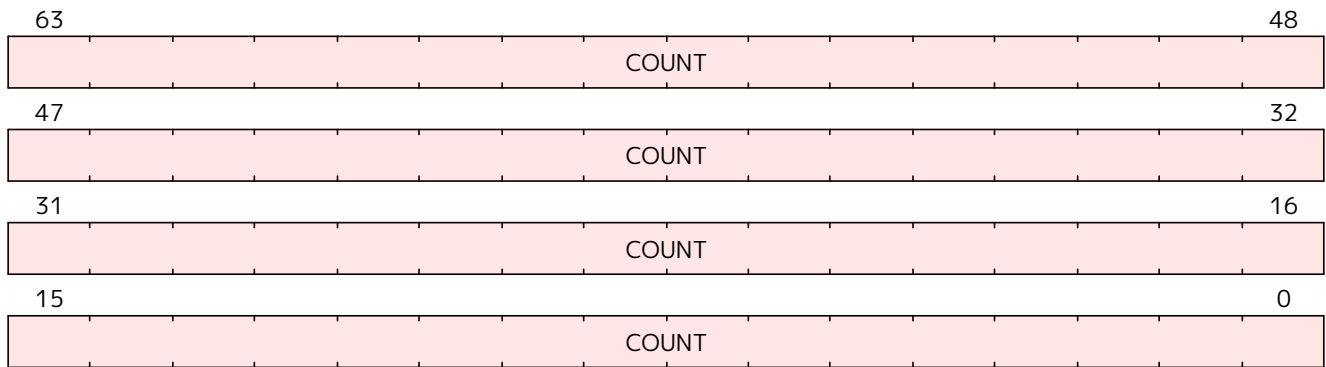


Figure 17. hpmcounter14 format

DRAFT

C.17. hpmcounter14h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter14h](#).

Privilege mode access is controlled with mcounteren.HPM14, scounteren.HPM14, and hcounteren.HPM14 as follows:

mcounteren. HPM14	scounteren. HPM14	hcounteren. HPM14	hpmcounter14h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.17.1. Attributes

CSR Address	0xc8e
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.17.2. Format

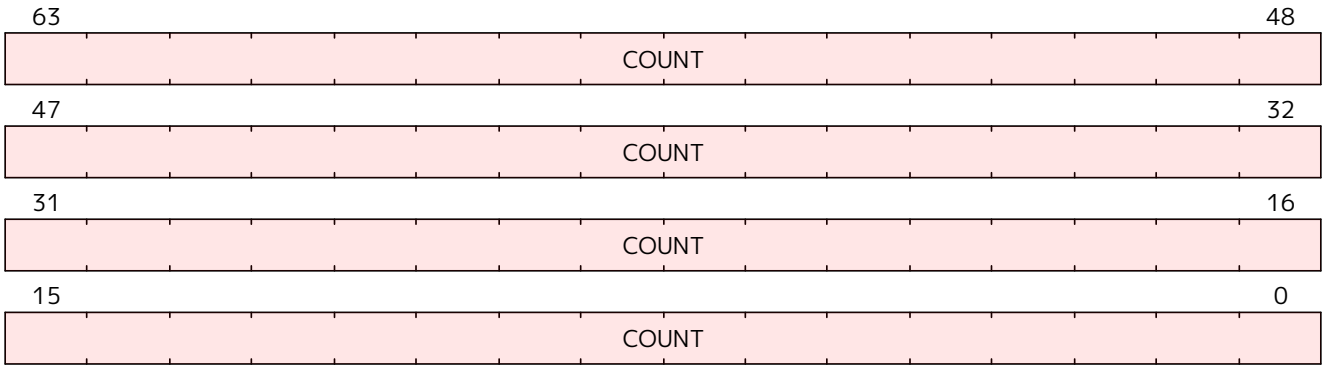


Figure 18. hpmcounter14h format

C.18. hpmcounter15

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter15](#).

Privilege mode access is controlled with mcounteren.HPM15 <%- if ext?(:S) -%> , scounteren.HPM15 <%- if ext?(:H) -%> , and hcounteren.HPM15 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM15	scounteren. HPM15	hcounteren. HPM15	hpmcounter15 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM15	scounteren.HPM15	hpmcounter15 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM15	hpmcounter15 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.18.1. Attributes

CSR Address	0xc0f
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.18.2. Format

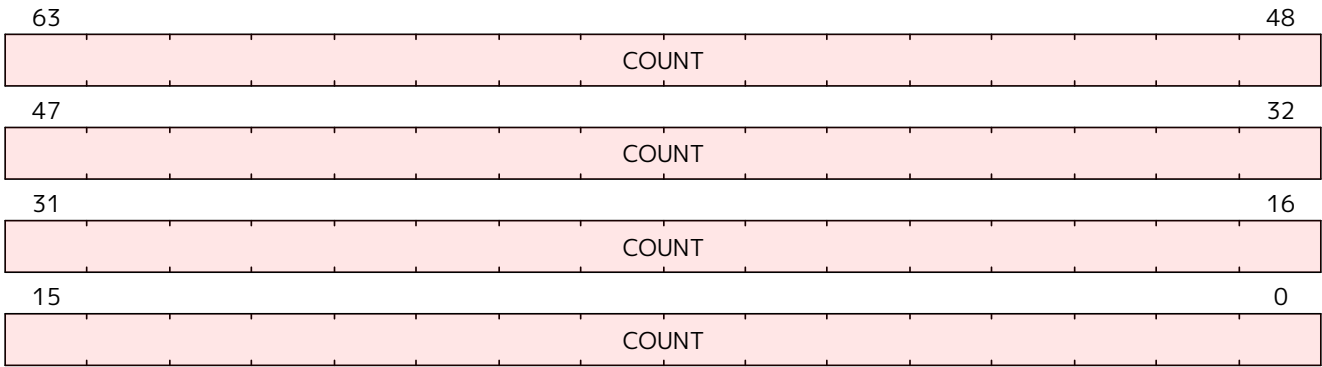


Figure 19. hpmcounter15 format

DRAFT

C.19. hpmcounter15h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter15h](#).

Privilege mode access is controlled with mcounteren.HPM15, scounteren.HPM15, and hcounteren.HPM15 as follows:

mcounteren. HPM15	scounteren. HPM15	hcounteren. HPM15	hpmcounter15h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.19.1. Attributes

CSR Address	0xc8f
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.19.2. Format

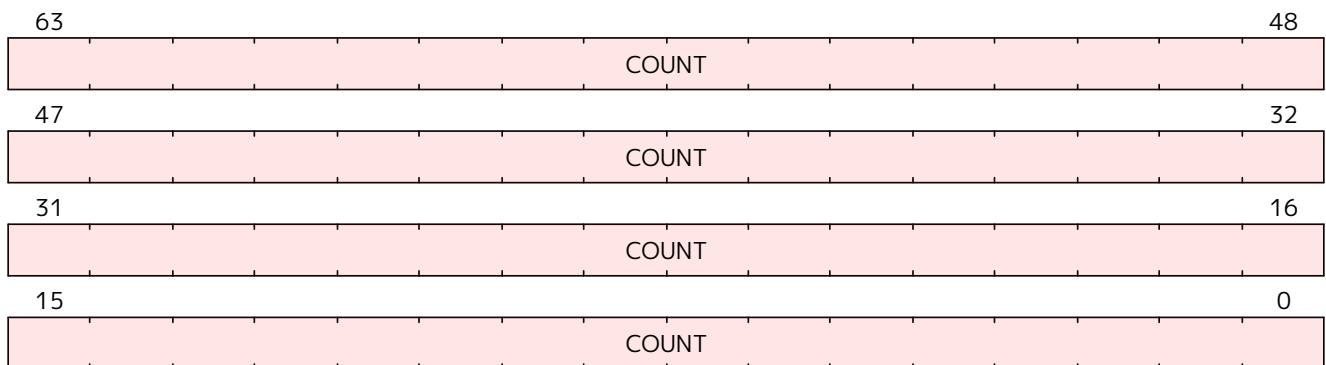


Figure 20. hpmcounter15h format

C.20. hpmcounter16

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter16](#).

Privilege mode access is controlled with mcounteren.HPM16 <%- if ext?(:S) -%> , scounteren.HPM16 <%- if ext?(:H) -%> , and hcounteren.HPM16 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM16	scounteren.HPM16	hcounteren.HPM16	hpmcounter16 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM16	scounteren.HPM16	hpmcounter16 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM16	hpmcounter16 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.20.1. Attributes

CSR Address	0xc10
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.20.2. Format

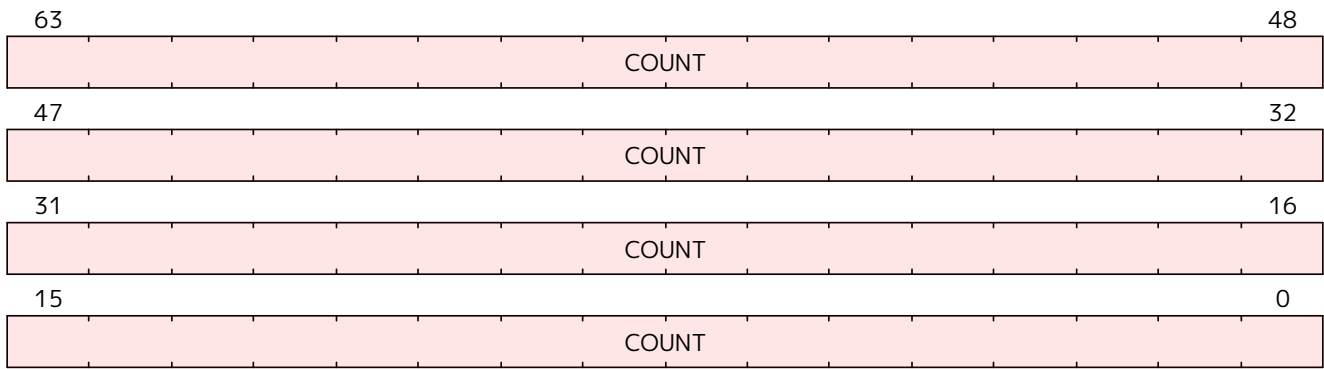


Figure 21. hpmcounter16 format

DRAFT

C.21. hpmcounter16h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter16h](#).

Privilege mode access is controlled with mcounteren.HPM16, scounteren.HPM16, and hcounteren.HPM16 as follows:

mcounteren. HPM16	scounteren. HPM16	hcounteren. HPM16	hpmcounter16h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.21.1. Attributes

CSR Address	0xc90
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.21.2. Format

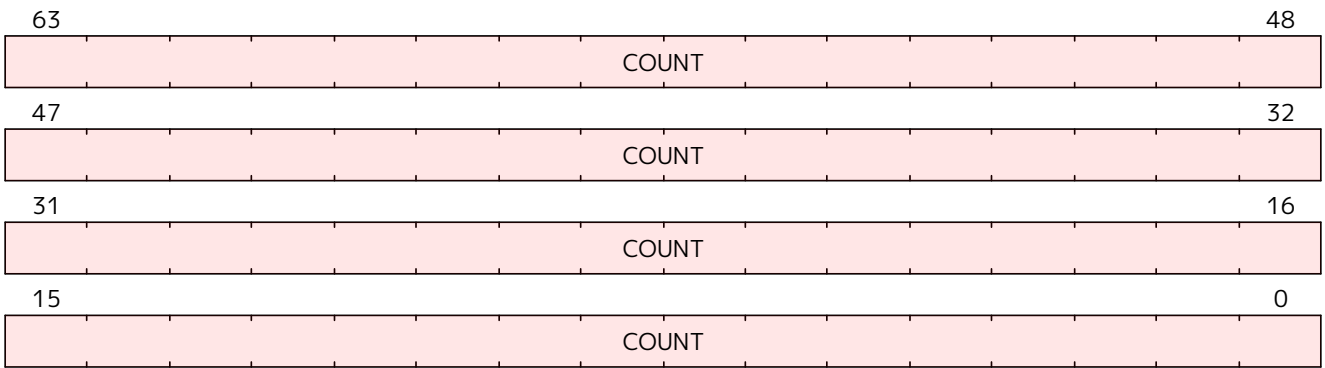


Figure 22. hpmcounter16h format

C.22. hpmcounter17

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter17](#).

Privilege mode access is controlled with mcounteren.HPM17 <%- if ext?(:S) -%> , scounteren.HPM17 <%- if ext?(:H) -%> , and hcounteren.HPM17 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM17	scounteren. HPM17	hcounteren. HPM17	hpmcounter17 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM17	scounteren.HPM17	hpmcounter17 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM17	hpmcounter17 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.22.1. Attributes

CSR Address	0xc11
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.22.2. Format

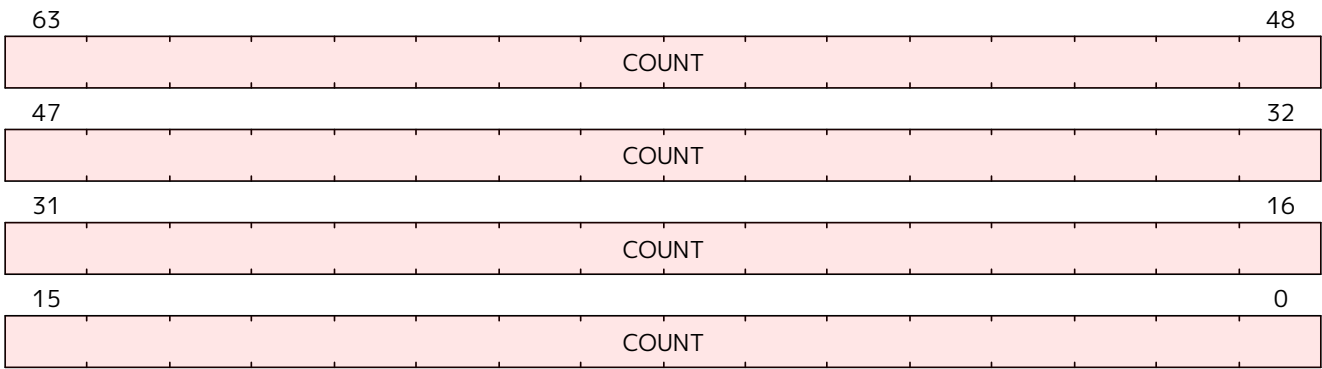


Figure 23. hpmcounter17 format

DRAFT

C.23. hpmcounter17h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter17h](#).

Privilege mode access is controlled with mcounteren.HPM17, scounteren.HPM17, and hcounteren.HPM17 as follows:

mcounteren. HPM17	scounteren. HPM17	hcounteren. HPM17	hpmcounter17h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.23.1. Attributes

CSR Address	0xc91
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.23.2. Format

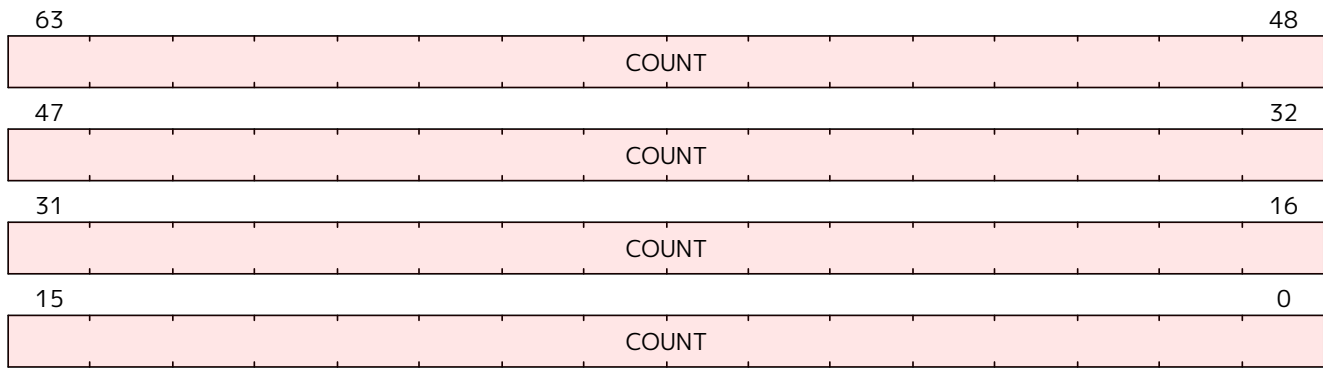


Figure 24. hpmcounter17h format

C.24. hpmcounter18

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter18](#).

Privilege mode access is controlled with mcounteren.HPM18 `<%- if ext?(:S) -%>` , scounteren.HPM18 `<%- if ext?(:H) -%>` , and hcounteren.HPM18 `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM18	scounteren. HPM18	hcounteren. HPM18	hpmcounter18 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM18	scounteren.HPM18	hpmcounter18 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM18	hpmcounter18 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.24.1. Attributes

CSR Address	0xc12
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.24.2. Format

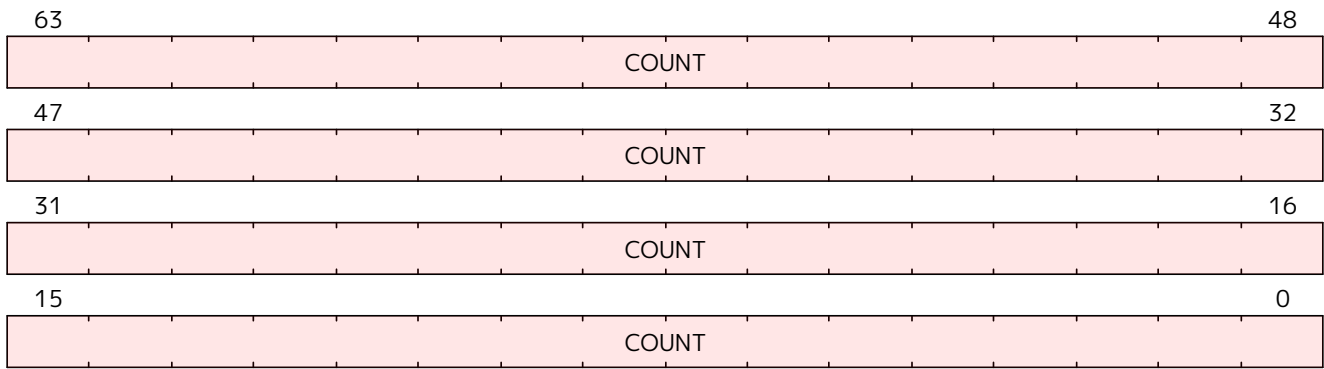


Figure 25. hpmcounter18 format

DRAFT

C.25. hpmcounter18h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter18h](#).

Privilege mode access is controlled with mcounteren.HPM18, scounteren.HPM18, and hcounteren.HPM18 as follows:

mcounteren. HPM18	scounteren. HPM18	hcounteren. HPM18	hpmcounter18h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.25.1. Attributes

CSR Address	0xc92
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.25.2. Format

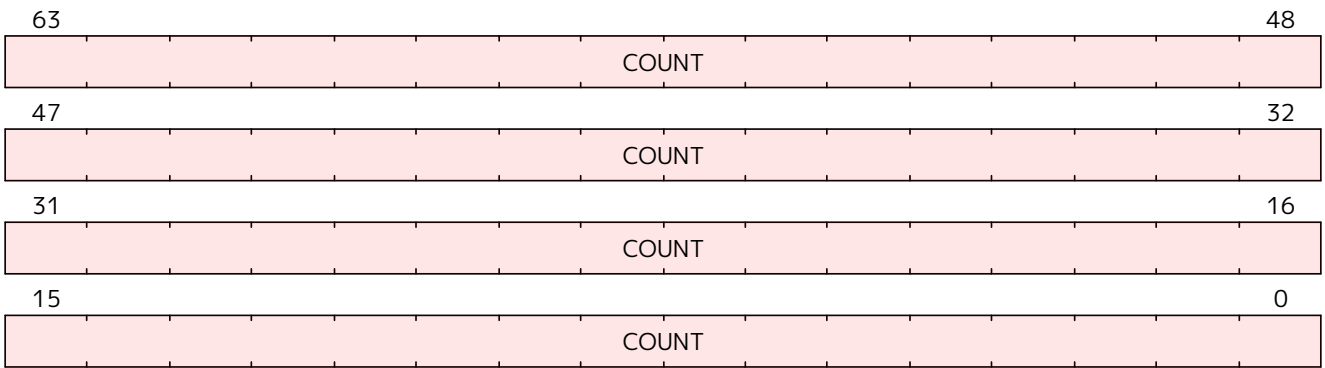


Figure 26. hpmcounter18h format

C.26. hpmcounter19

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter19](#).

Privilege mode access is controlled with mcounteren.HPM19 <%- if ext?(:S) -%> , scounteren.HPM19 <%- if ext?(:H) -%> , and hcounteren.HPM19 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM19	scounteren. HPM19	hcounteren. HPM19	hpmcounter19 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM19	scounteren.HPM19	hpmcounter19 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM19	hpmcounter19 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.26.1. Attributes

CSR Address	0xc13
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.26.2. Format

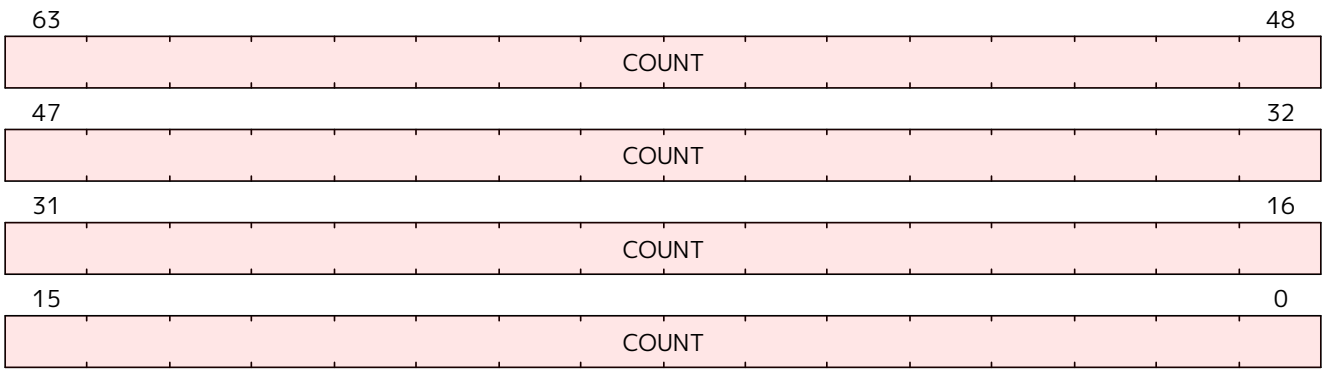


Figure 27. hpmcounter19 format

DRAFT

C.27. hpmcounter19h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter19h](#).

Privilege mode access is controlled with mcounteren.HPM19, scounteren.HPM19, and hcounteren.HPM19 as follows:

mcounteren. HPM19	scounteren. HPM19	hcounteren. HPM19	hpmcounter19h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.27.1. Attributes

CSR Address	0xc93
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.27.2. Format

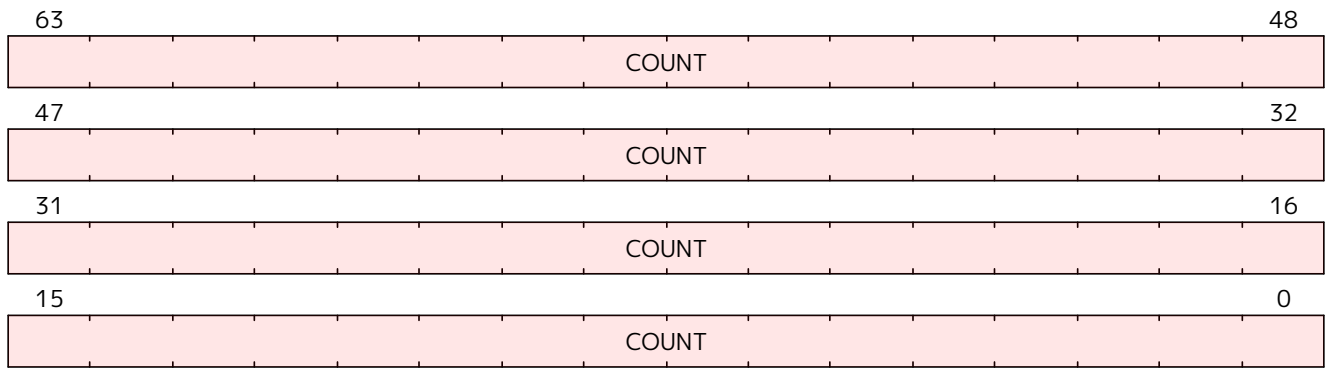


Figure 28. hpmcounter19h format

C.28. hpmcounter20

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter20](#).

Privilege mode access is controlled with mcounteren.HPM20 <%- if ext?(:S) -%> , scounteren.HPM20 <%- if ext?(:H) -%> , and hcounteren.HPM20 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM20	scounteren. HPM20	hcounteren. HPM20	hpmcounter20 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM20	scounteren.HPM20	hpmcounter20 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM20	hpmcounter20 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.28.1. Attributes

CSR Address	0xc14
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.28.2. Format

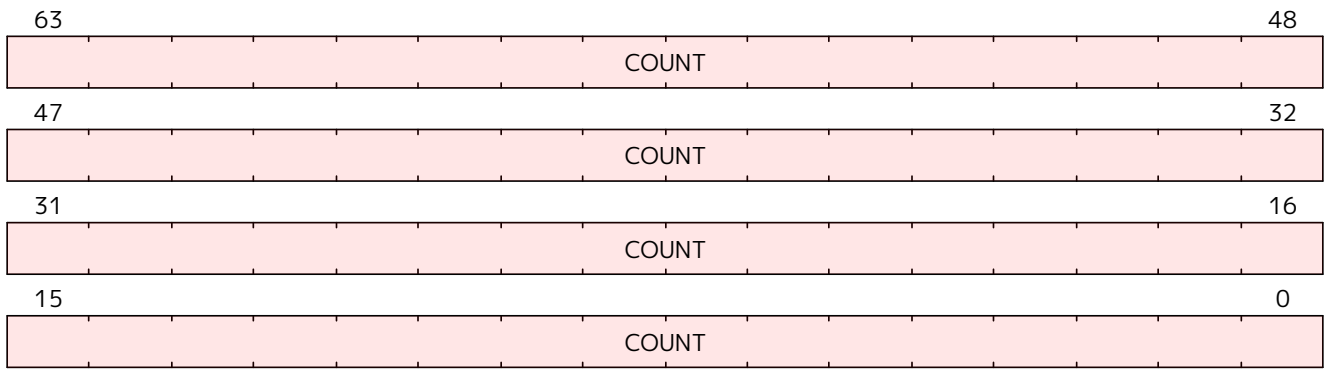


Figure 29. hpmcounter20 format

DRAFT

C.29. hpmcounter20h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter20h](#).

Privilege mode access is controlled with mcounteren.HPM20, scounteren.HPM20, and hcounteren.HPM20 as follows:

mcounteren. HPM20	scounteren. HPM20	hcounteren. HPM20	hpmcounter20h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.29.1. Attributes

CSR Address	0xc94
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.29.2. Format

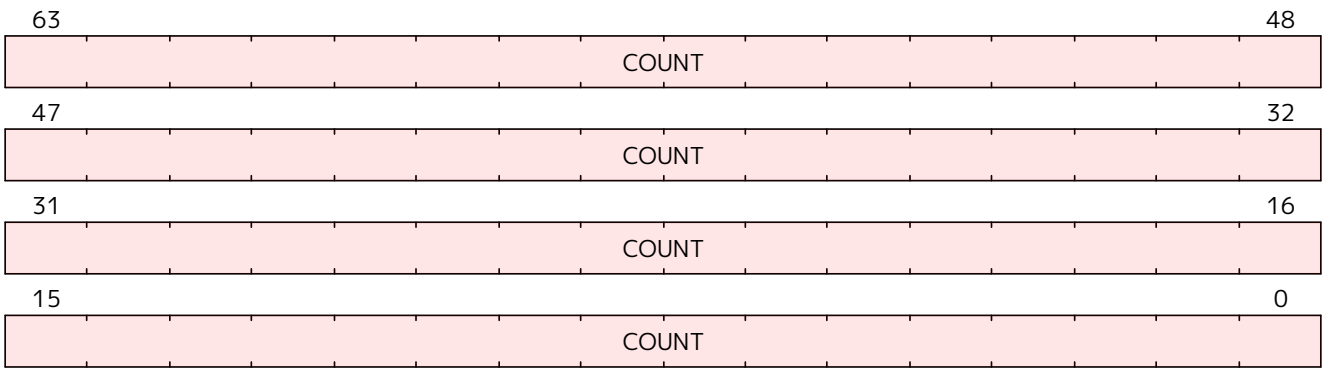


Figure 30. hpmcounter20h format

C.30. hpmcounter21

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter21](#).

Privilege mode access is controlled with mcounteren.HPM21 <%- if ext?(:S) -%> , scounteren.HPM21 <%- if ext?(:H) -%> , and hcounteren.HPM21 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM21	scounteren. HPM21	hcounteren. HPM21	hpmcounter21 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM21	scounteren.HPM21	hpmcounter21 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM21	hpmcounter21 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.30.1. Attributes

CSR Address	0xc15
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.30.2. Format

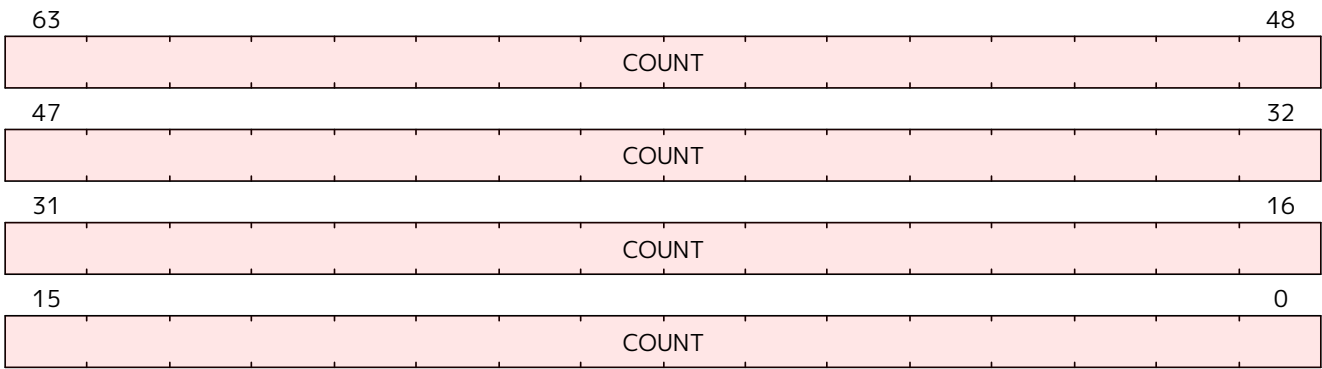


Figure 31. hpmcounter21 format

DRAFT

C.31. hpmcounter21h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter21h](#).

Privilege mode access is controlled with mcounteren.HPM21, scounteren.HPM21, and hcounteren.HPM21 as follows:

mcounteren. HPM21	scounteren. HPM21	hcounteren. HPM21	hpmcounter21h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.31.1. Attributes

CSR Address	0xc95
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.31.2. Format

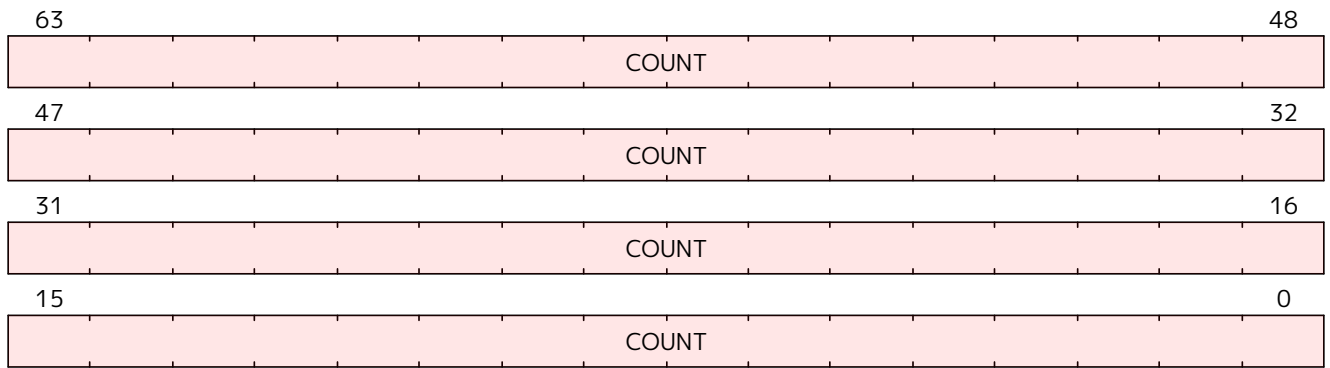


Figure 32. hpmcounter21h format

C.32. hpmcounter22

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter22](#).

Privilege mode access is controlled with mcounteren.HPM22 <%- if ext?(:S) -%> , scounteren.HPM22 <%- if ext?(:H) -%> , and hcounteren.HPM22 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM22	scounteren. HPM22	hcounteren. HPM22	hpmcounter22 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM22	scounteren.HPM22	hpmcounter22 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM22	hpmcounter22 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.32.1. Attributes

CSR Address	0xc16
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.32.2. Format

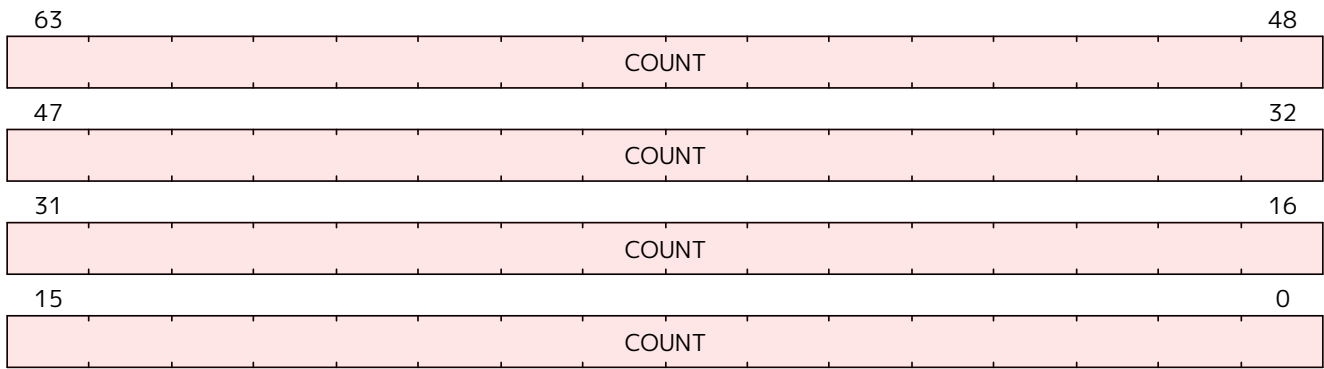


Figure 33. hpmcounter22 format

DRAFT

C.33. hpmcounter22h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter22h](#).

Privilege mode access is controlled with mcounteren.HPM22, scounteren.HPM22, and hcounteren.HPM22 as follows:

mcounteren. HPM22	scounteren. HPM22	hcounteren. HPM22	hpmcounter22h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.33.1. Attributes

CSR Address	0xc96
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.33.2. Format

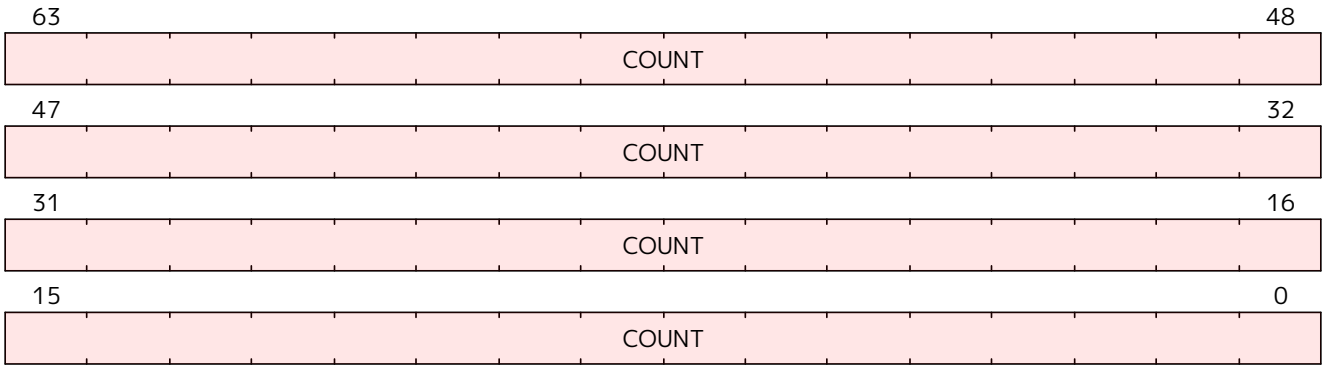


Figure 34. hpmcounter22h format

C.34. hpmcounter23

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter23](#).

Privilege mode access is controlled with mcounteren.HPM23 <%- if ext?(:S) -%> , scounteren.HPM23 <%- if ext?(:H) -%> , and hcounteren.HPM23 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM23	scounteren. HPM23	hcounteren. HPM23	hpmcounter23 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM23	scounteren.HPM23	hpmcounter23 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM23	hpmcounter23 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.34.1. Attributes

CSR Address	0xc17
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.34.2. Format

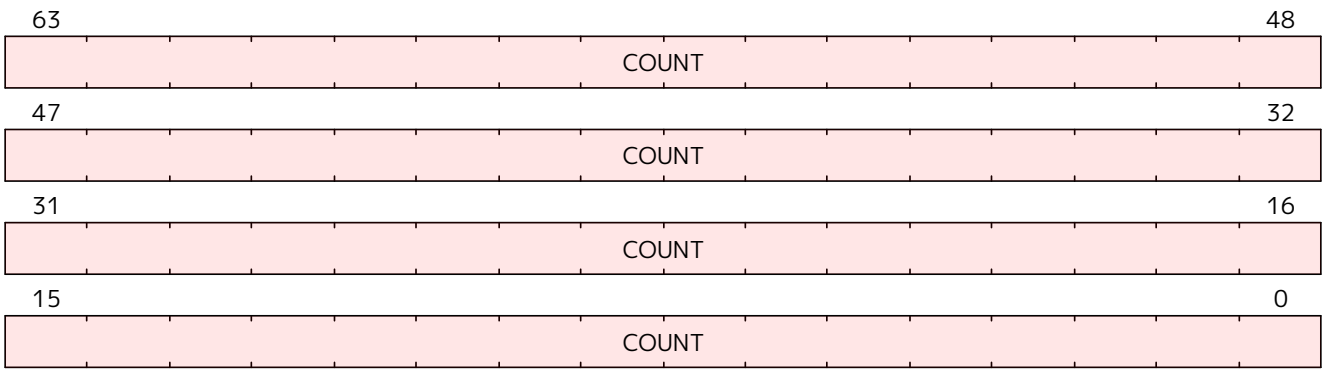


Figure 35. hpmcounter23 format

DRAFT

C.35. hpmcounter23h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter23h](#).

Privilege mode access is controlled with mcounteren.HPM23, scounteren.HPM23, and hcounteren.HPM23 as follows:

mcounteren. HPM23	scounteren. HPM23	hcounteren. HPM23	hpmcounter23h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.35.1. Attributes

CSR Address	0xc97
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.35.2. Format

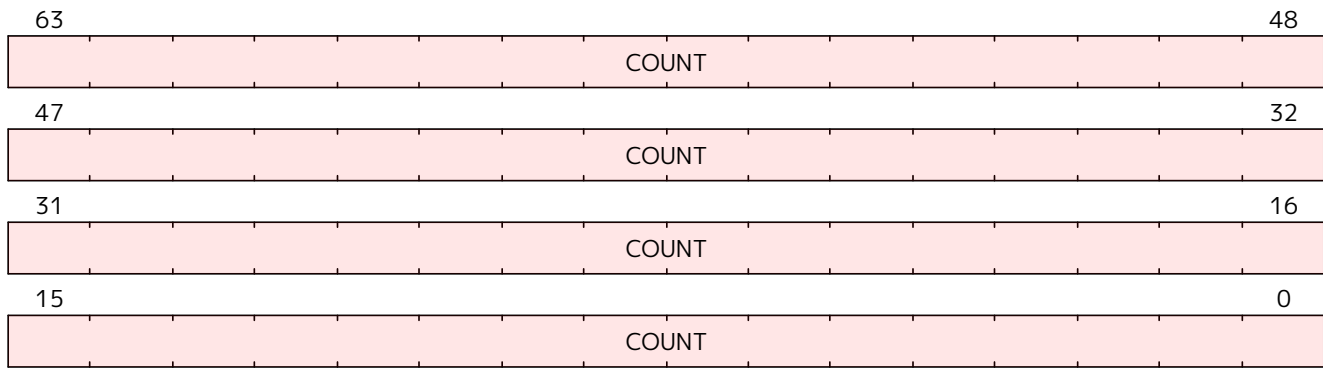


Figure 36. hpmcounter23h format

C.36. hpmcounter24

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter24](#).

Privilege mode access is controlled with mcounteren.HPM24 <%- if ext?(:S) -%> , scounteren.HPM24 <%- if ext?(:H) -%> , and hcounteren.HPM24 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM24	scounteren.HPM24	hcounteren.HPM24	hpmcounter24 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM24	scounteren.HPM24	hpmcounter24 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM24	hpmcounter24 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.36.1. Attributes

CSR Address	0xc18
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.36.2. Format

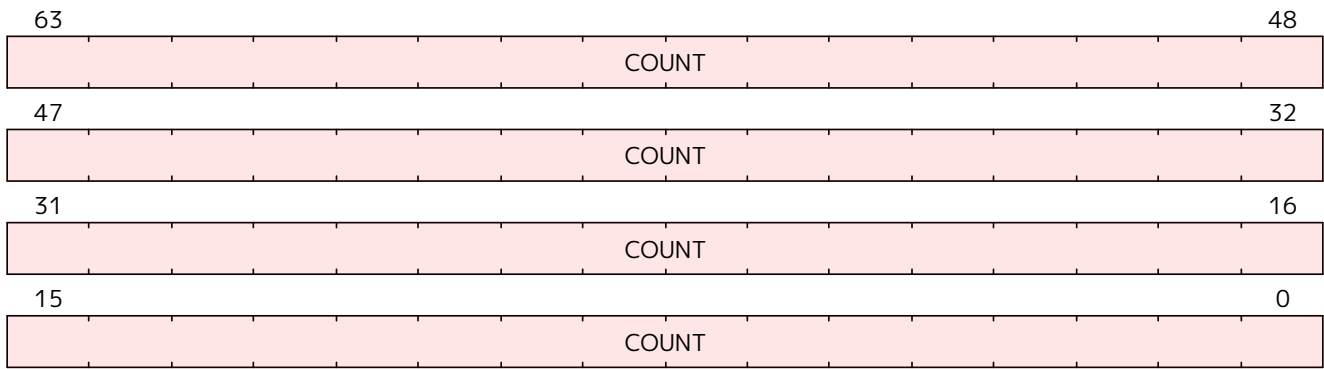


Figure 37. hpmcounter24 format

DRAFT

C.37. hpmcounter24h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter24h](#).

Privilege mode access is controlled with mcounteren.HPM24, scounteren.HPM24, and hcounteren.HPM24 as follows:

mcounteren. HPM24	scounteren. HPM24	hcounteren. HPM24	hpmcounter24h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.37.1. Attributes

CSR Address	0xc98
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.37.2. Format

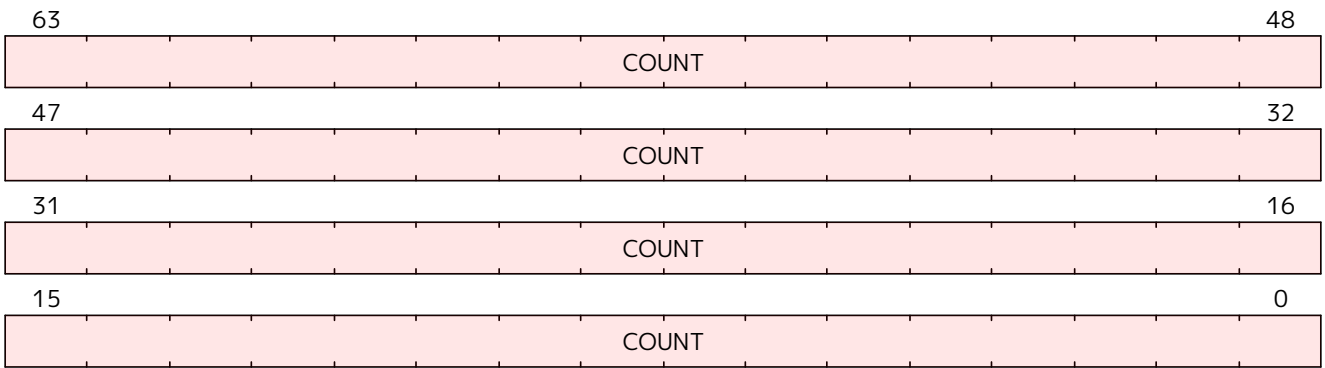


Figure 38. hpmcounter24h format

C.38. hpmcounter25

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter25](#).

Privilege mode access is controlled with mcounteren.HPM25 <%- if ext?(:S) -%> , scounteren.HPM25 <%- if ext?(:H) -%> , and hcounteren.HPM25 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM25	scounteren. HPM25	hcounteren. HPM25	hpmcounter25 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM25	scounteren.HPM25	hpmcounter25 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM25	hpmcounter25 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.38.1. Attributes

CSR Address	0xc19
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.38.2. Format

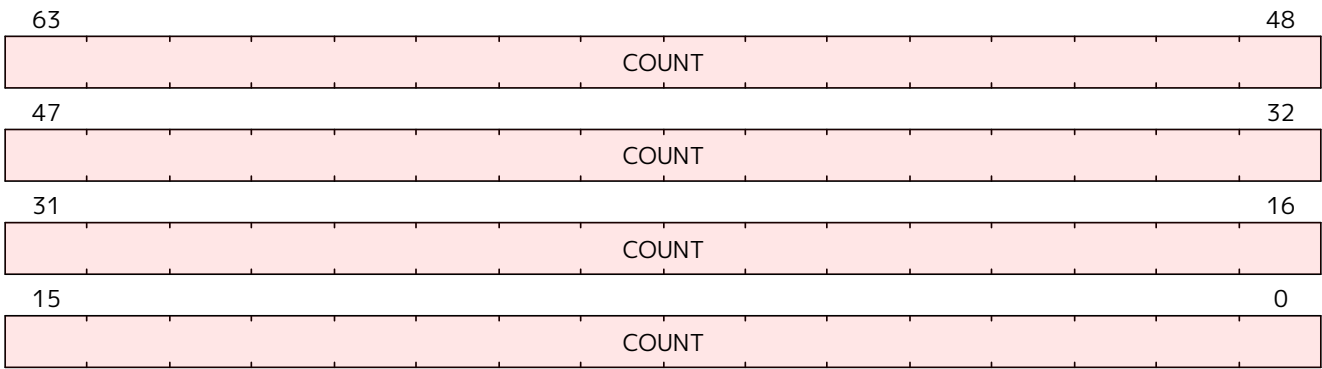


Figure 39. hpmcounter25 format

DRAFT

C.39. hpmcounter25h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter25h](#).

Privilege mode access is controlled with mcounteren.HPM25, scounteren.HPM25, and hcounteren.HPM25 as follows:

mcounteren. HPM25	scounteren. HPM25	hcounteren. HPM25	hpmcounter25h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.39.1. Attributes

CSR Address	0xc99
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.39.2. Format

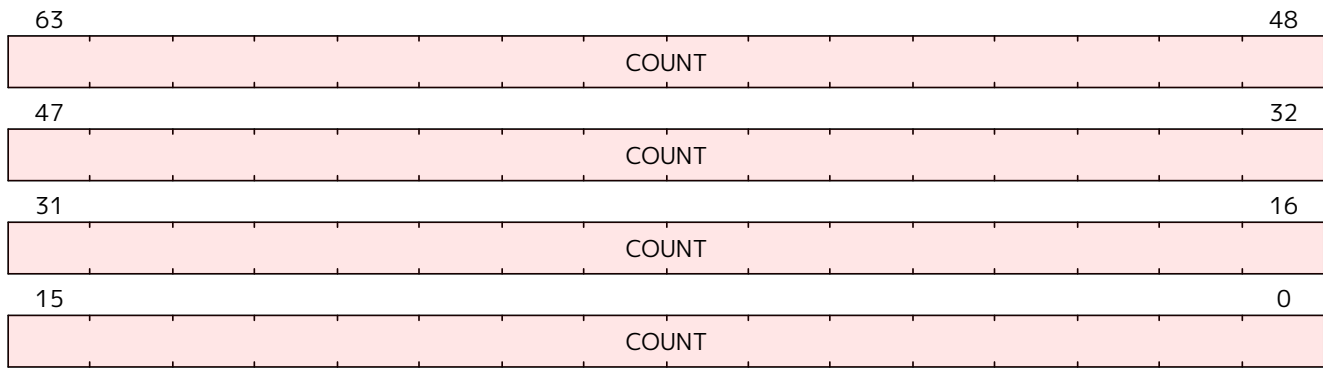


Figure 40. hpmcounter25h format

C.40. hpmcounter26

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter26](#).

Privilege mode access is controlled with `mcounteren.HPM26` `<%- if ext?(:S) -%>`, `scounteren.HPM26` `<%- if ext?(:H) -%>`, and `hcounteren.HPM26` `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM26	scounteren. HPM26	hcounteren. HPM26	hpmcounter26 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM26	scounteren.HPM26	hpmcounter26 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM26	hpmcounter26 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.40.1. Attributes

CSR Address	0xc1a
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.40.2. Format

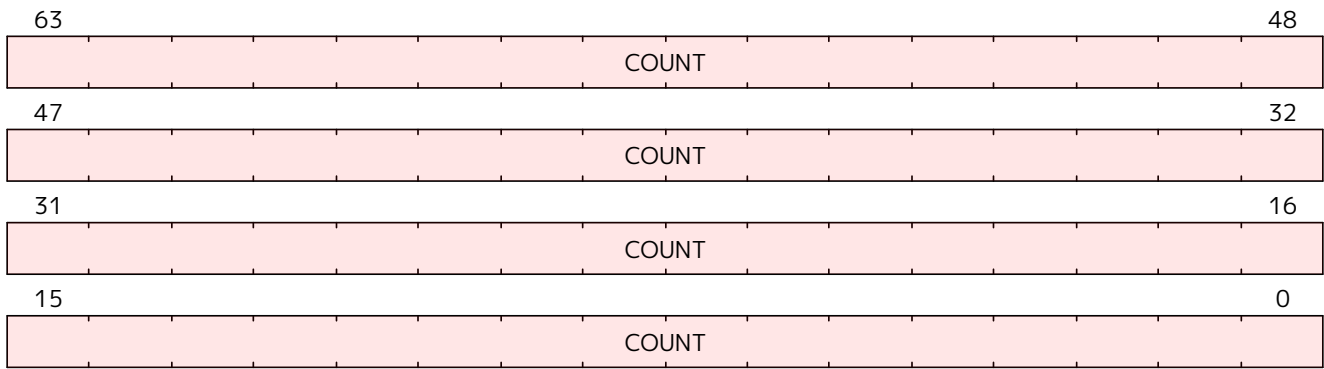


Figure 41. hpmcounter26 format

DRAFT

C.41. hpmcounter26h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter26h](#).

Privilege mode access is controlled with mcounteren.HPM26, scounteren.HPM26, and hcounteren.HPM26 as follows:

mcounteren. HPM26	scounteren. HPM26	hcounteren. HPM26	hpmcounter26h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.41.1. Attributes

CSR Address	0xc9a
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.41.2. Format

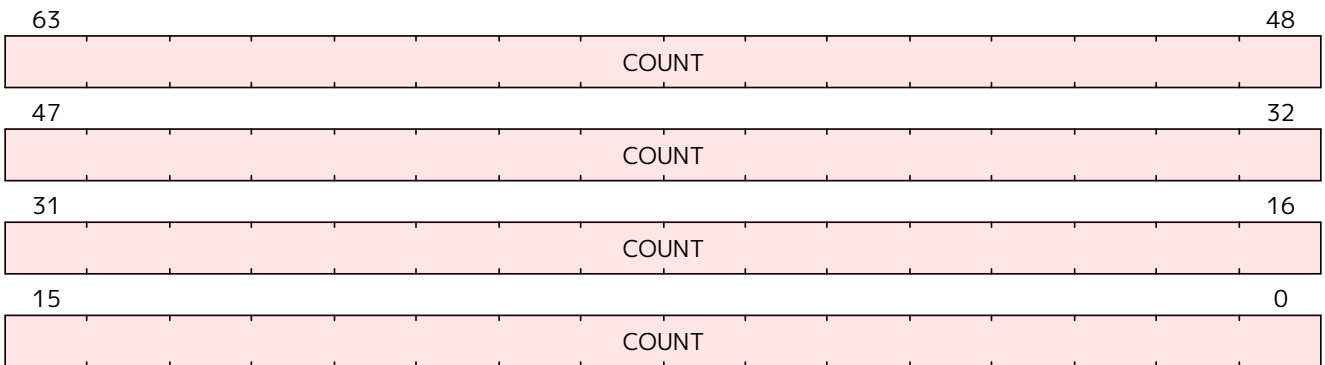


Figure 42. hpmcounter26h format

C.42. hpmcounter27

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter27](#).

Privilege mode access is controlled with mcounteren.HPM27 <%- if ext?(:S) -%> , scounteren.HPM27 <%- if ext?(:H) -%> , and hcounteren.HPM27 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM27	scounteren. HPM27	hcounteren. HPM27	hpmcounter27 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM27	scounteren.HPM27	hpmcounter27 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM27	hpmcounter27 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.42.1. Attributes

CSR Address	0xc1b
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.42.2. Format

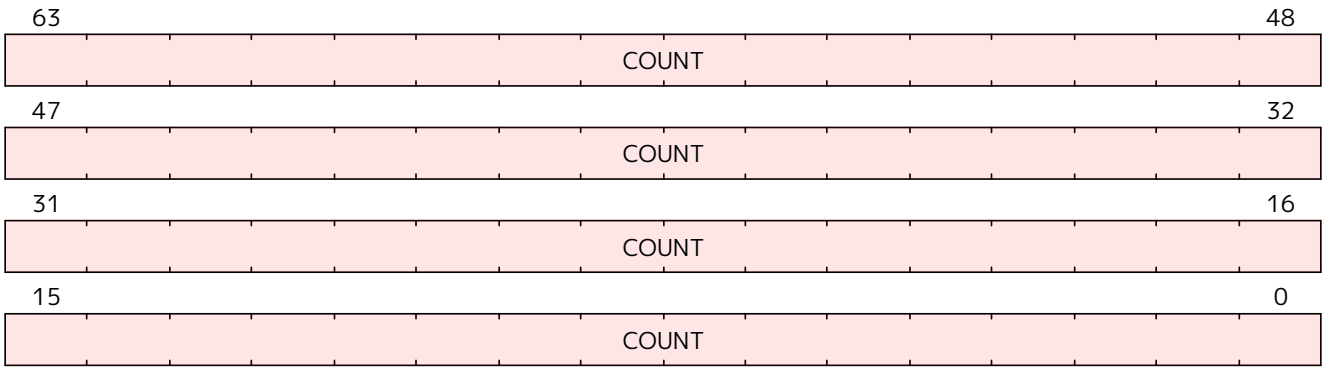


Figure 43. hpmcounter27 format

DRAFT

C.43. hpmcounter27h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter27h](#).

Privilege mode access is controlled with mcounteren.HPM27, scounteren.HPM27, and hcounteren.HPM27 as follows:

mcounteren. HPM27	scounteren. HPM27	hcounteren. HPM27	hpmcounter27h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.43.1. Attributes

CSR Address	0xc9b
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.43.2. Format

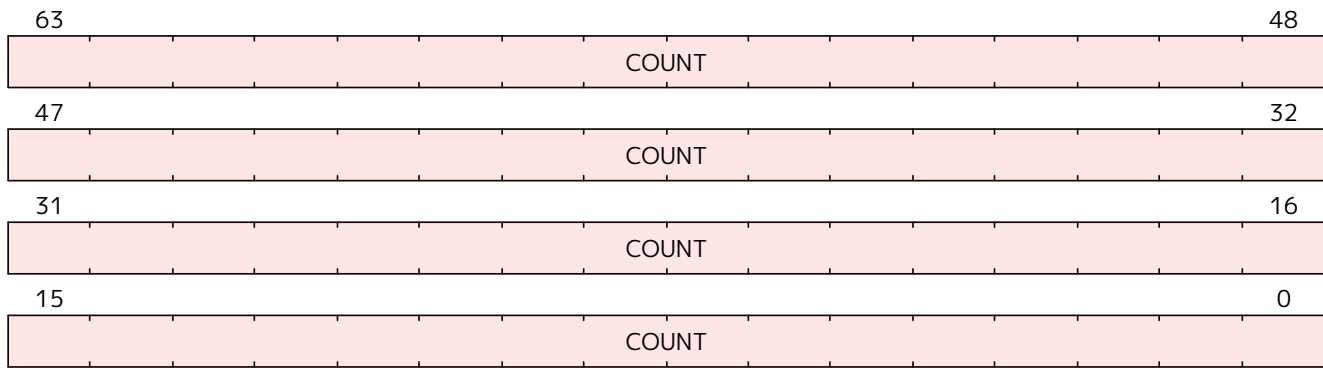


Figure 44. hpmcounter27h format

C.44. hpmcounter28

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter28](#).

Privilege mode access is controlled with `mcounteren.HPM28` `<%- if ext?(:S) -%>`, `scounteren.HPM28` `<%- if ext?(:H) -%>`, and `hcounteren.HPM28` `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM28	scounteren. HPM28	hcounteren. HPM28	hpmcounter28 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM28	scounteren.HPM28	hpmcounter28 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM28	hpmcounter28 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.44.1. Attributes

CSR Address	0xc1c
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.44.2. Format

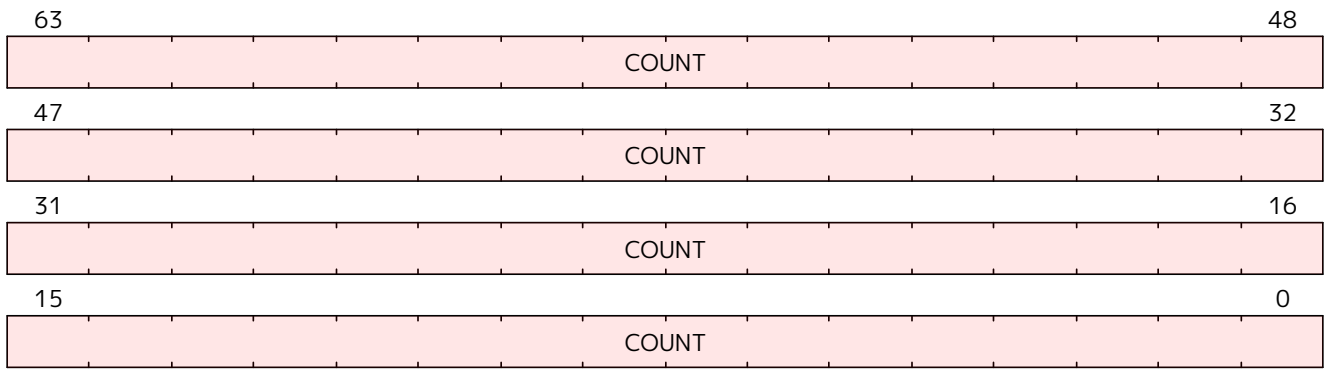


Figure 45. hpmcounter28 format

DRAFT

C.45. hpmcounter28h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter28h](#).

Privilege mode access is controlled with mcounteren.HPM28, scounteren.HPM28, and hcounteren.HPM28 as follows:

mcounteren. HPM28	scounteren. HPM28	hcounteren. HPM28	hpmcounter28h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.45.1. Attributes

CSR Address	0xc9c
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.45.2. Format

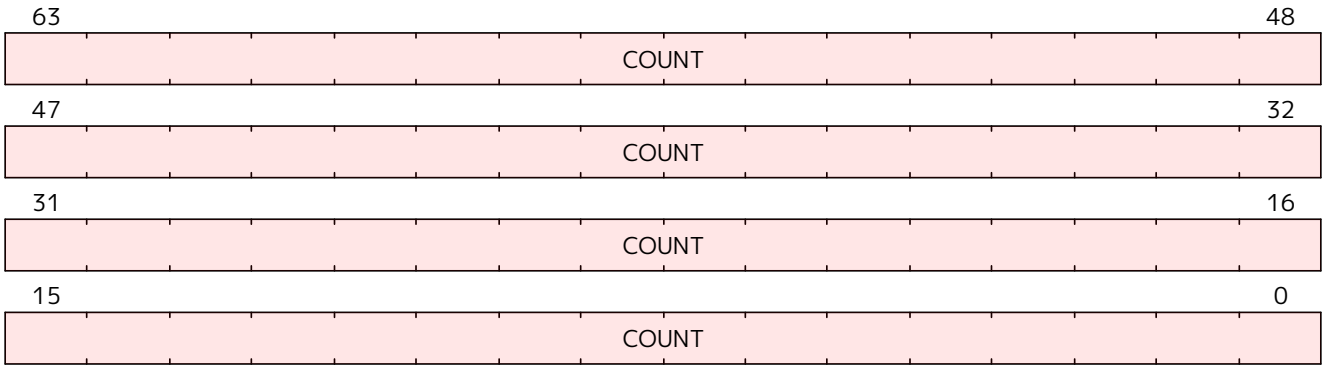


Figure 46. hpmcounter28h format

C.46. hpmcounter29

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter29](#).

Privilege mode access is controlled with `mcounteren.HPM29` `<%- if ext?(:S) -%>`, `scounteren.HPM29` `<%- if ext?(:H) -%>`, and `hcounteren.HPM29` `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM29	scounteren. HPM29	hcounteren. HPM29	hpmcounter29 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM29	scounteren.HPM29	hpmcounter29 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM29	hpmcounter29 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.46.1. Attributes

CSR Address	0xc1d
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.46.2. Format

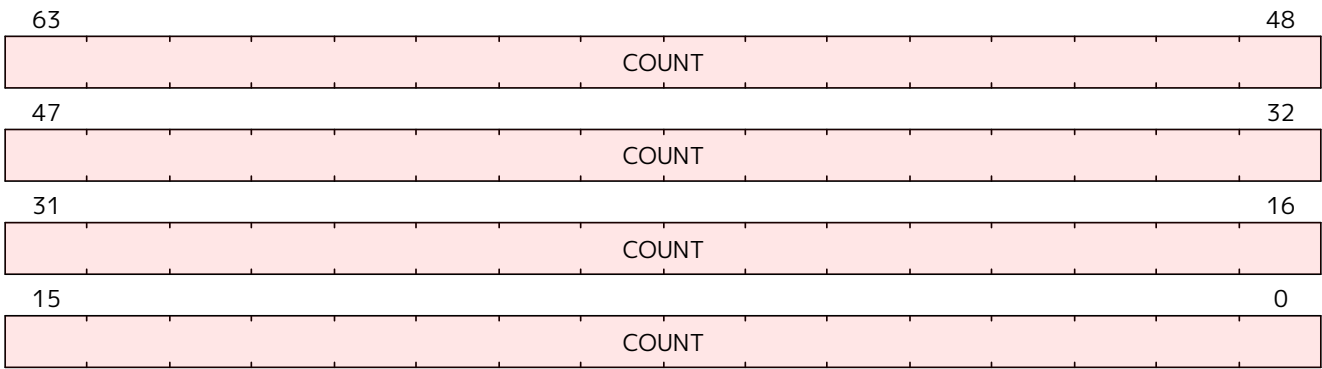


Figure 47. hpmcounter29 format

DRAFT

C.47. hpmcounter29h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter29h](#).

Privilege mode access is controlled with mcounteren.HPM29, scounteren.HPM29, and hcounteren.HPM29 as follows:

mcounteren. HPM29	scounteren. HPM29	hcounteren. HPM29	hpmcounter29h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.47.1. Attributes

CSR Address	0xc9d
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.47.2. Format

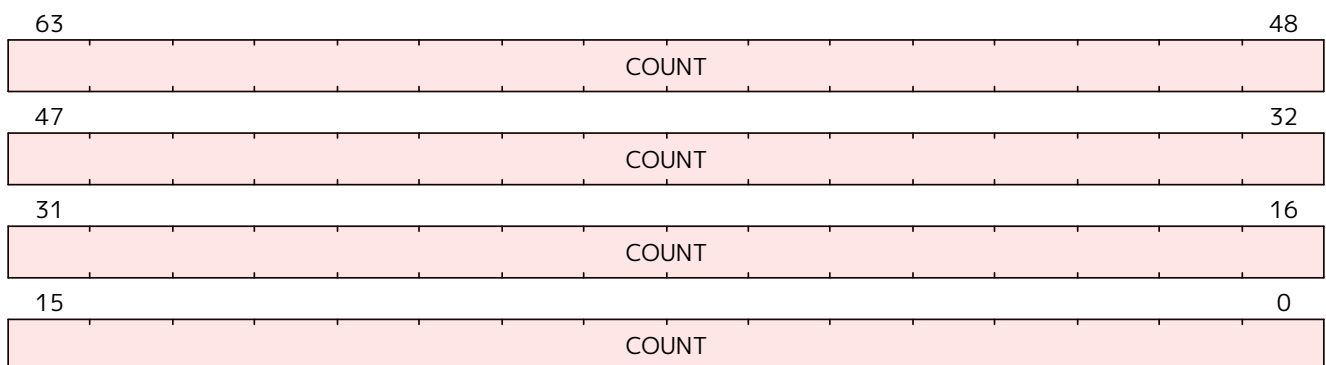


Figure 48. hpmcounter29h format

C.48. hpmcounter3

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter3](#).

Privilege mode access is controlled with mcounteren.HPM3 `<%- if ext?(:S) -%>` , scounteren.HPM3 `<%- if ext?(:H) -%>` , and hcounteren.HPM3 `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM3	scounteren. HPM3	hcounteren. HPM3	hpmcounter3 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM3	scounteren.HPM3	hpmcounter3 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM3	hpmcounter3 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.48.1. Attributes

CSR Address	0xc03
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.48.2. Format

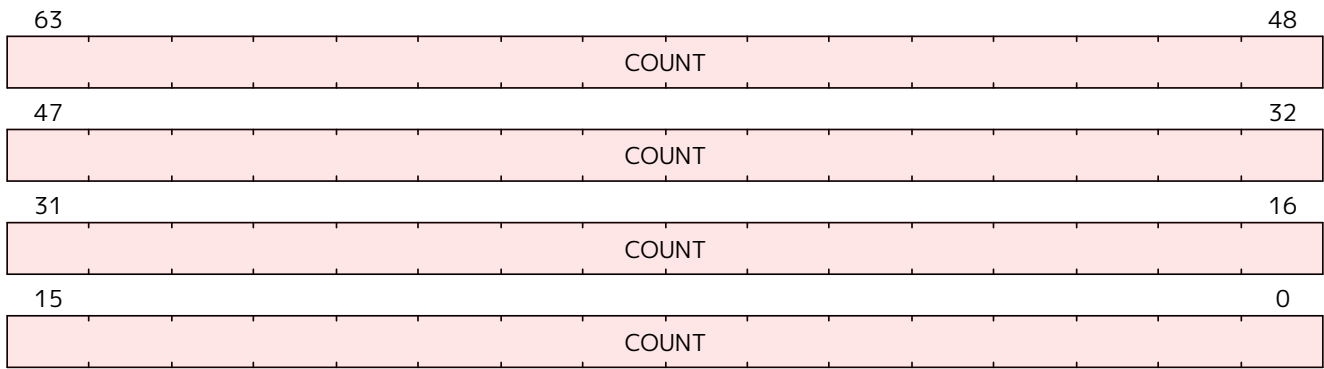


Figure 49. hpmcounter3 format

DRAFT

C.49. hpmcounter30

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter30](#).

Privilege mode access is controlled with mcounteren.HPM30 <%- if ext?(:S) -%> , scounteren.HPM30 <%- if ext?(:H) -%> , and hcounteren.HPM30 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM30	scounteren.HPM30	hcounteren.HPM30	hpmcounter30 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM30	scounteren.HPM30	hpmcounter30 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM30	hpmcounter30 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.49.1. Attributes

CSR Address	0xc1e
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.49.2. Format

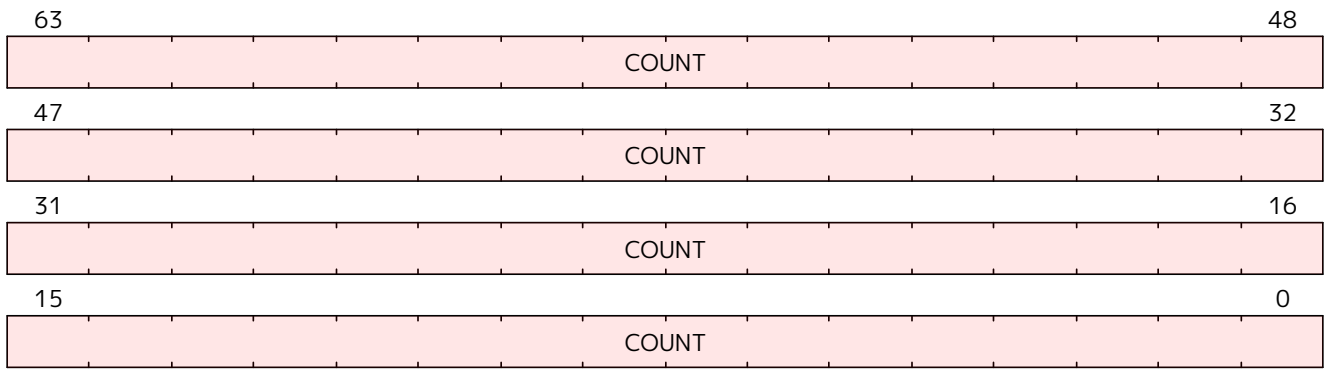


Figure 50. hpmcounter30 format

DRAFT

C.50. hpmcounter30h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter30h](#).

Privilege mode access is controlled with mcounteren.HPM30, scounteren.HPM30, and hcounteren.HPM30 as follows:

mcounteren. HPM30	scounteren. HPM30	hcounteren. HPM30	hpmcounter30h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.50.1. Attributes

CSR Address	0xc9e
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.50.2. Format

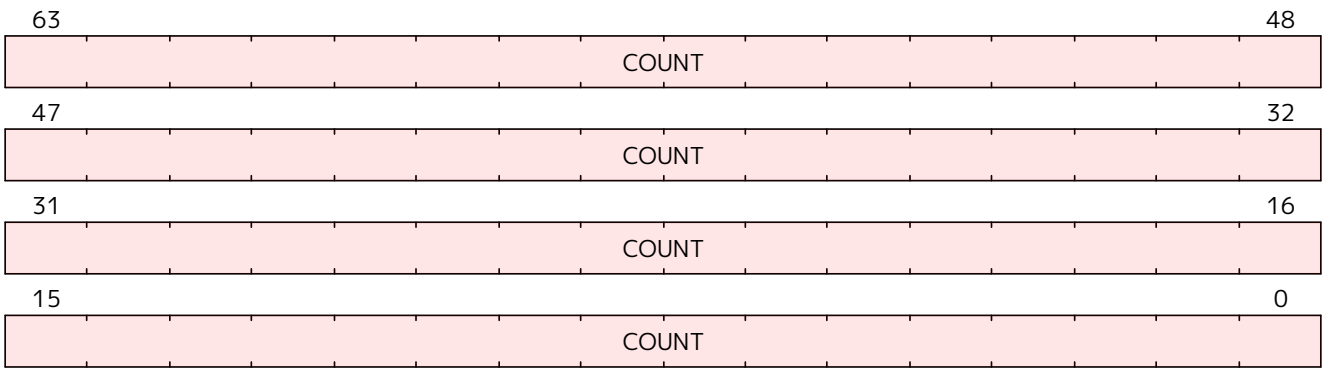


Figure 51. hpmcounter30h format

C.51. hpmcounter31

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter31](#).

Privilege mode access is controlled with mcounteren.HPM31 <%- if ext?(:S) -%> , scounteren.HPM31 <%- if ext?(:H) -%> , and hcounteren.HPM31 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM31	scounteren. HPM31	hcounteren. HPM31	hpmcounter31 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM31	scounteren.HPM31	hpmcounter31 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM31	hpmcounter31 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.51.1. Attributes

CSR Address	0xc1f
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.51.2. Format

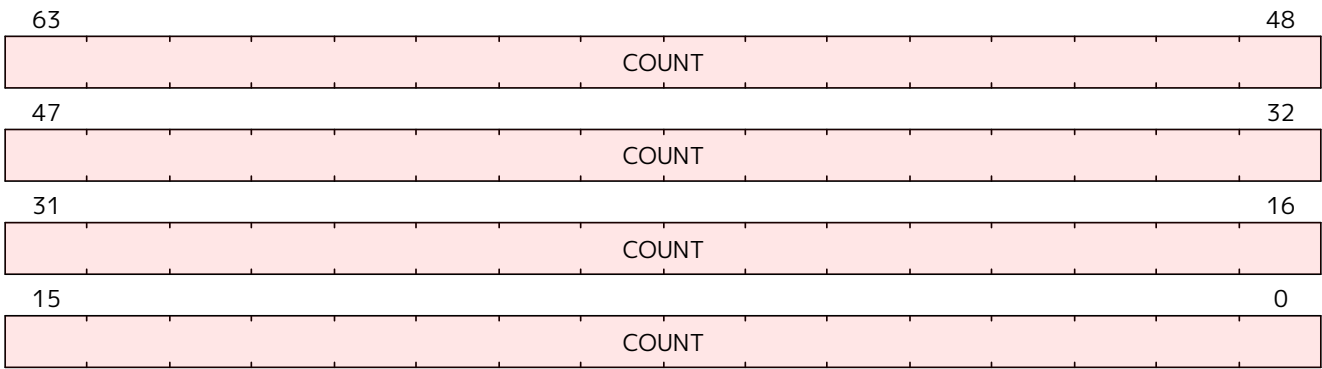


Figure 52. hpmcounter31 format

DRAFT

C.52. hpmcounter31h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter31h](#).

Privilege mode access is controlled with mcounteren.HPM31, scounteren.HPM31, and hcounteren.HPM31 as follows:

mcounteren. HPM31	scounteren. HPM31	hcounteren. HPM31	hpmcounter31h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.52.1. Attributes

CSR Address	0xc9f
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.52.2. Format

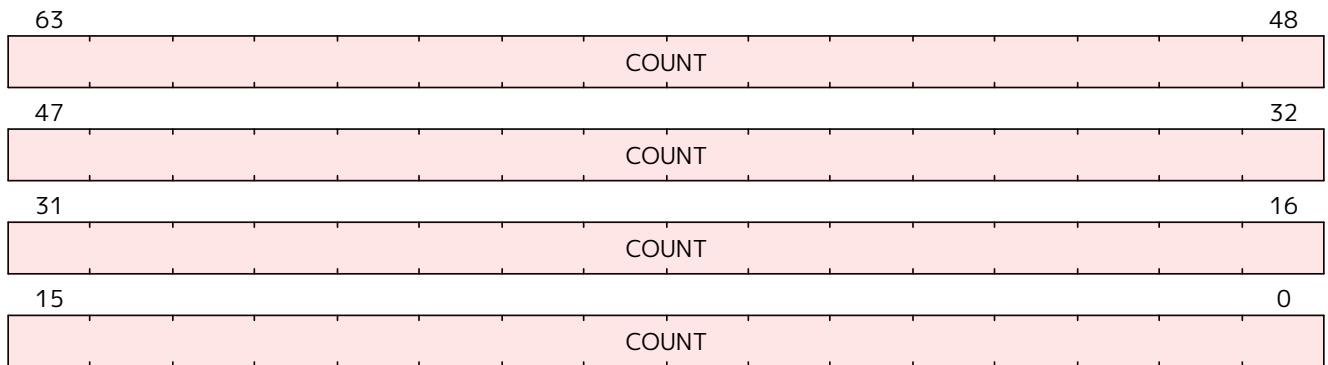


Figure 53. hpmcounter31h format

C.53. hpmcounter3h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter3h](#).

Privilege mode access is controlled with mcounteren.HPM3, scounteren.HPM3, and hcounteren.HPM3 as follows:

mcounteren. HPM3	scounteren. HPM3	hcounteren. HPM3	hpmcounter3h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.53.1. Attributes

CSR Address	0xc83
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.53.2. Format

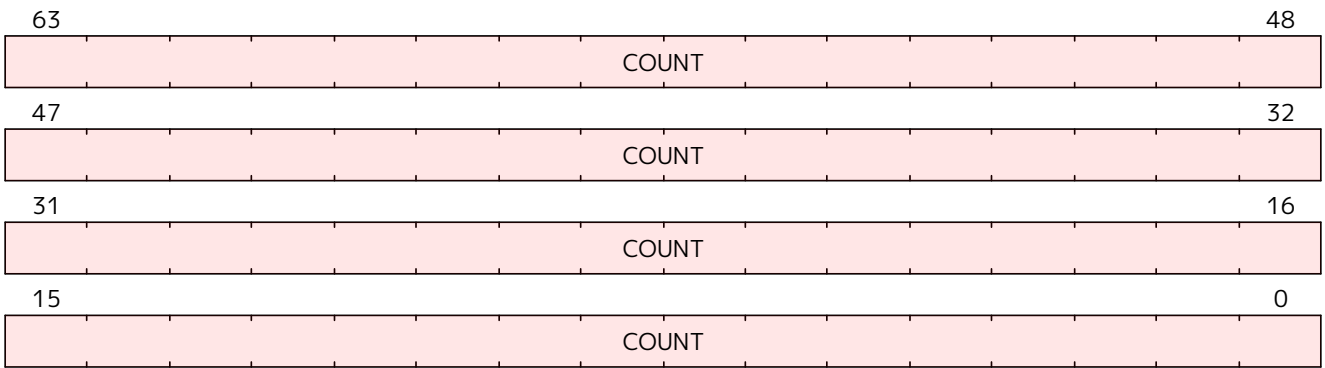


Figure 54. hpmcounter3h format

C.54. hpmcounter4

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter4](#).

Privilege mode access is controlled with mcounteren.HPM4 <%- if ext?(S) -%> , scounteren.HPM4 <%- if ext?(H) -%> , and hcounteren.HPM4 <%- end -%> <%- end -%> as follows:

<%- if ext?(H) -%>

mcounteren. HPM4	scounteren. HPM4	hcounteren. HPM4	hpmcounter4 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(S) -%>

mcounteren.HPM4	scounteren.HPM4	hpmcounter4 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM4	hpmcounter4 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.54.1. Attributes

CSR Address	0xc04
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.54.2. Format

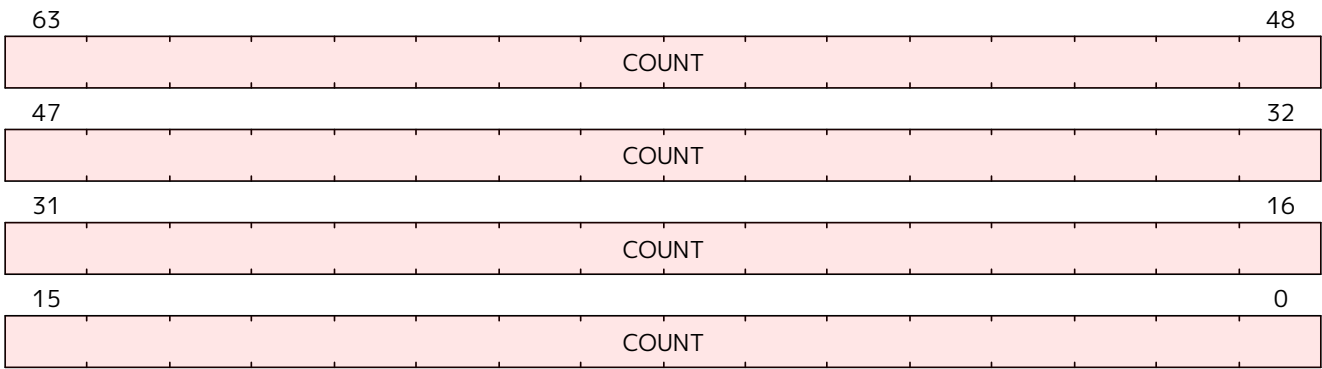


Figure 55. hpmcounter4 format

DRAFT

C.55. hpmcounter4h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter4h](#).

Privilege mode access is controlled with mcounteren.HPM4, scounteren.HPM4, and hcounteren.HPM4 as follows:

mcounteren. HPM4	scounteren. HPM4	hcounteren. HPM4	hpmcounter4h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.55.1. Attributes

CSR Address	0xc84
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.55.2. Format

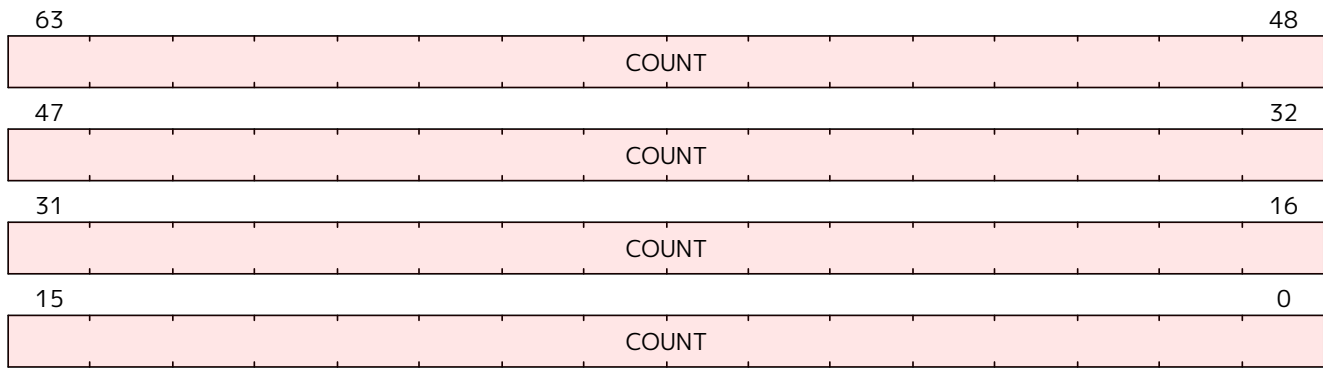


Figure 56. hpmcounter4h format

C.56. hpmcounter5

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter5](#).

Privilege mode access is controlled with mcounteren.HPM5 `<%- if ext?(:S) -%>` , scounteren.HPM5 `<%- if ext?(:H) -%>` , and hcounteren.HPM5 `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren. HPM5	scounteren. HPM5	hcounteren. HPM5	hpmcounter5 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM5	scounteren.HPM5	hpmcounter5 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM5	hpmcounter5 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.56.1. Attributes

CSR Address	0xc05
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.56.2. Format

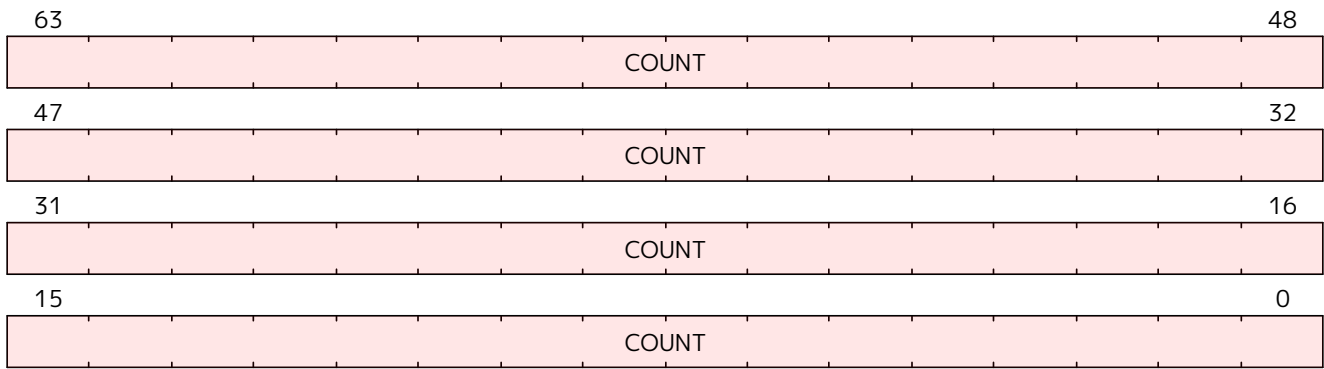


Figure 57. hpmcounter5 format

DRAFT

C.57. hpmcounter5h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter5h](#).

Privilege mode access is controlled with mcounteren.HPM5, scounteren.HPM5, and hcounteren.HPM5 as follows:

mcounteren. HPM5	scounteren. HPM5	hcounteren. HPM5	hpmcounter5h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.57.1. Attributes

CSR Address	0xc85
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.57.2. Format

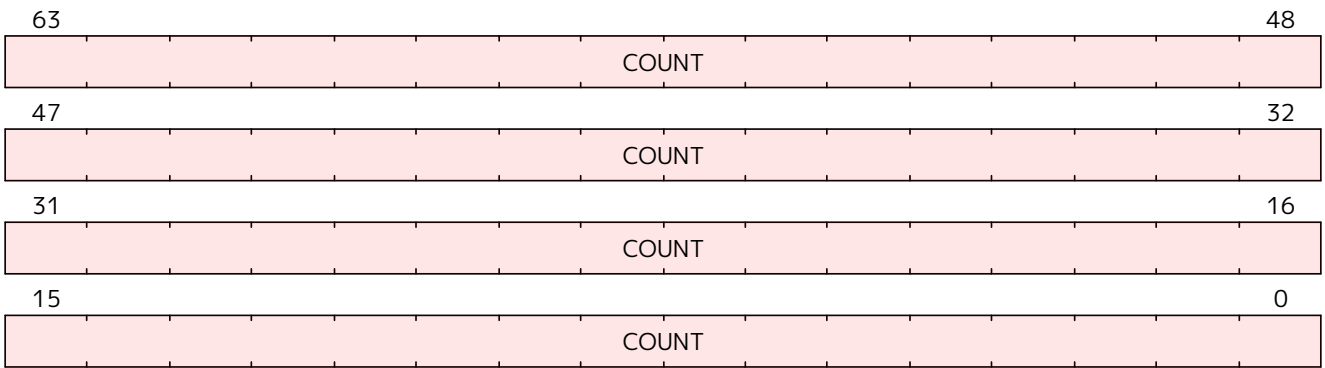


Figure 58. hpmcounter5h format

C.58. hpmcounter6

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter6](#).

Privilege mode access is controlled with mcounteren.HPM6 <%- if ext?(:S) -%> , scounteren.HPM6 <%- if ext?(:H) -%> , and hcounteren.HPM6 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM6	scounteren. HPM6	hcounteren. HPM6	hpmcounter6 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM6	scounteren.HPM6	hpmcounter6 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM6	hpmcounter6 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.58.1. Attributes

CSR Address	0xc06
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.58.2. Format

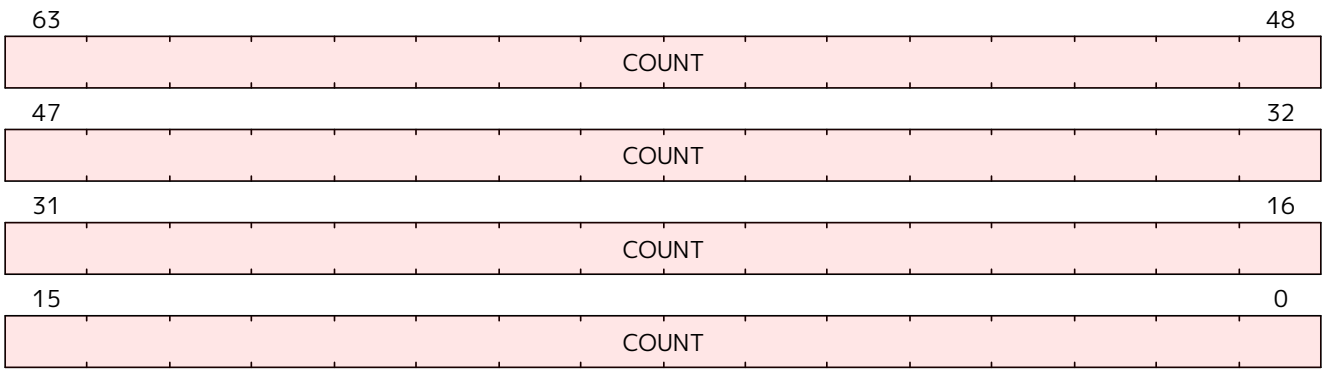


Figure 59. hpmcounter6 format

DRAFT

C.59. hpmcounter6h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter6h](#).

Privilege mode access is controlled with mcounteren.HPM6, scounteren.HPM6, and hcounteren.HPM6 as follows:

mcounteren. HPM6	scounteren. HPM6	hcounteren. HPM6	hpmcounter6h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.59.1. Attributes

CSR Address	0xc86
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.59.2. Format

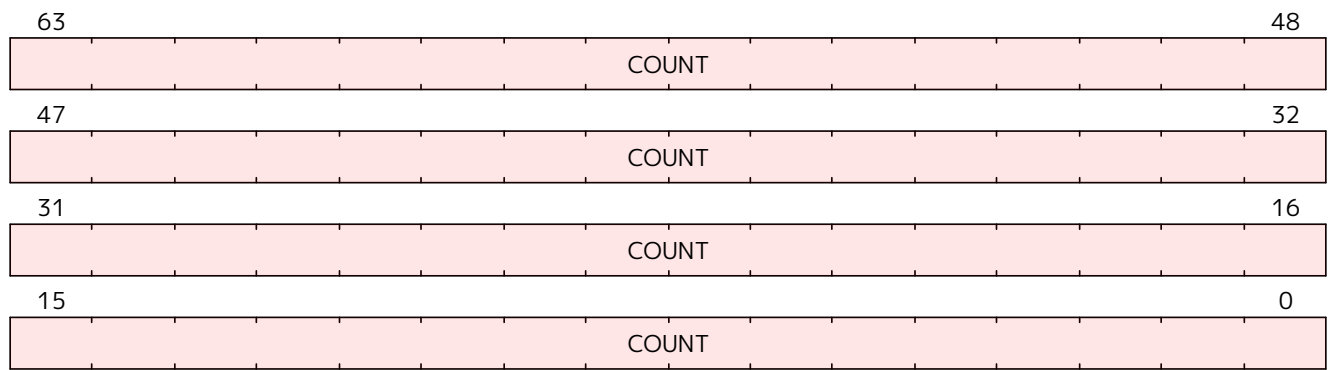


Figure 60. hpmcounter6h format

C.60. hpmcounter7

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter7](#).

Privilege mode access is controlled with mcounteren.HPM7 `<%- if ext?(:S) -%>` , scounteren.HPM7 `<%- if ext?(:H) -%>` , and hcounteren.HPM7 `<%- end -%>` `<%- end -%>` as follows:

`<%- if ext?(:H) -%>`

mcounteren.HPM7	scounteren.HPM7	hcounteren.HPM7	hpmcounter7 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

`<%- elsif ext?(:S) -%>`

mcounteren.HPM7	scounteren.HPM7	hpmcounter7 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

`<%- else -%>`

mcounteren.HPM7	hpmcounter7 behavior
	U-mode
0	IllegalInstruction
1	read-only

`<%- end -%>`

C.60.1. Attributes

CSR Address	0xc07
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.60.2. Format

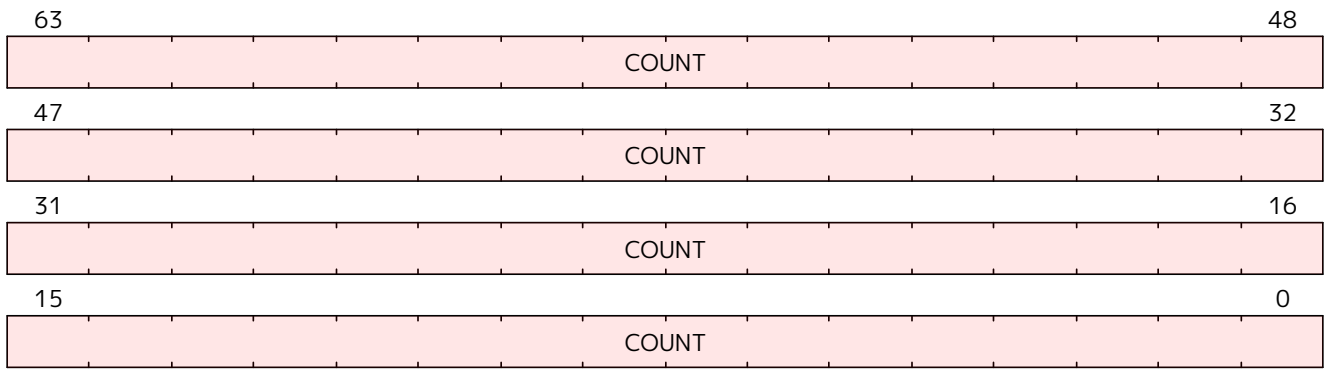


Figure 61. hpmcounter7 format

DRAFT

C.61. hpmcounter7h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter7h](#).

Privilege mode access is controlled with mcounteren.HPM7, scounteren.HPM7, and hcounteren.HPM7 as follows:

mcounteren. HPM7	scounteren. HPM7	hcounteren. HPM7	hpmcounter7h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.61.1. Attributes

CSR Address	0xc87
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.61.2. Format

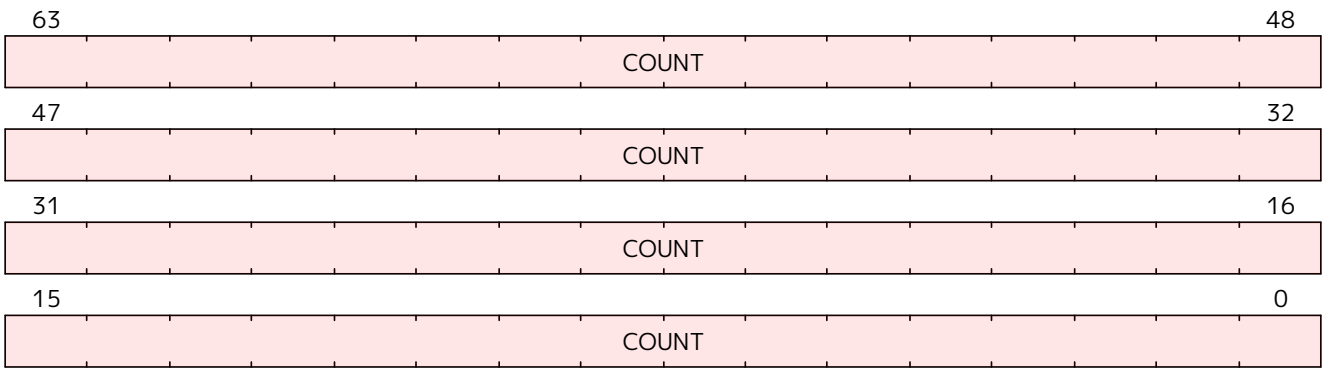


Figure 62. hpmcounter7h format

C.62. hpmcounter8

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter8](#).

Privilege mode access is controlled with mcounteren.HPM8 <%- if ext?(:S) -%> , scounteren.HPM8 <%- if ext?(:H) -%> , and hcounteren.HPM8 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren. HPM8	scounteren. HPM8	hcounteren. HPM8	hpmcounter8 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM8	scounteren.HPM8	hpmcounter8 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM8	hpmcounter8 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.62.1. Attributes

CSR Address	0xc08
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.62.2. Format

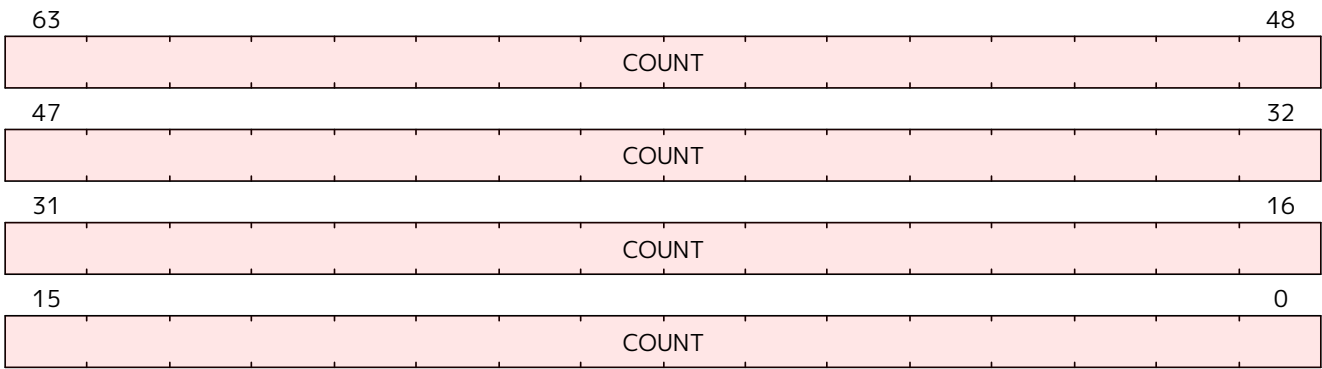


Figure 63. hpmcounter8 format

DRAFT

C.63. hpmcounter8h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter8h](#).

Privilege mode access is controlled with mcounteren.HPM8, scounteren.HPM8, and hcounteren.HPM8 as follows:

mcounteren. HPM8	scounteren. HPM8	hcounteren. HPM8	hpmcounter8h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.63.1. Attributes

CSR Address	0xc88
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.63.2. Format

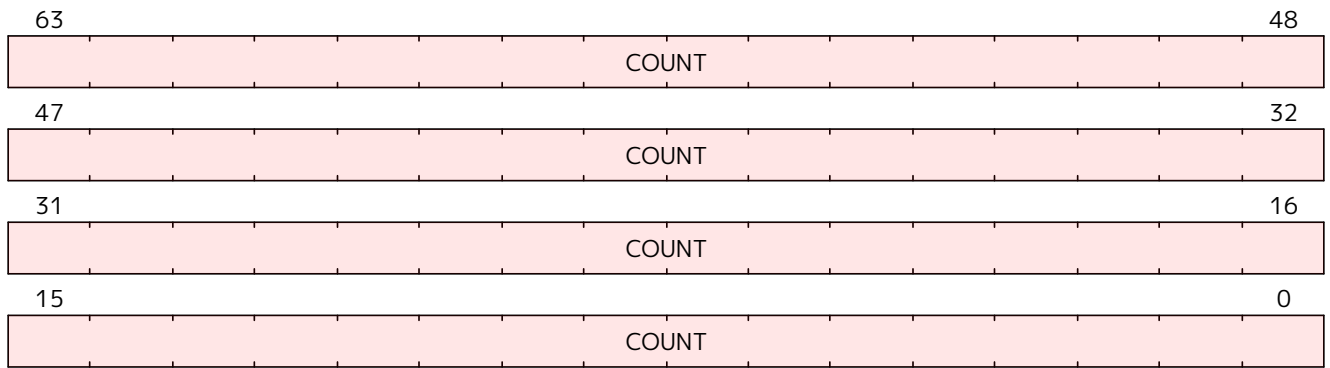


Figure 64. hpmcounter8h format

C.64. hpmcounter9

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mhpmcounter9](#).

Privilege mode access is controlled with mcounteren.HPM9 <%- if ext?(:S) -%> , scounteren.HPM9 <%- if ext?(:H) -%> , and hcounteren.HPM9 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM9	scounteren.HPM9	hcounteren.HPM9	hpmcounter9 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM9	scounteren.HPM9	hpmcounter9 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM9	hpmcounter9 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.64.1. Attributes

CSR Address	0xc09
Defining extension	Zihpm >= 0
Length	64-bit

Privilege Mode	U
----------------	---

C.64.2. Format

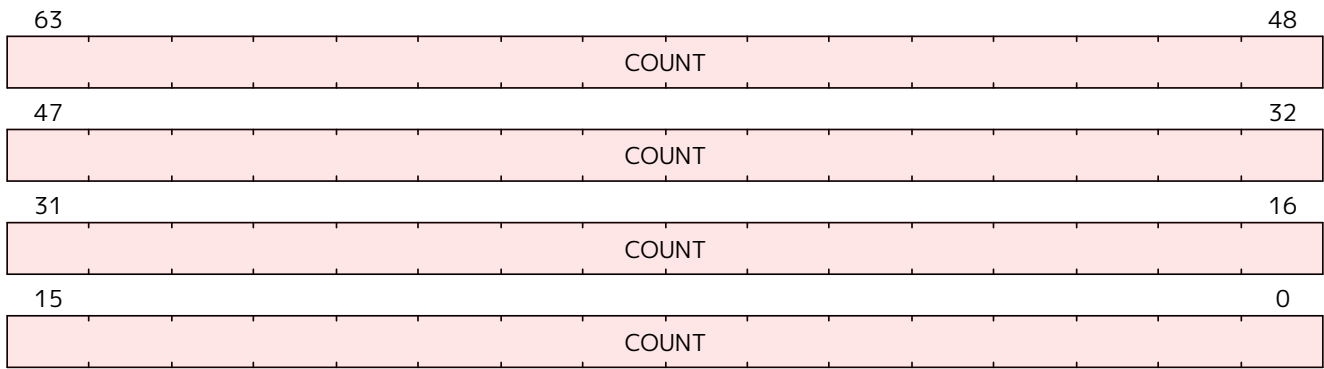


Figure 65. hpmcounter9 format

DRAFT

C.65. hpmcounter9h

Cycle counter for RDCYCLE Instruction, high half

Alias for M-mode CSR [mhpmcounter9h](#).

Privilege mode access is controlled with mcounteren.HPM9, scounteren.HPM9, and hcounteren.HPM9 as follows:

mcounteren. HPM9	scounteren. HPM9	hcounteren. HPM9	hpmcounter9h behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction	IllegalInst ruction
1	0	0	read-only	IllegalInst ruction	VirtualInst ruction	VirtualInst ruction
1	1	0	read-only	read-only	VirtualInst ruction	VirtualInst ruction
1	0	1	read-only	IllegalInst ruction	read-only	VirtualInst ruction
1	1	1	read-only	read-only	read-only	read-only

C.65.1. Attributes

CSR Address	0xc89
Defining extension	Sscofpmf >= 0
Length	64-bit
Privilege Mode	U

C.65.2. Format

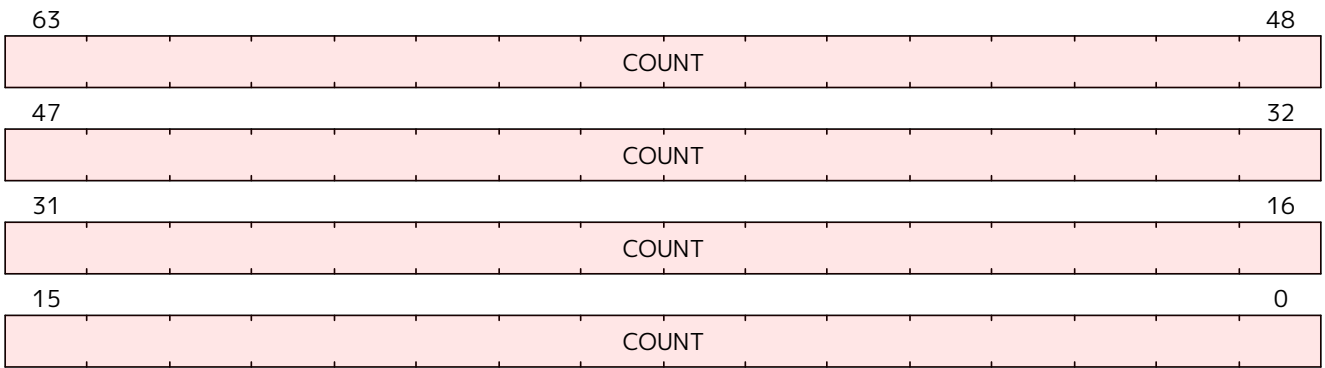


Figure 66. hpmcounter9h format

C.66. hstatus

Hypervisor Status

The hstatus register tracks and controls a VS-mode guest.

Unlike fields in `sstatus`, which are all aliases of fields `mstatus`, bits in `hstatus` are independent bits and do not have aliases.

C.66.1. Attributes

CSR Address	0x600
Defining extension	H >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.66.2. Format

This CSR format changes dynamically with XLEN.

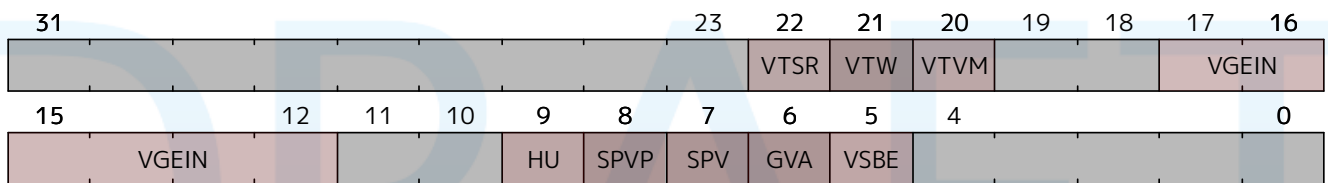


Figure 67. `hstatus` Format when `CSR[mstatus].SXL == 0`

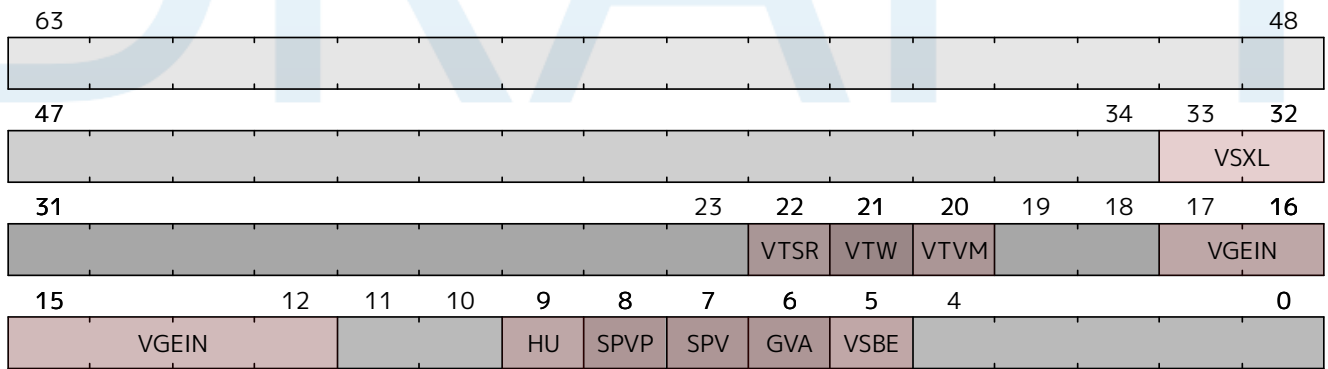


Figure 68. `hstatus` Format when `CSR[mstatus].SXL == 1`

C.67. htimedelta

Hypervisor time delta

The [htimedelta](#) CSR is a 64-bit read/write register that contains the delta between the value of the [time](#) CSR and the value returned in VS-mode or VU-mode. That is, reading the [time](#) CSR in VS or VU mode returns the sum of the contents of [htimedelta](#) and the actual value of [time](#).



Because overflow is ignored when summing [htimedelta](#) and [time](#), large values of [htimedelta](#) may be used to represent negative time offsets.

C.67.1. Attributes

CSR Address	0x605
Defining extension	H >= 0
Length	64-bit
Privilege Mode	S

C.67.2. Format

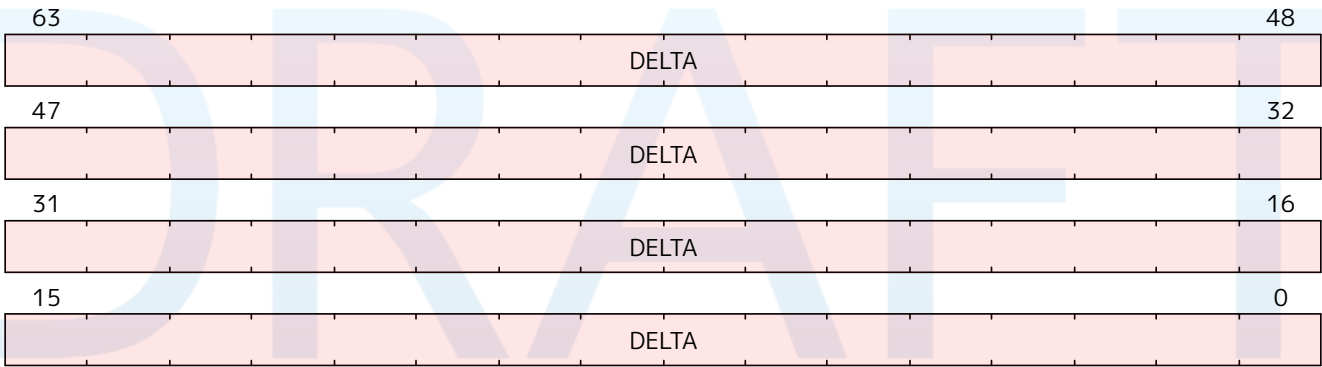


Figure 69. *htimedelta* format

C.68. htimedeltah

Hypervisor time delta, upper half

Upper half of the [htimedelta](#) CSR.

C.68.1. Attributes

CSR Address	0x615
Defining extension	H >= 0
Length	32-bit
Privilege Mode	S

C.68.2. Format

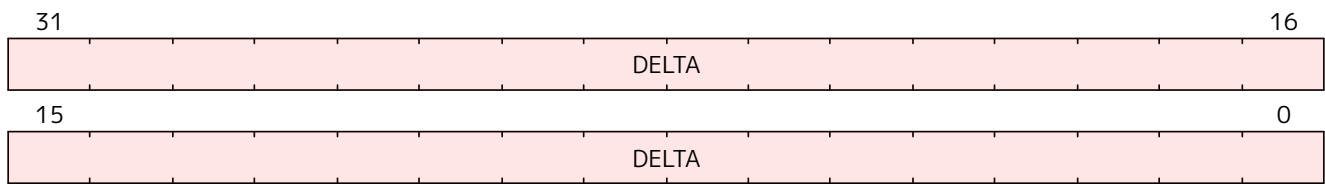


Figure 70. htimedeltah format

DRAFT

C.69. instret

Instructions retired counter for RDINSTRET Instruction

Alias for M-mode CSR [minstret](#).

Privilege mode access is controlled with mcounteren.IR, scounteren.IR, and hcounteren.IR as follows:

mcounteren.IR	scounteren.IR	hcounteren.IR	instret behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.69.1. Attributes

CSR Address	0xc02
Defining extension	Zicntr >= 0
Length	64-bit
Privilege Mode	U

C.69.2. Format

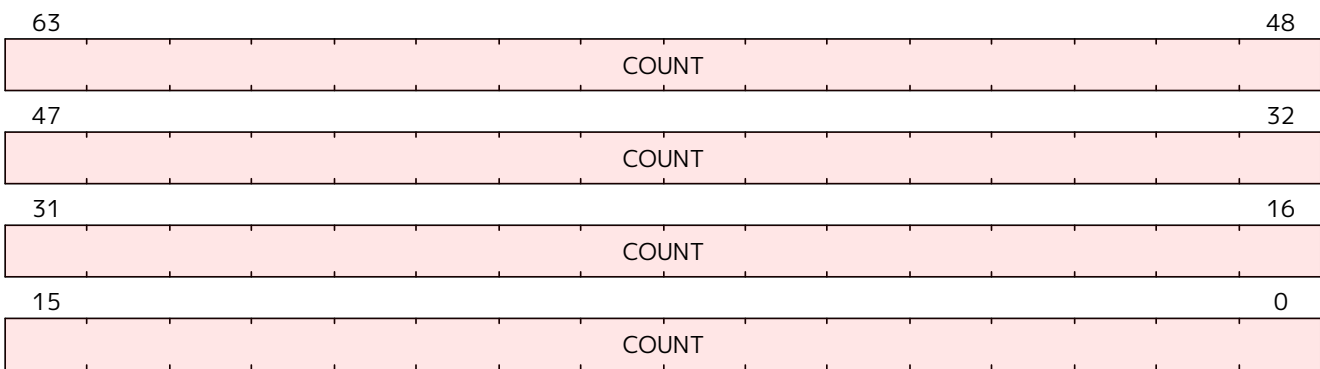


Figure 71. instret format

C.70. instreth

Instructions retired counter, high bits

i | *instreth* is only defined in RV32.

Alias for high bits of M-mode CSR *minstret*[63:32].

Privilege mode access is controlled with *mcouteren*.IR, *scouteren*.IR, and *hcouteren*.IR as follows:

mcouteren.IR	scouteren.IR	hcouteren.IR	instret behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.70.1. Attributes

CSR Address	0xc82
Defining extension	Zicntr >= 0
Length	32-bit
Privilege Mode	U

C.70.2. Format

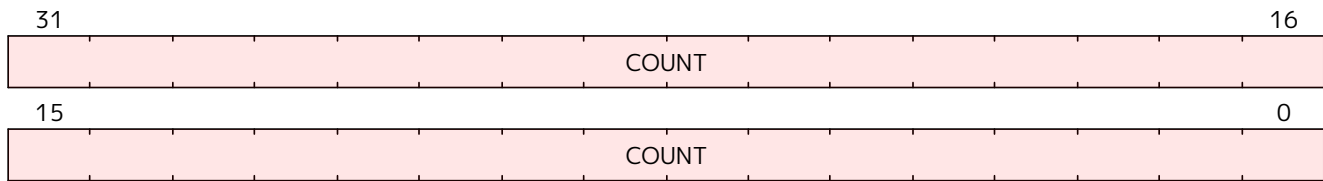


Figure 72. *instreth* format

C.71. marchid

Machine Architecture ID

The [marchid](#) CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of [mvendorid](#) and [marchid](#) should uniquely identify the type of hart microarchitecture that is implemented.

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.



The intent is for the architecture ID to represent the microarchitecture associated with the repo around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs are administered by RISC-V International and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The [misa](#) register also helps distinguish different variants of a design.

C.71.1. Attributes

CSR Address	0xf12
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.71.2. Format

This CSR format changes dynamically with XLEN.

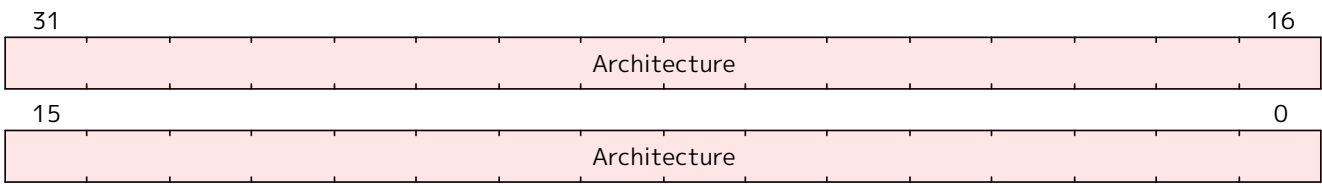


Figure 73. marchid Format when CSR[misa].MXL == 0

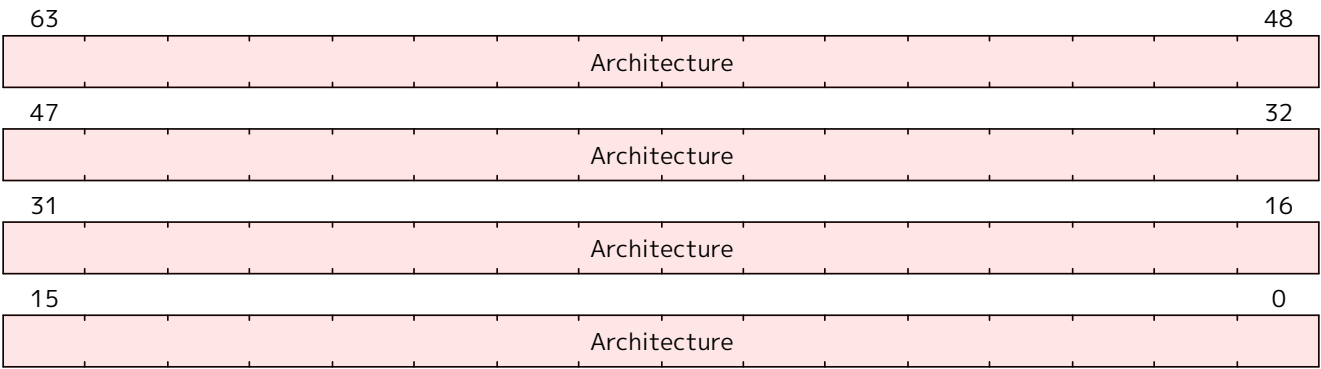


Figure 74. marchid Format when $CSR[misa].MXL == 1$

DRAFT

C.72. mcause

Machine Cause

Reports the cause of the latest exception.

C.72.1. Attributes

CSR Address	0x342
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.72.2. Format

This CSR format changes dynamically with XLEN.

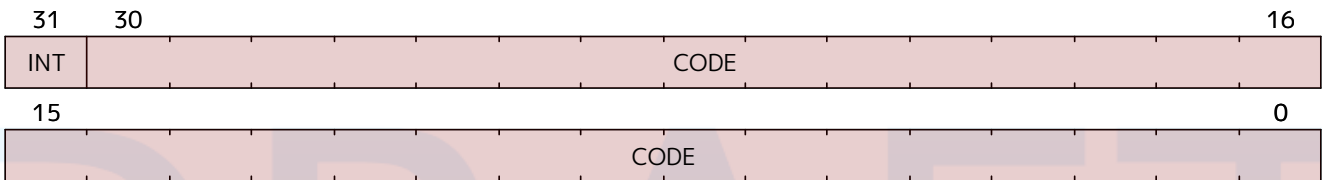


Figure 75. mcause Format when CSR[misa].MXL == 0

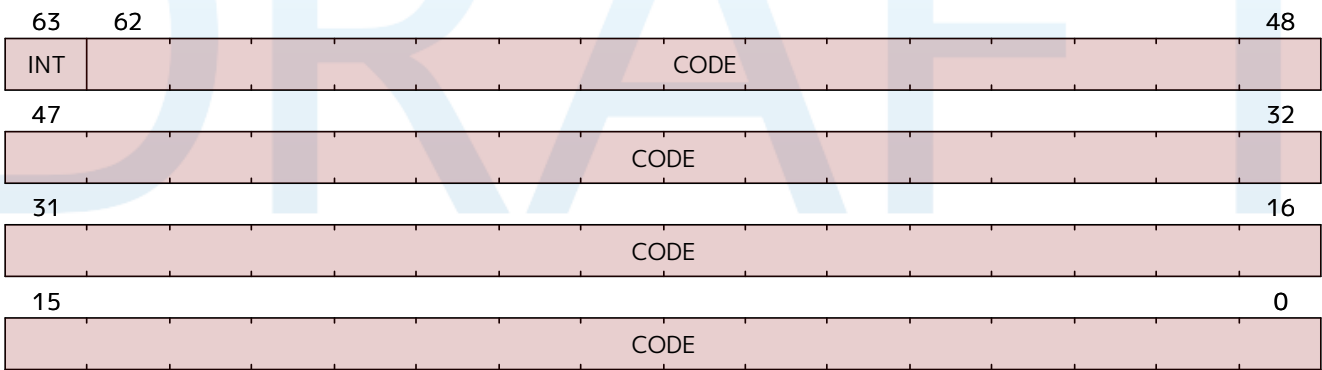


Figure 76. mcause Format when CSR[misa].MXL == 1

C.73. mconfigptr

Machine Configuration Pointer

Holds a physical address pointer to the unified discovery data structure in Memory.

The `mconfigptr` holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration.

The pointer alignment in bits must be no smaller than MXLEN: i.e., if MXLEN is $8 \times n$, then `mconfigptr[\log_2 n - 1:0]` must be zero.

The `mconfigptr` register must be implemented, but it may be zero to indicate the configuration data structure does not exist or that an alternative mechanism must be used to locate it.

The format and schema of the configuration data structure have yet to be standardized.



While the `mconfigptr` register will simply be hardwired in some implementations, other implementations may provide a means to configure the value returned on CSR reads. For example, `mconfigptr` might present the value of a memory-mapped register that is programmed by the platform or by M-mode software towards the beginning of the boot process.

C.73.1. Attributes

CSR Address	0xf15
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.73.2. Format

This CSR format changes dynamically with XLEN.

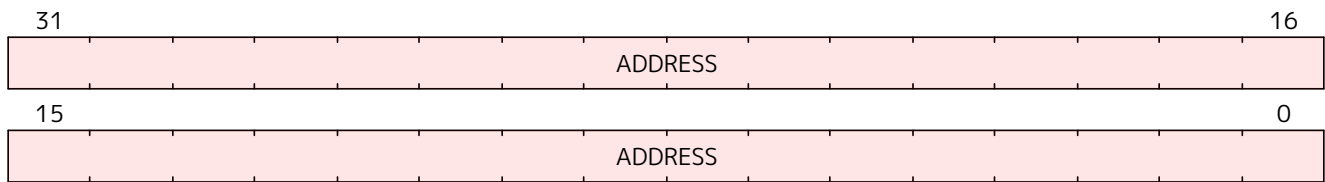


Figure 77. `mconfigptr` Format when `CSR[misa].MXL == 0`

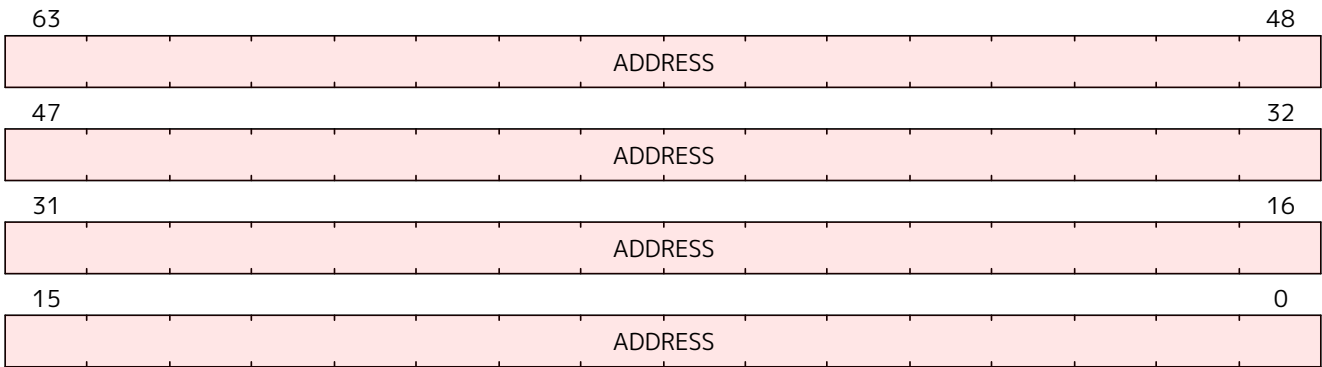


Figure 78. mconfigptr Format when CSR[misa].MXL == 1

DRAFT

C.74. mcounteren

Machine Counter Enable

The counter-enable `mcounteren` register is a 32-bit register that controls the availability of the hardware performance-monitoring counters to `<%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> the next-lower privileged mode <%- end -%> .`

The settings in this register only control accessibility. The act of reading or writing this register does not affect the underlying counters, which continue to increment even when not accessible.

When the CY, TM, IR, or HPMn bit in the `mcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcountern` register while executing in `<%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> S-mode or U-mode <%- end -%>` will cause an **IllegalInstruction** exception. When one of these bits is set, access to the corresponding register is permitted in `<%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> the next implemented privilege mode (S-mode if implemented, otherwise U-mode). <%- end -%>`



The counter-enable bits support two common use cases with minimal hardware. For harts that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For harts that need high-performance timers and counters but are not concerned with obfuscating the underlying hardware counters, the counters can be directly exposed to lower privilege modes.

The `cycle`, `instret`, and `hpmcountern` CSRs are read-only shadows of `mcycle`, `minstret`, and `mhpmcountern`, respectively. The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. `<%- if possible_xlens.include?(32) -%>` Analogously, on RV32I the `cycleh`, `instreth` and `hpmcounternh` CSRs are read-only shadows of `mcycleh`, `minstreth` and `mhpmcounternh`, respectively. On RV32I the `timeh` CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while time shadows only the lower 32 bits of `mtime`. `<%- end -%>`



Implementations can convert reads of the `time` and `timeh` CSRs into loads to the memory-mapped `mtime` register, or emulate this functionality on behalf of less-privileged modes in M-mode software.

`<%- if !ext?(:U) -%>` In harts with U-mode, the `mcounteren` CSR must be implemented, but all fields are WARL and may be read-only zero, indicating reads to the corresponding counter will cause an **IllegalInstruction** exception when executing in a less-privileged mode. In harts without U-mode, the `mcounteren` register should not exist. `<%- end -%>`

`<%- if ext?(:S) -%>`

The `cycle`, `instret`, and `hpmcountern` CSRs can also be made available to U-mode through the `scounteren` CSR `<%- if ext?(:H) -%>` and to VS-mode and/or VU-mode through `hcounteren` `<%- end -%> . <%- end -%>`

C.74.1. Attributes

CSR Address	0x306
Defining extension	U ≥ 0

Length	32-bit
Privilege Mode	M

C.74.2. Format

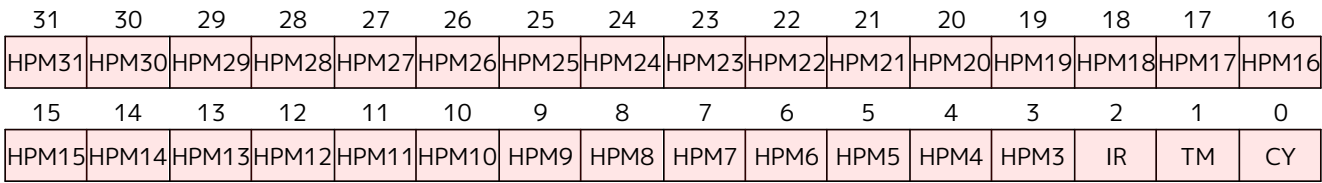


Figure 79. mcounteren format

DRAFT

C.75. mcountinhibit

Machine Counter Inhibit

Bits to inhibit (stops counting) performance counters.

The counter-inhibit register `mcountinhibit` is a WARL register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register.

When the CY, IR, or HPM_n bit in the `mcountinhibit` register is clear, the `mcycle`, `minstret`, or `mhpmcountern` register increments as usual. When the CY, IR, or HPM_n bit is set, the corresponding counter does not increment.

The `mcycle` CSR may be shared between harts on the same core, in which case the `mcountinhibit.CY` field is also shared between those harts, and so writes to `mcountinhibit.CY` will be visible to those harts.

If the `mcountinhibit` register is not implemented, the implementation behaves as though the register were set to zero.



When the `mcycle` and `minstret` counters are not needed, it is desirable to conditionally inhibit them to reduce energy consumption. Providing a single CSR to inhibit all counters also allows the counters to be atomically sampled.

Because the `mtime` counter can be shared between multiple cores, it cannot be inhibited with the `mcountinhibit` mechanism.

C.75.1. Attributes

CSR Address	0x320
Defining extension	Zicntr >= 0, Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.75.2. Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
HPM31	HPM30	HPM29	HPM28	HPM27	HPM26	HPM25	HPM24	HPM23	HPM22	HPM21	HPM20	HPM19	HPM18	HPM17	HPM16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HPM15	HPM14	HPM13	HPM12	HPM11	HPM10	HPM9	HPM8	HPM7	HPM6	HPM5	HPM4	HPM3	IR		CY

Figure 80. mcountinhibit format

C.76. mcycle

Machine Cycle Counter

Counts the number of clock cycles executed by the processor core on which the hart is running. The counter has 64-bit precision on all RV32 and RV64 harts.

The `mcycle` CSR may be shared between harts on the same core, in which case writes to `mcycle` will be visible to those harts. The platform should provide a mechanism to indicate which harts share an `mcycle` CSR.

C.76.1. Attributes

CSR Address	0xb00
Defining extension	Zicntr >= 0
Length	64-bit
Privilege Mode	M

C.76.2. Format

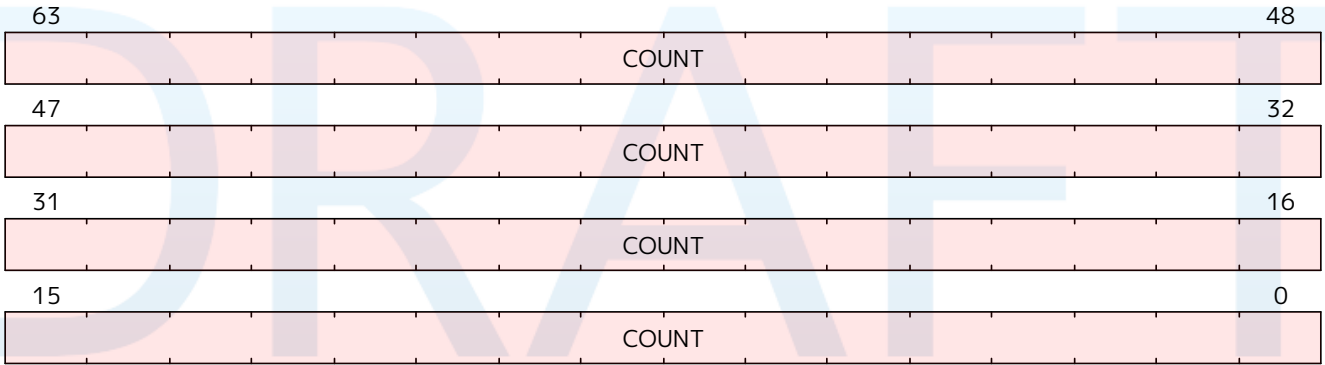


Figure 81. mcycle format

C.77. mcycleh

High-half machine Cycle Counter

i | *mcycleh* is only defined in RV32.

High-half alias of *mcycle*.

C.77.1. Attributes

CSR Address	0xb80
Defining extension	Zicntr >= 0
Length	32-bit
Privilege Mode	M

C.77.2. Format

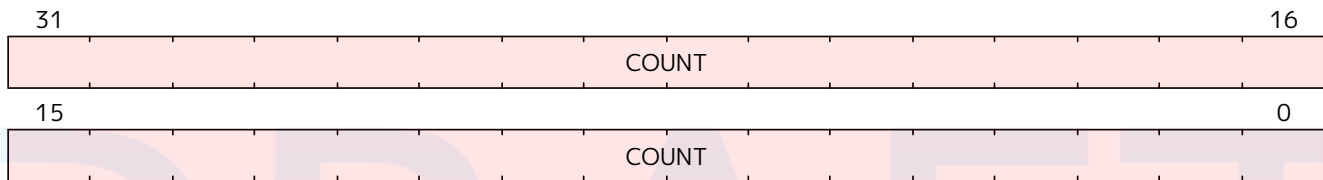


Figure 82. mcycleh format

C.78. medeleg

Machine Exception Delegation

Controls exception delegation from M-mode to (H)S-mode <%- if ext?(:H) -%> or, in conjunction with [hedeleg](#), to VS-mode <%- end -%> .

An exception cause is delegated to (H)S-mode when all of the following hold:

- The corresponding field in [medeleg](#) is set.
- The current privilege level is not M-mode. <%- if ext?(:H) -%>
- The same field in [hedeleg](#) is clear. <%- end -%>

<%- if ext?(:H) -%> An exception cause is delegated to VS-mode when all of the following hold:

- The corresponding field in [medeleg](#) is set.
- The corresponding field in [hedeleg](#) is set.
- The current privilege level is not M-mode or HS-mode. <%- end -%>

Otherwise, an exception cause is handled by M-mode.

See [interrupt documentation](#) for more details.

C.78.1. Attributes

CSR Address	0x302
Defining extension	S >= 0
Length	64-bit
Privilege Mode	M

C.78.2. Format

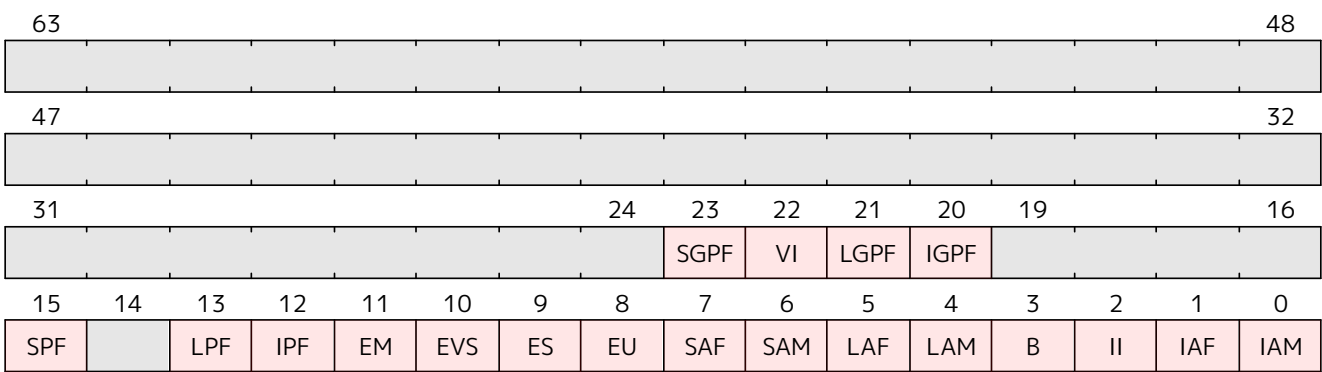



Figure 83. medeleg format

C.79. medelegh

Machine Exception Delegation, High bits

 *medelegh* is only defined in RV32.

Alias of the upper 32 bits of [medeleg](#).

C.79.1. Attributes

CSR Address	0x312
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.79.2. Format

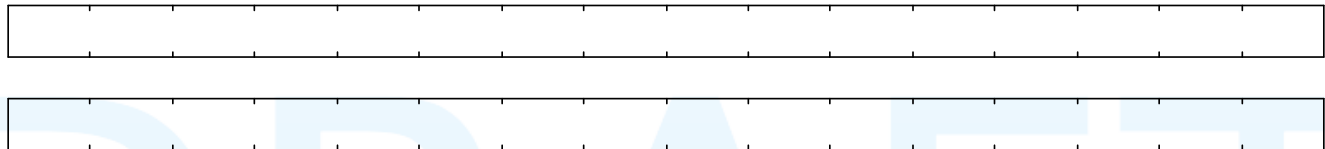


Figure 84. *medelegh* format

C.80. menvcfg

Machine Environment Configuration

Contains bits to enable/disable extensions

The [menvcfg](#) CSR controls certain characteristics of the execution environment for modes less privileged than M.

If bit FIOM (Fence of I/O implies Memory) is set to one in [menvcfg](#), FENCE instructions executed in modes less privileged than M are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. [Table 55](#) details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO for modes less privileged than M when FIOM=1.

Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

If S-mode is not supported, or if [satp](#).MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 55. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1.

Instruction bit Meaning when set	PI PO
Predecessor device input and memory reads (PR implied)	SI SO
Predecessor device output and memory writes (PW implied)	



Bit FIOM is needed in [menvcfg](#) so M-mode can emulate the H (hypervisor) extension, which has an equivalent FIOM bit in the hypervisor CSR [henvcfg](#).

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode and G-stage address translation (i.e., for page tables pointed to by [satp](#) or [hgatp](#)). When PBMTE=1, Svpbmt is available for S-mode and G-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero. Furthermore, for implementations with the hypervisor extension, [henvcfg](#).PBMTE is read-only zero if [menvcfg](#).PBMTE is zero.

After changing [menvcfg](#).PBMTE, executing an SFENCE.VMA instruction with *rs1*=**x0** and *rs2*=**x0** suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields. See [\[hyp-mm-fences\]](#) for additional synchronization requirements when the hypervisor extension is implemented.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for S-mode and G-stage address translations. When ADUE=1, hardware updating of PTE A/D bits is enabled during S-mode address translation, and the implementation behaves as though the Svade extension were not implemented for S-mode address translation. When the hypervisor extension is implemented, if ADUE=1, hardware updating of PTE A/D bits is enabled during G-stage address translation, and the implementation behaves as though the Svade extension were not implemented for G-stage address translation. When ADUE=0, the implementation behaves as though Svade were implemented for S-mode and G-stage address translation. If Svadu is not implemented,

ADUE is read-only zero. Furthermore, for implementations with the hypervisor extension, **menvcfg**.ADUE is read-only zero if **menvcfg**.ADUE is zero.



The Svade extension requires page-fault exceptions be raised when PTE A/D bits need be set, hence Svade is implemented when ADUE=0.

If the Smcdeleg extension is implemented, the CDE (Counter Delegation Enable) bit controls whether Zicntr and Zihpm counters can be delegated to S-mode. When CDE=1, the Smcdeleg extension is enabled, see [smcdeleg]. When CDE=0, the Smcdeleg and Ssccfg extensions appear to be not implemented. If Smcdeleg is not implemented, CDE is read-only zero.

The definition of the STCE field is furnished by the Sstc extension.

The definition of the CBZE field is furnished by the Zicboz extension.

The definitions of the CBCFE and CBIE fields are furnished by the Zicbom extension.

The definition of the PMM field will be furnished by the forthcoming Smnprn extension. Its allocation within **menvcfg** may change prior to the ratification of that extension.

The Zicfilp extension adds the **LPE** field in **menvcfg**. When the **LPE** field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If **LPE** field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode. When the **LPE** field is 0, the Zicfilp extension is not enabled in S-mode, and the following rules apply to S-mode. If the **LPE** field is 0 and S-mode is not implemented, then the same rules apply to U-mode.

- The hart does not update the **ELP** state; it remains as **NO_LP_EXPECTED**.
- The **LPAD** instruction operates as a no-op.

The Zicfiss extension adds the **SSE** field to **menvcfg**. When the **SSE** field is set to 1 the Zicfiss extension is activated in S-mode. When **SSE** field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The **pte.xwr=010b** encoding in VS/S-stage page tables becomes reserved.
- The **menvcfg**.**SSE** and **senvcfg** fields will read as zero and are read-only.
- **SSAMOSWAP.W/D** raises an illegal-instruction exception.

The Ssdbltrp extension adds the double-trap-enable (**DTE**) field in **menvcfg**. When **menvcfg** is zero, the implementation behaves as though Ssdbltrp is not implemented. When Ssdbltrp is not implemented **sstatus**, **vsstatus**, and **menvcfg**.**DTE** bits are read-only zero.

When XLEN=32, **menvcfgh** is a 32-bit read/write register that aliases bits 63:32 of **menvcfg**. The **menvcfgh** register does not exist when XLEN=64.

If U-mode is not supported, then registers **menvcfg** and **menvcfgh** do not exist.

C.80.1. Attributes

CSR Address	0x30a
Defining extension	U >= 0
Length	64-bit
Privilege Mode	M

C.80.2. Format

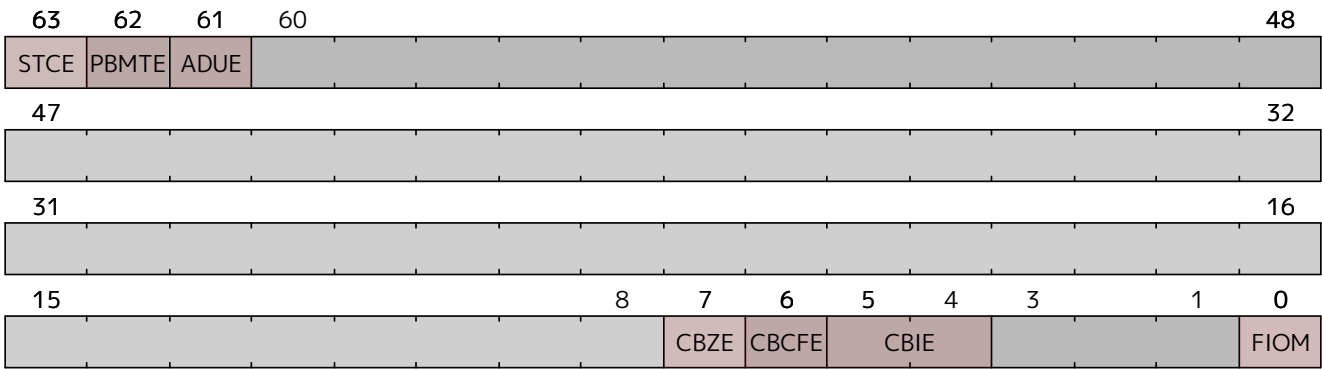


Figure 85. menvcfg format

DRAFT

C.81. menvcfgh

Machine Environment Configuration

Contains bits to enable/disable extensions

C.81.1. Attributes

CSR Address	0x31a
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.81.2. Format



Figure 86. menvcfgh format

C.82. mepc

Machine Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in M-mode.

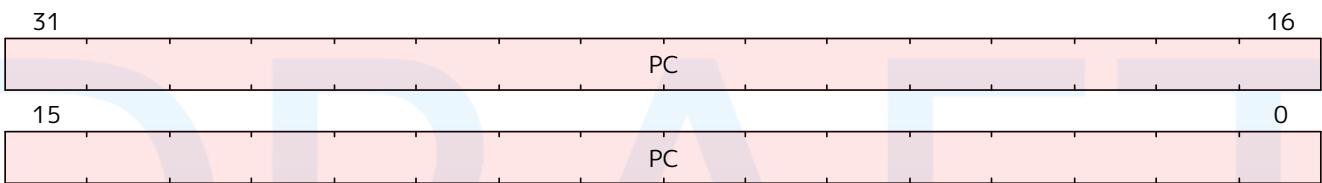
Also controls where the hart jumps on an exception return from M-mode.

C.82.1. Attributes

CSR Address	0x341
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.82.2. Format

This CSR format changes dynamically with XLEN.



C.83. mhartid

Machine Hart ID

Reports the unique hart-specific ID in the system.

C.83.1. Attributes

CSR Address	0xf14
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.83.2. Format

This CSR format changes dynamically with XLEN.

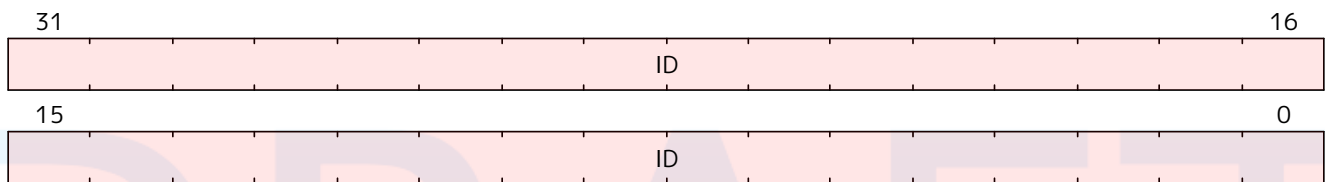


Figure 89. mhartid Format when CSR[misa].MXL == 0

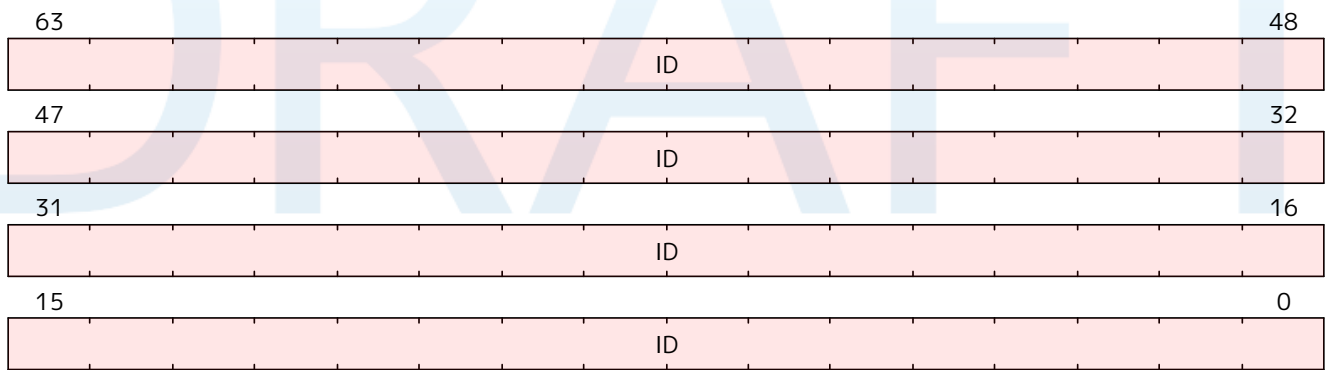


Figure 90. mhartid Format when CSR[misa].MXL == 1

C.84. mhpmcounter10

Machine Hardware Performance Counter 10

Programmable hardware performance counter.

C.84.1. Attributes

CSR Address	0xb0a
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.84.2. Format

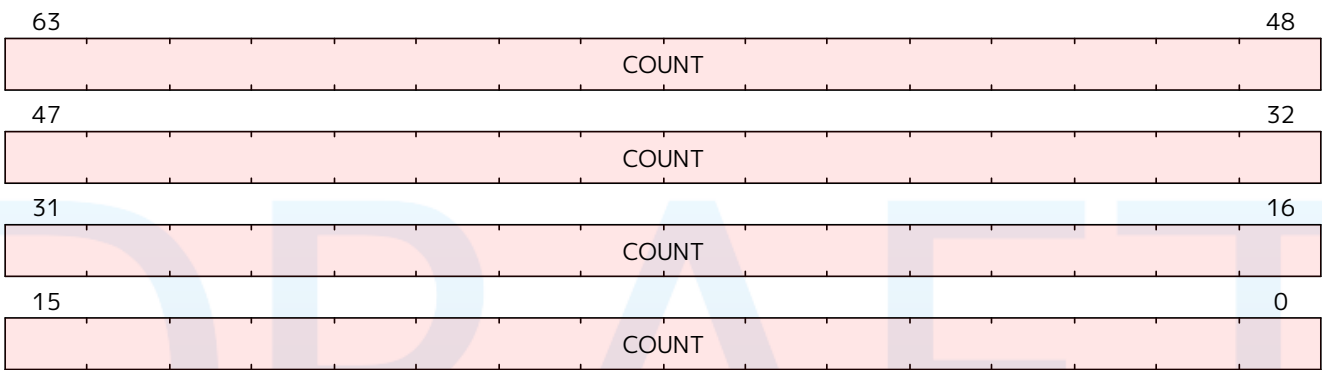


Figure 91. mhpmcounter10 format

C.85. mhpmpcounter10h

Machine Hardware Performance Counter 10, Upper half

 *mhpmpcounter10h* is only defined in RV32.

Upper half of mhpmpcounter10.

C.85.1. Attributes

CSR Address	0xb8a
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.85.2. Format

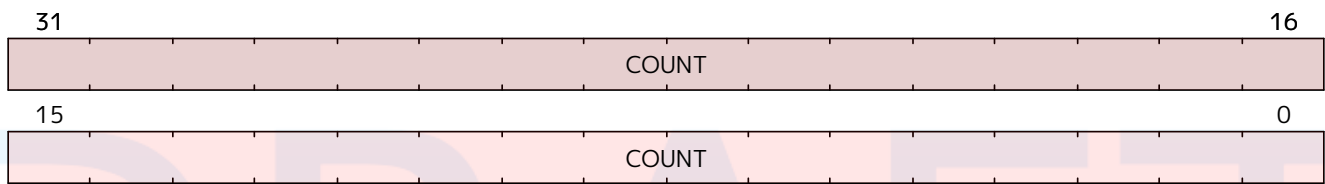


Figure 92. mhpmpcounter10h format

C.86. mhpmcounter11

Machine Hardware Performance Counter 11

Programmable hardware performance counter.

C.86.1. Attributes

CSR Address	0xb0b
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.86.2. Format

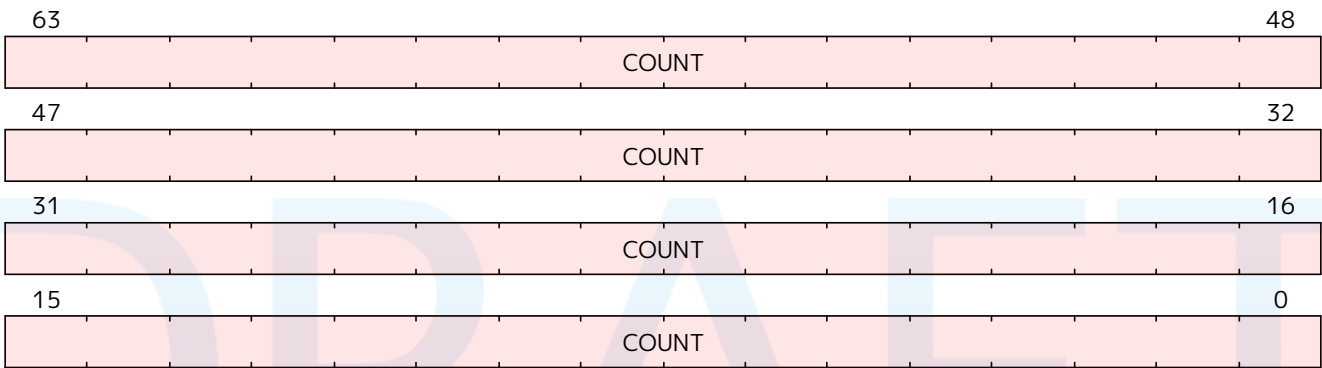


Figure 93. mhpmcounter11 format

C.87. mhpmcounter11h

Machine Hardware Performance Counter 11, Upper half

 *mhpmcounter11h* is only defined in RV32.

Upper half of mhpmcounter11.

C.87.1. Attributes

CSR Address	0xb8b
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.87.2. Format

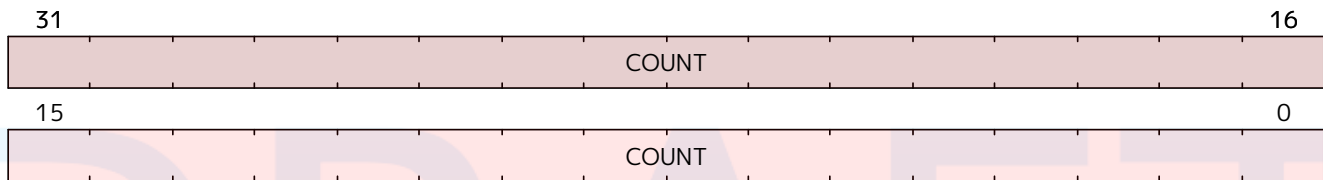


Figure 94. mhpmcounter11h format

C.88. mhpmcounter12

Machine Hardware Performance Counter 12

Programmable hardware performance counter.

C.88.1. Attributes

CSR Address	0xb0c
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.88.2. Format

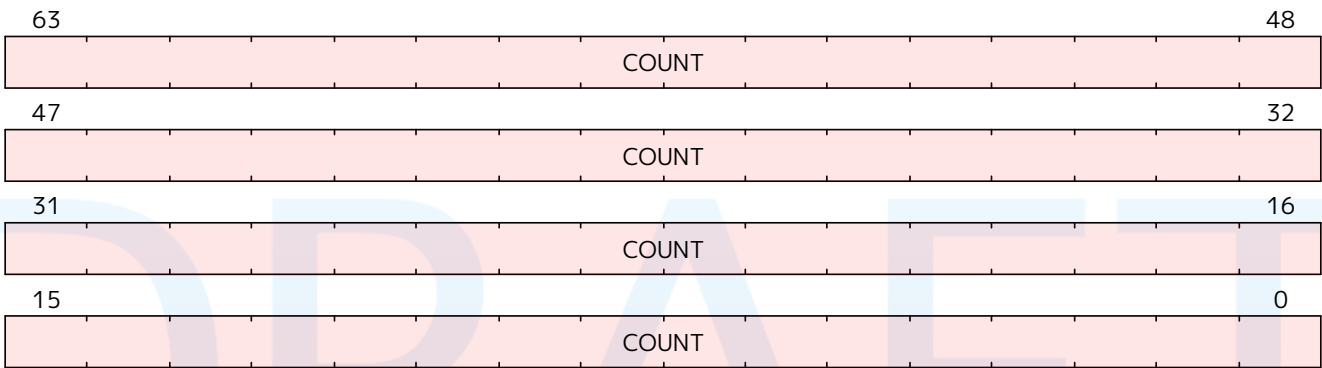


Figure 95. mhpmcounter12 format

C.89. mhpcounter12h

Machine Hardware Performance Counter 12, Upper half

 *mhpcounter12h* is only defined in RV32.

Upper half of mhpcounter12.

C.89.1. Attributes

CSR Address	0xb8c
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.89.2. Format

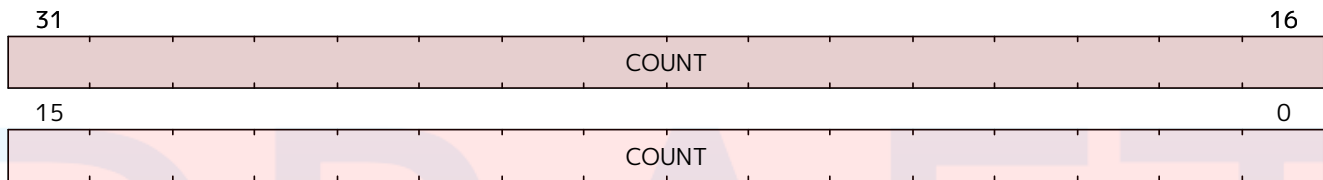


Figure 96. mhpcounter12h format

C.90. mhpmcounter13

Machine Hardware Performance Counter 13

Programmable hardware performance counter.

C.90.1. Attributes

CSR Address	0xb0d
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.90.2. Format

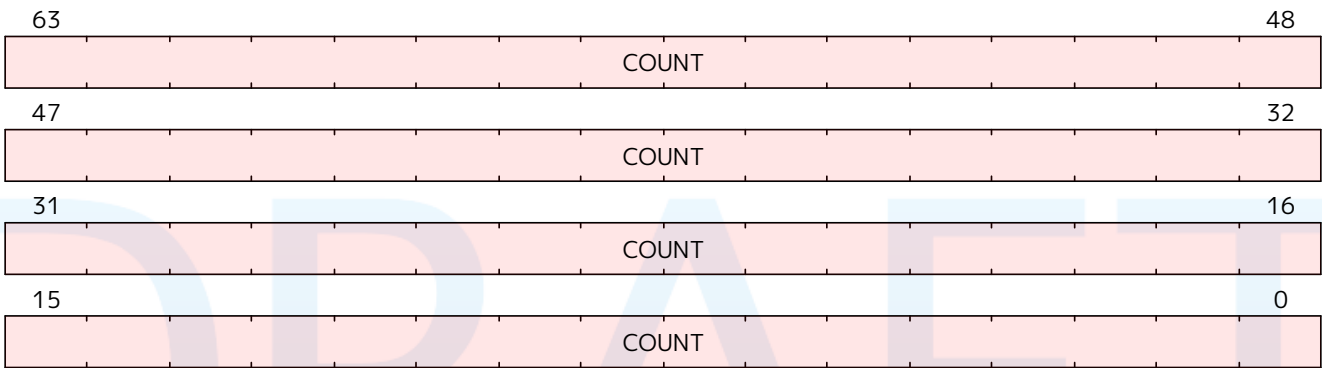


Figure 97. mhpmcounter13 format

C.91. mhpmcounter13h

Machine Hardware Performance Counter 13, Upper half

 *mhpmcounter13h* is only defined in RV32.

Upper half of mhpmcounter13.

C.91.1. Attributes

CSR Address	0xb8d
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.91.2. Format

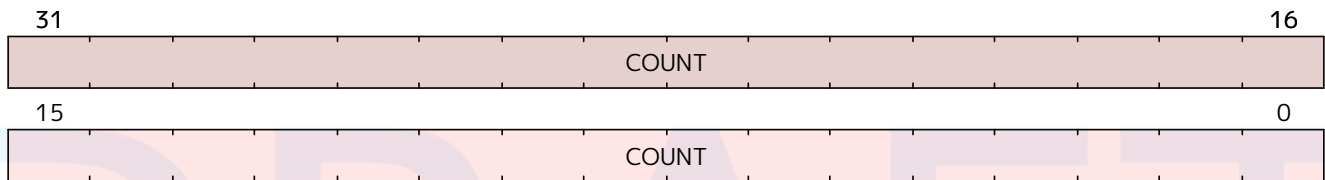


Figure 98. mhpmcounter13h format

C.92. mhpmcounter14

Machine Hardware Performance Counter 14

Programmable hardware performance counter.

C.92.1. Attributes

CSR Address	0xb0e
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.92.2. Format

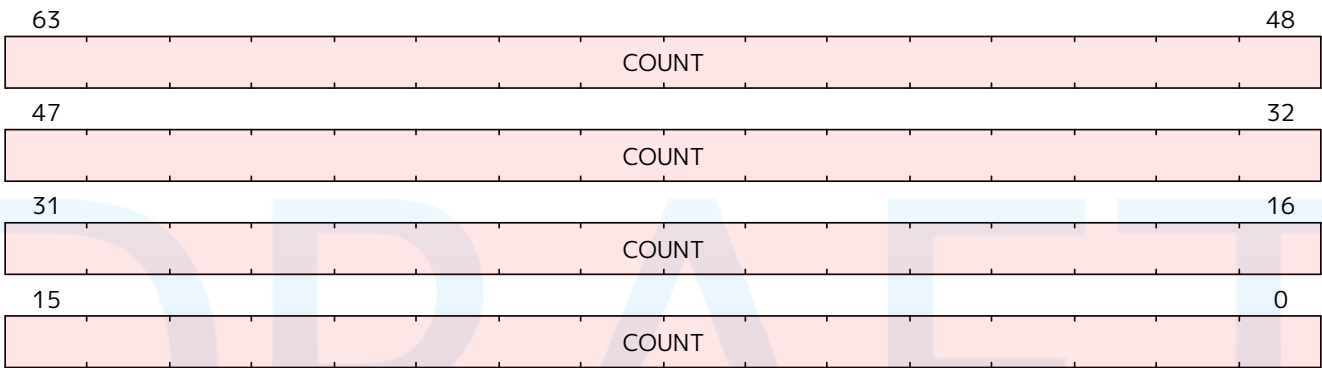


Figure 99. mhpmcounter14 format

C.93. mhpcounter14h

Machine Hardware Performance Counter 14, Upper half

 *mhpcounter14h* is only defined in RV32.

Upper half of mhpcounter14.

C.93.1. Attributes

CSR Address	0xb8e
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.93.2. Format

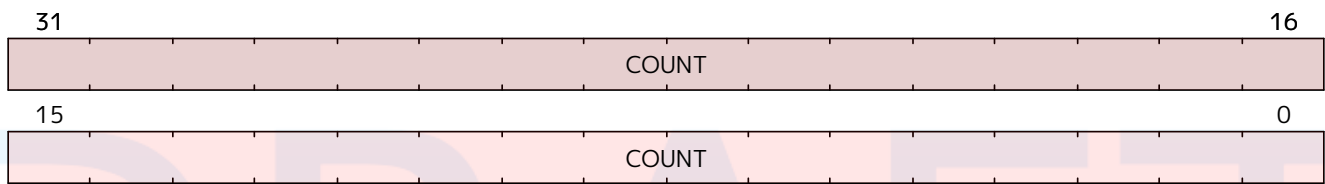


Figure 100. mhpcounter14h format

C.94. mhpmcounter15

Machine Hardware Performance Counter 15

Programmable hardware performance counter.

C.94.1. Attributes

CSR Address	0xb0f
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.94.2. Format

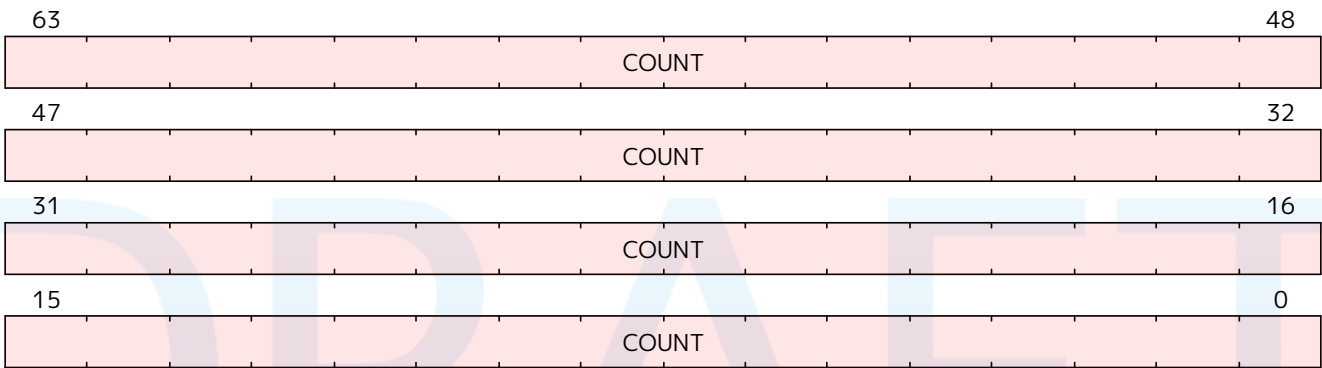


Figure 101. mhpmcounter15 format

C.95. mhpmcounter15h

Machine Hardware Performance Counter 15, Upper half

 *mhpmcounter15h* is only defined in RV32.

Upper half of mhpmcounter15.

C.95.1. Attributes

CSR Address	0xb8f
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.95.2. Format

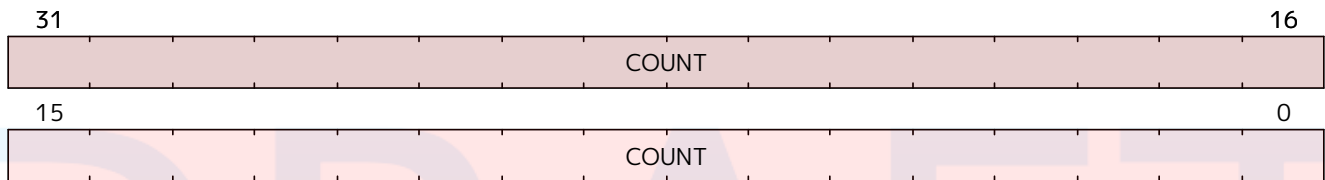


Figure 102. mhpmcounter15h format

C.96. mhpmcounter16

Machine Hardware Performance Counter 16

Programmable hardware performance counter.

C.96.1. Attributes

CSR Address	0xb10
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.96.2. Format

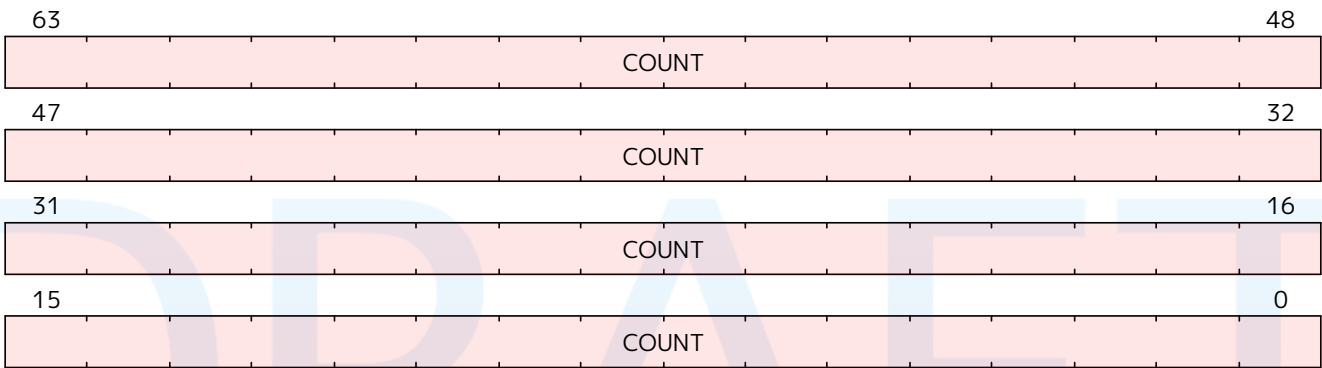


Figure 103. mhpmcounter16 format

C.97. mhpmcounter16h

Machine Hardware Performance Counter 16, Upper half

 *mhpmcounter16h* is only defined in RV32.

Upper half of mhpmcounter16.

C.97.1. Attributes

CSR Address	0xb90
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.97.2. Format

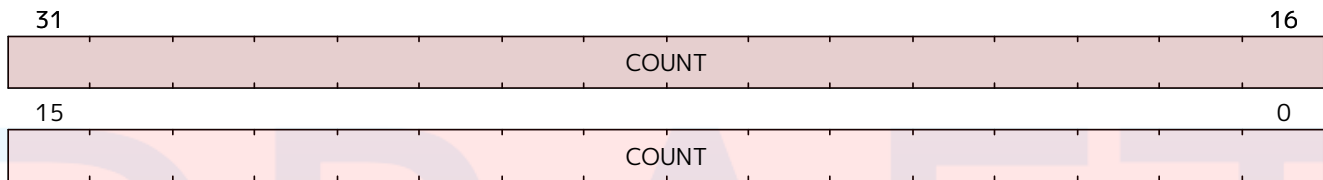


Figure 104. mhpmcounter16h format

C.98. mhpmcounter17

Machine Hardware Performance Counter 17

Programmable hardware performance counter.

C.98.1. Attributes

CSR Address	0xb11
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.98.2. Format

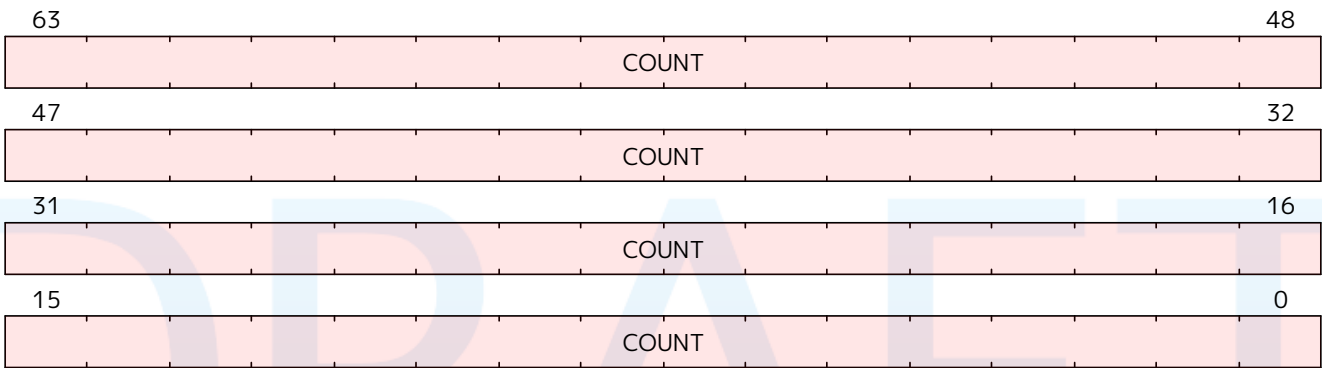


Figure 105. mhpmcounter17 format

C.99. mhpmcounter17h

Machine Hardware Performance Counter 17, Upper half

 *mhpmcounter17h* is only defined in RV32.

Upper half of mhpmcounter17.

C.99.1. Attributes

CSR Address	0xb91
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.99.2. Format

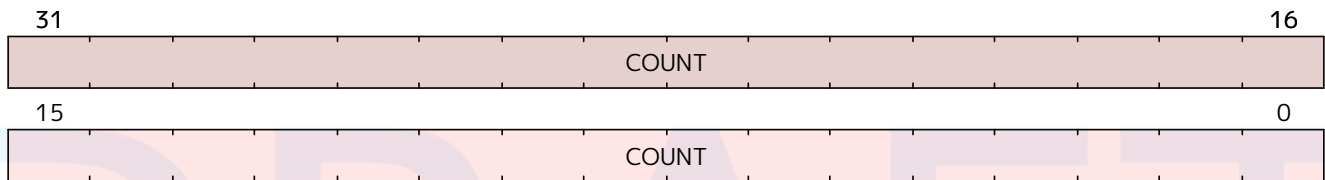


Figure 106. mhpmcounter17h format

C.100. mhpmcounter18

Machine Hardware Performance Counter 18

Programmable hardware performance counter.

C.100.1. Attributes

CSR Address	0xb12
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.100.2. Format

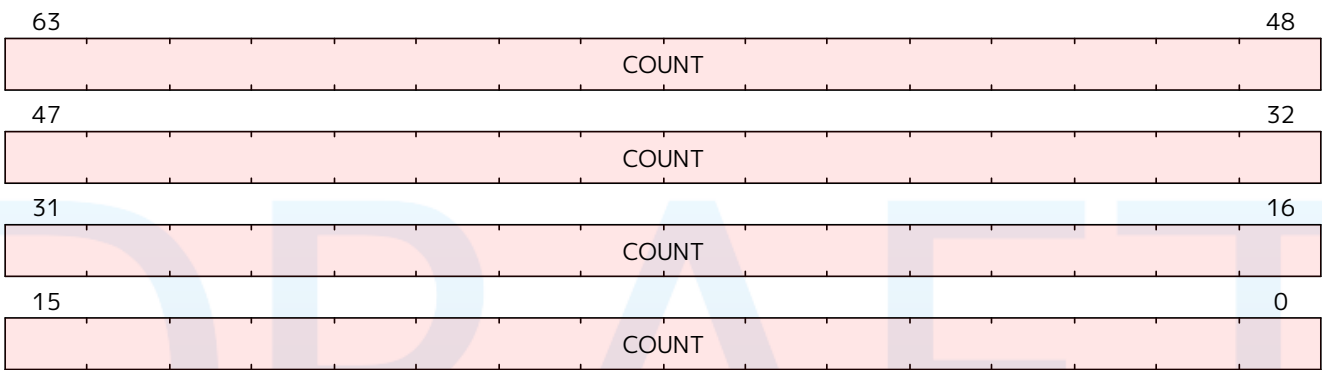


Figure 107. mhpmcounter18 format

C.101. mhpmcounter18h

Machine Hardware Performance Counter 18, Upper half

 *mhpmcounter18h* is only defined in RV32.

Upper half of mhpmcounter18.

C.101.1. Attributes

CSR Address	0xb92
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.101.2. Format

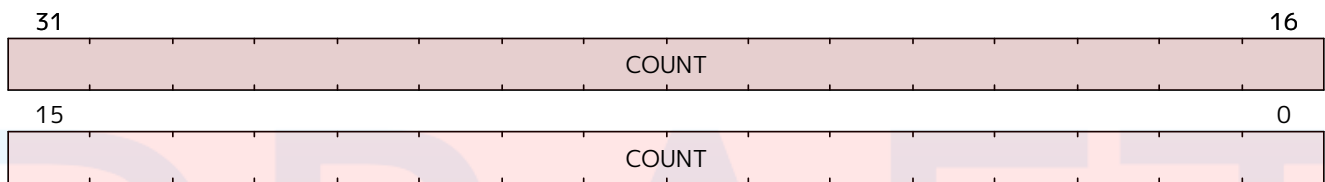


Figure 108. mhpmcounter18h format

C.102. mhpmcounter19

Machine Hardware Performance Counter 19

Programmable hardware performance counter.

C.102.1. Attributes

CSR Address	0xb13
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.102.2. Format

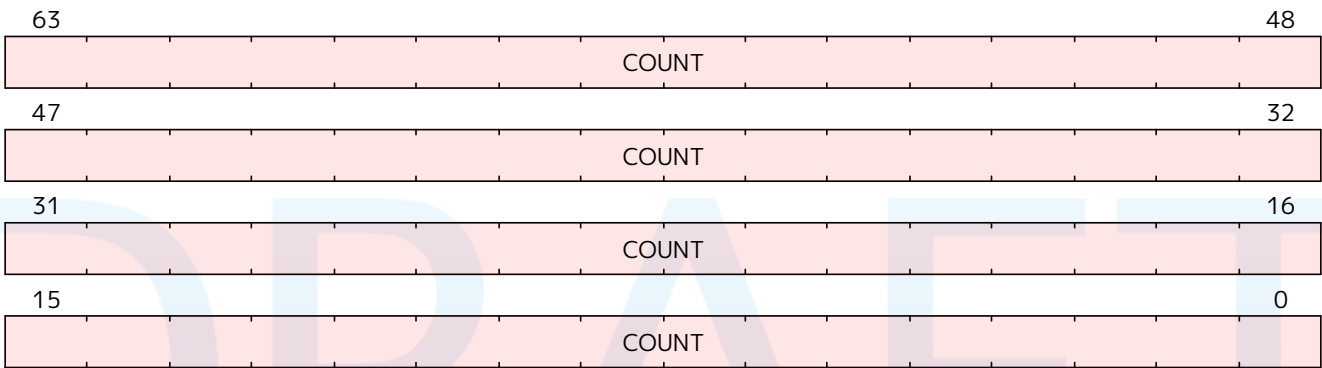


Figure 109. mhpmcounter19 format

C.103. mhpmcounter19h

Machine Hardware Performance Counter 19, Upper half



mhpmcounter19h is only defined in RV32.

Upper half of mhpmcounter19.

C.103.1. Attributes

CSR Address	0xb93
Defining extension	Zihpm ≥ 0
Length	32-bit
Privilege Mode	M

C.103.2. Format

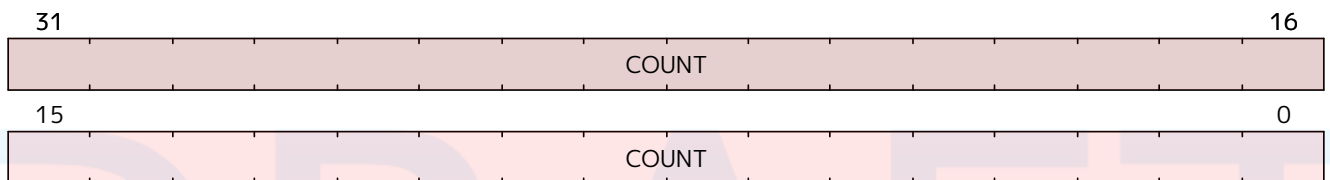


Figure 110. mhpmcounter19h format

C.104. mhpmcounter20

Machine Hardware Performance Counter 20

Programmable hardware performance counter.

C.104.1. Attributes

CSR Address	0xb14
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.104.2. Format

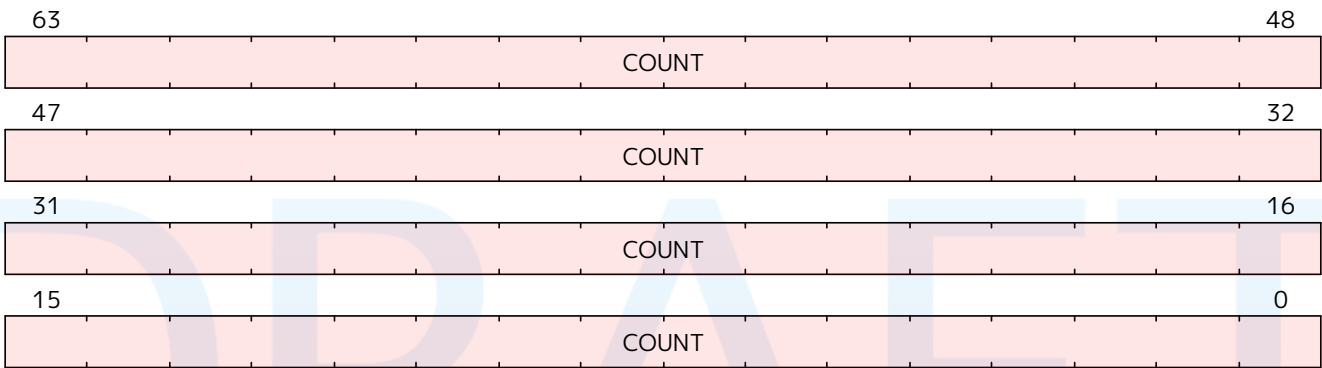


Figure 111. mhpmcounter20 format

C.105. mhpmcounter20h

Machine Hardware Performance Counter 20, Upper half

i | *mhpmcounter20h* is only defined in RV32.

Upper half of mhpmcounter20.

C.105.1. Attributes

CSR Address	0xb94
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.105.2. Format

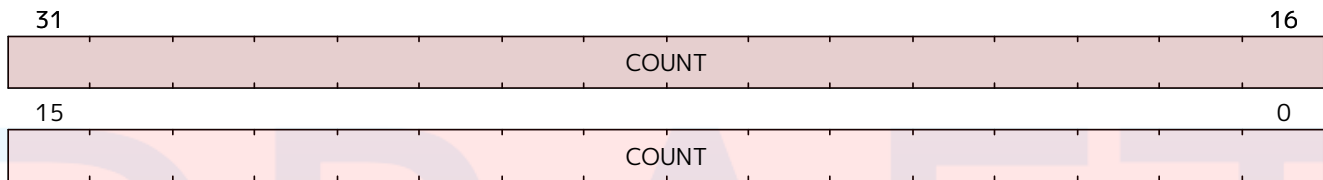


Figure 112. mhpmcounter20h format

C.106. mhpmcounter21

Machine Hardware Performance Counter 21

Programmable hardware performance counter.

C.106.1. Attributes

CSR Address	0xb15
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.106.2. Format

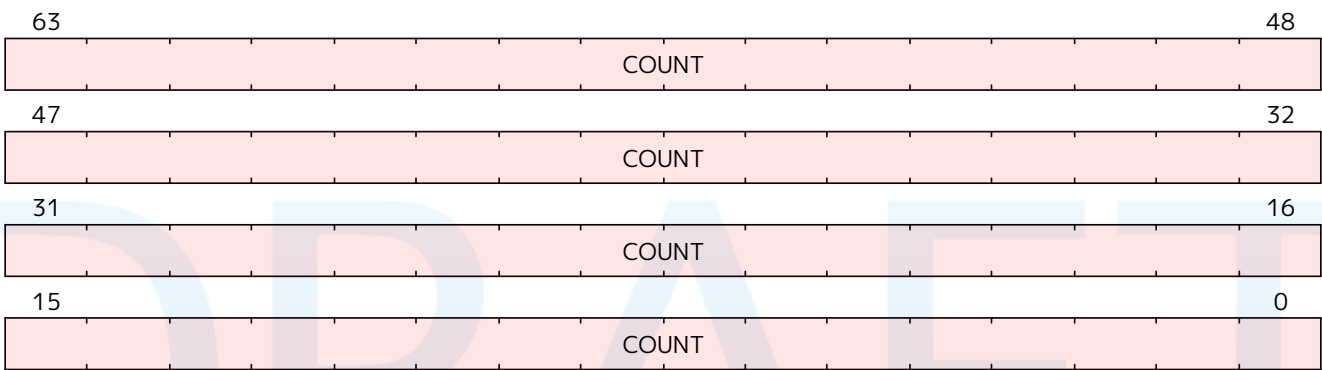


Figure 113. mhpmcounter21 format

C.107. mhpmcounter21h

Machine Hardware Performance Counter 21, Upper half



mhpmcounter21h is only defined in RV32.

Upper half of mhpmcounter21.

C.107.1. Attributes

CSR Address	0xb95
Defining extension	Zihpm \geq 0
Length	32-bit
Privilege Mode	M

C.107.2. Format

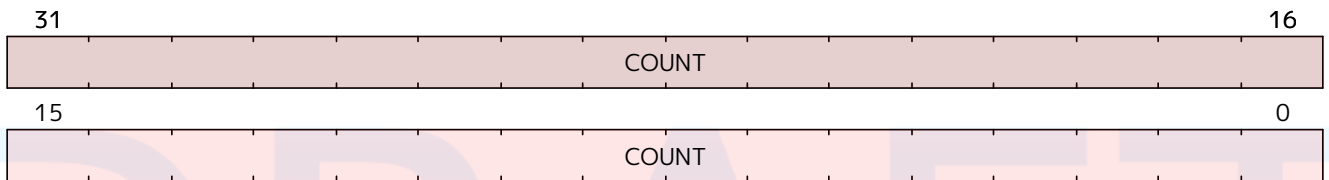


Figure 114. mhpmcounter21h format

C.108. mhpmcounter22

Machine Hardware Performance Counter 22

Programmable hardware performance counter.

C.108.1. Attributes

CSR Address	0xb16
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.108.2. Format

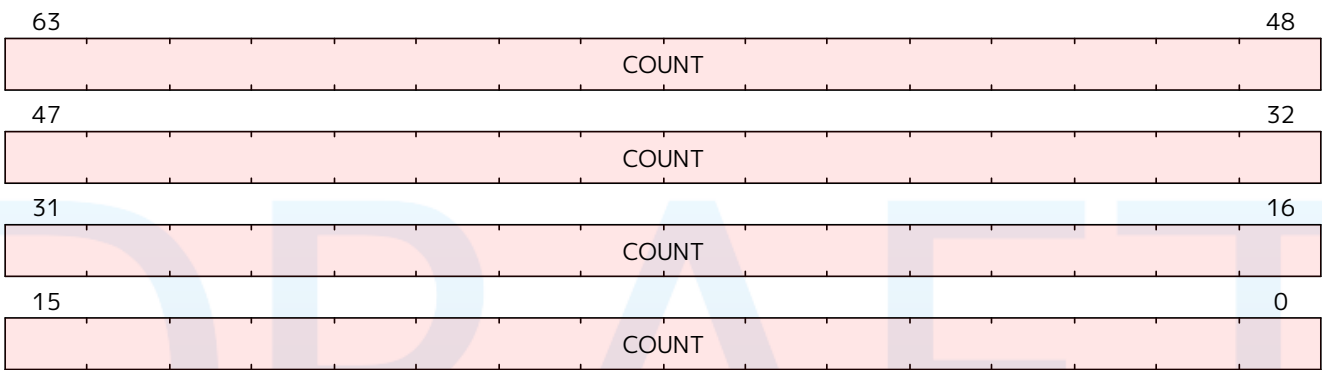


Figure 115. mhpmcounter22 format

C.109. mhpmcounter22h

Machine Hardware Performance Counter 22, Upper half

 *mhpmcounter22h* is only defined in RV32.

Upper half of mhpmcounter22.

C.109.1. Attributes

CSR Address	0xb96
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.109.2. Format

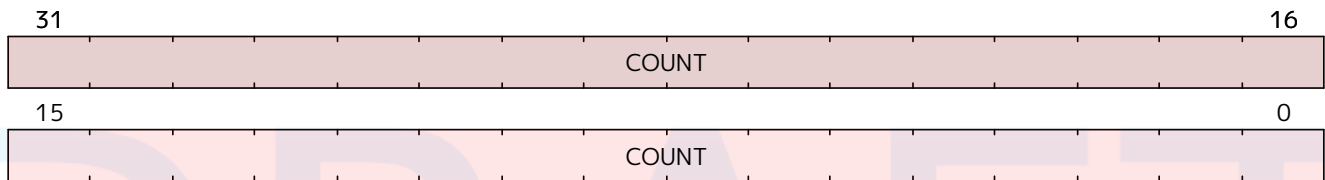


Figure 116. mhpmcounter22h format

C.110. mhpcounter23

Machine Hardware Performance Counter 23

Programmable hardware performance counter.

C.110.1. Attributes

CSR Address	0xb17
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.110.2. Format

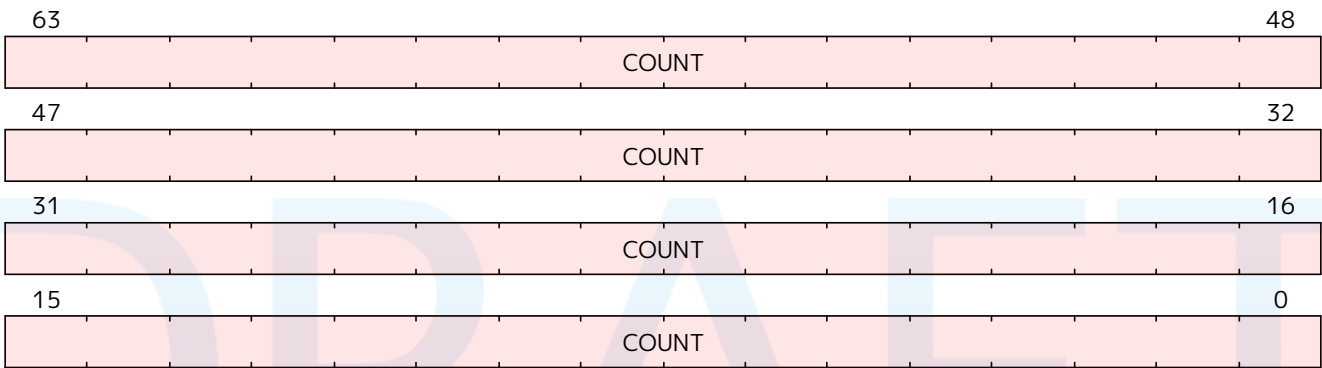


Figure 117. mhpcounter23 format

C.111. mhpmpcounter23h

Machine Hardware Performance Counter 23, Upper half

 *mhpmpcounter23h* is only defined in RV32.

Upper half of mhpmpcounter23.

C.111.1. Attributes

CSR Address	0xb97
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.111.2. Format

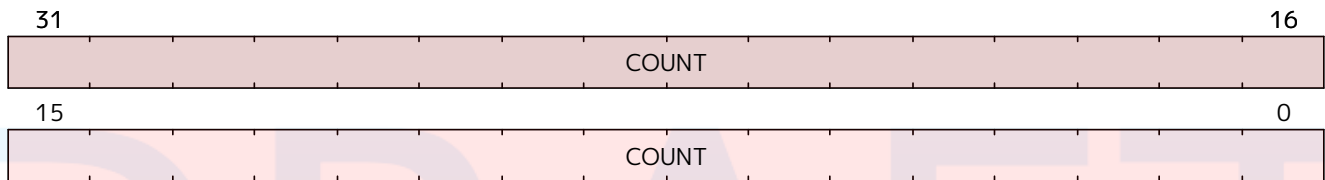


Figure 118. mhpmpcounter23h format

C.112. mhpmcounter24

Machine Hardware Performance Counter 24

Programmable hardware performance counter.

C.112.1. Attributes

CSR Address	0xb18
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.112.2. Format

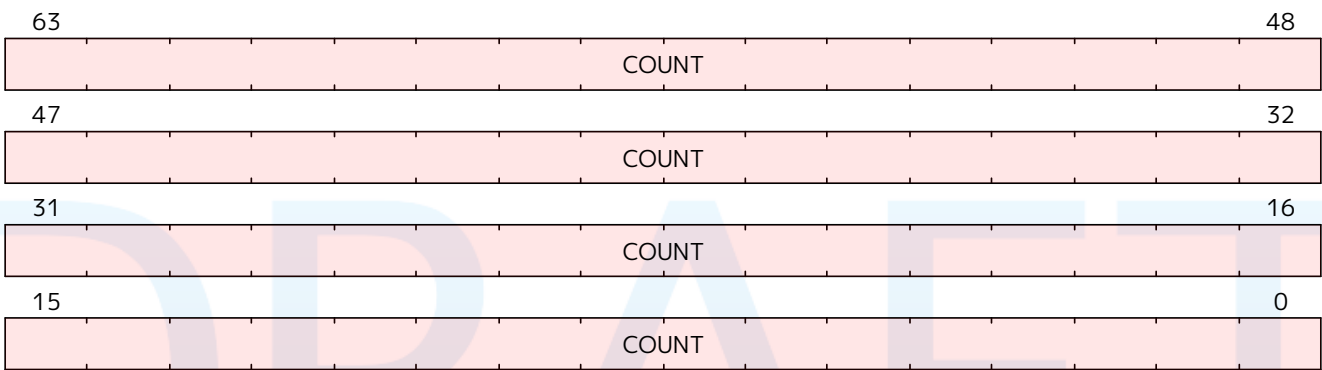


Figure 119. mhpmcounter24 format

C.113. mhpmcounter24h

Machine Hardware Performance Counter 24, Upper half

 *mhpmcounter24h* is only defined in RV32.

Upper half of mhpmcounter24.

C.113.1. Attributes

CSR Address	0xb98
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.113.2. Format

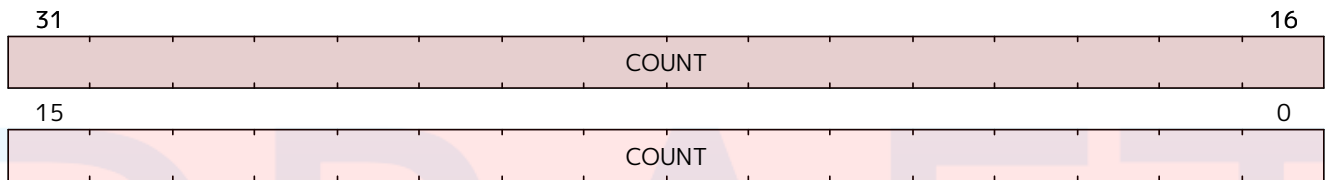


Figure 120. mhpmcounter24h format

C.114. mhpmcounter25

Machine Hardware Performance Counter 25

Programmable hardware performance counter.

C.114.1. Attributes

CSR Address	0xb19
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.114.2. Format

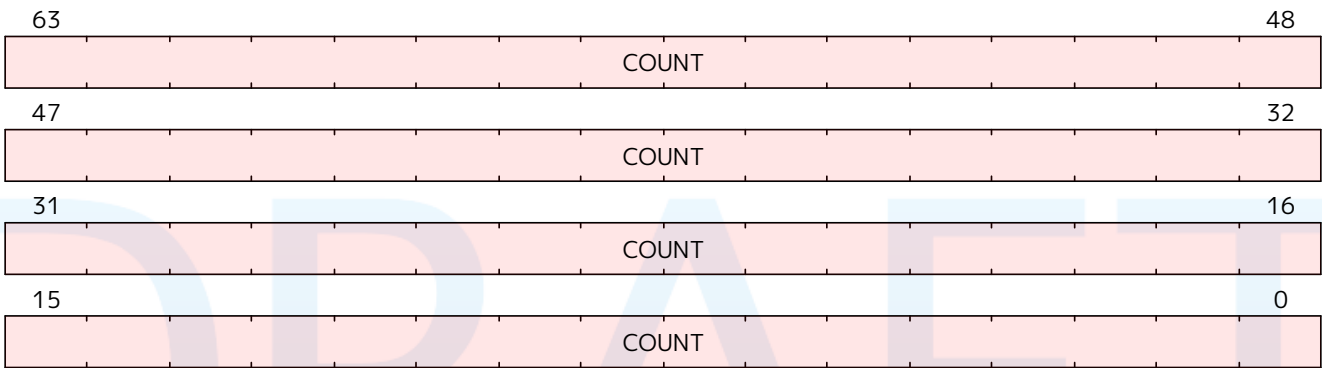


Figure 121. mhpmcounter25 format

C.115. mhpmcounter25h

Machine Hardware Performance Counter 25, Upper half

 *mhpmcounter25h* is only defined in RV32.

Upper half of mhpmcounter25.

C.115.1. Attributes

CSR Address	0xb99
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.115.2. Format

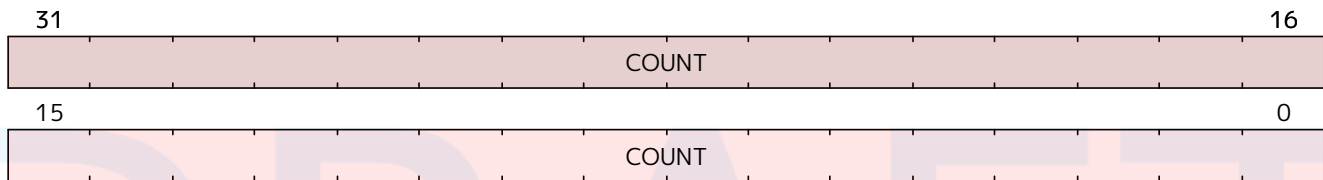


Figure 122. mhpmcounter25h format

C.116. mhpmcounter26

Machine Hardware Performance Counter 26

Programmable hardware performance counter.

C.116.1. Attributes

CSR Address	0xb1a
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.116.2. Format

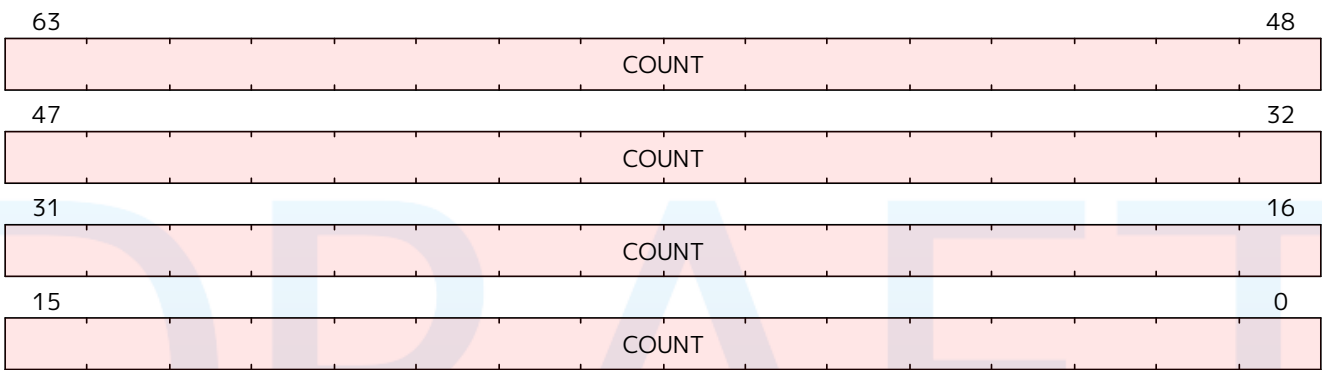


Figure 123. mhpmcounter26 format

C.117. mhpmcounter26h

Machine Hardware Performance Counter 26, Upper half

 *mhpmcounter26h* is only defined in RV32.

Upper half of mhpmcounter26.

C.117.1. Attributes

CSR Address	0xb9a
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.117.2. Format

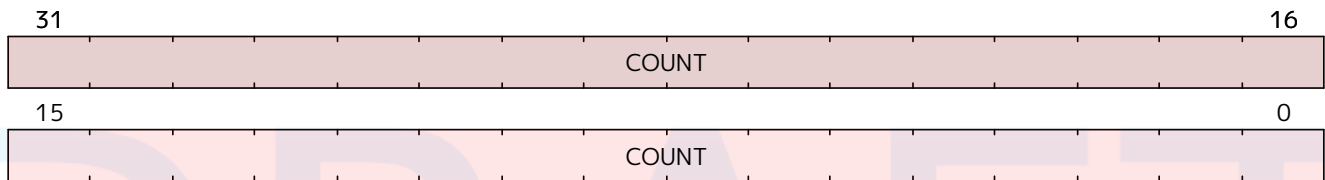


Figure 124. mhpmcounter26h format

C.118. mhpmpcounter27

Machine Hardware Performance Counter 27

Programmable hardware performance counter.

C.118.1. Attributes

CSR Address	0xb1b
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.118.2. Format

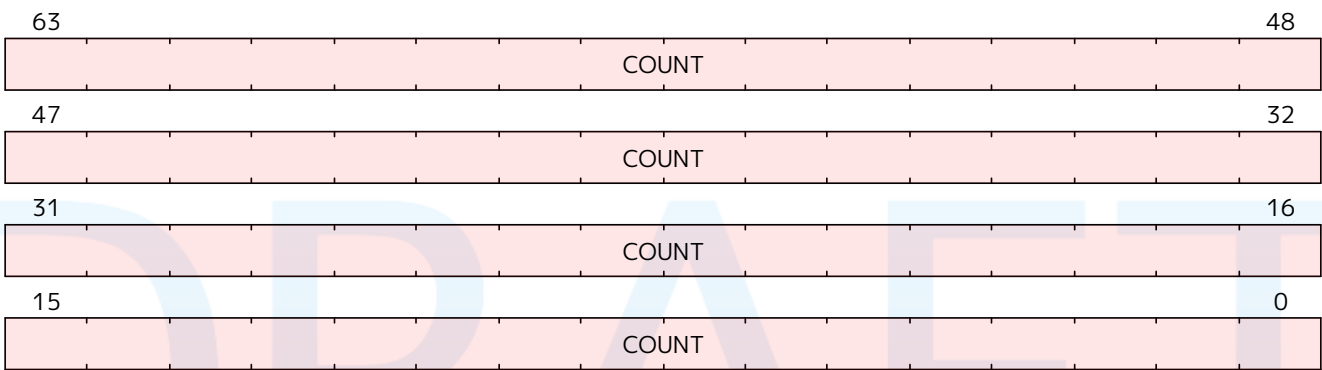


Figure 125. mhpmpcounter27 format

C.119. mhpmpcounter27h

Machine Hardware Performance Counter 27, Upper half



mhpmpcounter27h is only defined in RV32.

Upper half of mhpmpcounter27.

C.119.1. Attributes

CSR Address	0xb9b
Defining extension	Zihpm \geq 0
Length	32-bit
Privilege Mode	M

C.119.2. Format

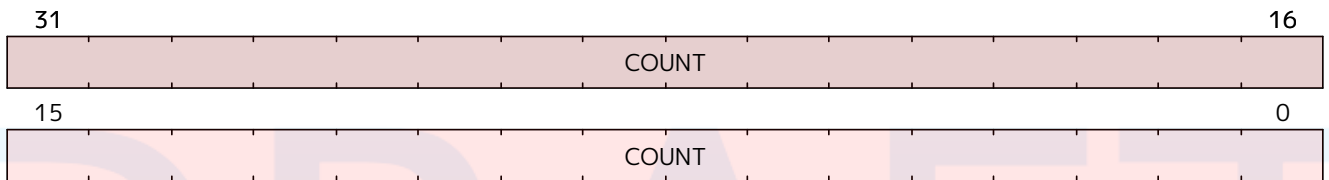


Figure 126. mhpmpcounter27h format

C.120. mhpmcounter28

Machine Hardware Performance Counter 28

Programmable hardware performance counter.

C.120.1. Attributes

CSR Address	0xb1c
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.120.2. Format

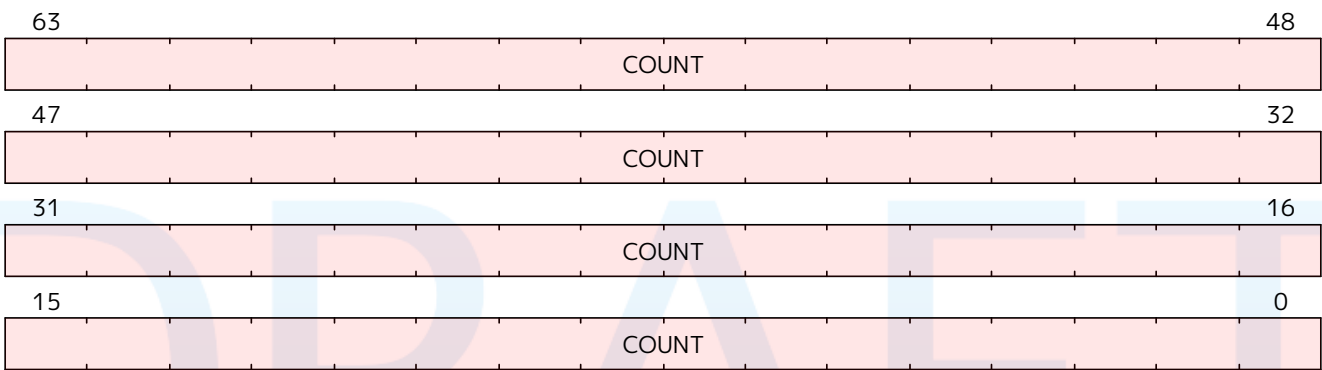


Figure 127. mhpmcounter28 format

C.121. mhpmcounter28h

Machine Hardware Performance Counter 28, Upper half

 *mhpmcounter28h* is only defined in RV32.

Upper half of mhpmcounter28.

C.121.1. Attributes

CSR Address	0xb9c
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.121.2. Format

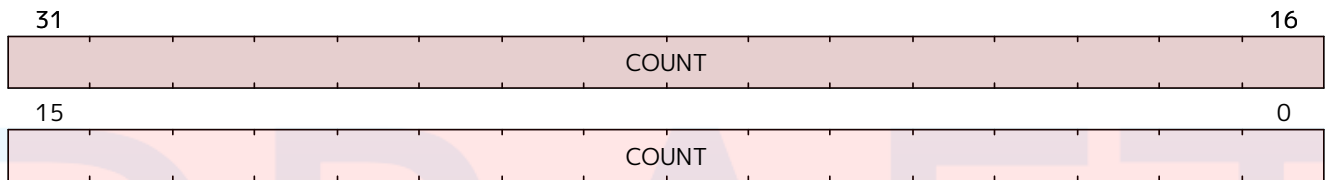


Figure 128. mhpmcounter28h format

C.122. mhpmcounter29

Machine Hardware Performance Counter 29

Programmable hardware performance counter.

C.122.1. Attributes

CSR Address	0xb1d
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.122.2. Format

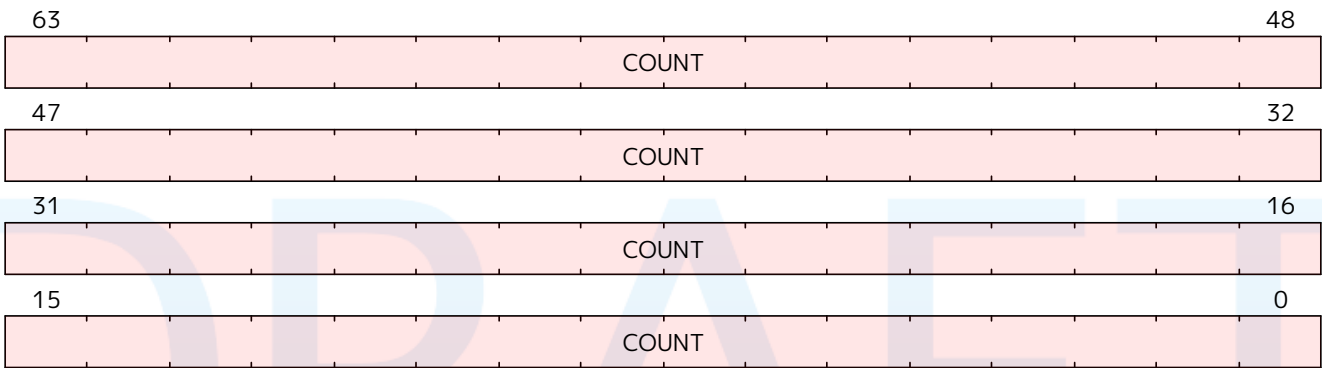


Figure 129. mhpmcounter29 format

C.123. mhpmpcounter29h

Machine Hardware Performance Counter 29, Upper half

 *mhpmpcounter29h* is only defined in RV32.

Upper half of mhpmpcounter29.

C.123.1. Attributes

CSR Address	0xb9d
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.123.2. Format

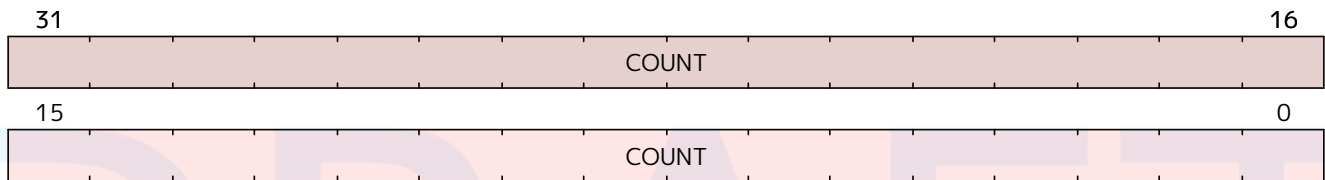


Figure 130. mhpmpcounter29h format

C.124. mhpmcounter3

Machine Hardware Performance Counter 3

Programmable hardware performance counter.

C.124.1. Attributes

CSR Address	0xb03
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.124.2. Format

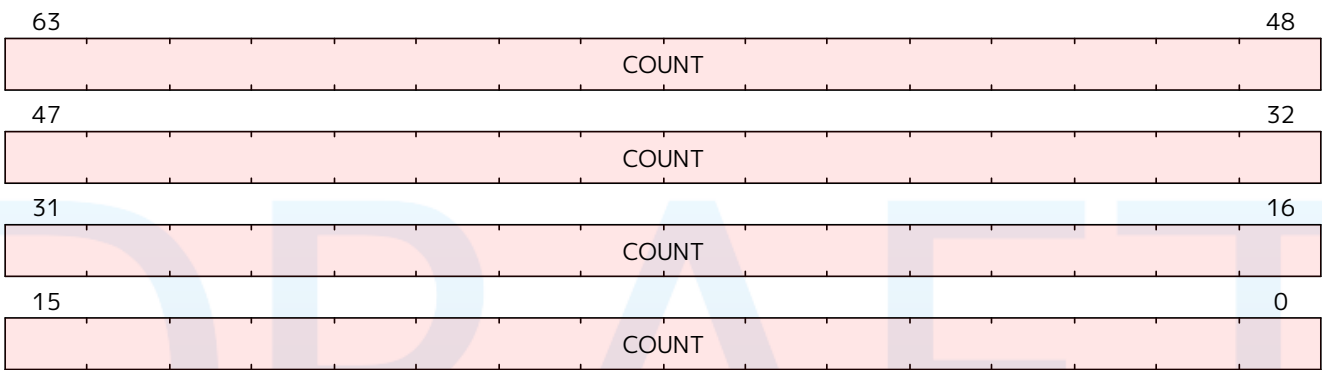


Figure 131. mhpmcounter3 format

C.125. mhpmcounter30

Machine Hardware Performance Counter 30

Programmable hardware performance counter.

C.125.1. Attributes

CSR Address	0xb1e
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.125.2. Format

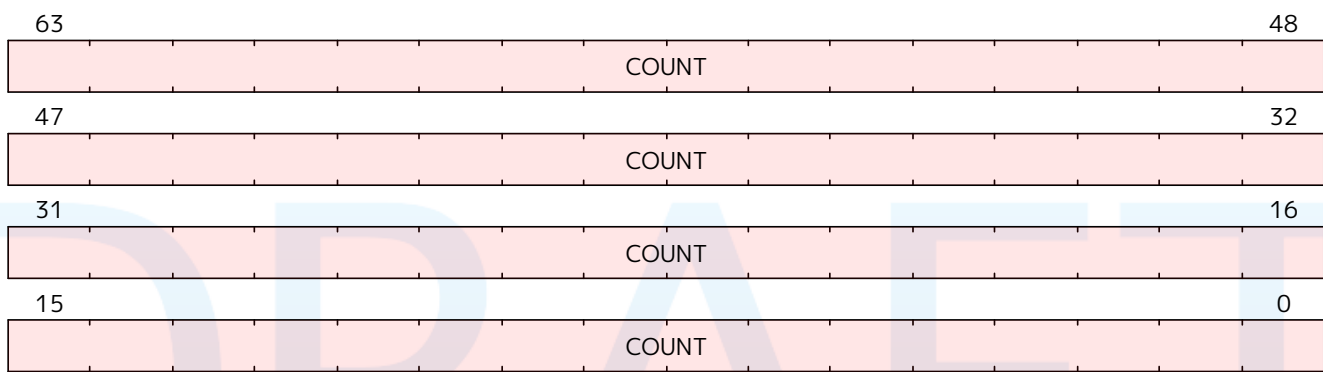


Figure 132. mhpmcounter30 format

C.126. mhpmcounter30h

Machine Hardware Performance Counter 30, Upper half

 *mhpmcounter30h* is only defined in RV32.

Upper half of mhpmcounter30.

C.126.1. Attributes

CSR Address	0xb9e
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.126.2. Format

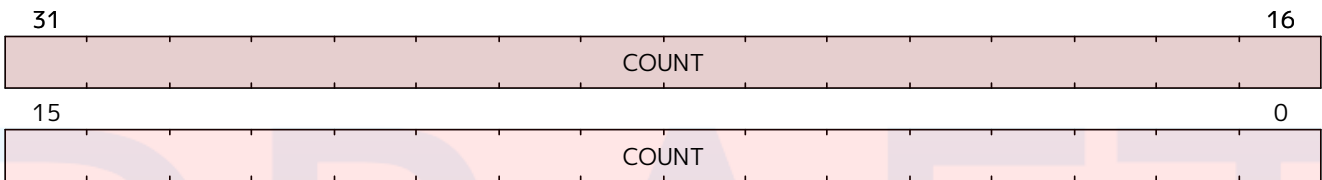


Figure 133. mhpmcounter30h format

C.127. mhpmcounter31

Machine Hardware Performance Counter 31

Programmable hardware performance counter.

C.127.1. Attributes

CSR Address	0xb1f
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.127.2. Format

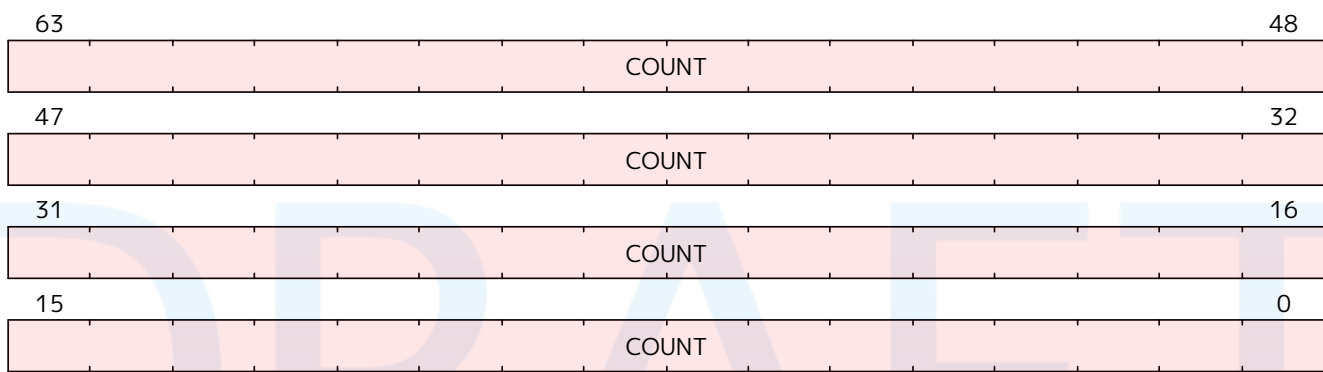


Figure 134. mhpmcounter31 format

C.128. mhpmpcounter31h

Machine Hardware Performance Counter 31, Upper half

 *mhpmpcounter31h* is only defined in RV32.

Upper half of mhpmpcounter31.

C.128.1. Attributes

CSR Address	0xb9f
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.128.2. Format

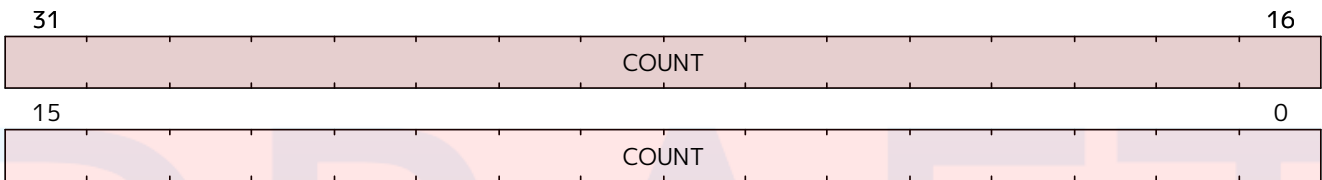


Figure 135. mhpmpcounter31h format

C.129. mhpmcounter3h

Machine Hardware Performance Counter 3, Upper half



mhpmcounter3h is only defined in RV32.

Upper half of mhpmcounter3.

C.129.1. Attributes

CSR Address	0xb83
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.129.2. Format

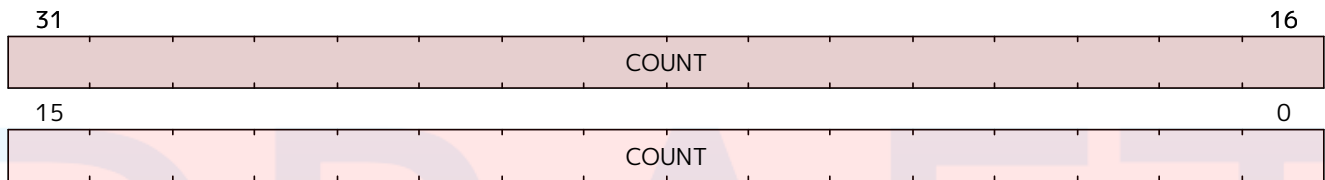


Figure 136. mhpmcounter3h format

C.130. mhpmcounter4

Machine Hardware Performance Counter 4

Programmable hardware performance counter.

C.130.1. Attributes

CSR Address	0xb04
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.130.2. Format

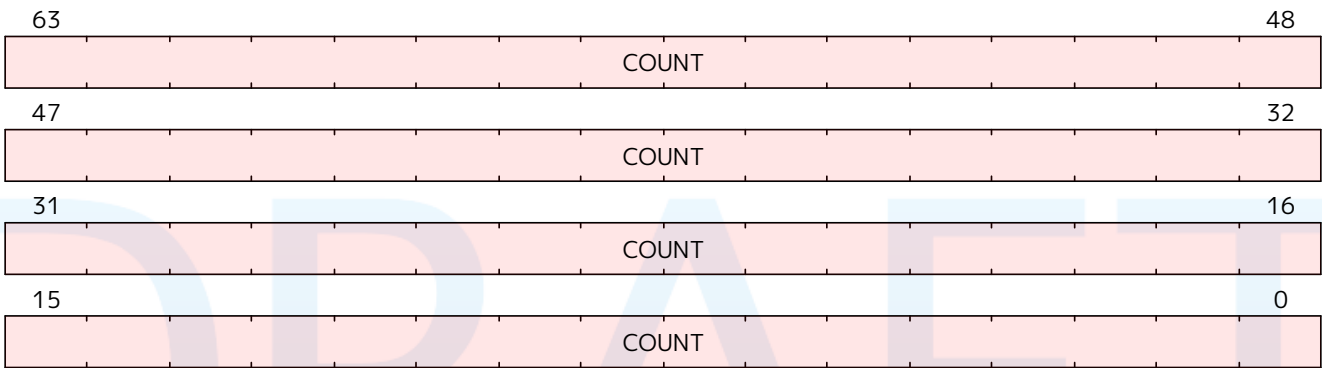


Figure 137. mhpmcounter4 format

C.131. mhpmcounter4h

Machine Hardware Performance Counter 4, Upper half

 *mhpmcounter4h* is only defined in RV32.

Upper half of mhpmcounter4.

C.131.1. Attributes

CSR Address	0xb84
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.131.2. Format

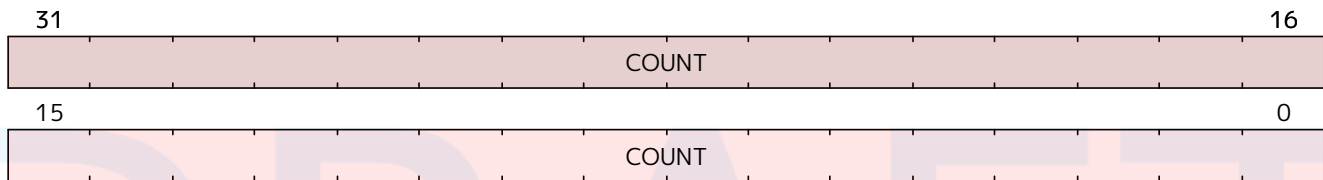


Figure 138. mhpmcounter4h format

C.132. mhpmcounter5

Machine Hardware Performance Counter 5

Programmable hardware performance counter.

C.132.1. Attributes

CSR Address	0xb05
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.132.2. Format

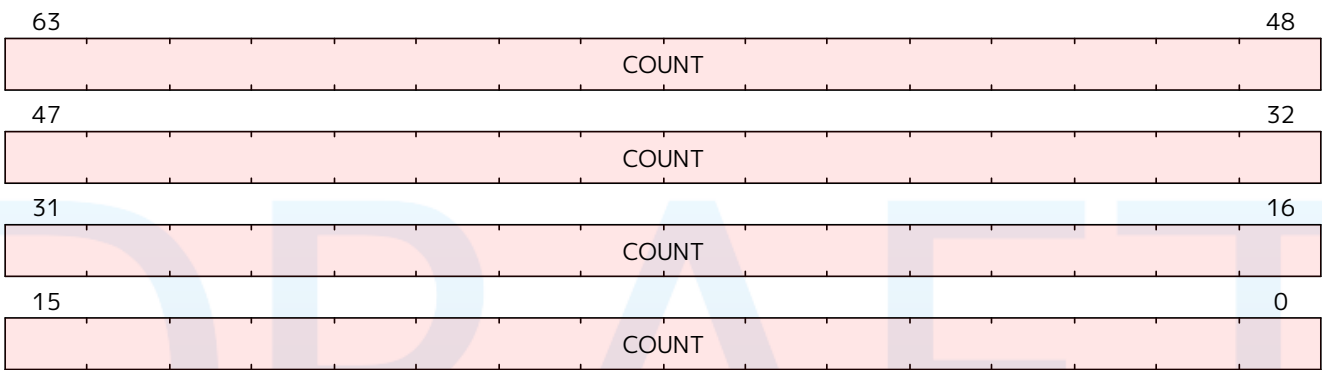


Figure 139. mhpmcounter5 format

C.133. mhpmpcounter5h

Machine Hardware Performance Counter 5, Upper half

 *mhpmpcounter5h* is only defined in RV32.

Upper half of mhpmpcounter5.

C.133.1. Attributes

CSR Address	0xb85
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.133.2. Format

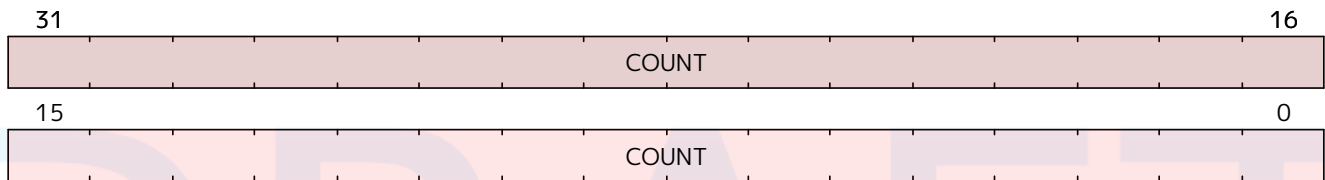


Figure 140. mhpmpcounter5h format

C.134. mhpmcounter6

Machine Hardware Performance Counter 6

Programmable hardware performance counter.

C.134.1. Attributes

CSR Address	0xb06
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.134.2. Format

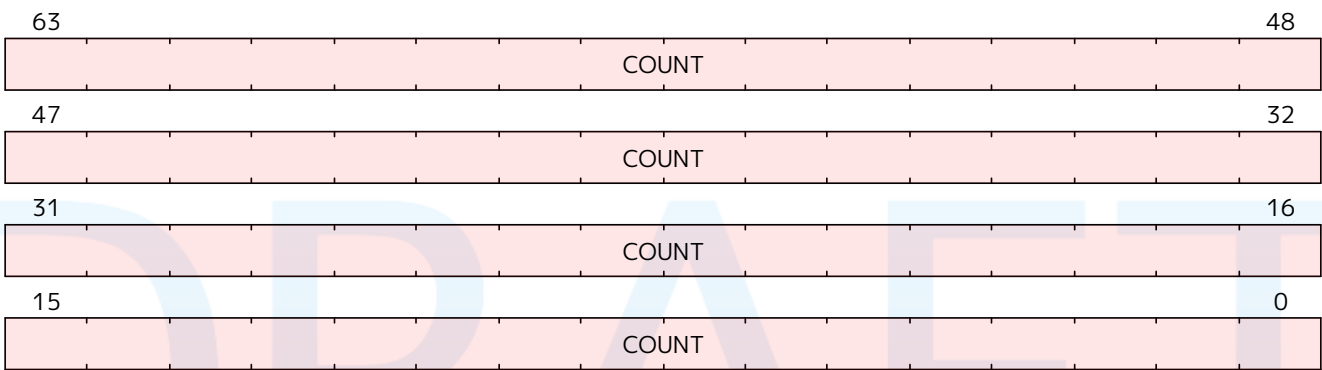


Figure 141. mhpmcounter6 format

C.135. mhpmcounter6h

Machine Hardware Performance Counter 6, Upper half

 *mhpmcounter6h* is only defined in RV32.

Upper half of mhpmcounter6.

C.135.1. Attributes

CSR Address	0xb86
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.135.2. Format

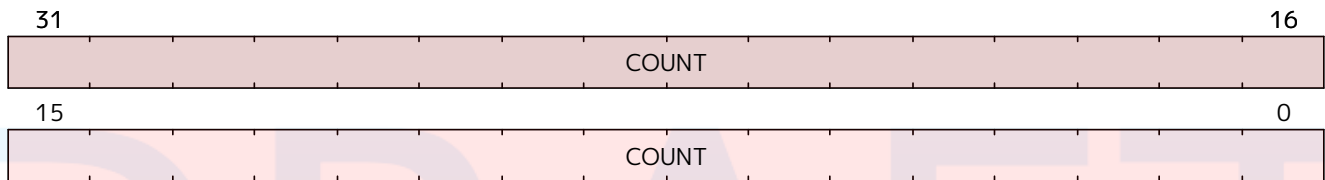


Figure 142. mhpmcounter6h format

C.136. mhpmcounter7

Machine Hardware Performance Counter 7

Programmable hardware performance counter.

C.136.1. Attributes

CSR Address	0xb07
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.136.2. Format

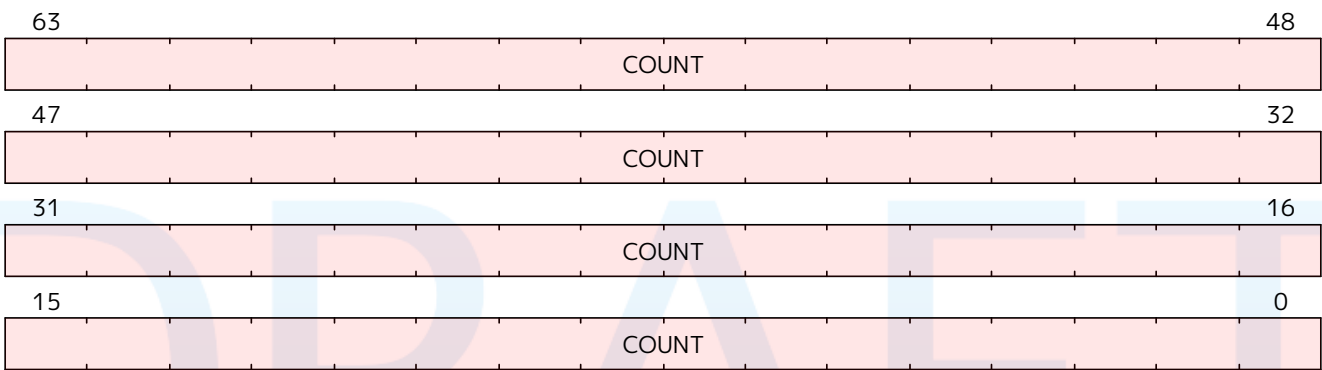


Figure 143. mhpmcounter7 format

C.137. mhpmcounter7h

Machine Hardware Performance Counter 7, Upper half

 *mhpmcounter7h* is only defined in RV32.

Upper half of mhpmcounter7.

C.137.1. Attributes

CSR Address	0xb87
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.137.2. Format

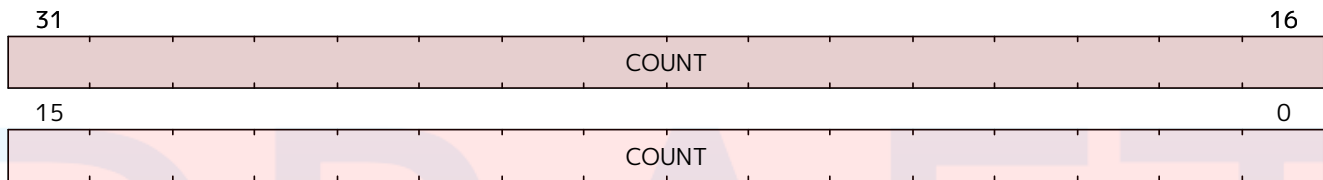


Figure 144. mhpmcounter7h format

C.138. mhpmcounter8

Machine Hardware Performance Counter 8

Programmable hardware performance counter.

C.138.1. Attributes

CSR Address	0xb08
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.138.2. Format

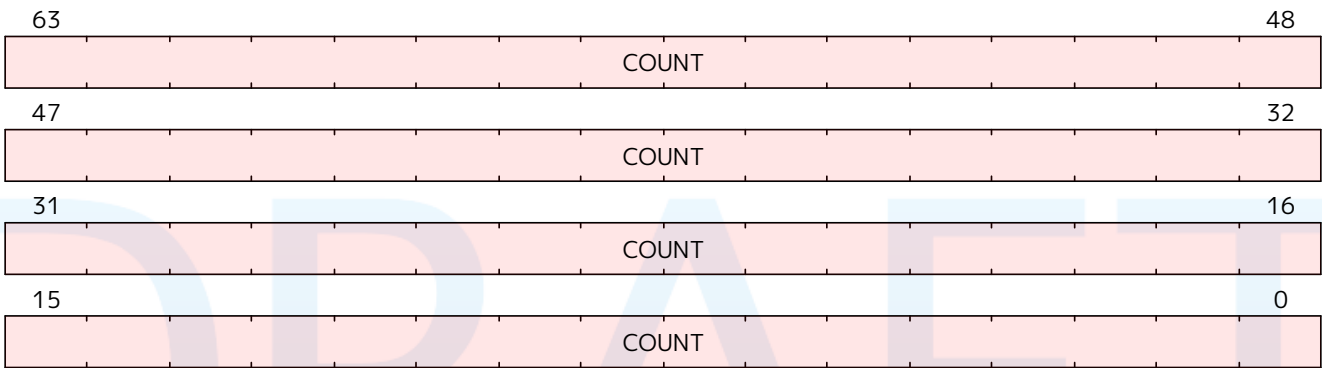


Figure 145. mhpmcounter8 format

C.139. mhpmcounter8h

Machine Hardware Performance Counter 8, Upper half

 *mhpmcounter8h* is only defined in RV32.

Upper half of mhpmcounter8.

C.139.1. Attributes

CSR Address	0xb88
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.139.2. Format

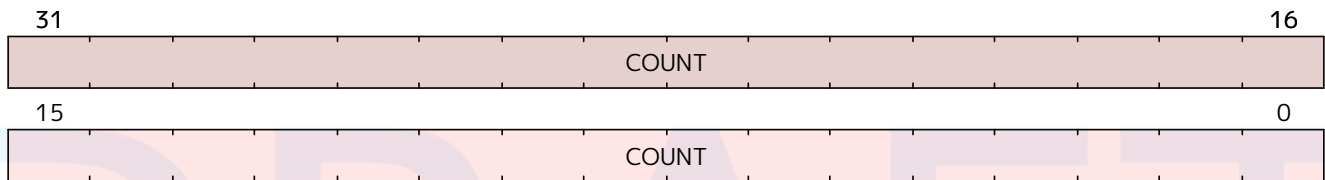


Figure 146. mhpmcounter8h format

C.140. mhpmcounter9

Machine Hardware Performance Counter 9

Programmable hardware performance counter.

C.140.1. Attributes

CSR Address	0xb09
Defining extension	Zihpm >= 0
Length	64-bit
Privilege Mode	M

C.140.2. Format

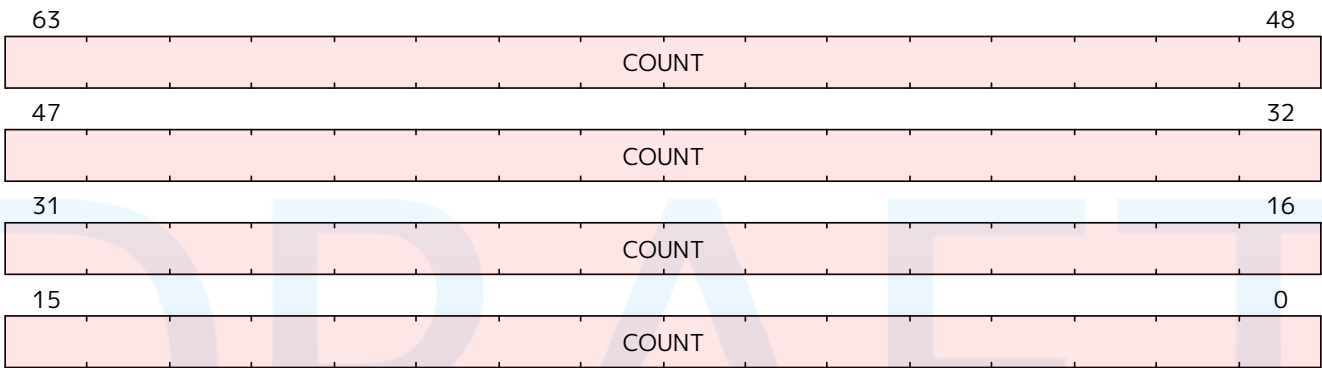


Figure 147. mhpmcounter9 format

C.141. mhpmcounter9h

Machine Hardware Performance Counter 9, Upper half



mhpmcounter9h is only defined in RV32.

Upper half of mhpmcounter9.

C.141.1. Attributes

CSR Address	0xb89
Defining extension	Zihpm >= 0
Length	32-bit
Privilege Mode	M

C.141.2. Format

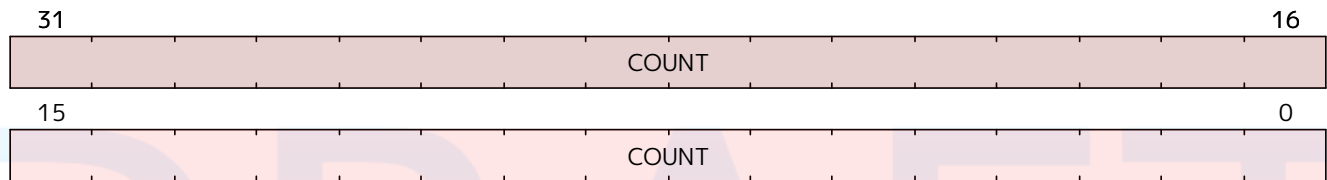


Figure 148. mhpmcounter9h format

C.142. mhpmevent10

Machine Hardware Performance Counter 10 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.142.1. Attributes

CSR Address	0x32a
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.142.2. Format

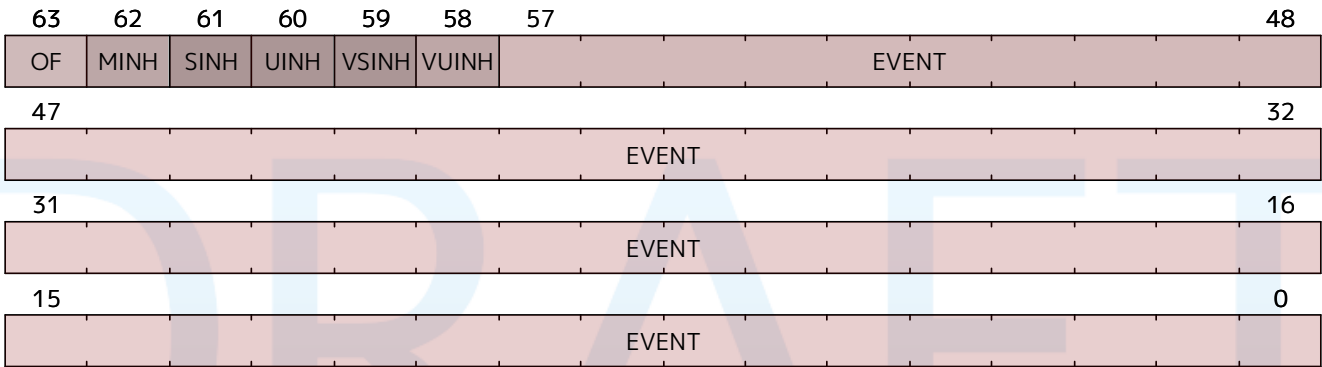


Figure 149. mhpmevent10 format

C.143. mhpmevent10h

Machine Hardware Performance Counter 10 Control, High half

 *mhpmevent10h* is only defined in RV32.

Alias of [mhpmevent10](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.143.1. Attributes

CSR Address	0x72a
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.143.2. Format

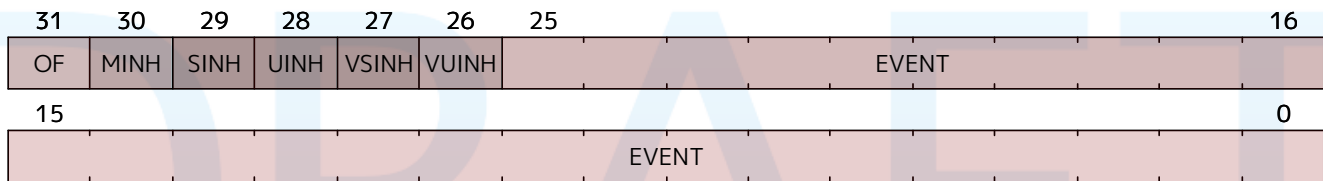


Figure 150. mhpmevent10h format

C.144. mhpmevent11

Machine Hardware Performance Counter 11 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.144.1. Attributes

CSR Address	0x32b
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.144.2. Format

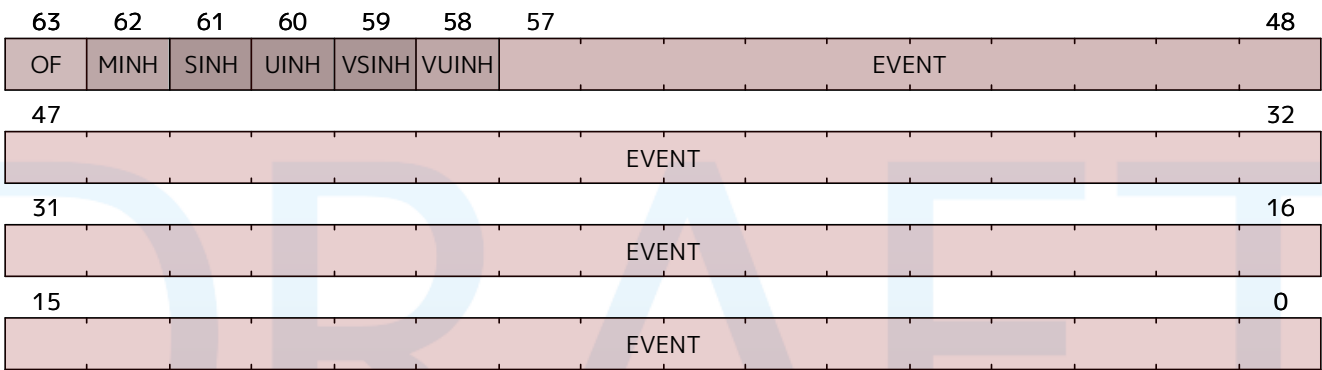


Figure 151. mhpmevent11 format

C.145. mhpmevent11h

Machine Hardware Performance Counter 11 Control, High half



mhpmevent11h is only defined in RV32.

Alias of [mhpmevent11](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.145.1. Attributes

CSR Address	0x72b
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.145.2. Format

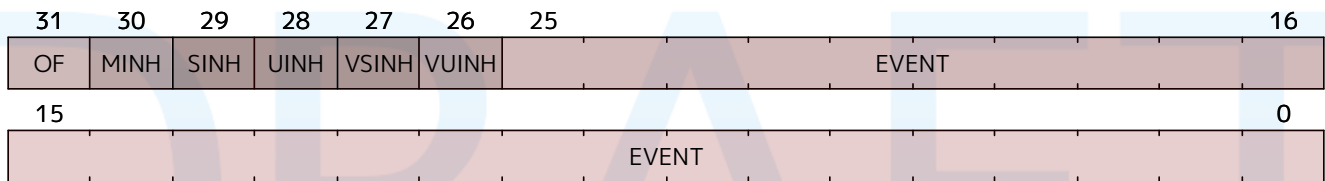


Figure 152. mhpmevent11h format

C.146. mhpmevent12

Machine Hardware Performance Counter 12 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.146.1. Attributes

CSR Address	0x32c
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.146.2. Format

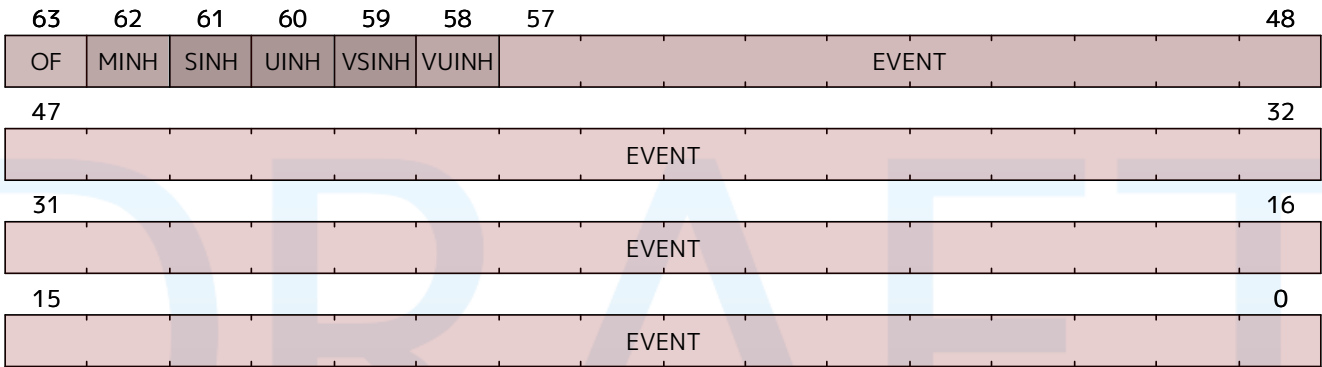


Figure 153. mhpmevent12 format

C.147. mhpmevent12h

Machine Hardware Performance Counter 12 Control, High half



mhpmevent12h is only defined in RV32.

Alias of [mhpmevent12](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.147.1. Attributes

CSR Address	0x72c
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.147.2. Format

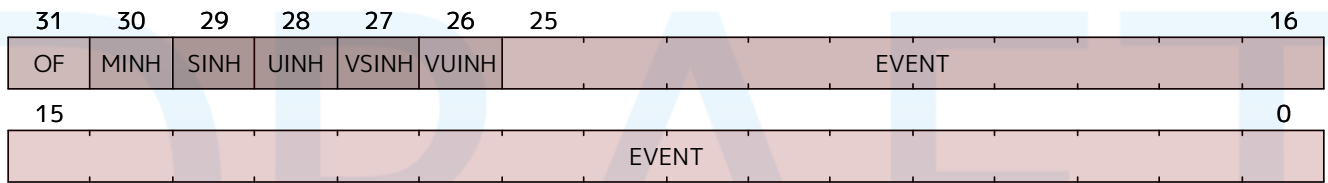


Figure 154. mhpmevent12h format

C.148. mhpmevent13

Machine Hardware Performance Counter 13 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.148.1. Attributes

CSR Address	0x32d
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.148.2. Format

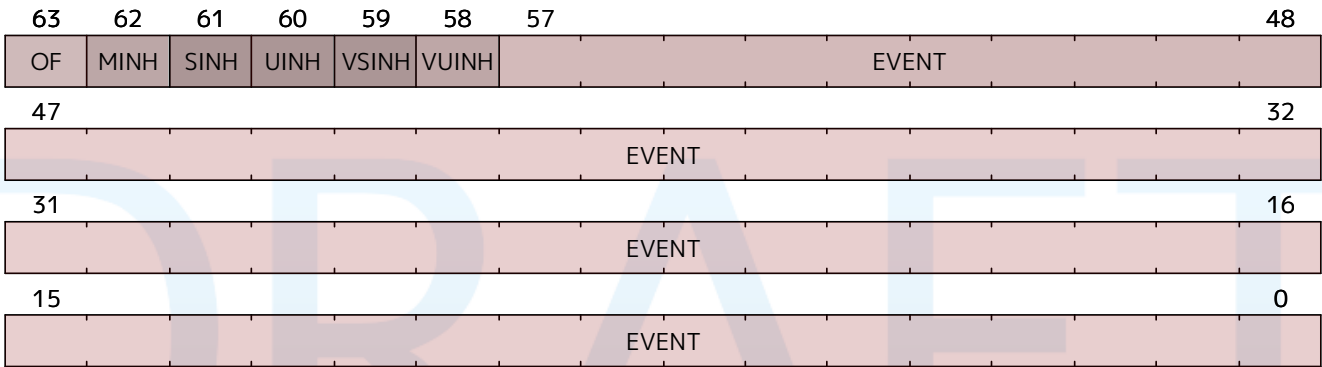


Figure 155. mhpmevent13 format

C.149. mhpmevent13h

Machine Hardware Performance Counter 13 Control, High half



mhpmevent13h is only defined in RV32.

Alias of [mhpmevent13](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.149.1. Attributes

CSR Address	0x72d
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.149.2. Format

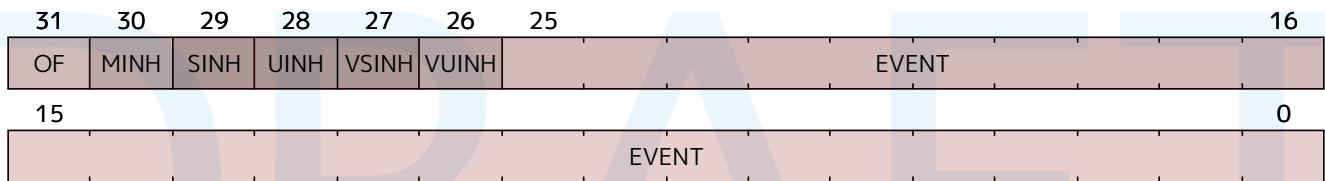


Figure 156. *mhpmevent13h* format

C.150. mhpmevent14

Machine Hardware Performance Counter 14 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.150.1. Attributes

CSR Address	0x32e
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.150.2. Format

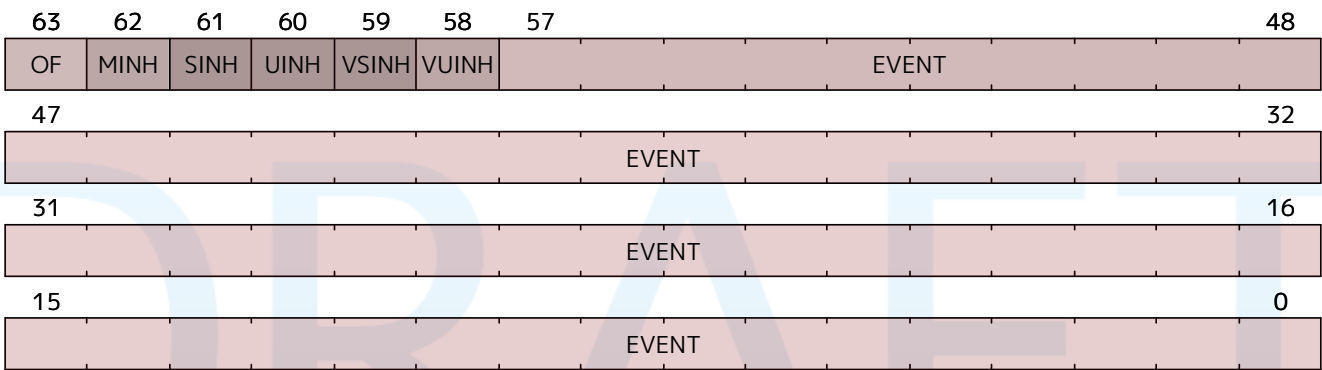


Figure 157. mhpmevent14 format

C.151. mhpmevent14h

Machine Hardware Performance Counter 14 Control, High half

 *mhpmevent14h* is only defined in RV32.

Alias of [mhpmevent14](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.151.1. Attributes

CSR Address	0x72e
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.151.2. Format

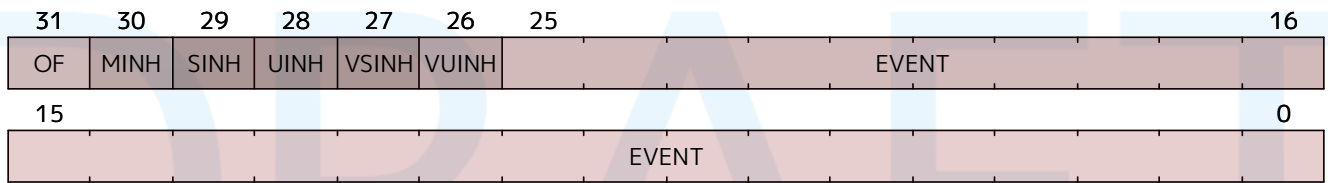


Figure 158. *mhpmevent14h* format

C.152. mhpmevent15

Machine Hardware Performance Counter 15 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.152.1. Attributes

CSR Address	0x32f
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.152.2. Format

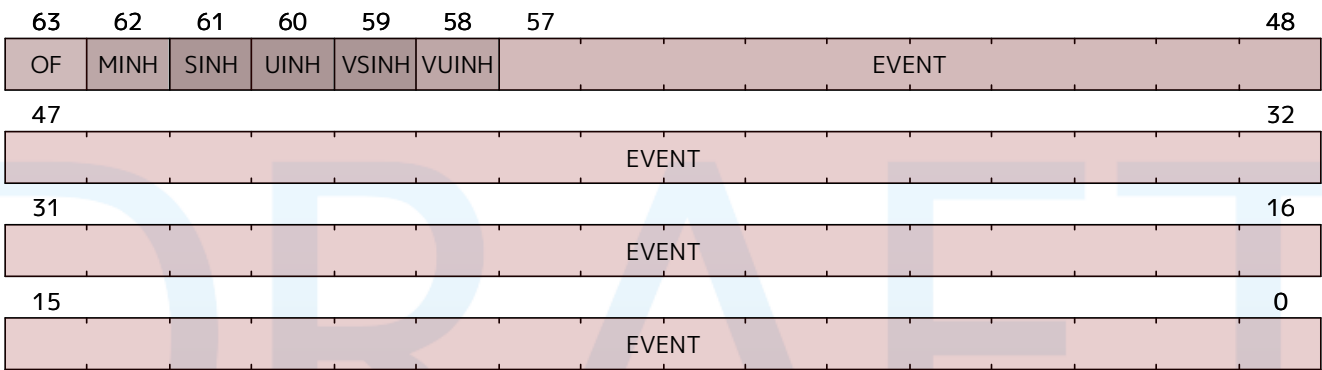


Figure 159. mhpmevent15 format

C.153. mhpmevent15h

Machine Hardware Performance Counter 15 Control, High half

i | *mhpmevent15h* is only defined in RV32.

Alias of [mhpmevent15](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.153.1. Attributes

CSR Address	0x72f
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.153.2. Format

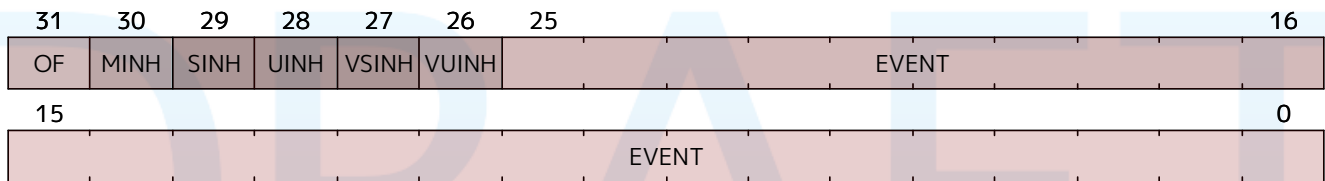


Figure 160. mhpmevent15h format

C.154. mhpmevent16

Machine Hardware Performance Counter 16 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.154.1. Attributes

CSR Address	0x330
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.154.2. Format

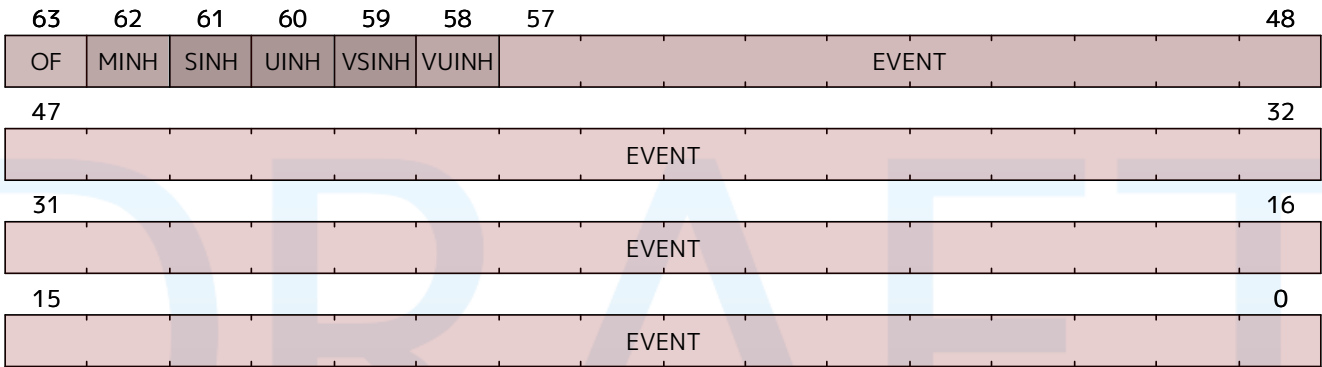


Figure 161. mhpmevent16 format

C.155. mhpmevent16h

Machine Hardware Performance Counter 16 Control, High half

 *mhpmevent16h* is only defined in RV32.

Alias of [mhpmevent16](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.155.1. Attributes

CSR Address	0x730
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.155.2. Format

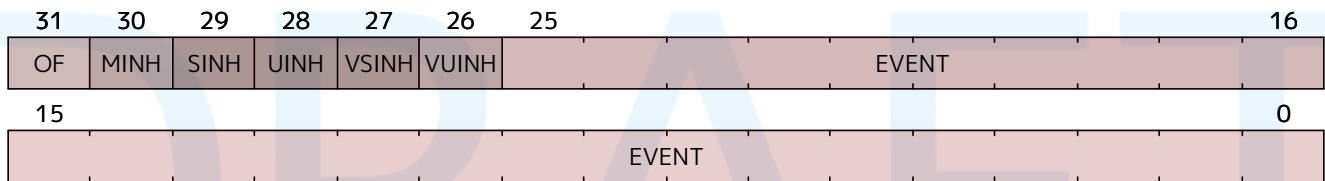


Figure 162. mhpmevent16h format

C.156. mhpmevent17

Machine Hardware Performance Counter 17 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.156.1. Attributes

CSR Address	0x331
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.156.2. Format

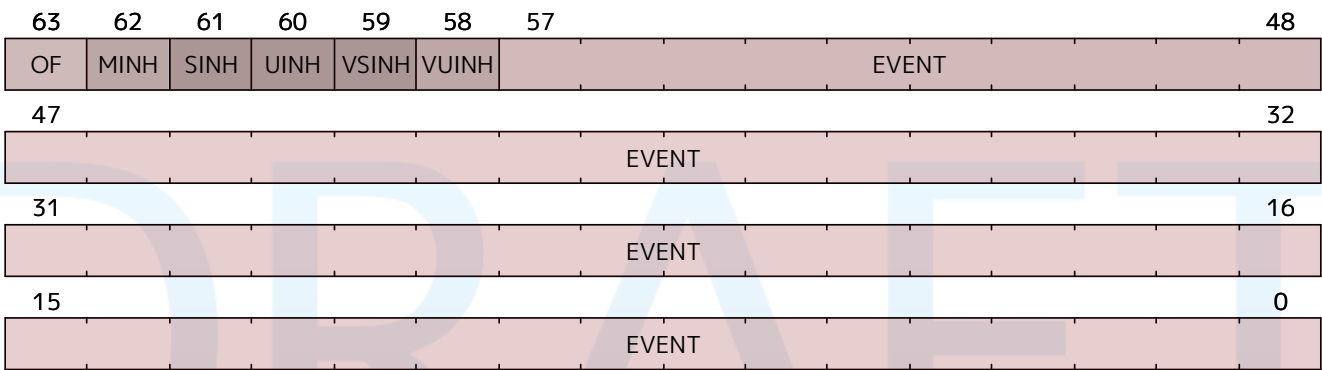


Figure 163. mhpmevent17 format

C.157. mhpmevent17h

Machine Hardware Performance Counter 17 Control, High half

 *mhpmevent17h* is only defined in RV32.

Alias of [mhpmevent17](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.157.1. Attributes

CSR Address	0x731
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.157.2. Format

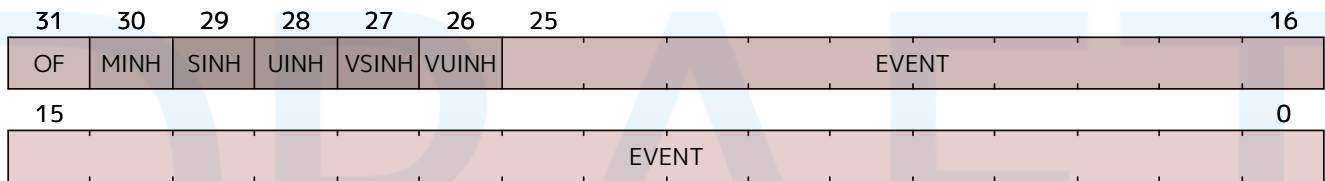


Figure 164. mhpmevent17h format

C.158. mhpmevent18

Machine Hardware Performance Counter 18 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.158.1. Attributes

CSR Address	0x332
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.158.2. Format

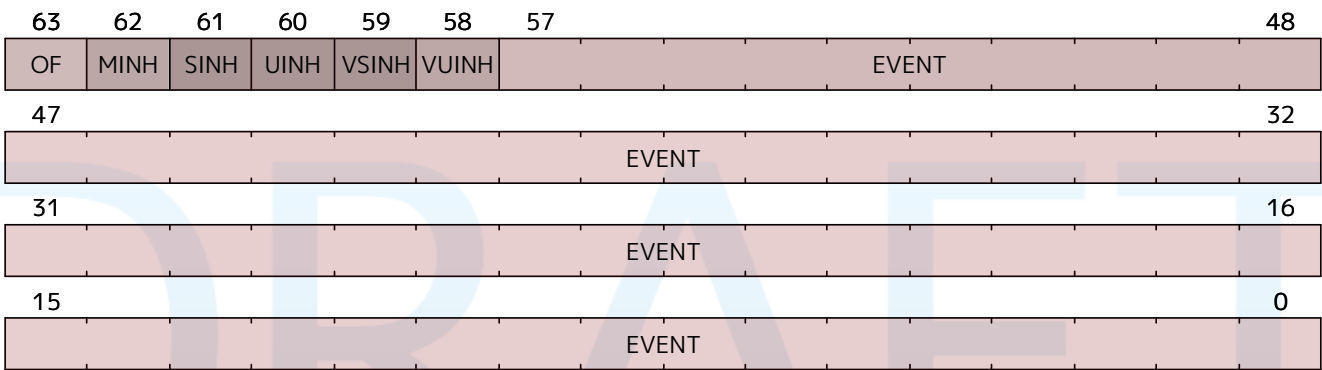


Figure 165. mhpmevent18 format

C.159. mhpmevent18h

Machine Hardware Performance Counter 18 Control, High half

 *mhpmevent18h* is only defined in RV32.

Alias of [mhpmevent18](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.159.1. Attributes

CSR Address	0x732
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.159.2. Format

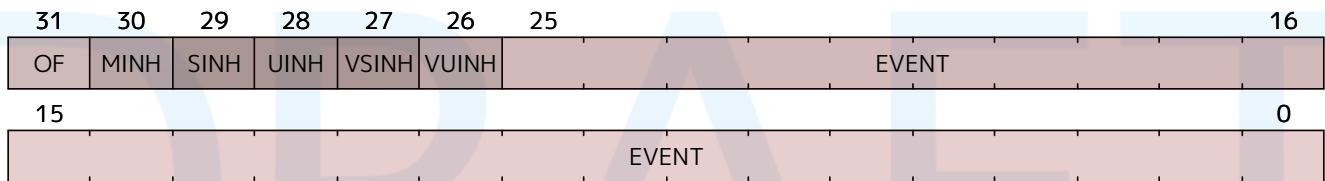


Figure 166. *mhpmevent18h* format

C.160. mhpmevent19

Machine Hardware Performance Counter 19 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.160.1. Attributes

CSR Address	0x333
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.160.2. Format

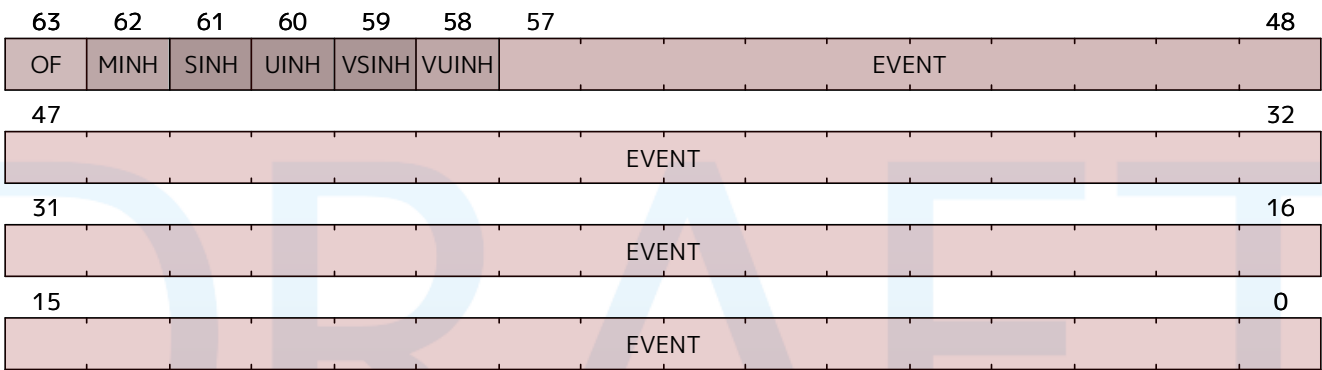


Figure 167. mhpmevent19 format

C.161. mhpmevent19h

Machine Hardware Performance Counter 19 Control, High half

 *mhpmevent19h* is only defined in RV32.

Alias of [mhpmevent19](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.161.1. Attributes

CSR Address	0x733
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.161.2. Format

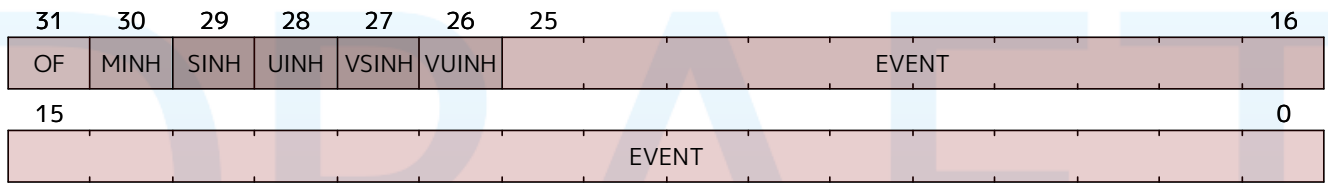


Figure 168. mhpmevent19h format

C.162. mhpmevent20

Machine Hardware Performance Counter 20 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.162.1. Attributes

CSR Address	0x334
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.162.2. Format

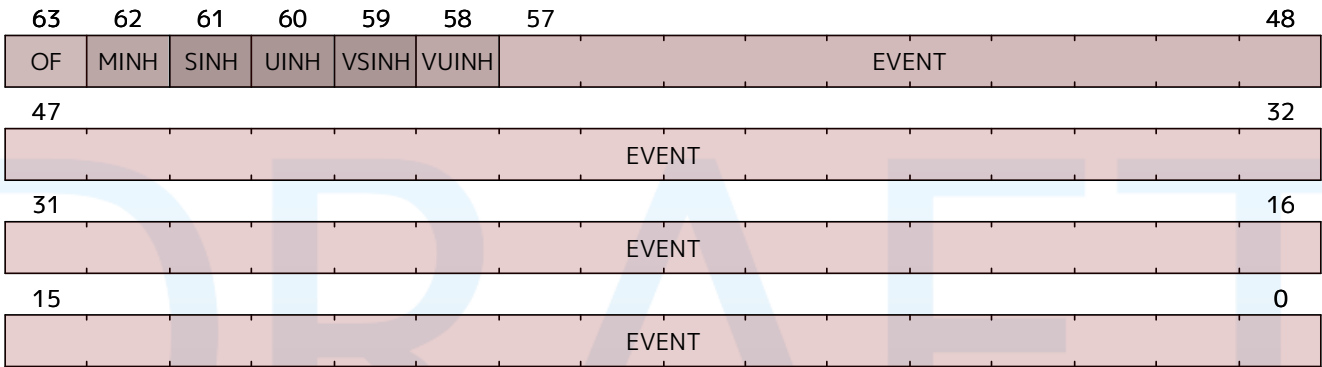


Figure 169. mhpmevent20 format

C.163. mhpmevent20h

Machine Hardware Performance Counter 20 Control, High half

 *mhpmevent20h* is only defined in RV32.

Alias of [mhpmevent20](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.163.1. Attributes

CSR Address	0x734
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.163.2. Format

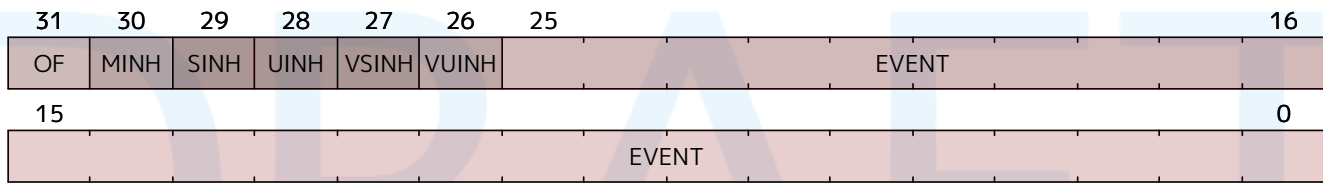


Figure 170. *mhpmevent20h* format

C.164. mhpmevent21

Machine Hardware Performance Counter 21 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.164.1. Attributes

CSR Address	0x335
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.164.2. Format

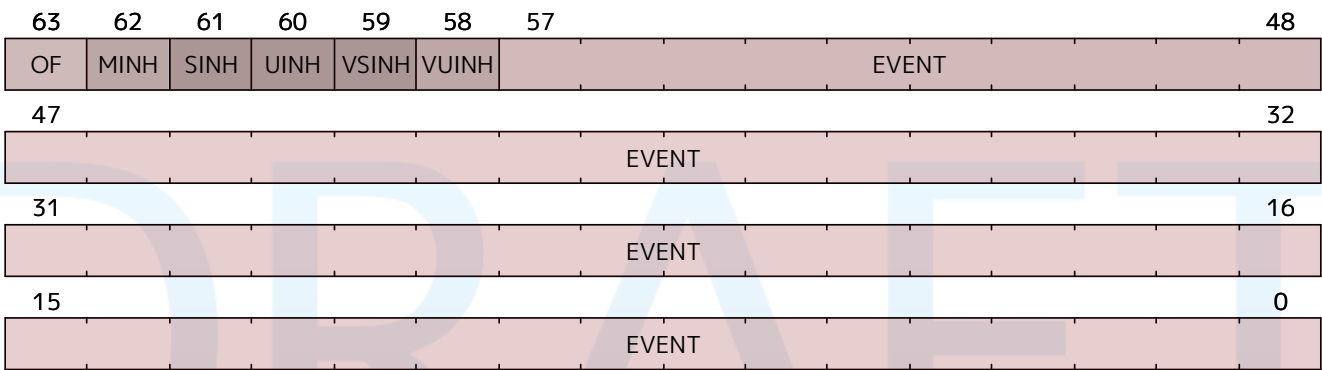


Figure 171. mhpmevent21 format

C.165. mhpmevent21h

Machine Hardware Performance Counter 21 Control, High half



mhpmevent21h is only defined in RV32.

Alias of [mhpmevent21](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.165.1. Attributes

CSR Address	0x735
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.165.2. Format

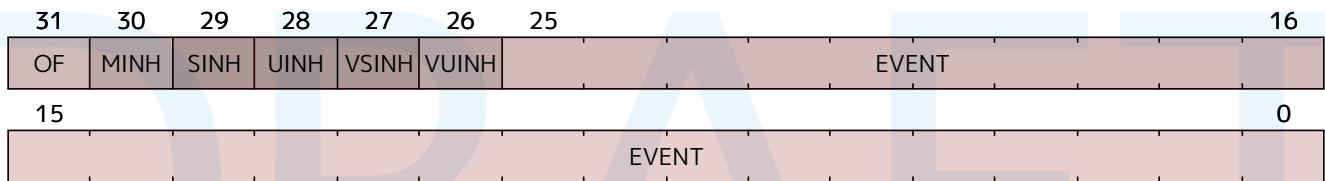


Figure 172. mhpmevent21h format

C.166. mhpmevent22

Machine Hardware Performance Counter 22 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.166.1. Attributes

CSR Address	0x336
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.166.2. Format

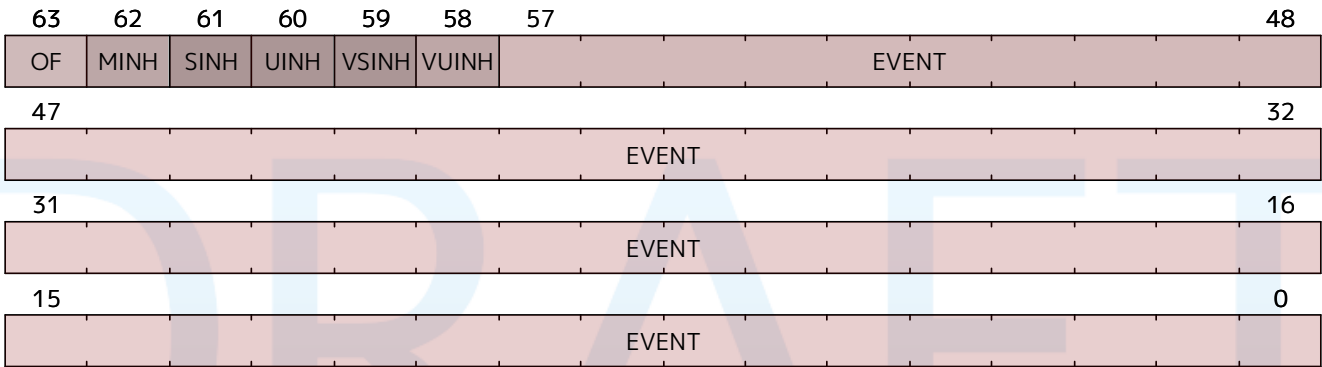


Figure 173. mhpmevent22 format

C.167. mhpmevent22h

Machine Hardware Performance Counter 22 Control, High half



mhpmevent22h is only defined in RV32.

Alias of [mhpmevent22](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.167.1. Attributes

CSR Address	0x736
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.167.2. Format

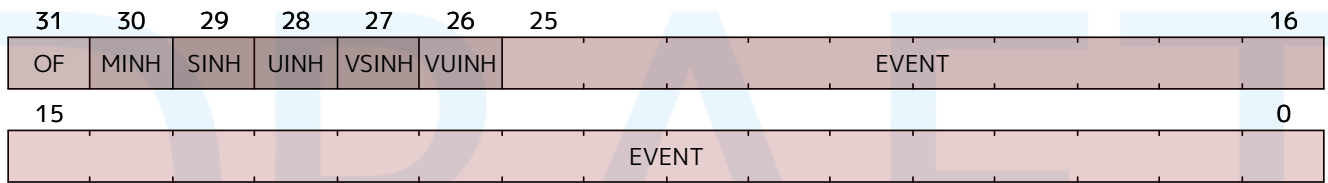


Figure 174. *mhpmevent22h* format

C.168. mhpmevent23

Machine Hardware Performance Counter 23 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.168.1. Attributes

CSR Address	0x337
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.168.2. Format

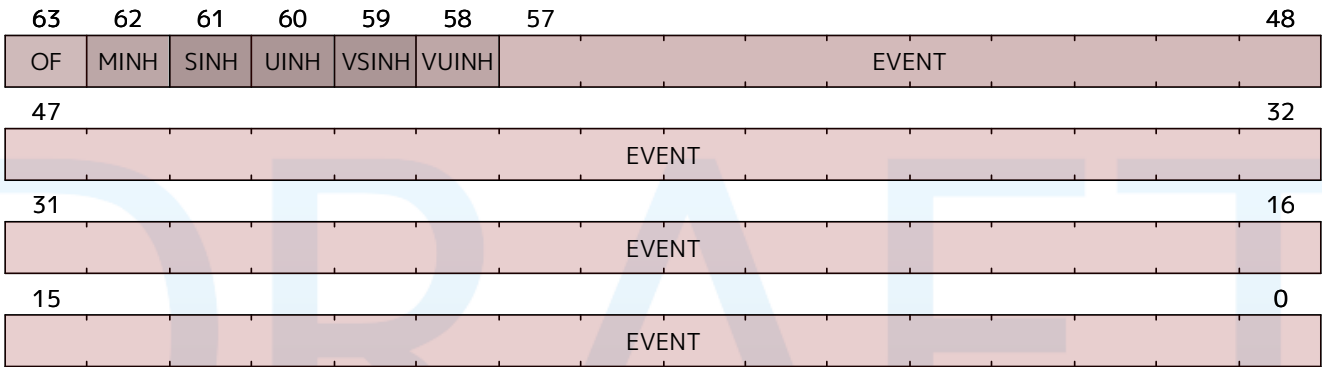


Figure 175. mhpmevent23 format

C.169. mhpmevent23h

Machine Hardware Performance Counter 23 Control, High half



mhpmevent23h is only defined in RV32.

Alias of [mhpmevent23](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.169.1. Attributes

CSR Address	0x737
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.169.2. Format

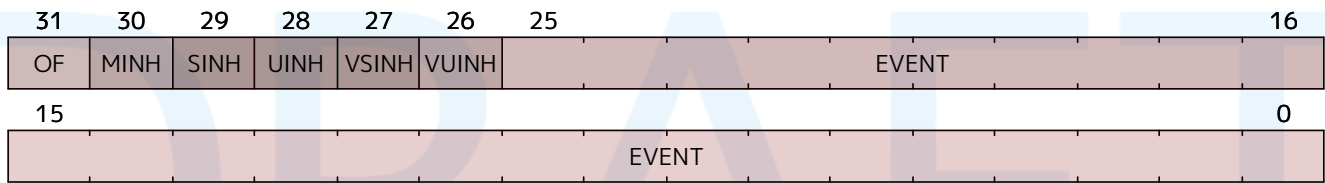


Figure 176. *mhpmevent23h* format

C.170. mhpmevent24

Machine Hardware Performance Counter 24 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.170.1. Attributes

CSR Address	0x338
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.170.2. Format

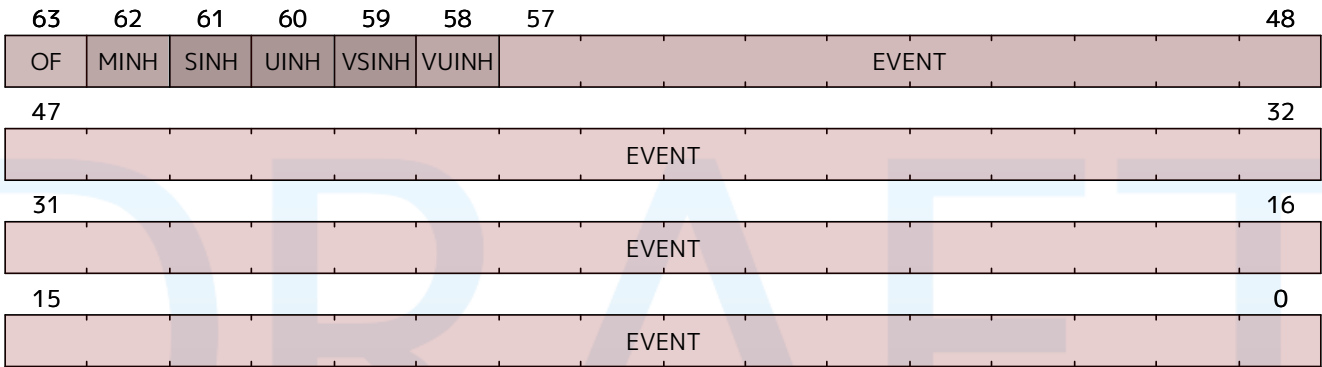


Figure 177. mhpmevent24 format

C.171. mhpmevent24h

Machine Hardware Performance Counter 24 Control, High half

 *mhpmevent24h* is only defined in RV32.

Alias of [mhpmevent24](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.171.1. Attributes

CSR Address	0x738
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.171.2. Format

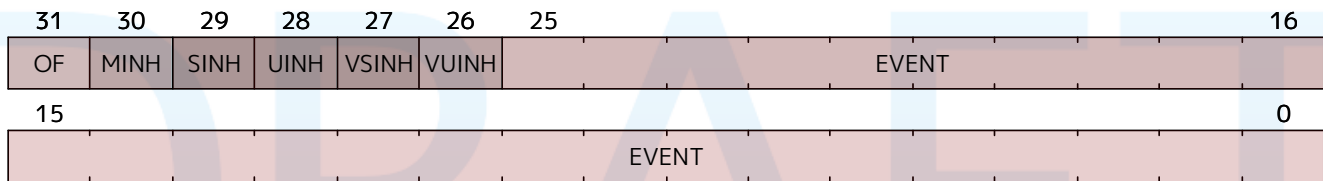


Figure 178. *mhpmevent24h* format

C.172. mhpmevent25

Machine Hardware Performance Counter 25 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.172.1. Attributes

CSR Address	0x339
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.172.2. Format

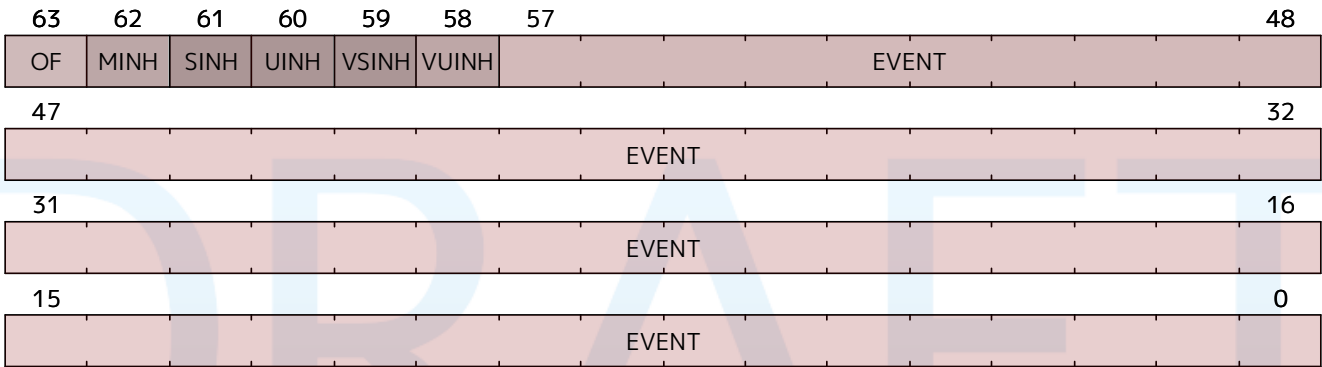


Figure 179. mhpmevent25 format

C.173. mhpmevent25h

Machine Hardware Performance Counter 25 Control, High half

 *mhpmevent25h* is only defined in RV32.

Alias of [mhpmevent25](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.173.1. Attributes

CSR Address	0x739
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.173.2. Format

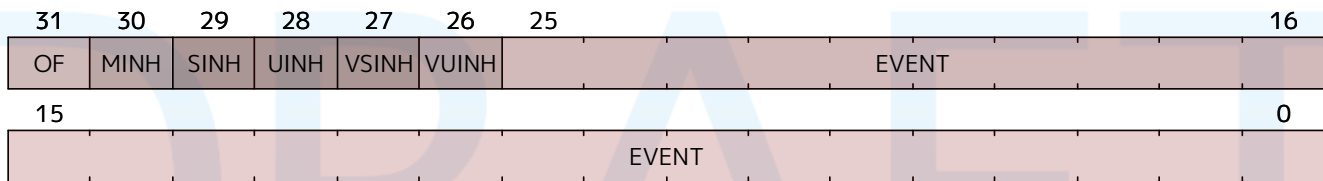


Figure 180. *mhpmevent25h* format

C.174. mhpmevent26

Machine Hardware Performance Counter 26 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.174.1. Attributes

CSR Address	0x33a
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.174.2. Format

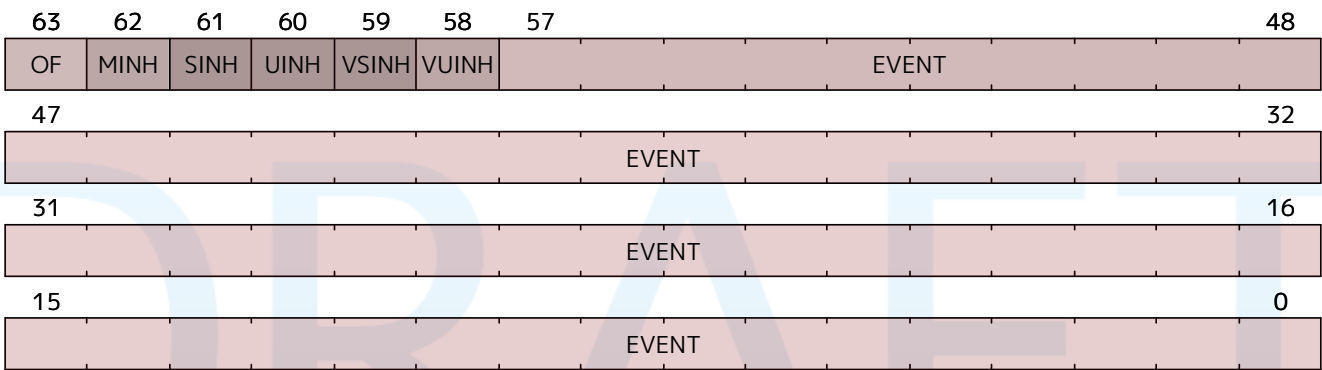


Figure 181. mhpmevent26 format

C.175. mhpmevent26h

Machine Hardware Performance Counter 26 Control, High half

 *mhpmevent26h* is only defined in RV32.

Alias of [mhpmevent26](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.175.1. Attributes

CSR Address	0x73a
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.175.2. Format

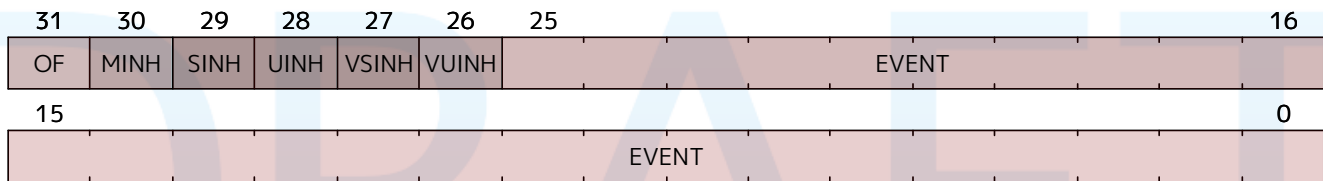


Figure 182. *mhpmevent26h* format

C.176. mhpmevent27

Machine Hardware Performance Counter 27 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.176.1. Attributes

CSR Address	0x33b
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.176.2. Format

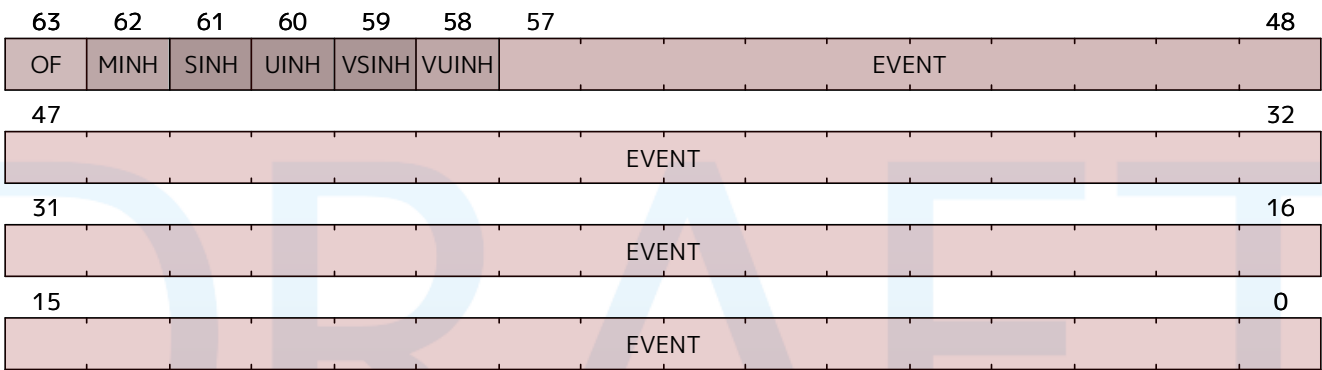


Figure 183. mhpmevent27 format

C.177. mhpmevent27h

Machine Hardware Performance Counter 27 Control, High half

 *mhpmevent27h* is only defined in RV32.

Alias of [mhpmevent27](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.177.1. Attributes

CSR Address	0x73b
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.177.2. Format

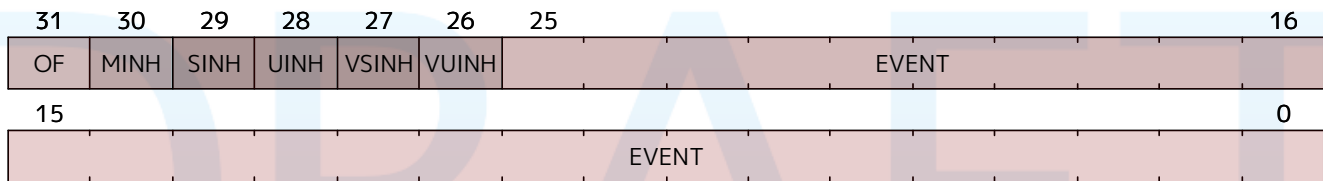


Figure 184. *mhpmevent27h* format

C.178. mhpmevent28

Machine Hardware Performance Counter 28 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.178.1. Attributes

CSR Address	0x33c
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.178.2. Format

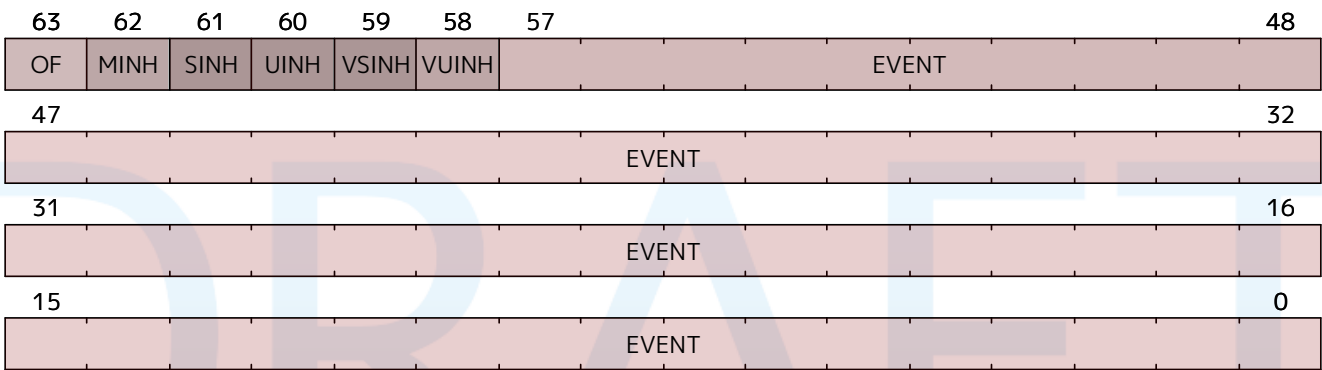


Figure 185. mhpmevent28 format

C.179. mhpmevent28h

Machine Hardware Performance Counter 28 Control, High half

 *mhpmevent28h* is only defined in RV32.

Alias of [mhpmevent28](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.179.1. Attributes

CSR Address	0x73c
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.179.2. Format

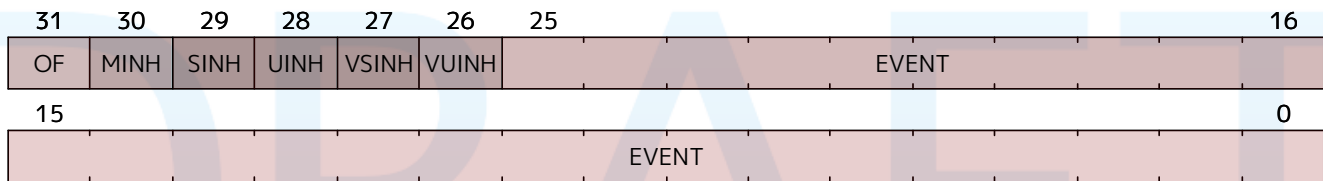


Figure 186. *mhpmevent28h* format

C.180. mhpmevent29

Machine Hardware Performance Counter 29 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.180.1. Attributes

CSR Address	0x33d
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.180.2. Format

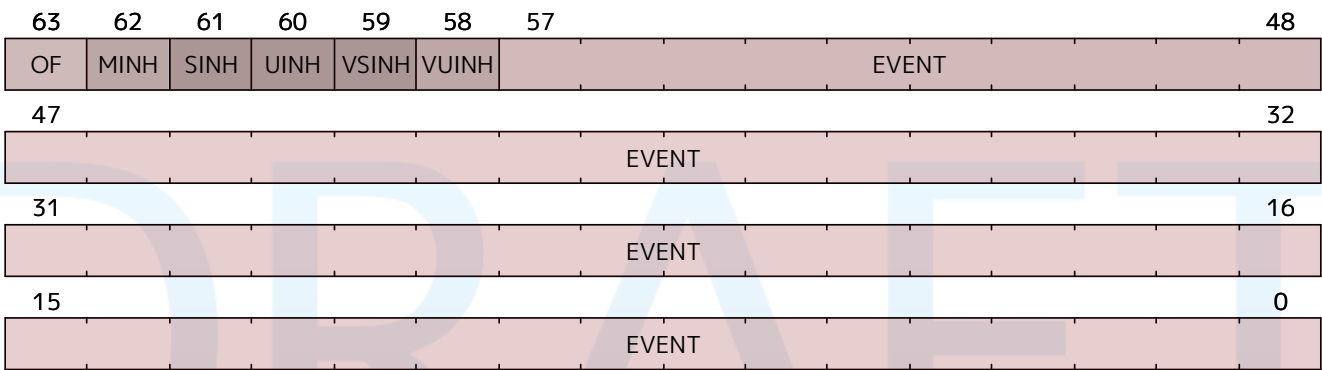


Figure 187. mhpmevent29 format

C.181. mhpmevent29h

Machine Hardware Performance Counter 29 Control, High half

 *mhpmevent29h* is only defined in RV32.

Alias of [mhpmevent29](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.181.1. Attributes

CSR Address	0x73d
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.181.2. Format

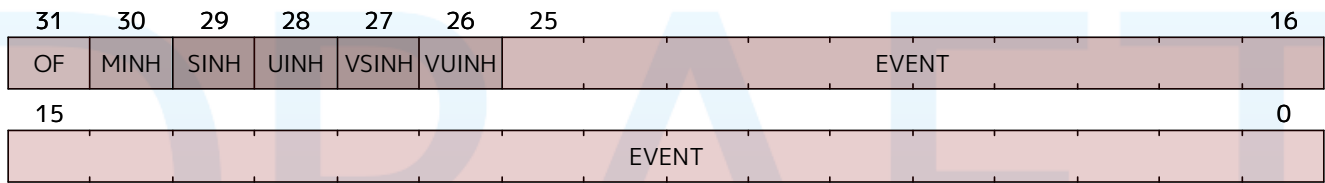


Figure 188. *mhpmevent29h* format

C.182. mhpmevent3

Machine Hardware Performance Counter 3 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.182.1. Attributes

CSR Address	0x323
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.182.2. Format

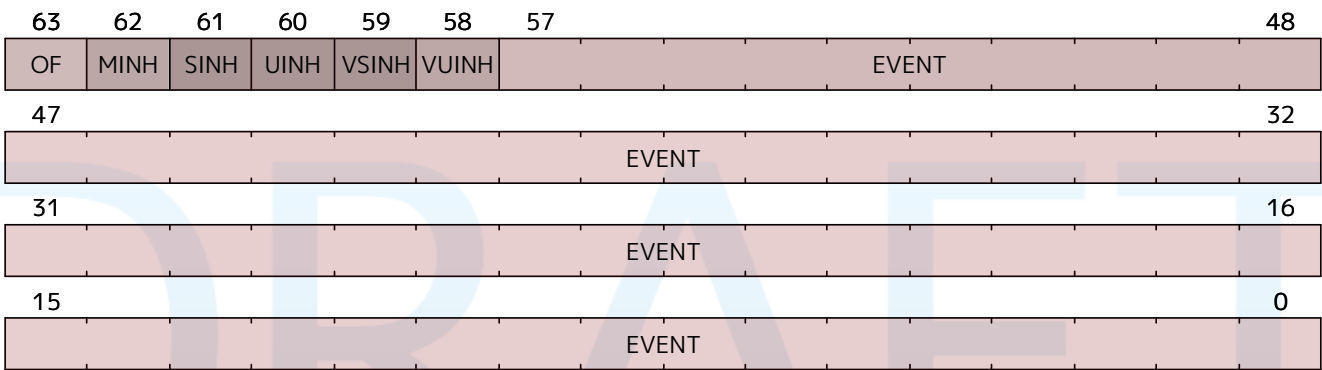


Figure 189. mhpmevent3 format

C.183. mhpmevent30

Machine Hardware Performance Counter 30 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.183.1. Attributes

CSR Address	0x33e
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.183.2. Format

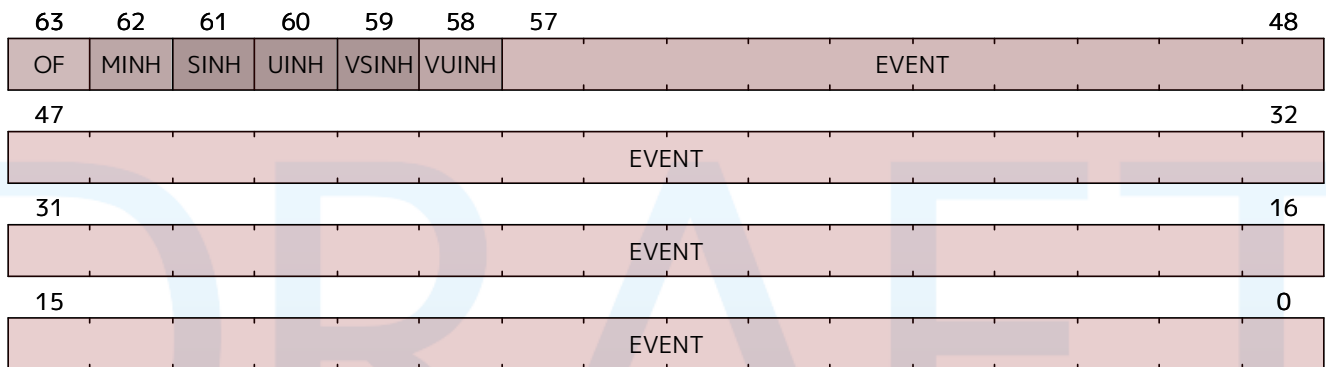


Figure 190. mhpmevent30 format

C.184. mhpmevent30h

Machine Hardware Performance Counter 30 Control, High half

 *mhpmevent30h* is only defined in RV32.

Alias of [mhpmevent30](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.184.1. Attributes

CSR Address	0x73e
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.184.2. Format

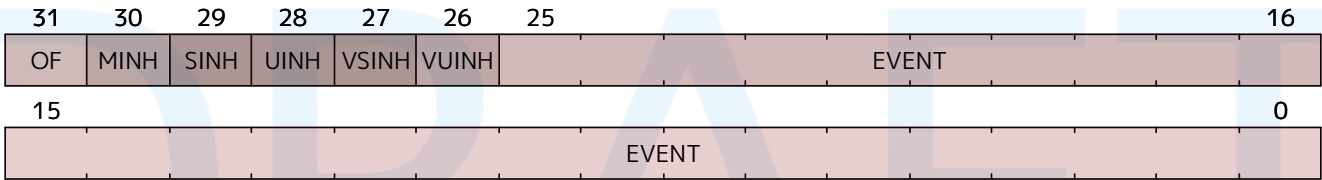


Figure 191. *mhpmevent30h* format

C.185. mhpmevent31

Machine Hardware Performance Counter 31 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.185.1. Attributes

CSR Address	0x33f
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.185.2. Format

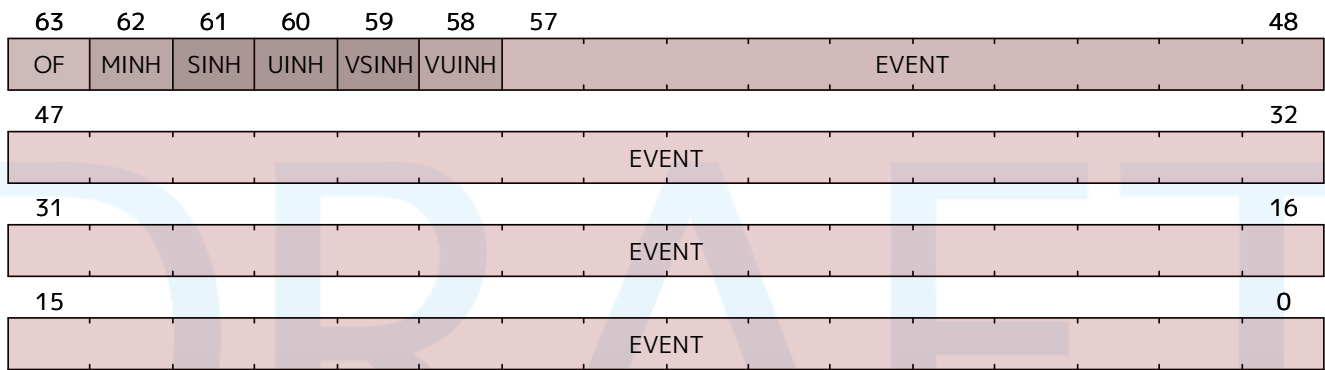


Figure 192. mhpmevent31 format

C.186. mhpmevent31h

Machine Hardware Performance Counter 31 Control, High half

 *mhpmevent31h* is only defined in RV32.

Alias of [mhpmevent31](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.186.1. Attributes

CSR Address	0x73f
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.186.2. Format

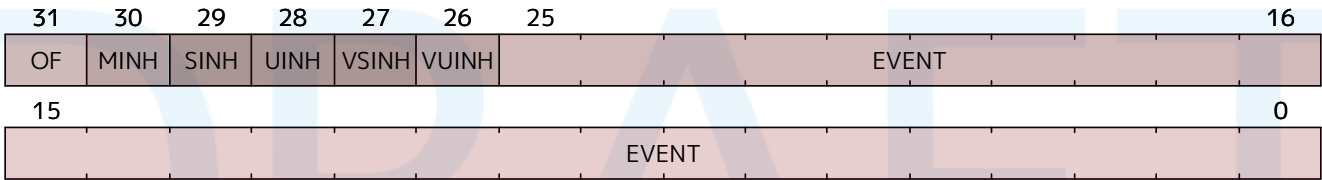


Figure 193. *mhpmevent31h* format

C.187. mhpmevent3h

Machine Hardware Performance Counter 3 Control, High half

 *mhpmevent3h* is only defined in RV32.

Alias of [mhpmevent3](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.187.1. Attributes

CSR Address	0x723
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.187.2. Format

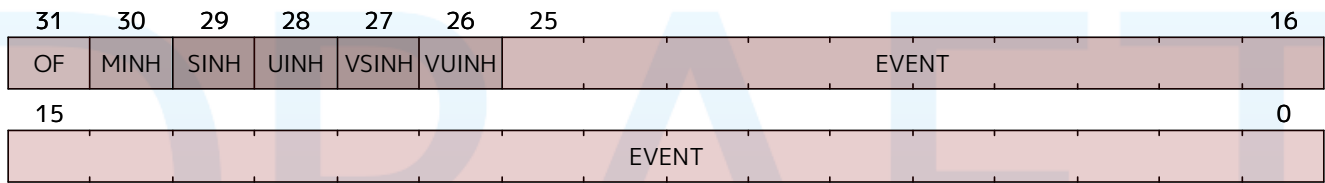


Figure 194. *mhpmevent3h* format

C.188. mhpmevent4

Machine Hardware Performance Counter 4 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.188.1. Attributes

CSR Address	0x324
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.188.2. Format

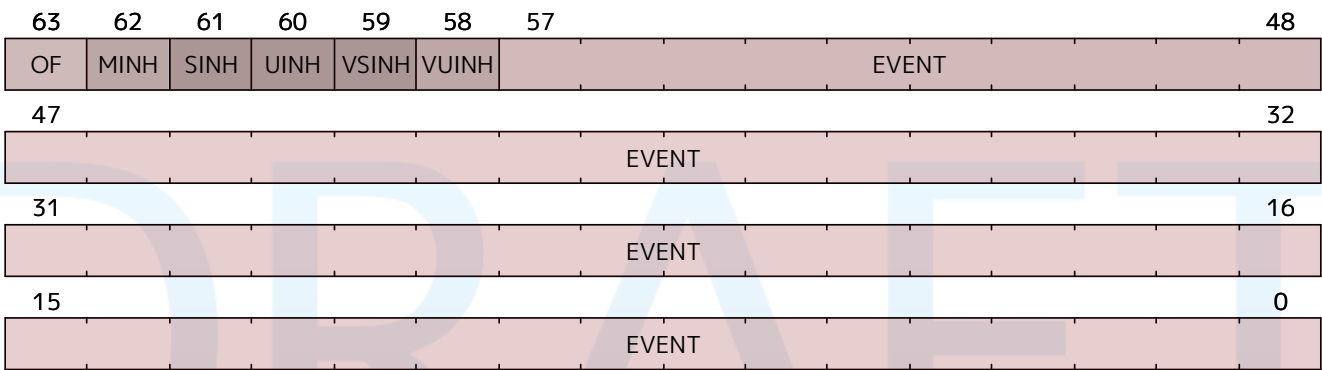



Figure 195. mhpmevent4 format

C.189. mhpmevent4h

Machine Hardware Performance Counter 4 Control, High half

 *mhpmevent4h* is only defined in RV32.

Alias of [mhpmevent4](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.189.1. Attributes

CSR Address	0x724
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.189.2. Format

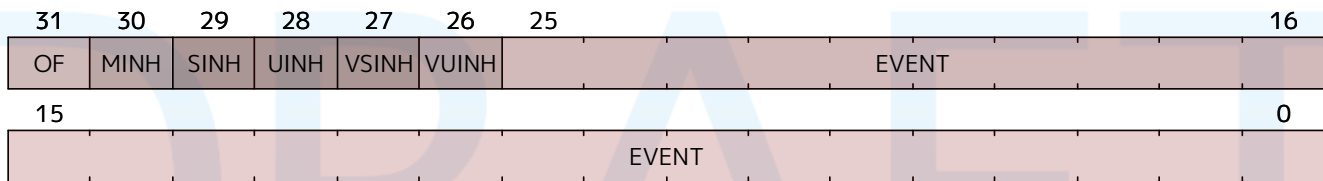


Figure 196. *mhpmevent4h* format

C.190. mhpmevent5

Machine Hardware Performance Counter 5 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.190.1. Attributes

CSR Address	0x325
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.190.2. Format

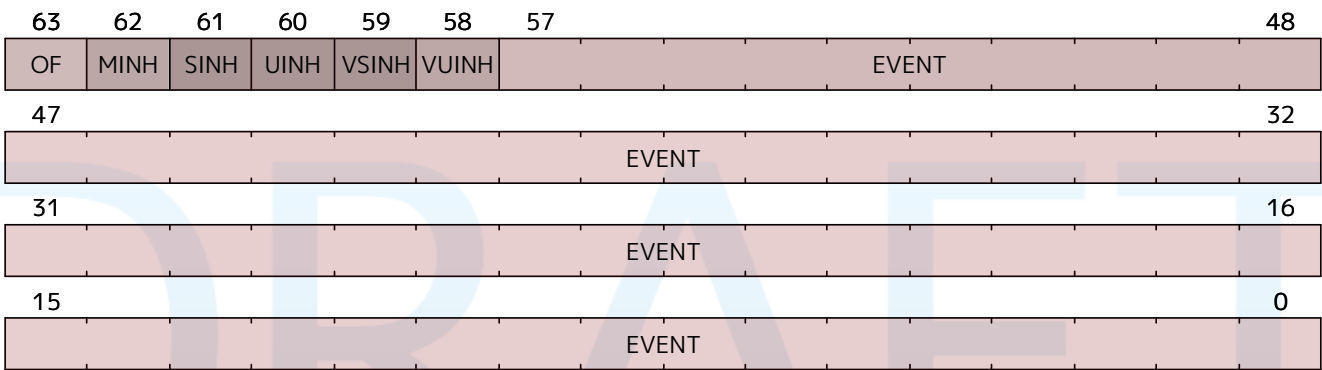


Figure 197. mhpmevent5 format

C.191. mhpmevent5h

Machine Hardware Performance Counter 5 Control, High half



mhpmevent5h is only defined in RV32.

Alias of [mhpmevent5](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.191.1. Attributes

CSR Address	0x725
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.191.2. Format

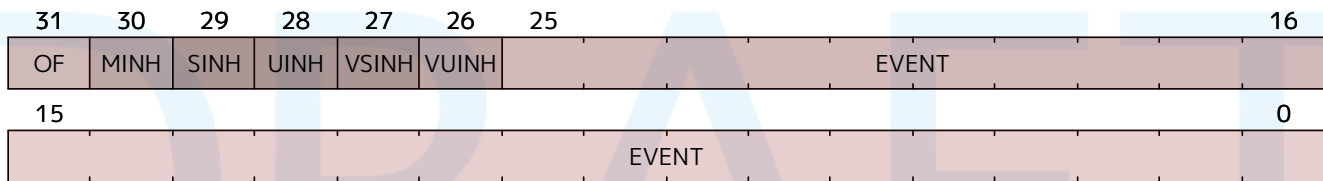


Figure 198. *mhpmevent5h* format

C.192. mhpmevent6

Machine Hardware Performance Counter 6 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.192.1. Attributes

CSR Address	0x326
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.192.2. Format

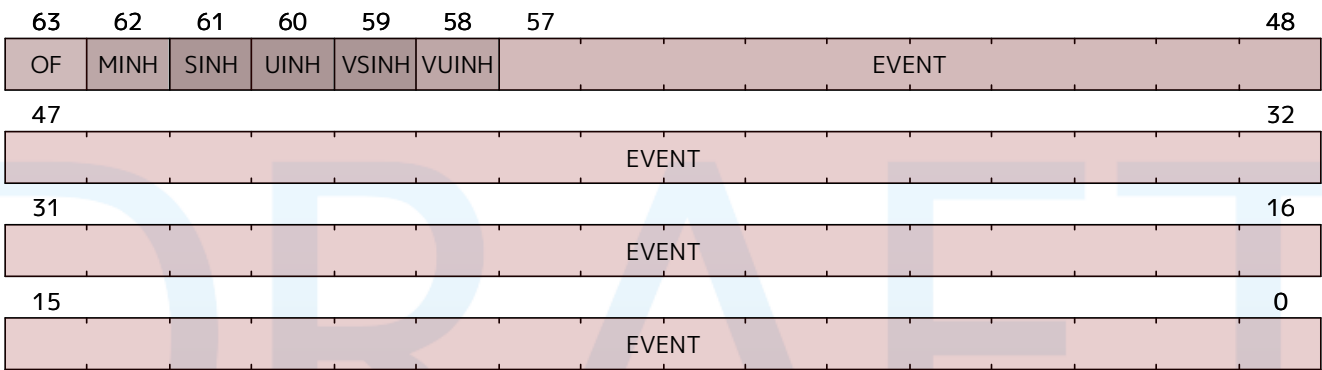


Figure 199. mhpmevent6 format

C.193. mhpmevent6h

Machine Hardware Performance Counter 6 Control, High half

i | *mhpmevent6h* is only defined in RV32.

Alias of *mhpmevent6*[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of *mhpmevent#{hpm_num}*.

C.193.1. Attributes

CSR Address	0x726
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.193.2. Format

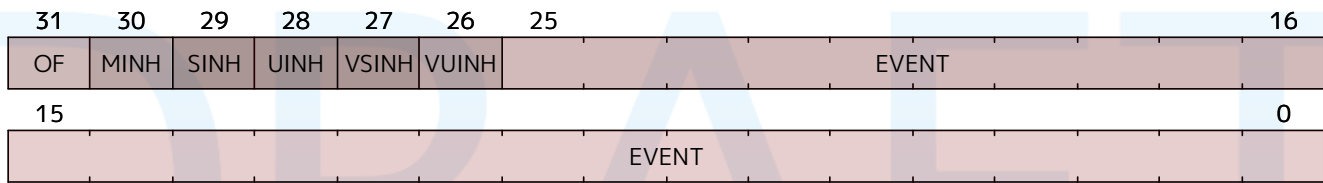


Figure 200. mhpmevent6h format

C.194. mhpmevent7

Machine Hardware Performance Counter 7 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.194.1. Attributes

CSR Address	0x327
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.194.2. Format

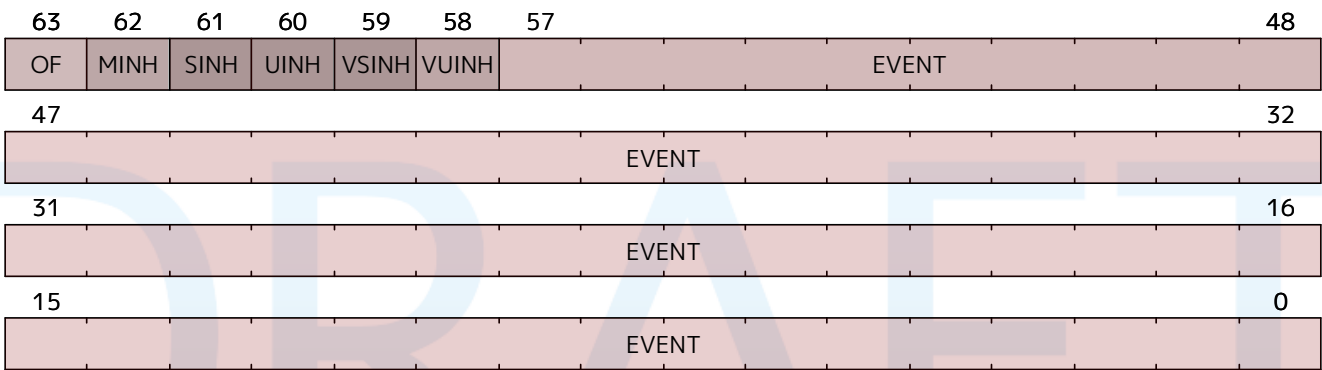


Figure 201. mhpmevent7 format

C.195. mhpmevent7h

Machine Hardware Performance Counter 7 Control, High half



mhpmevent7h is only defined in RV32.

Alias of [mhpmevent7](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.195.1. Attributes

CSR Address	0x727
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.195.2. Format

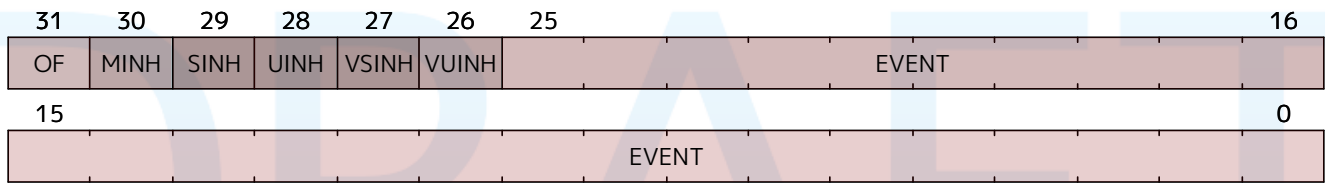


Figure 202. mhpmevent7h format

C.196. mhpmevent8

Machine Hardware Performance Counter 8 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.196.1. Attributes

CSR Address	0x328
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.196.2. Format

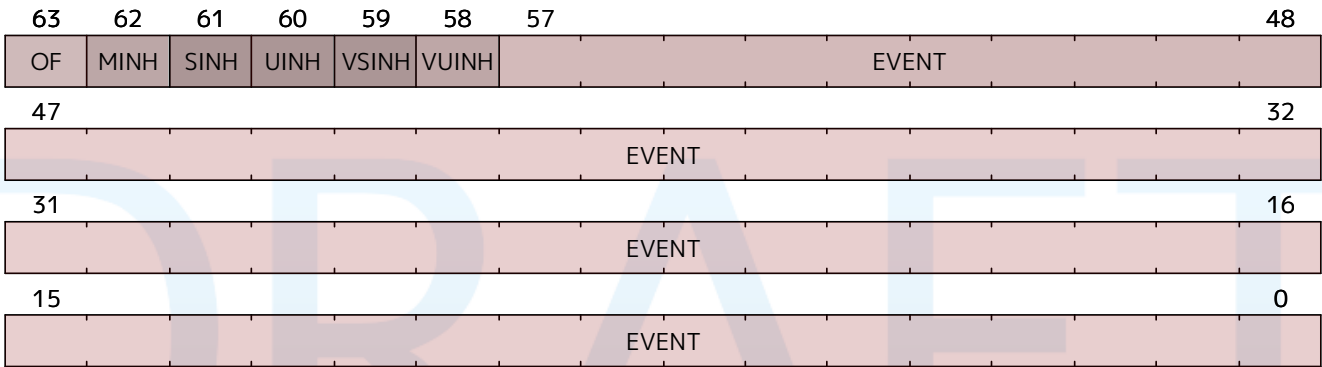


Figure 203. mhpmevent8 format

C.197. mhpmevent8h

Machine Hardware Performance Counter 8 Control, High half



mhpmevent8h is only defined in RV32.

Alias of [mhpmevent8](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.197.1. Attributes

CSR Address	0x728
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.197.2. Format

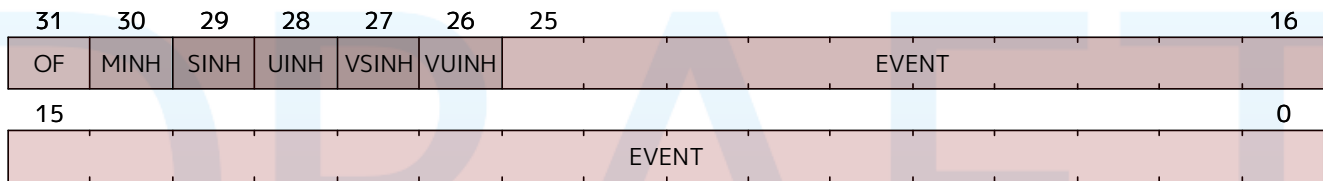


Figure 204. mhpmevent8h format

C.198. mhpmevent9

Machine Hardware Performance Counter 9 Control

Programmable hardware performance counter event selector <% if ext?(:Sscofpmf) %> and overflow/filtering control<% end %>

C.198.1. Attributes

CSR Address	0x329
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.198.2. Format

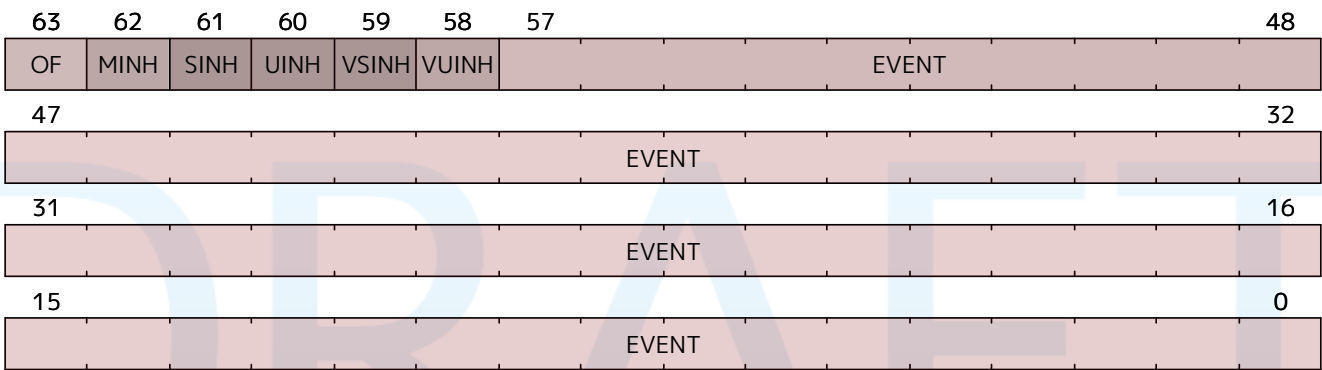


Figure 205. mhpmevent9 format

C.199. mhpmevent9h

Machine Hardware Performance Counter 9 Control, High half

 *mhpmevent9h* is only defined in RV32.

Alias of [mhpmevent9](#)[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

C.199.1. Attributes

CSR Address	0x729
Defining extension	Sscofpmf >= 0
Length	32-bit
Privilege Mode	M

C.199.2. Format

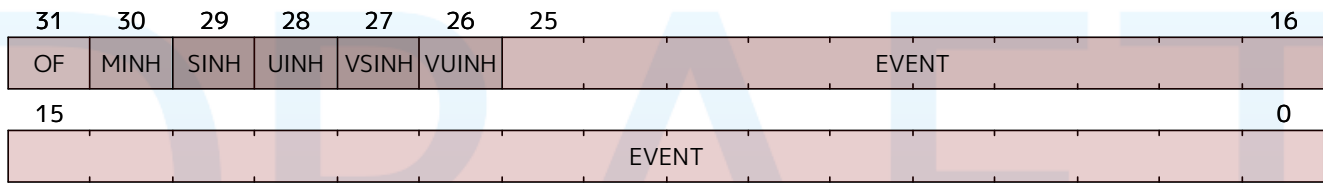


Figure 206. *mhpmevent9h* format

C.200. mideleg

Machine Interrupt Delegation

Controls exception delegation from M-mode to HS/S-mode

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the **MRET** instruction. To increase performance, implementations can provide individual read/write bits within **mideleg** to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level.

In harts with S-mode, the **mideleg** register must exist, and setting a bit **mideleg** will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler `<%- if ext?:(H) -%>` (which could further be delegated to VS-mode through **hideleg**) `<%- end -%>` . `<%- if ext?:(S, ">1.9.1") -%>` In harts without S-mode, the **mideleg** register should not exist.



*In versions 1.9.1 and earlier , this register existed but was hardwired to zero in M-mode only, or M/U without N harts. There is no reason to require they return zero in those cases, as the **misa** register indicates whether they exist.*

`<%- else -%>` In harts without S-mode, the **mideleg** register is read-only zero. `<%- end -%>`

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in **mideleg** to see which bit positions hold a one.



*Version 1.11 and earlier prohibited having any bits of **mideleg** be read-only one. Platform standards may always add such restrictions.*

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal-instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may be taken horizontally. Using the same example, if M-mode has delegated illegal-instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode.

Delegated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting **mideleg**[5], STIs will not be taken when executing in M-mode. By contrast, if **mideleg**[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

mideleg holds trap delegation bits for individual interrupts, with the layout of bits matching those in the **mip** register (i.e., STIP interrupt delegation control is located in bit 5).

For exceptions that cannot occur in less privileged modes, the corresponding **medeleg** bits should be read-only zero. In particular, **medeleg**[11] is read-only zero.

`<%- if ext?:(S, mdbltrap) || ext?:(S, sdbltrap) -%>` The **medeleg**[16] is read-only zero as double trap is not delegatable. `<%- end -%>`

C.200.1. Attributes

CSR Address	0x303
-------------	-------

Defining extension	S > 1.9.1, M < 1.0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.200.2. Format

This CSR format changes dynamically with XLEN.

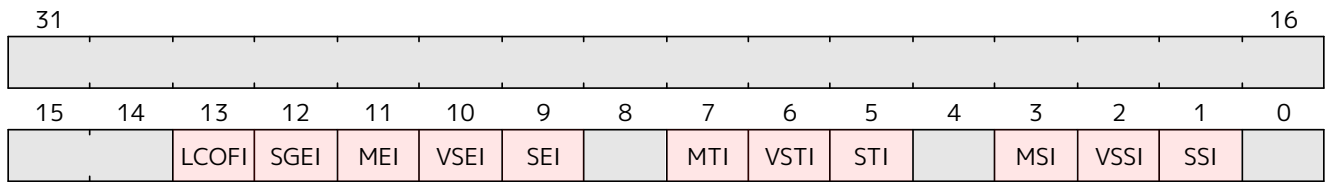


Figure 207. mideleg Format when CSR[misa].MXL == 0

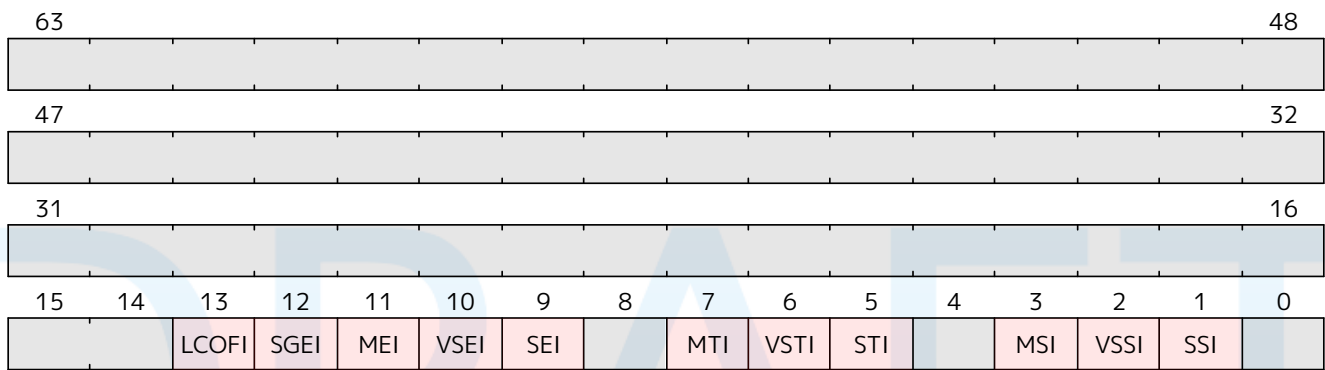


Figure 208. mideleg Format when CSR[misa].MXL == 1

C.201. mie

Machine Interrupt Enable

Per-type interrupt enables.

For a detailed description of interrupt handling, see [link_to\(:section, 'sec:interrupts'\) %>](#).

The [mie](#) register is an MXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number i (as reported in CSR [mcause](#)) corresponds with bit i in [mie](#). Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.



Interrupts designated for platform use may be designated for custom use at the platform's discretion.

An interrupt i will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true:

- either the current privilege mode is M and the MIE bit in the [mstatus](#) register is set, or the current privilege mode has less privilege than M-mode;
- bit i is set in both [mip](#) and [mie](#);
- if register [mideleg](#) exists, bit i is not set in [mideleg](#).

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in [mip](#), and must also be evaluated immediately following the execution of an xRET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including [mip](#), [mie](#), [mstatus](#), and [mideleg](#)).

Interrupts to M-mode take priority over any interrupts to lower privilege modes.

A bit in [mie](#) must be writable if the corresponding interrupt can ever become pending. Bits of [mie](#) that are not writable must be read-only zero.



The machine-level interrupt registers handle a few root interrupt sources which are assigned a fixed service priority for simplicity, while separate external interrupt controllers can implement a more complex prioritization scheme over a much larger set of interrupts that are then muxed into the machine-level interrupt sources.

The non-maskable interrupt is not made visible via the [mip](#) register as its presence is implicitly known when executing the NMI trap handler.

If supervisor mode is implemented, bits [mip](#).SEIP and [mie](#).SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in [mip](#), and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software-writable SEIP bit and the signal from the external interrupt controller. When [mip](#) is read with a CSR instruction, the value of the SEIP bit returned in the **rd** destination register is the logical-OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRRS or CSRRC instruction.



For example, if we name the software-writable SEIP bit *B* and the signal from the external interrupt controller *E*, then if `csrrs t0, mip, t1` is executed, `t0[9]` is written with `B || E`, then *B* is written with `B || t1[9]`. If `csrrw t0, mip, t1` is executed, then `t0[9]` is written with `B || E`, and *B* is simply written with `t1[9]`. In neither case does *B* depend upon *E*.

The SEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts cleanly, without losing any real external interrupts. The behavior of the CSR instructions is slightly modified from regular CSR accesses as a result.

If supervisor mode is implemented, bits `mip.STIP` and `mie.STIE` are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. STIP is writable in `mip`, and may be written by M-mode software to deliver timer interrupts to S-mode.

If supervisor mode is implemented, bits `mip.SSIP` and `mie.SSIE` are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in `mip` and may also be set to 1 by a platform-specific interrupt controller.

<%- if ext?(:Sscofpmf) -%> Bits `mip.LCOFIP` and `mie.LCOFIE` are the interrupt-pending and interrupt-enable bits for local counter-overflow interrupts. LCOFIP is read-write in `mip` and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the `mhpmeventn.OF` bits being set. If the Sscofpmf extension is not implemented, `mip.LCOFIP` and `mie.LCOFIE` are read-only zeros. <%- end -%>

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

The machine-level interrupt fixed-priority ordering rules were developed with the following rationale.

Interrupts for higher privilege modes must be serviced before interrupts for lower privilege modes to support preemption.

The platform-specific machine-level interrupt sources in bits 16 and above have platform-specific priority, but are typically chosen to have the highest service priority to support very fast local vectored interrupts.



External interrupts are handled before internal (timer/software) interrupts as external interrupts are usually generated by devices that might require low interrupt service times.

Software interrupts are handled before internal timer interrupts, because internal timer interrupts are usually intended for time slicing, where time precision is less important, whereas software interrupts are used for inter-processor messaging. Software interrupts can be avoided when high-precision timing is required, or high-precision timer interrupts can be routed via a different interrupt path. Software interrupts are located in the lowest four bits of `mip` as these are often written by software, and this position allows the use of a single CSR instruction with a five-bit immediate.

Restricted views of the `mip` and `mie` registers appear as the `sip` and `sie` registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the `mideleg` register, it becomes visible in the `sip` register and is maskable using the `sie` register. Otherwise, the corresponding bits in `sip` and `sie` are read-only zero.

C.201.1. Attributes

CSR Address	0x304
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.201.2. Format

This CSR format changes dynamically with XLEN.

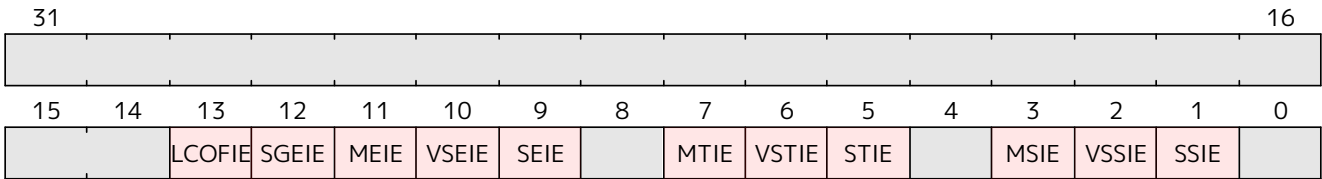


Figure 209. mie Format when CSR[misa].MXL == 0

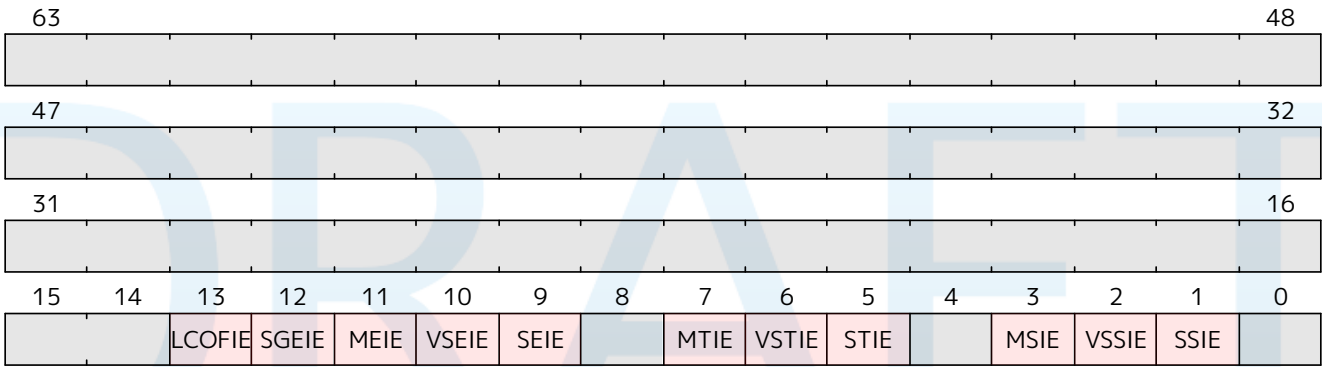


Figure 210. mie Format when CSR[misa].MXL == 1

C.202. mimpid

Machine Implementation ID

Reports the vendor-specific implementation ID.

The [mimpid](#) CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.



The format of this field is left to the provider of the architecture source code, but will often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value can be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

C.202.1. Attributes

CSR Address	0xf13
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.202.2. Format

This CSR format changes dynamically with XLEN.

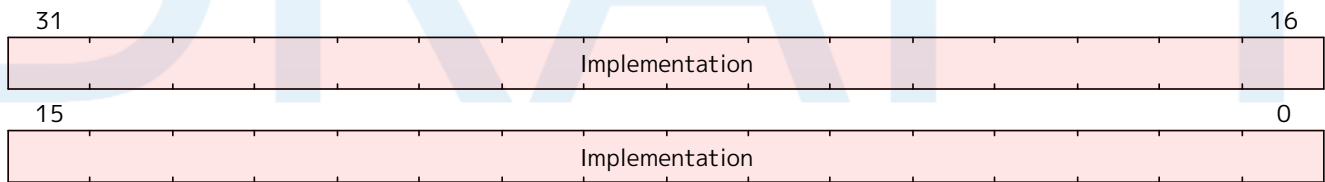


Figure 211. mimpid Format when CSR[misa].MXL == 0

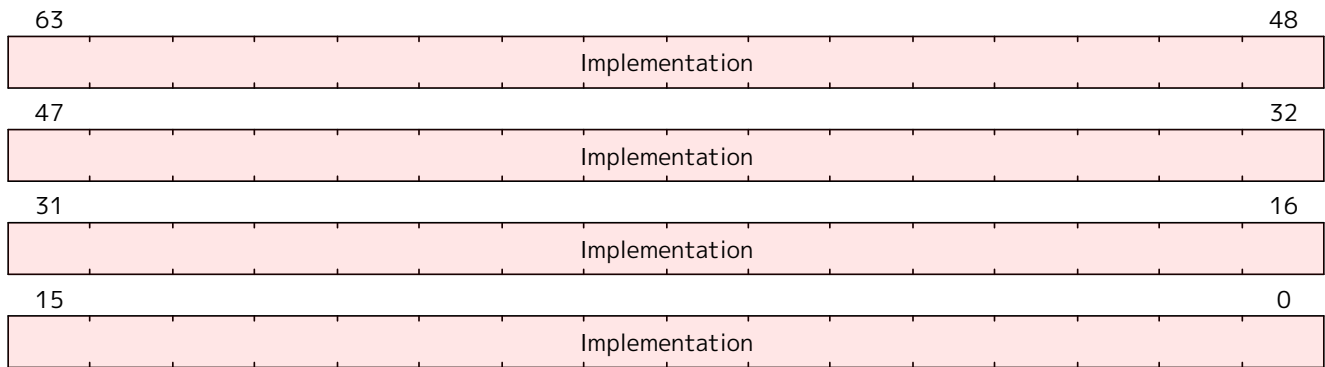


Figure 212. mimpid Format when CSR[misa].MXL == 1

C.203. minstret

Machine Instructions Retired Counter

Counts the number of instructions retired by this hart from some arbitrary start point in the past.



Instructions that cause synchronous exceptions, including *ecall* and *ebreak*, are not considered to retire and hence do not increment the *minstret* CSR.

C.203.1. Attributes

CSR Address	Oxb02
Defining extension	Zicntr >= 0
Length	64-bit
Privilege Mode	M

C.203.2. Format

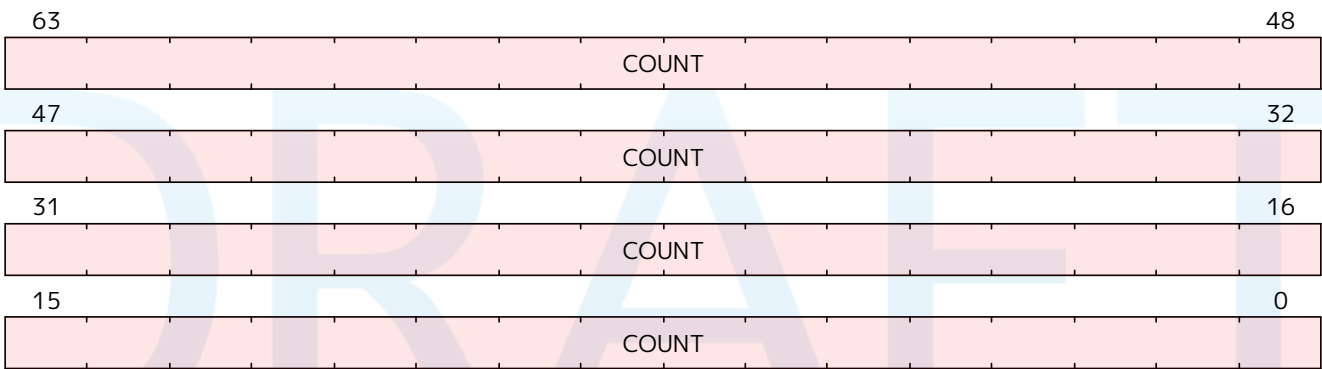


Figure 213. minstret format

C.204. minstreth

Machine Instructions Retired Counter

Upper half of 64-bit instructions retired counters.

See [minstret](#) for details.

C.204.1. Attributes

CSR Address	0xb02
Defining extension	Zicntr >= 0
Length	32-bit
Privilege Mode	M

C.204.2. Format

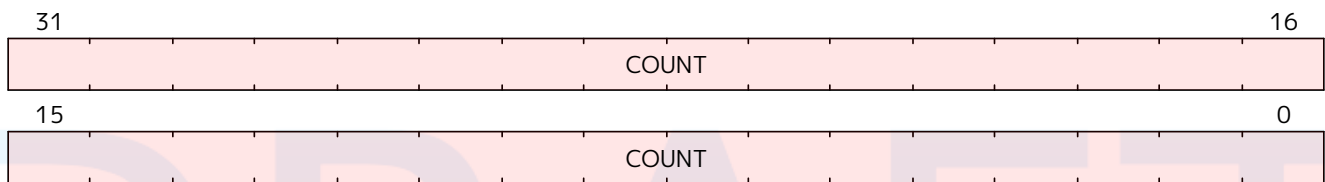


Figure 214. minstreth format

C.205. mip

Machine Interrupt Pending

Machine Interrupt Pending bits

C.205.1. Attributes

CSR Address	0x344
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.205.2. Format

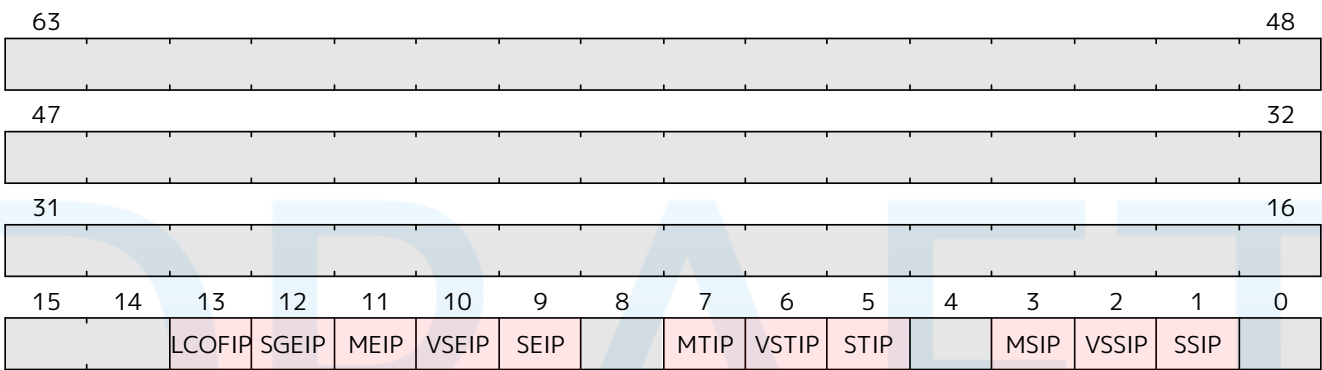


Figure 215. mip format

C.206. misa

Machine ISA Control

Reports the XLEN and "major" extensions supported by the ISA.

C.206.1. Attributes

CSR Address	0x301
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.206.2. Format



Figure 216. misa format

C.207. mscratch

Machine Scratch Register

Scratch register for software use. Bits are not interpreted by hardware.

C.207.1. Attributes

CSR Address	0x340
Defining extension	U >= 0
Length	64-bit
Privilege Mode	M

C.207.2. Format

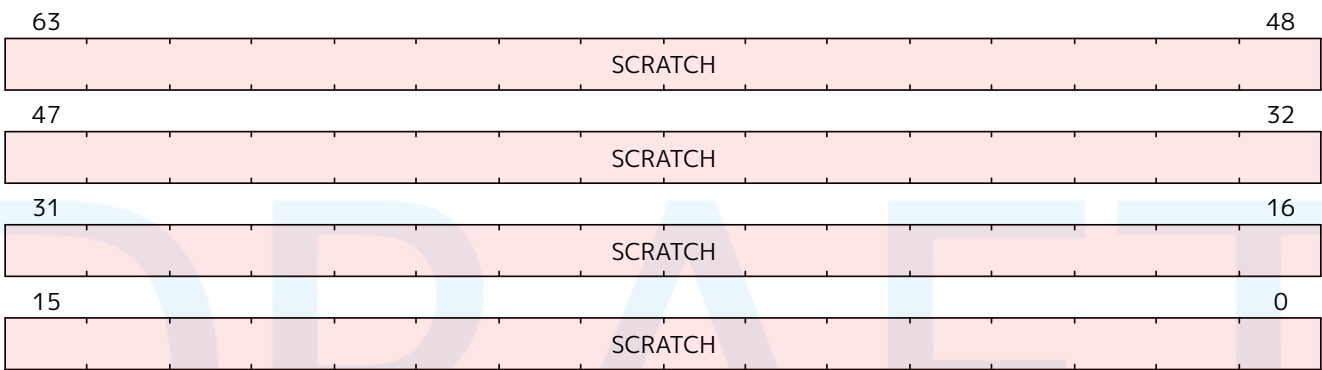


Figure 217. mscratch format

C.208. mseccfg

Machine Security Configuration

Machine Security Configuration

C.208.1. Attributes

CSR Address	0x747
Defining extension	M >= 0
Length	64-bit
Privilege Mode	M

C.208.2. Format

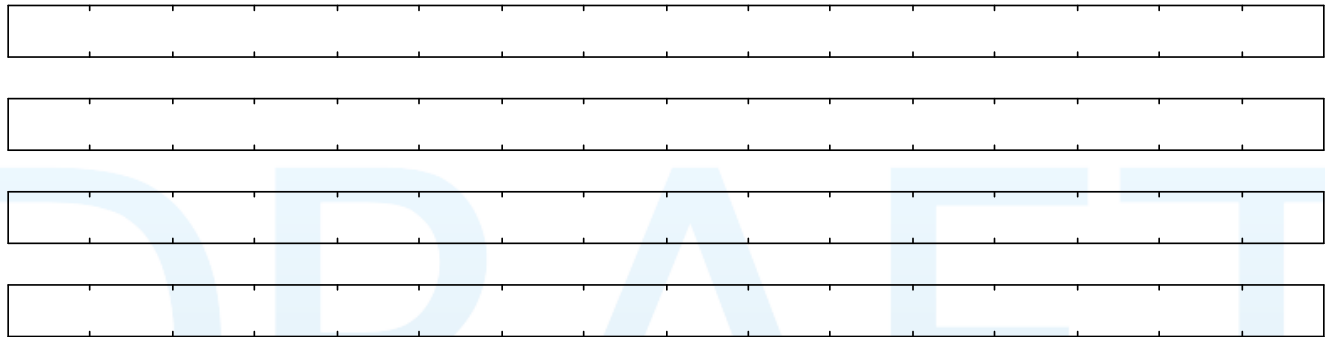


Figure 218. mseccfg format

C.209. mstatus

Machine Status

The mstatus register tracks and controls the hart’s current operating state.

C.209.1. Attributes

CSR Address	0x300
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.209.2. Format

This CSR format changes dynamically with XLEN.

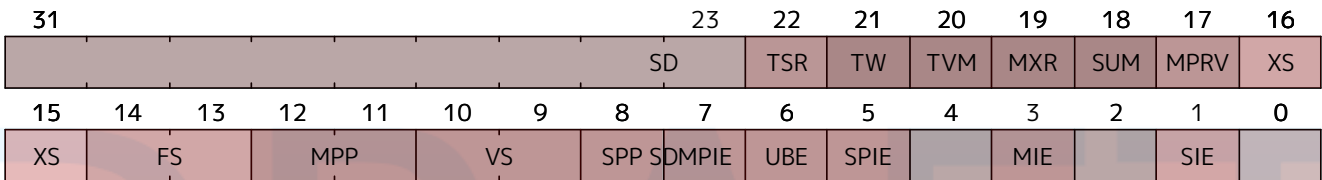


Figure 219. mstatus Format when CSR[misa].MXL == 0

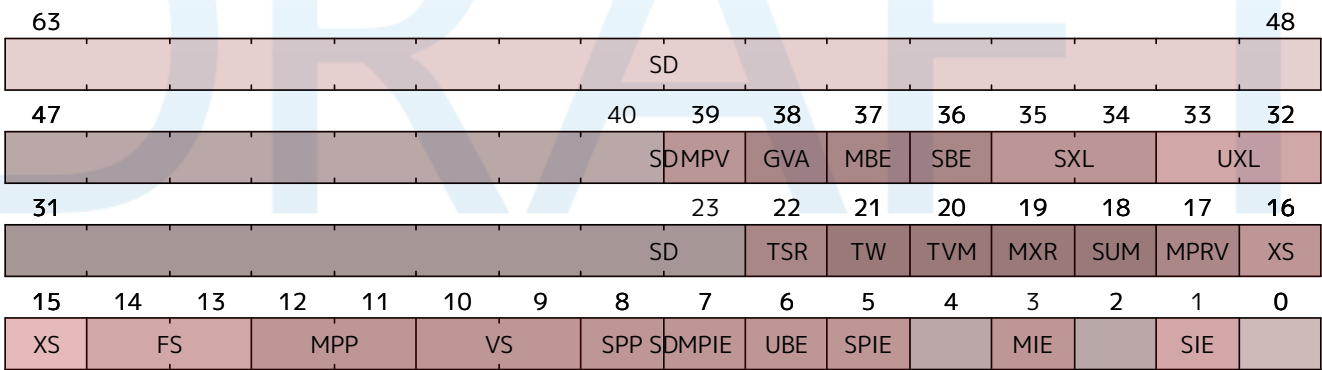


Figure 220. mstatus Format when CSR[misa].MXL == 1

C.210. mstatush

Machine Status High



msatush is only defined in RV32.

The mstatus register tracks and controls the hart’s current operating state.

C.210.1. Attributes

CSR Address	0x310
Defining extension	I \geq 0
Length	32-bit
Privilege Mode	M

C.210.2. Format

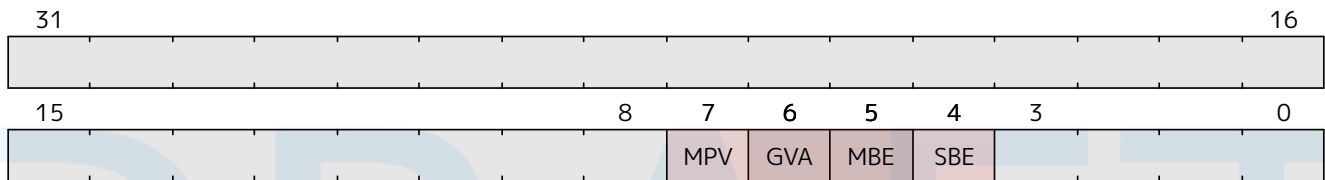


Figure 221. mstatush format

C.211. mtval

Machine Trap Value

Holds trap-specific information

C.211.1. Attributes

CSR Address	0x343
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.211.2. Format

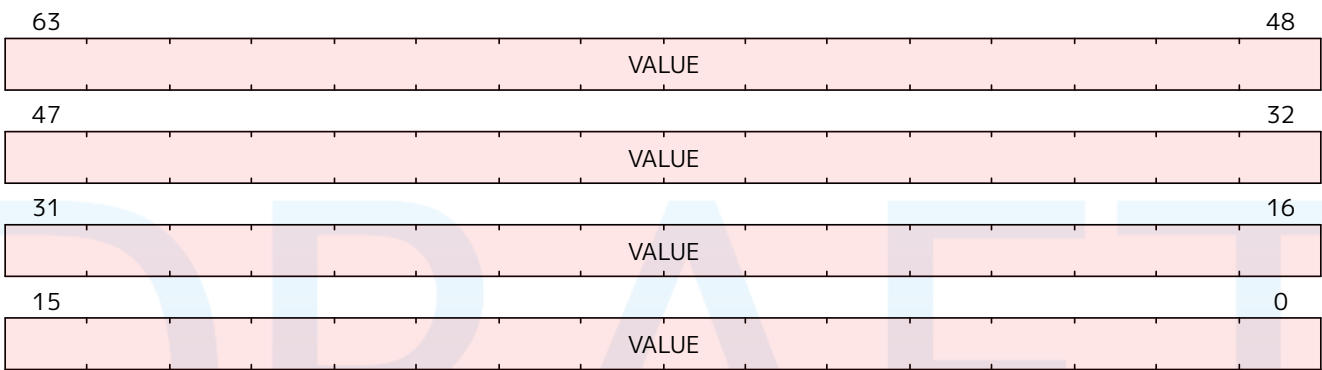


Figure 222. mtval format

C.212. mtvec

Machine Trap Vector Control

Controls where traps jump.

C.212.1. Attributes

CSR Address	0x305
Defining extension	I >= 0
Length	64-bit
Privilege Mode	M

C.212.2. Format

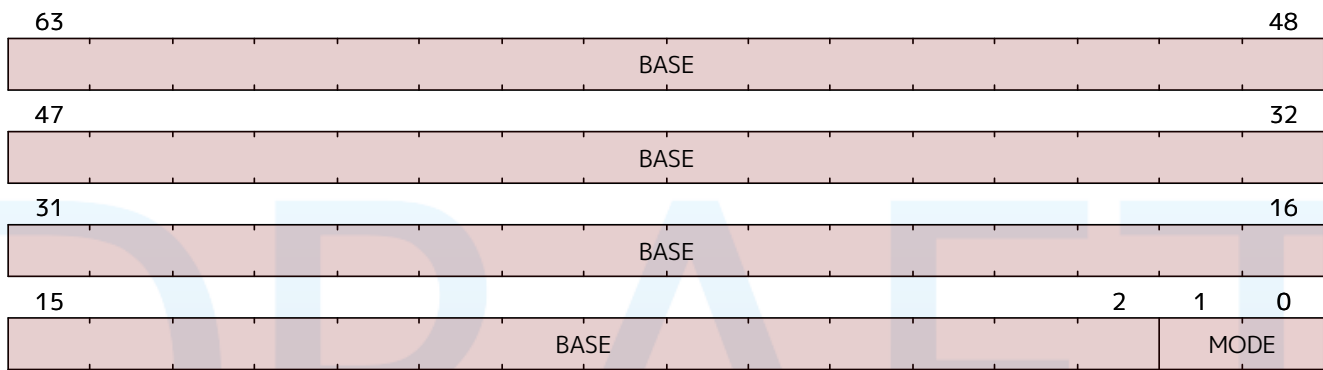


Figure 223. mtvec format

C.213. mvendorid

Machine Vendor ID

Reports the JEDEC manufacturer ID of the core.

C.213.1. Attributes

CSR Address	0xf11
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.213.2. Format

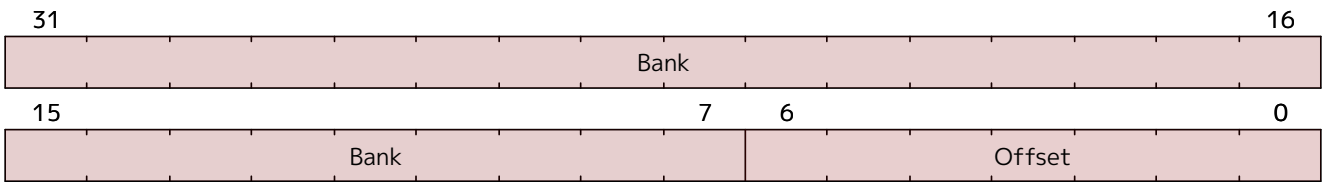


Figure 224. mvendorid format

C.214. pmpaddr0

PMP Address 0

PMP entry address

C.214.1. Attributes

CSR Address	0x3b0
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.214.2. Format

This CSR format changes dynamically with XLEN.

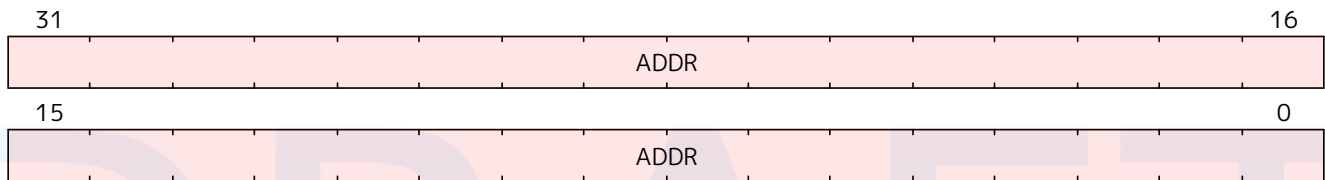


Figure 225. pmpaddr0 Format when CSR[misa].MXL == 0

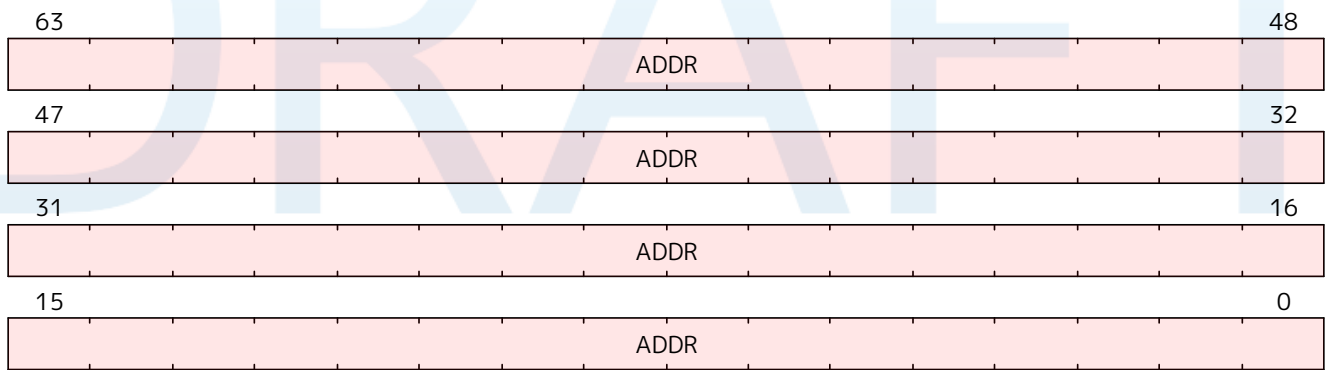


Figure 226. pmpaddr0 Format when CSR[misa].MXL == 1

C.215. pmpaddr1

PMP Address 1

PMP entry address

C.215.1. Attributes

CSR Address	0x3b1
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.215.2. Format

This CSR format changes dynamically with XLEN.

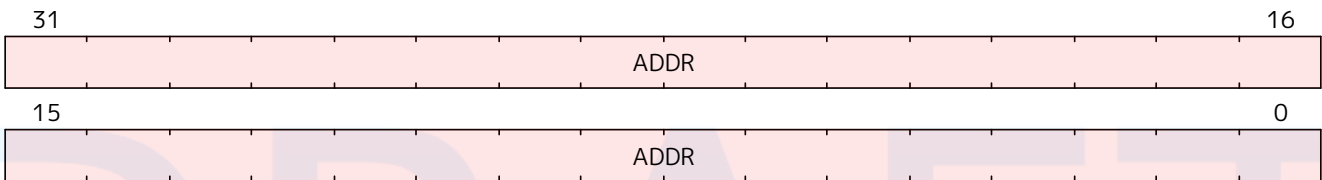


Figure 227. pmpaddr1 Format when CSR[misa].MXL == 0

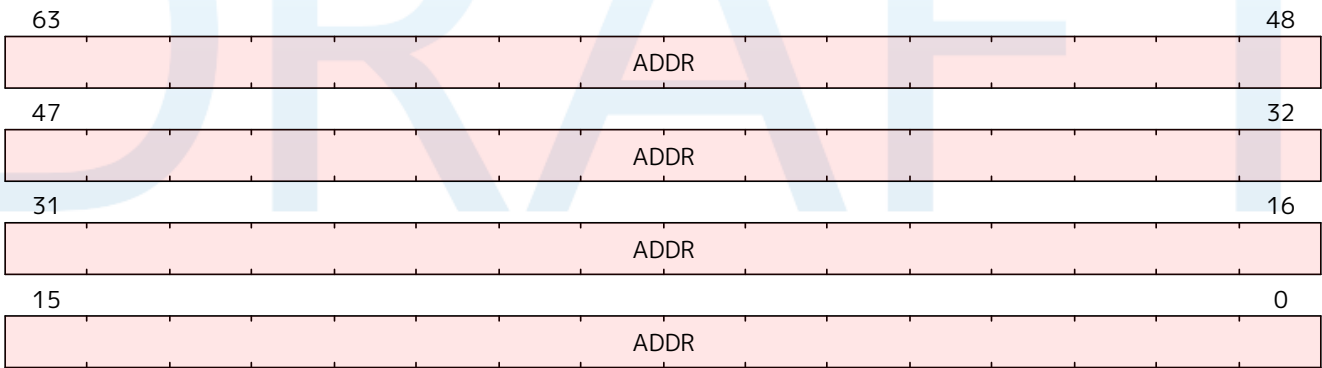


Figure 228. pmpaddr1 Format when CSR[misa].MXL == 1

C.216. pmpaddr10

PMP Address 10

PMP entry address

C.216.1. Attributes

CSR Address	0x3ba
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.216.2. Format

This CSR format changes dynamically with XLEN.

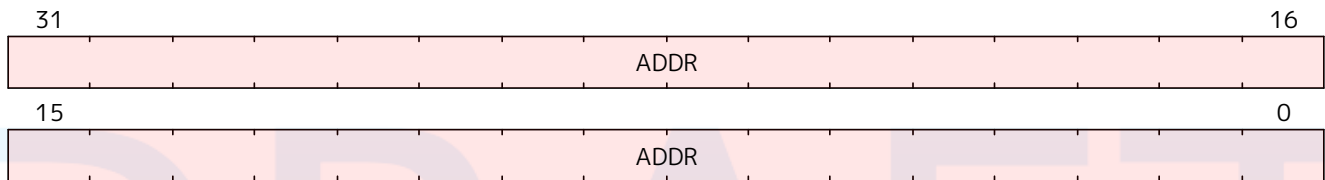


Figure 229. pmpaddr10 Format when CSR[misa].MXL == 0

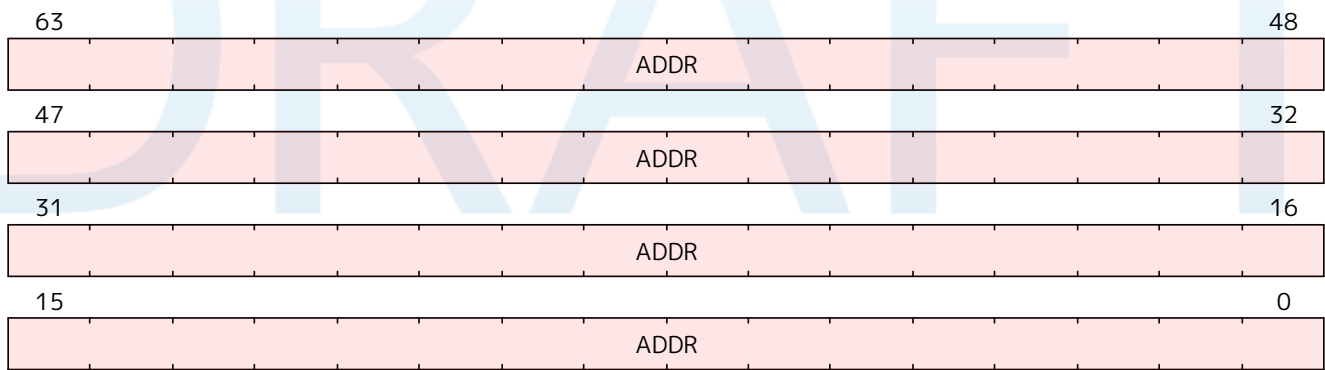


Figure 230. pmpaddr10 Format when CSR[misa].MXL == 1

C.217. pmpaddr11

PMP Address 11

PMP entry address

C.217.1. Attributes

CSR Address	0x3bb
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.217.2. Format

This CSR format changes dynamically with XLEN.

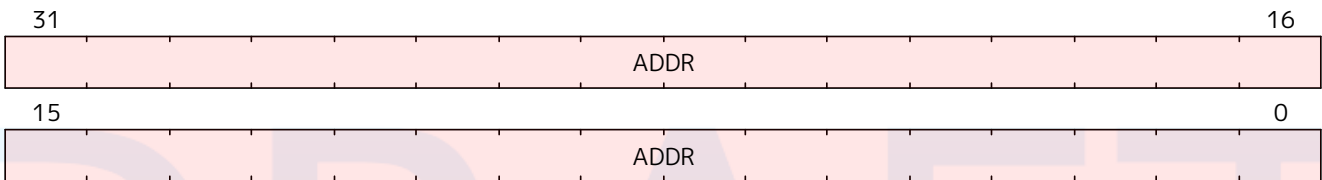


Figure 231. pmpaddr11 Format when CSR[misa].MXL == 0

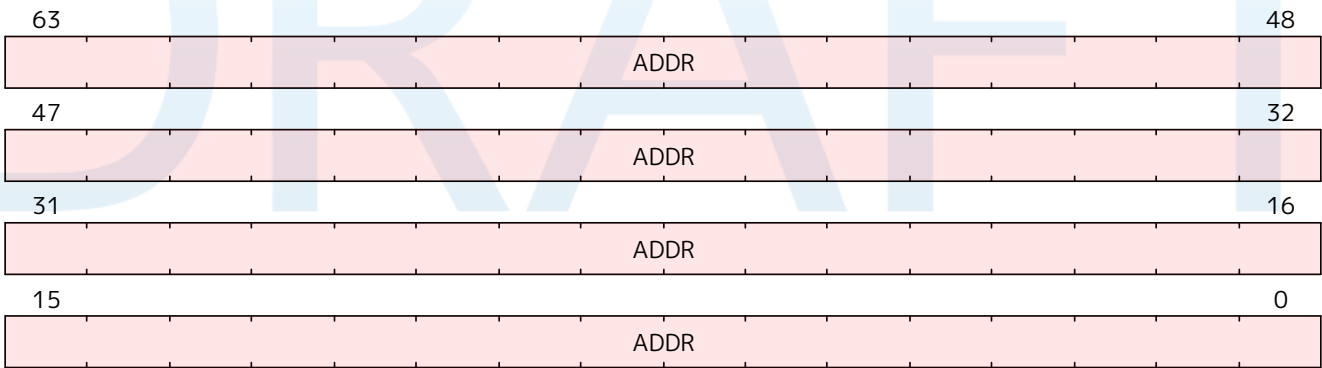


Figure 232. pmpaddr11 Format when CSR[misa].MXL == 1

C.218. pmpaddr12

PMP Address 12

PMP entry address

C.218.1. Attributes

CSR Address	0x3bc
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.218.2. Format

This CSR format changes dynamically with XLEN.

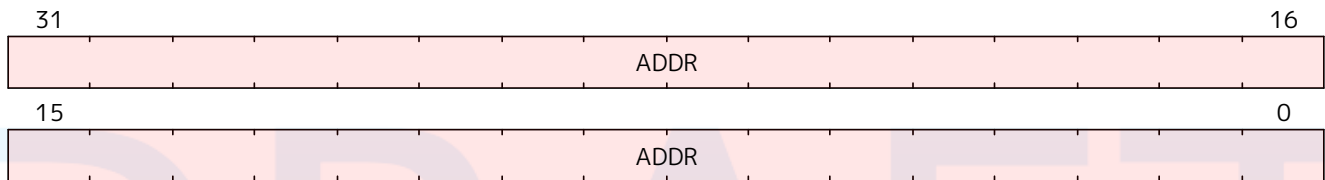


Figure 233. pmpaddr12 Format when CSR[misa].MXL == 0

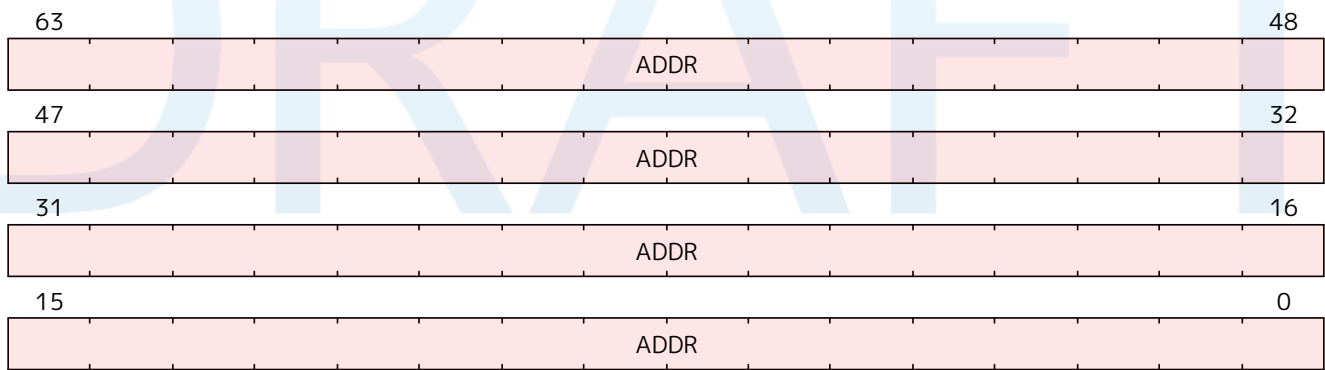


Figure 234. pmpaddr12 Format when CSR[misa].MXL == 1

C.219. pmpaddr13

PMP Address 13

PMP entry address

C.219.1. Attributes

CSR Address	0x3bd
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.219.2. Format

This CSR format changes dynamically with XLEN.

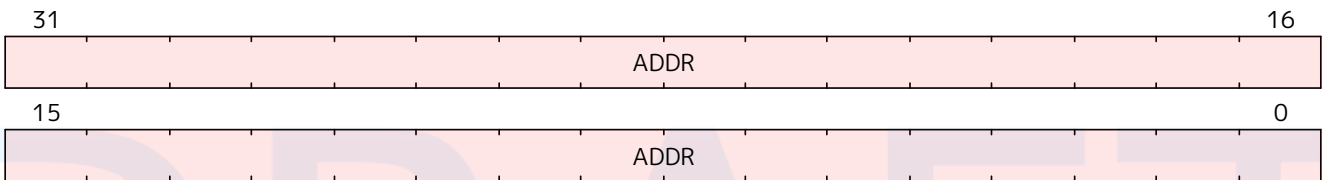


Figure 235. pmpaddr13 Format when CSR[misa].MXL == 0

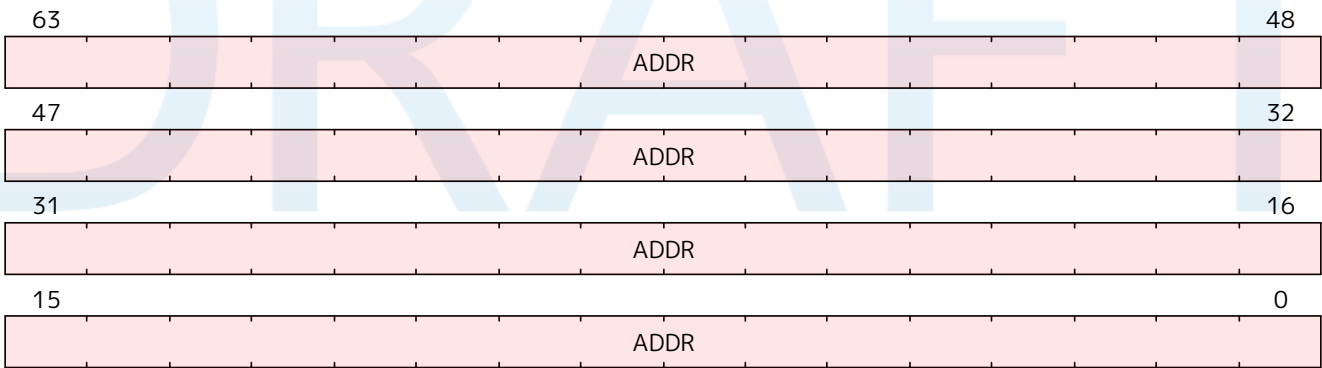


Figure 236. pmpaddr13 Format when CSR[misa].MXL == 1

C.220. pmpaddr14

PMP Address 14

PMP entry address

C.220.1. Attributes

CSR Address	0x3be
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.220.2. Format

This CSR format changes dynamically with XLEN.

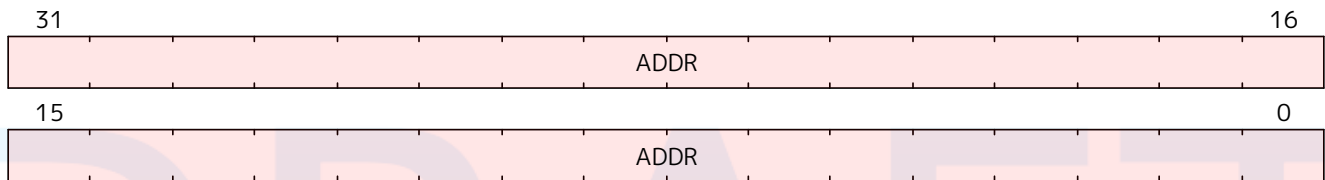


Figure 237. pmpaddr14 Format when CSR[misa].MXL == 0

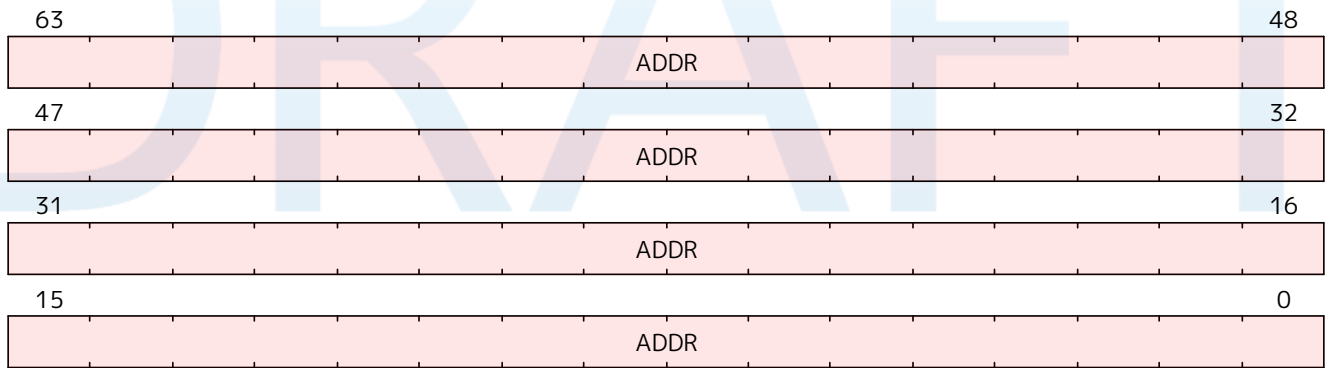


Figure 238. pmpaddr14 Format when CSR[misa].MXL == 1

C.221. pmpaddr15

PMP Address 15

PMP entry address

C.221.1. Attributes

CSR Address	0x3bf
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.221.2. Format

This CSR format changes dynamically with XLEN.

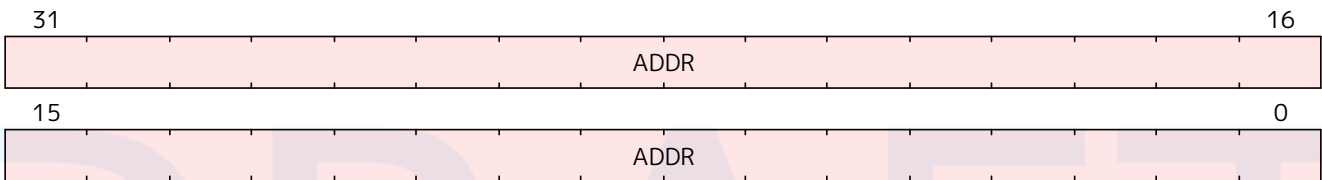


Figure 239. pmpaddr15 Format when CSR[misa].MXL == 0

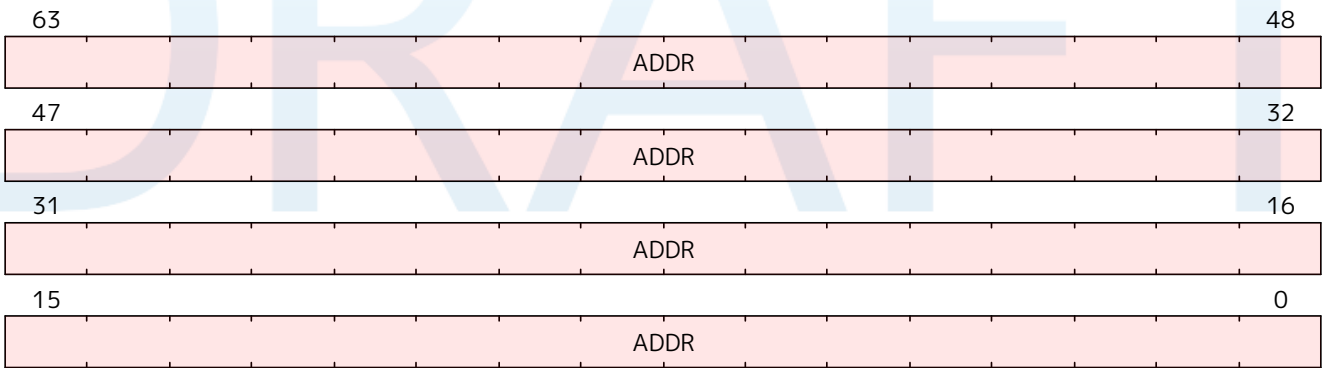


Figure 240. pmpaddr15 Format when CSR[misa].MXL == 1

C.222. pmpaddr16

PMP Address 16

PMP entry address

C.222.1. Attributes

CSR Address	0x3c0
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.222.2. Format

This CSR format changes dynamically with XLEN.

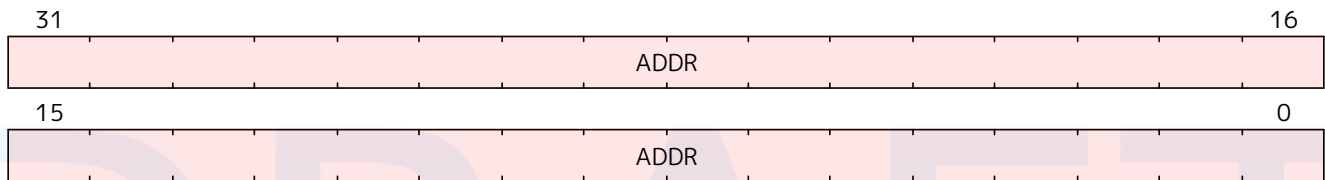


Figure 241. pmpaddr16 Format when CSR[misa].MXL == 0

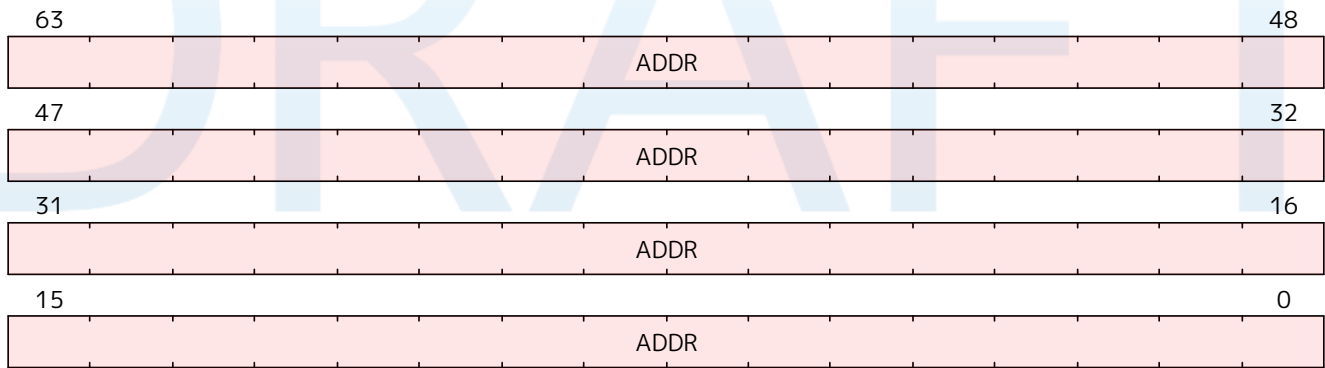


Figure 242. pmpaddr16 Format when CSR[misa].MXL == 1

C.223. pmpaddr17

PMP Address 17

PMP entry address

C.223.1. Attributes

CSR Address	0x3c1
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.223.2. Format

This CSR format changes dynamically with XLEN.

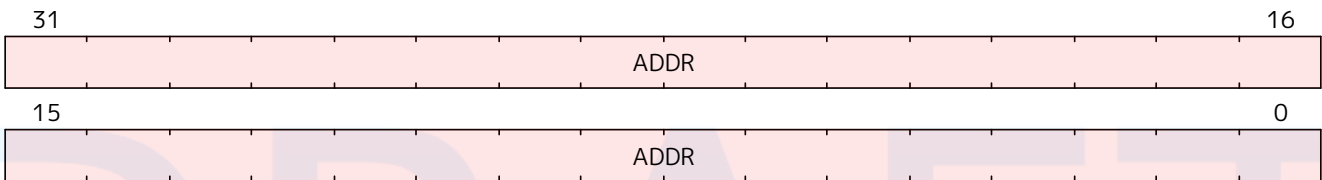


Figure 243. pmpaddr17 Format when CSR[misa].MXL == 0

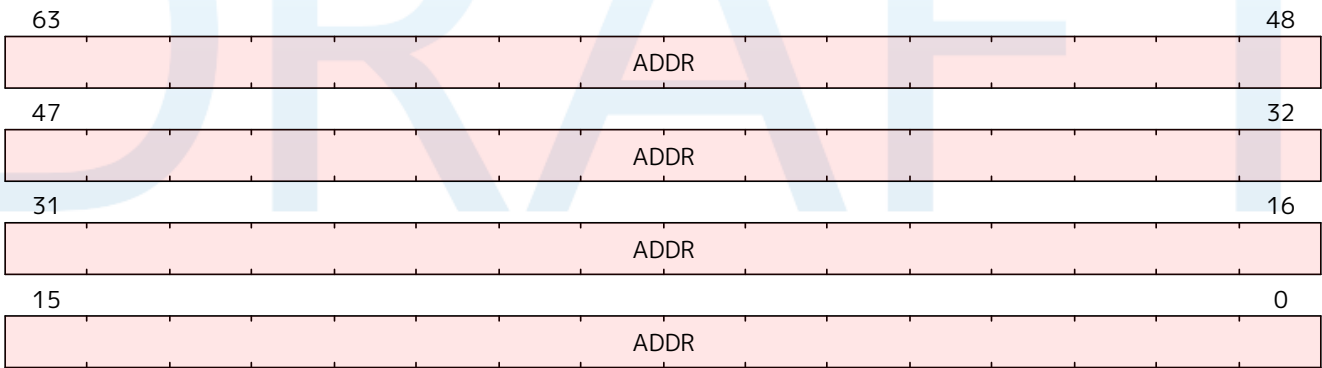


Figure 244. pmpaddr17 Format when CSR[misa].MXL == 1

C.224. pmpaddr18

PMP Address 18

PMP entry address

C.224.1. Attributes

CSR Address	0x3c2
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.224.2. Format

This CSR format changes dynamically with XLEN.

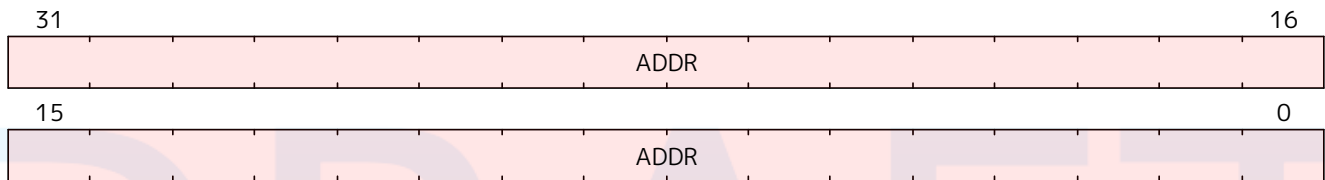


Figure 245. pmpaddr18 Format when CSR[misa].MXL == 0

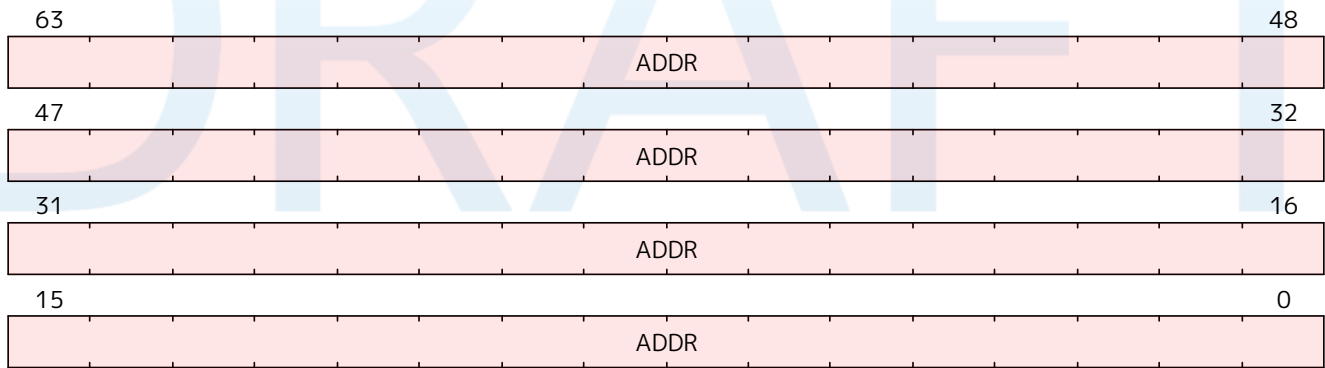


Figure 246. pmpaddr18 Format when CSR[misa].MXL == 1

C.225. pmpaddr19

PMP Address 19

PMP entry address

C.225.1. Attributes

CSR Address	0x3c3
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.225.2. Format

This CSR format changes dynamically with XLEN.

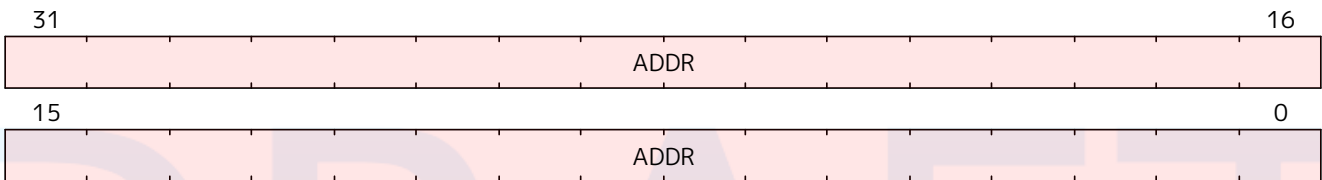


Figure 247. pmpaddr19 Format when CSR[misa].MXL == 0

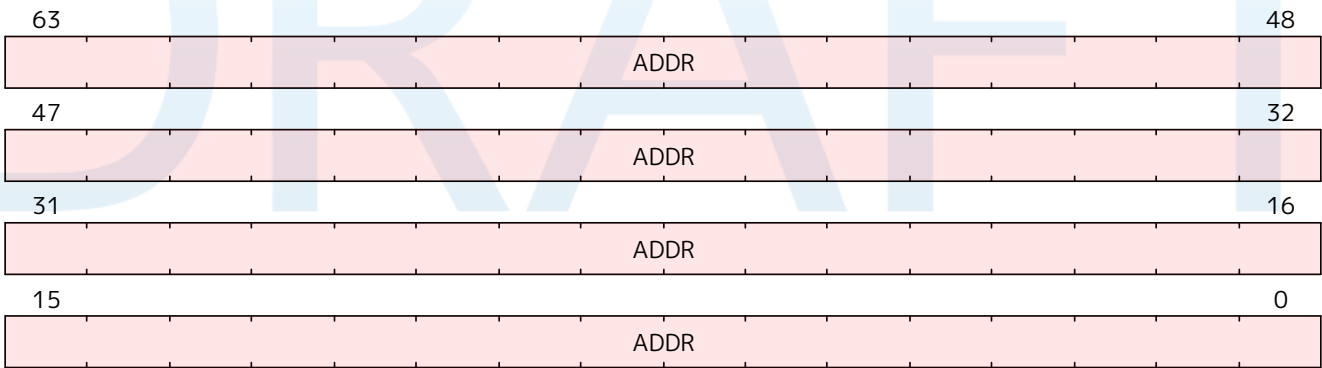


Figure 248. pmpaddr19 Format when CSR[misa].MXL == 1

C.226. pmpaddr2

PMP Address 2

PMP entry address

C.226.1. Attributes

CSR Address	0x3b2
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.226.2. Format

This CSR format changes dynamically with XLEN.

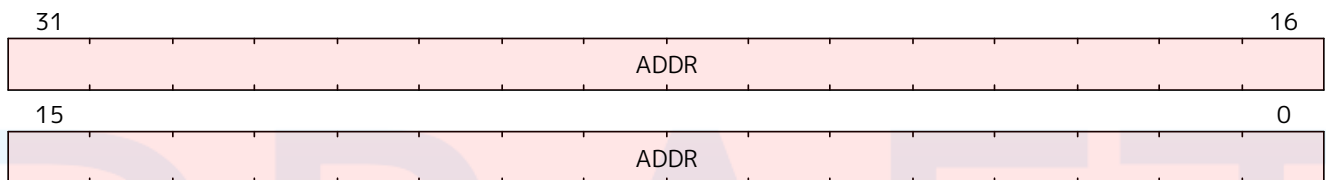


Figure 249. pmpaddr2 Format when CSR[misa].MXL == 0

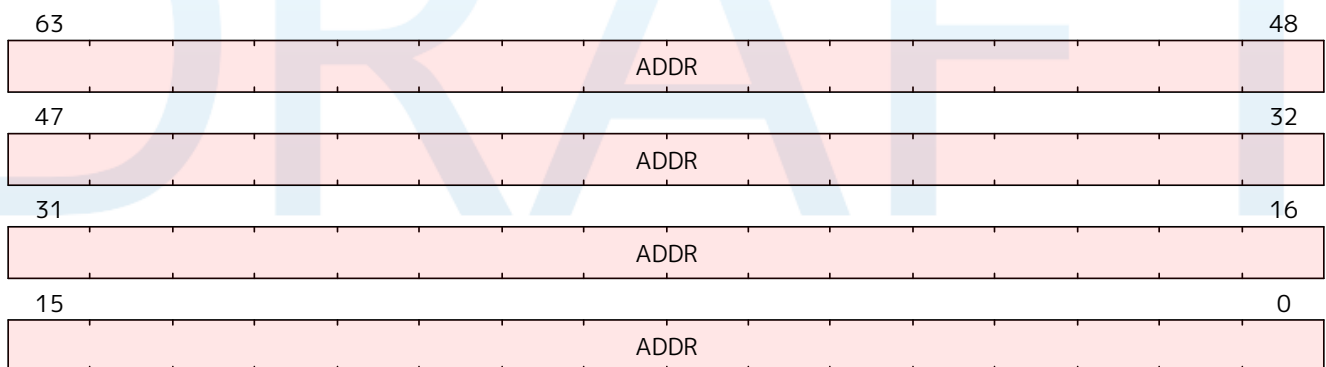


Figure 250. pmpaddr2 Format when CSR[misa].MXL == 1

C.227. pmpaddr20

PMP Address 20

PMP entry address

C.227.1. Attributes

CSR Address	0x3c4
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.227.2. Format

This CSR format changes dynamically with XLEN.

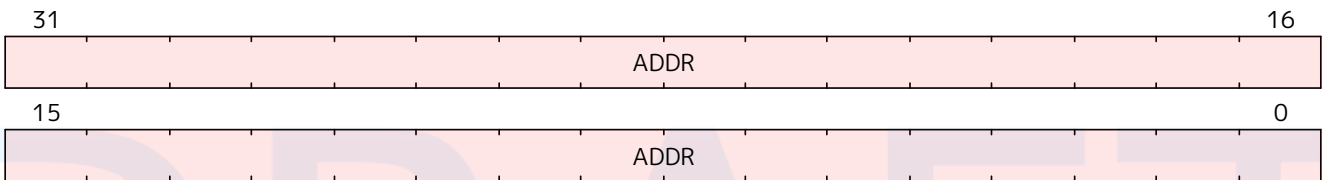


Figure 251. pmpaddr20 Format when CSR[misa].MXL == 0

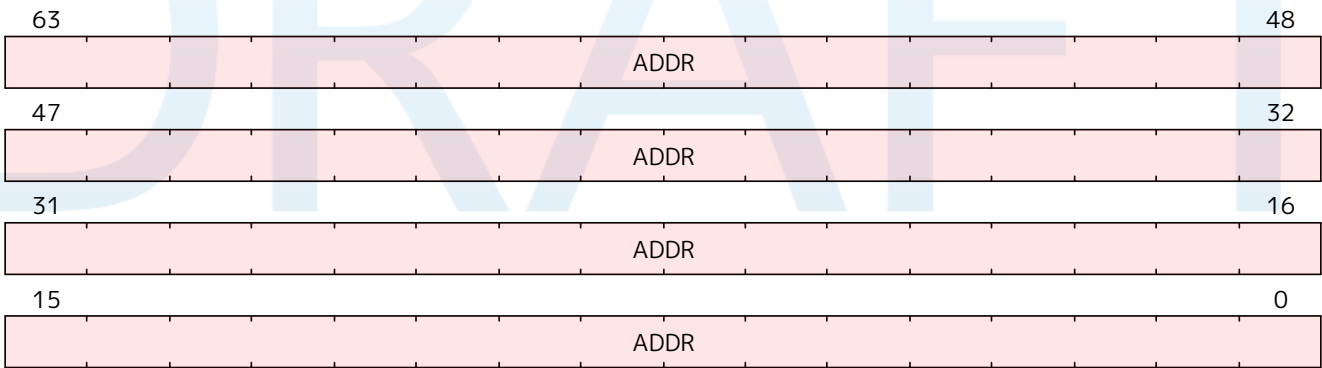


Figure 252. pmpaddr20 Format when CSR[misa].MXL == 1

C.228. pmpaddr21

PMP Address 21

PMP entry address

C.228.1. Attributes

CSR Address	0x3c5
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.228.2. Format

This CSR format changes dynamically with XLEN.

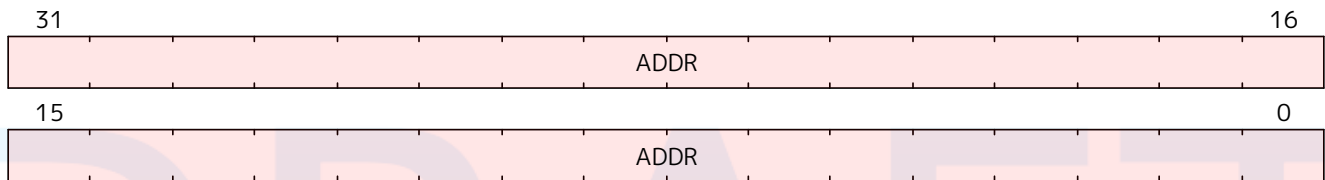


Figure 253. pmpaddr21 Format when CSR[misa].MXL == 0

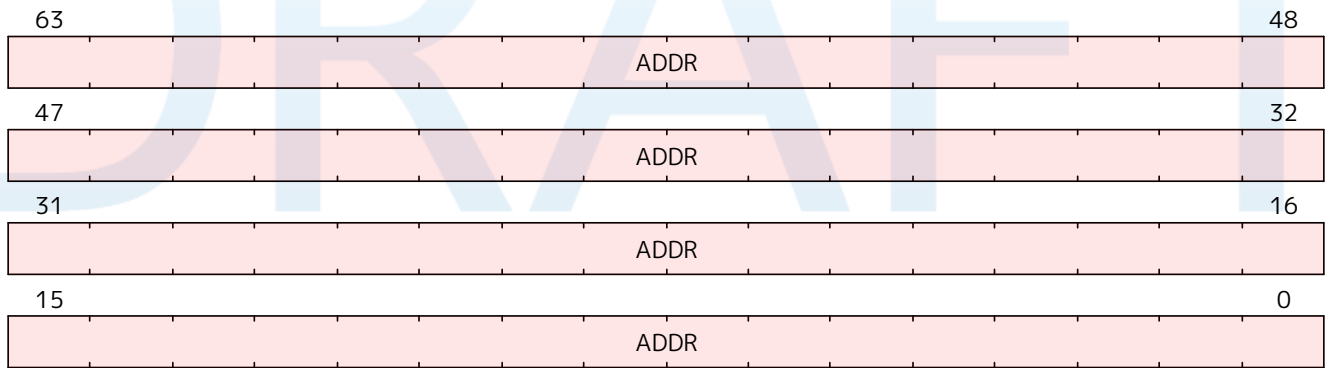


Figure 254. pmpaddr21 Format when CSR[misa].MXL == 1

C.229. pmpaddr22

PMP Address 22

PMP entry address

C.229.1. Attributes

CSR Address	0x3c6
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.229.2. Format

This CSR format changes dynamically with XLEN.

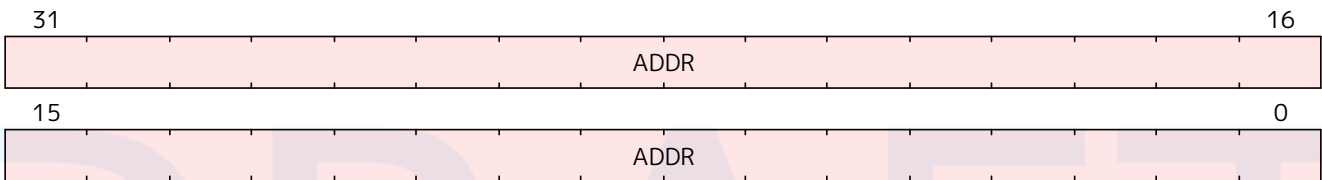


Figure 255. pmpaddr22 Format when CSR[misa].MXL == 0

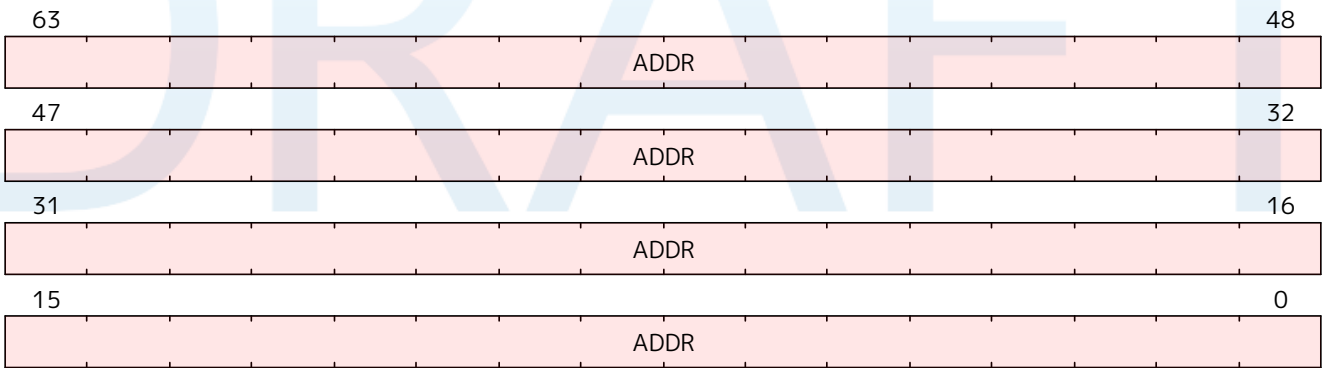


Figure 256. pmpaddr22 Format when CSR[misa].MXL == 1

C.230. pmpaddr23

PMP Address 23

PMP entry address

C.230.1. Attributes

CSR Address	0x3c7
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.230.2. Format

This CSR format changes dynamically with XLEN.

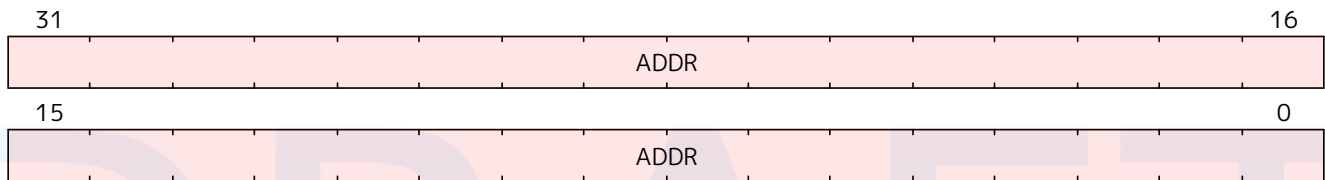


Figure 257. pmpaddr23 Format when CSR[misa].MXL == 0

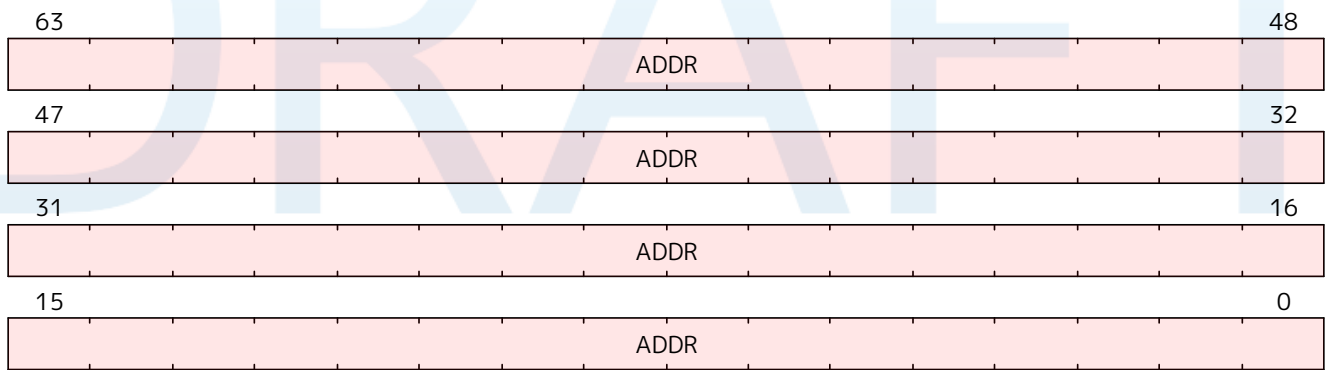


Figure 258. pmpaddr23 Format when CSR[misa].MXL == 1

C.231. pmpaddr24

PMP Address 24

PMP entry address

C.231.1. Attributes

CSR Address	0x3c8
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.231.2. Format

This CSR format changes dynamically with XLEN.

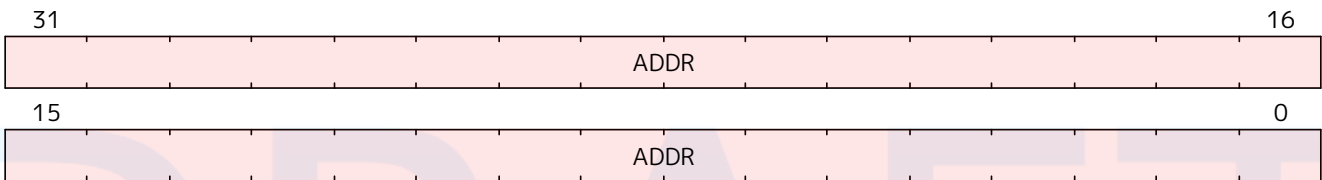


Figure 259. pmpaddr24 Format when CSR[misa].MXL == 0

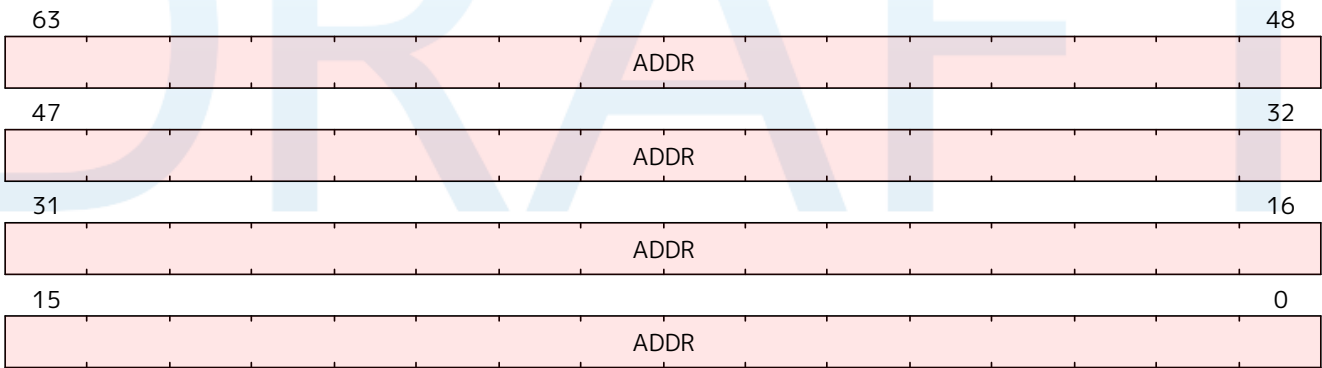


Figure 260. pmpaddr24 Format when CSR[misa].MXL == 1

C.232. pmpaddr25

PMP Address 25

PMP entry address

C.232.1. Attributes

CSR Address	0x3c9
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.232.2. Format

This CSR format changes dynamically with XLEN.

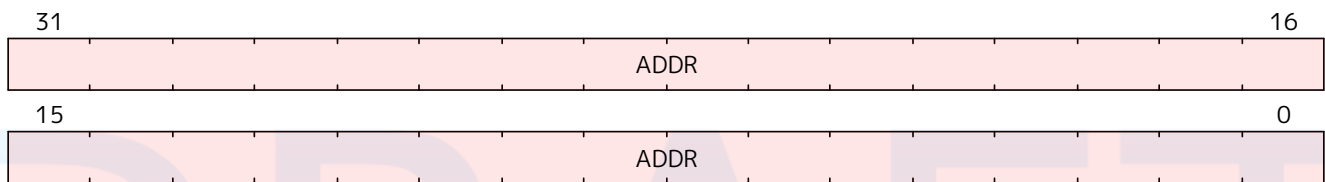


Figure 261. pmpaddr25 Format when CSR[misa].MXL == 0

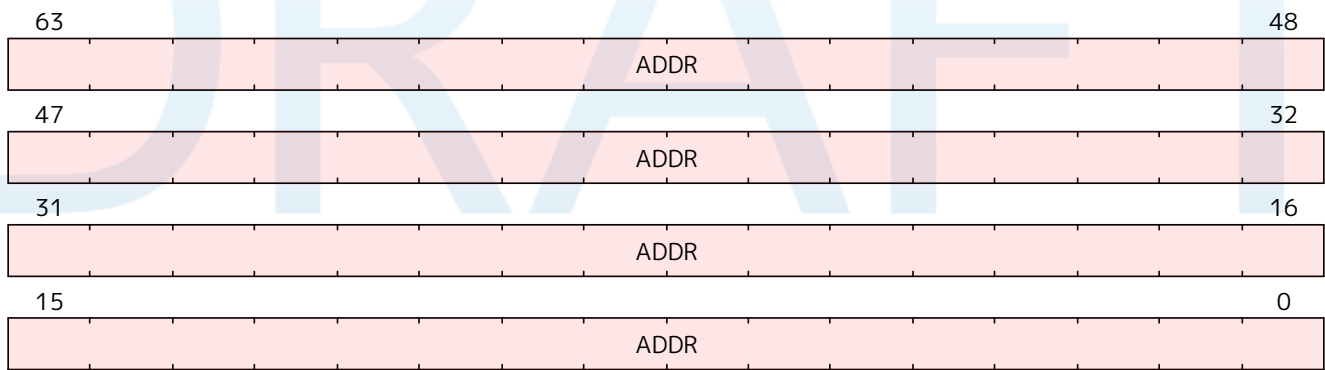


Figure 262. pmpaddr25 Format when CSR[misa].MXL == 1

C.233. pmpaddr26

PMP Address 26

PMP entry address

C.233.1. Attributes

CSR Address	0x3ca
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.233.2. Format

This CSR format changes dynamically with XLEN.

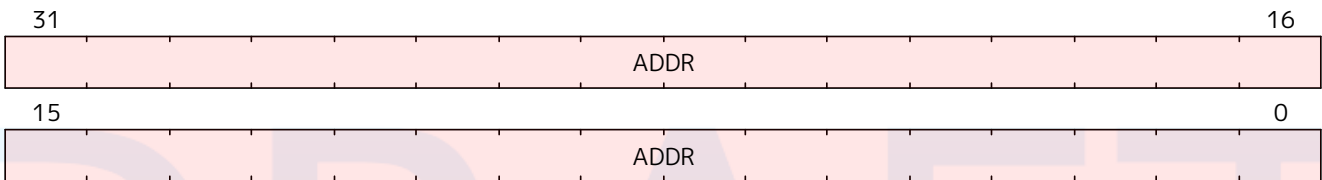


Figure 263. pmpaddr26 Format when CSR[misa].MXL == 0

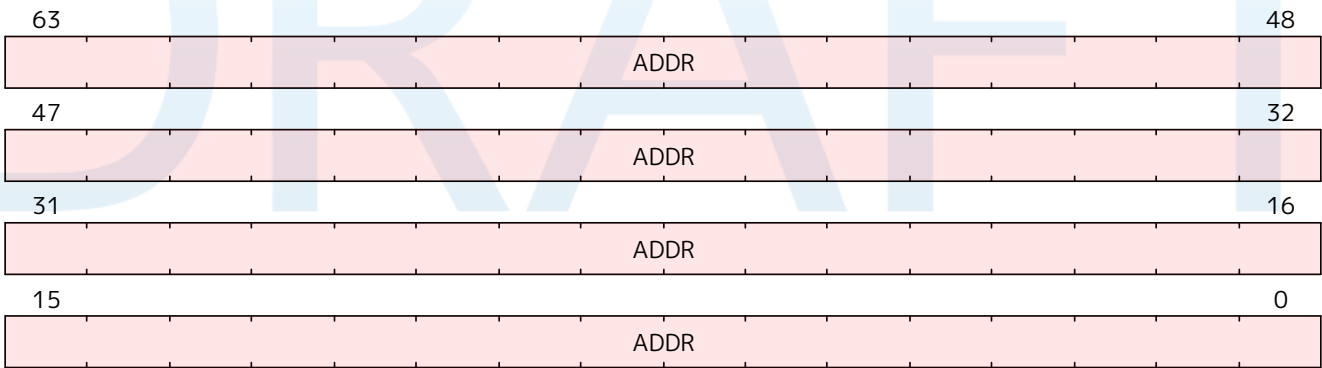


Figure 264. pmpaddr26 Format when CSR[misa].MXL == 1

C.234. pmpaddr27

PMP Address 27

PMP entry address

C.234.1. Attributes

CSR Address	0x3cb
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.234.2. Format

This CSR format changes dynamically with XLEN.

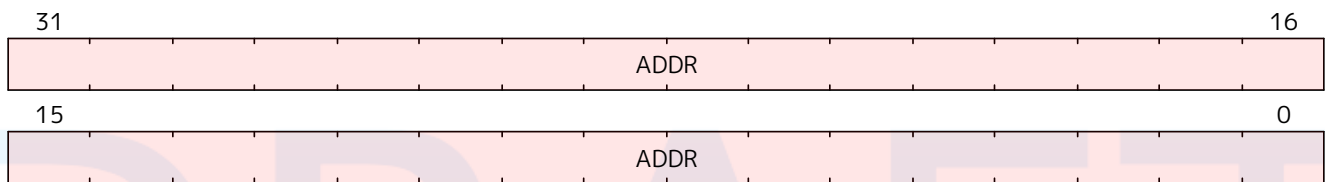


Figure 265. pmpaddr27 Format when CSR[misa].MXL == 0

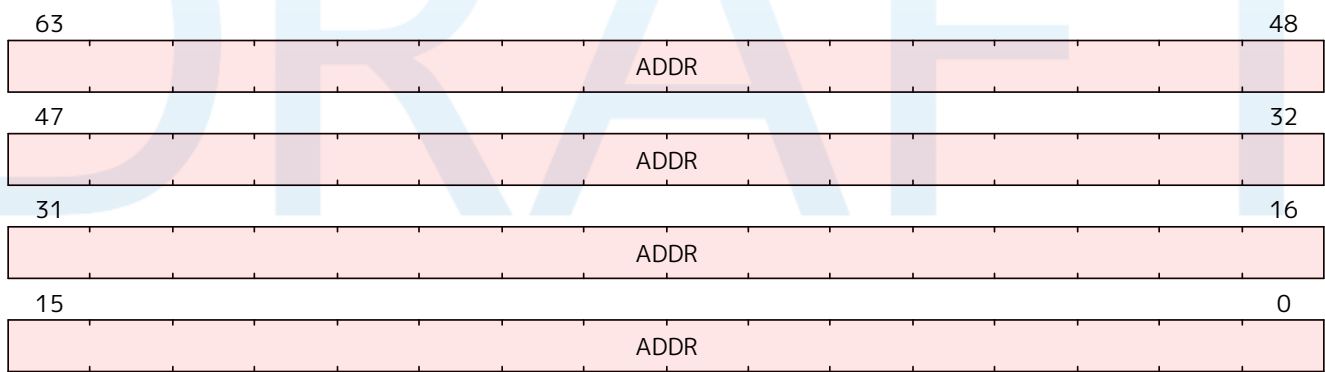


Figure 266. pmpaddr27 Format when CSR[misa].MXL == 1

C.235. pmpaddr28

PMP Address 28

PMP entry address

C.235.1. Attributes

CSR Address	0x3cc
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.235.2. Format

This CSR format changes dynamically with XLEN.

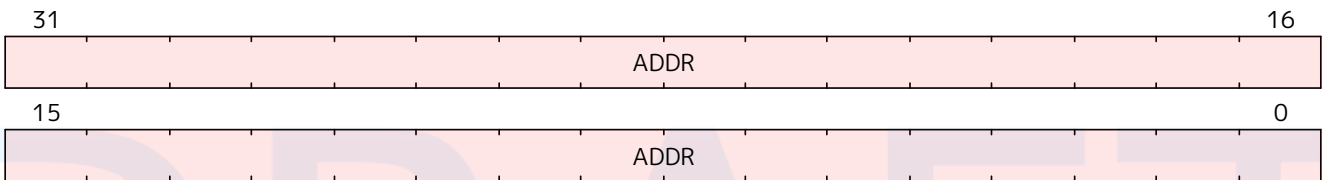


Figure 267. pmpaddr28 Format when CSR[misa].MXL == 0

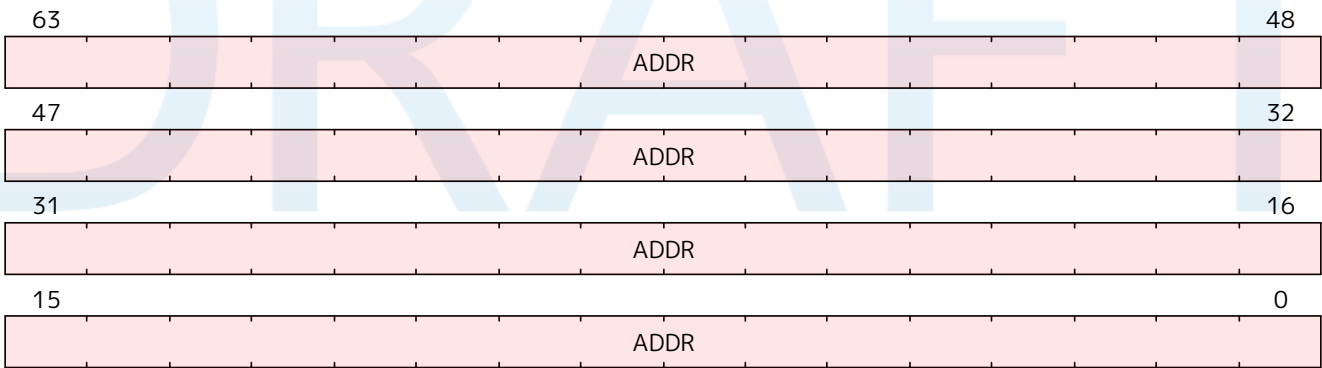


Figure 268. pmpaddr28 Format when CSR[misa].MXL == 1

C.236. pmpaddr29

PMP Address 29

PMP entry address

C.236.1. Attributes

CSR Address	0x3cd
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.236.2. Format

This CSR format changes dynamically with XLEN.

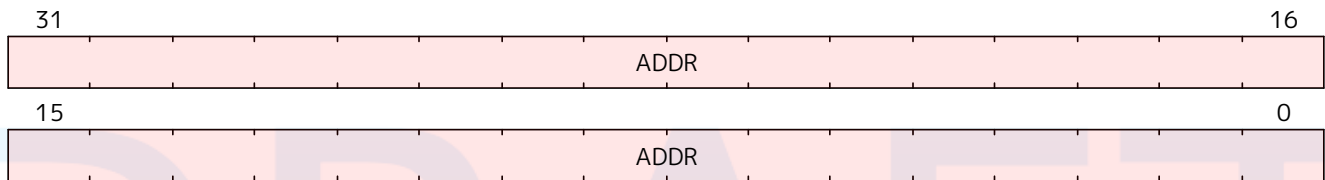


Figure 269. pmpaddr29 Format when CSR[misa].MXL == 0

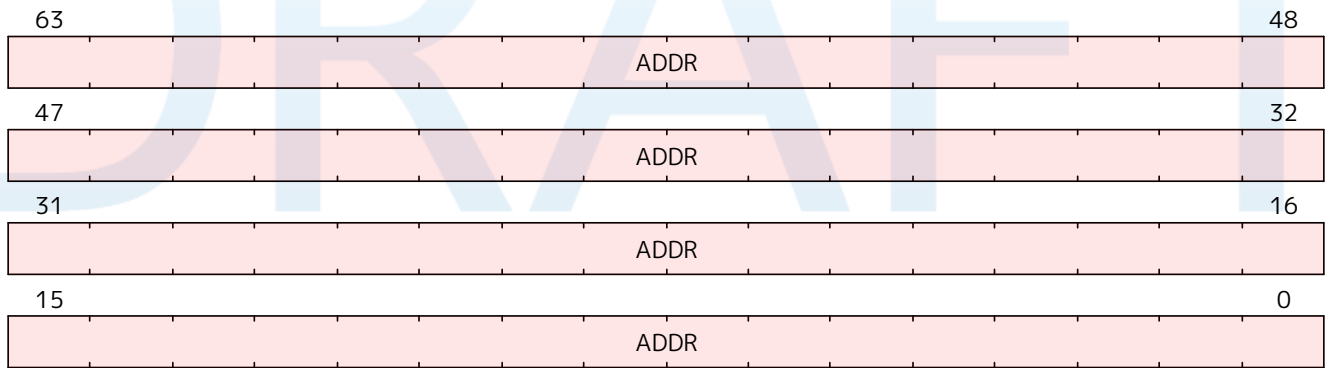


Figure 270. pmpaddr29 Format when CSR[misa].MXL == 1

C.237. pmpaddr3

PMP Address 3

PMP entry address

C.237.1. Attributes

CSR Address	0x3b3
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.237.2. Format

This CSR format changes dynamically with XLEN.

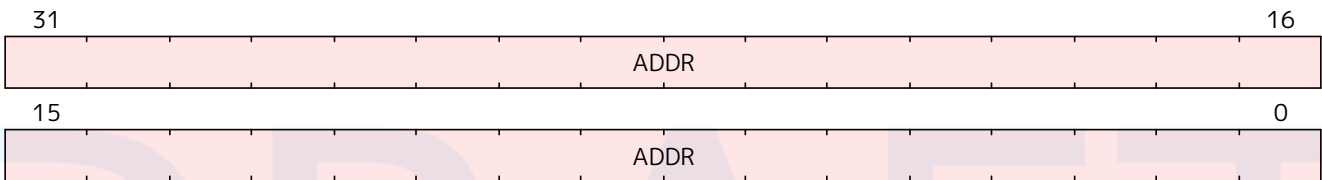


Figure 271. pmpaddr3 Format when CSR[misa].MXL == 0

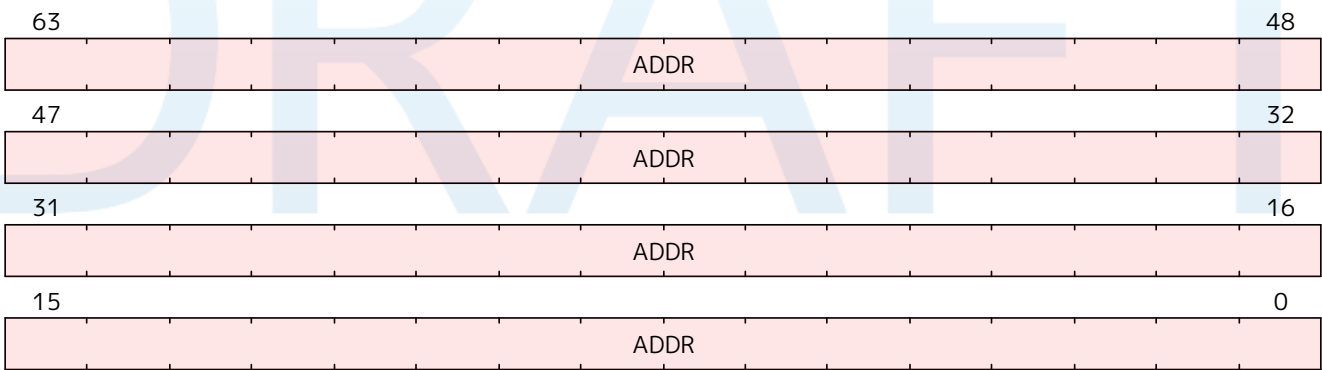


Figure 272. pmpaddr3 Format when CSR[misa].MXL == 1

C.238. pmpaddr30

PMP Address 30

PMP entry address

C.238.1. Attributes

CSR Address	0x3ce
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.238.2. Format

This CSR format changes dynamically with XLEN.

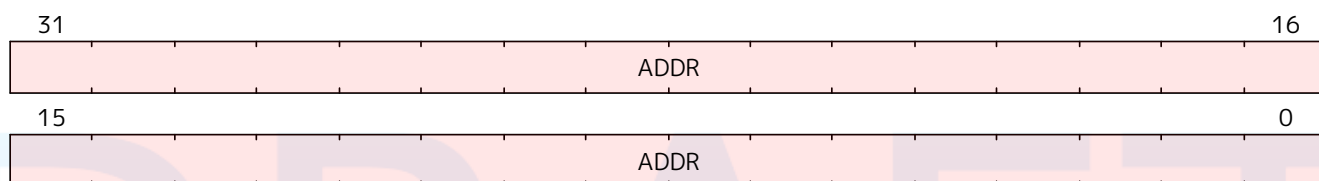


Figure 273. pmpaddr30 Format when CSR[misa].MXL == 0

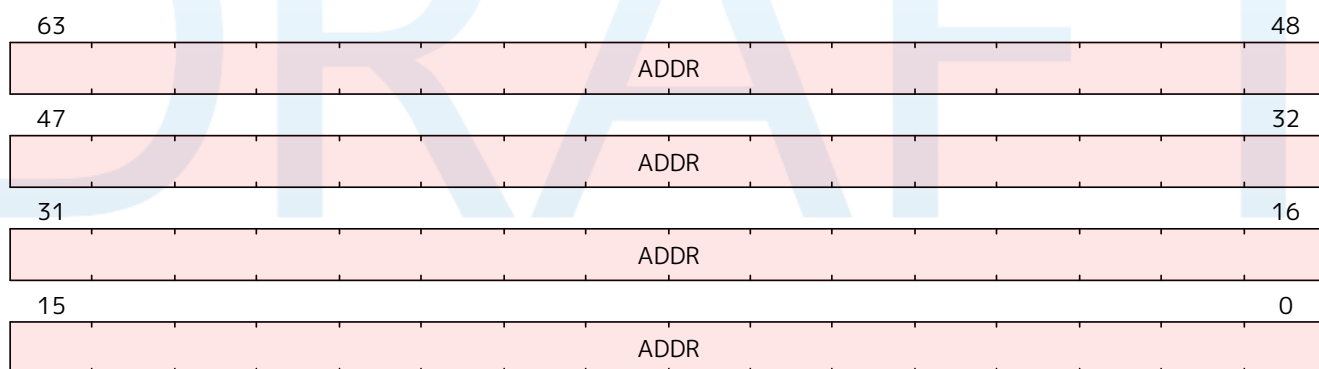


Figure 274. pmpaddr30 Format when CSR[misa].MXL == 1

C.239. pmpaddr31

PMP Address 31

PMP entry address

C.239.1. Attributes

CSR Address	0x3cf
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.239.2. Format

This CSR format changes dynamically with XLEN.

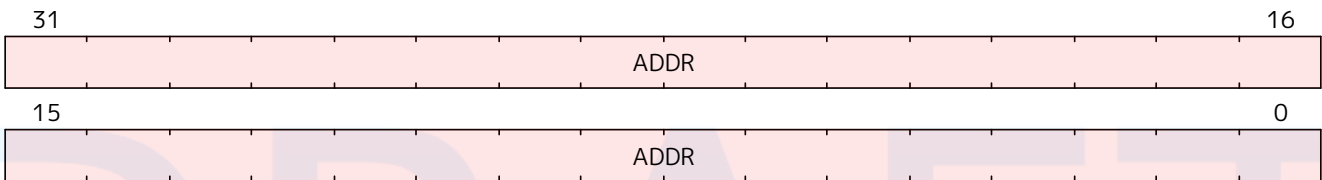


Figure 275. pmpaddr31 Format when CSR[misa].MXL == 0

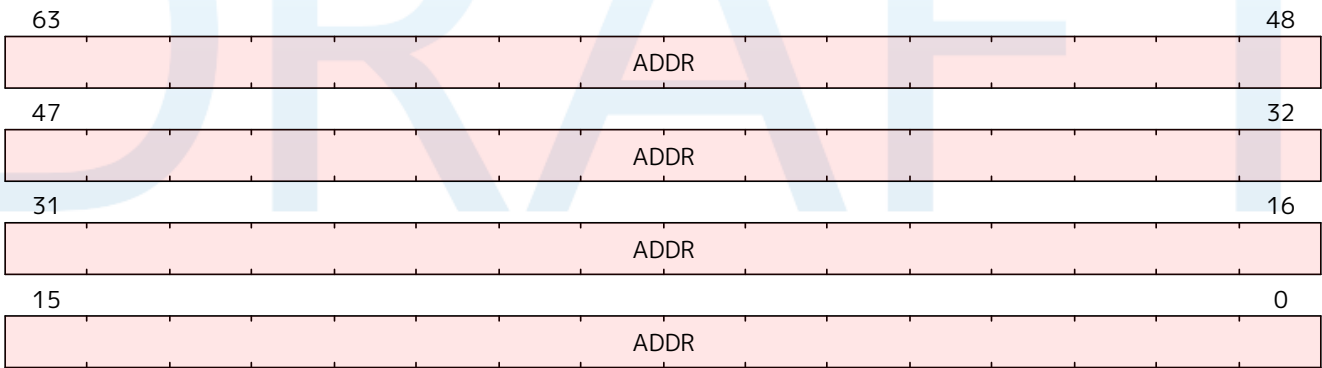


Figure 276. pmpaddr31 Format when CSR[misa].MXL == 1

C.240. pmpaddr32

PMP Address 32

PMP entry address

C.240.1. Attributes

CSR Address	0x3d0
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.240.2. Format

This CSR format changes dynamically with XLEN.

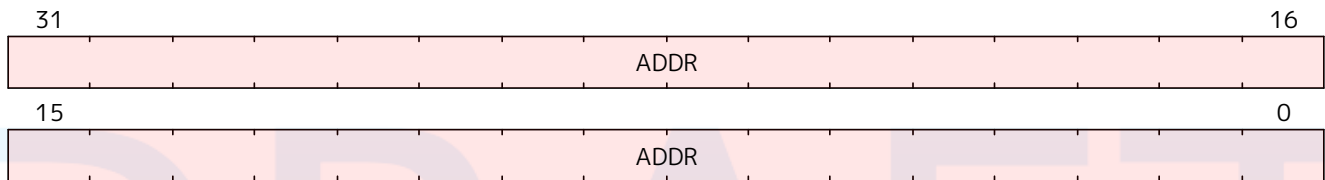


Figure 277. pmpaddr32 Format when CSR[misa].MXL == 0

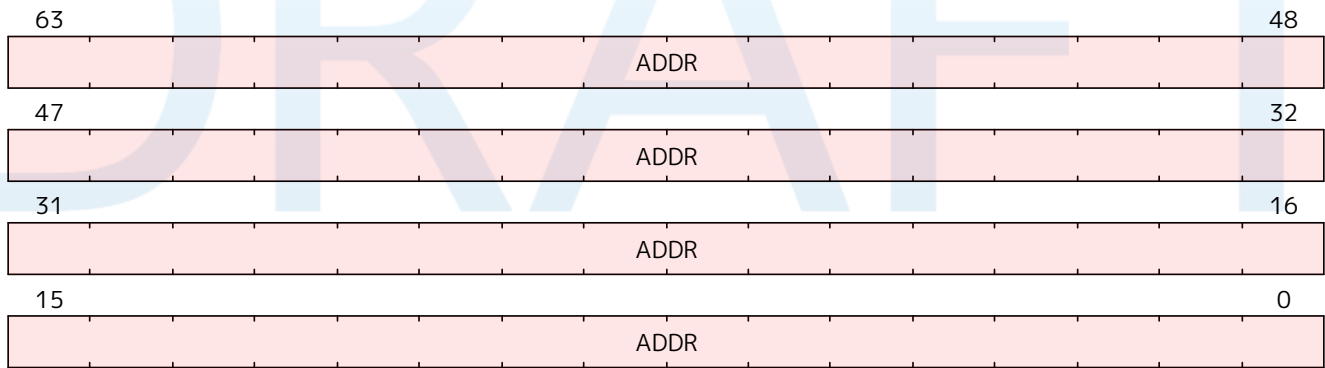


Figure 278. pmpaddr32 Format when CSR[misa].MXL == 1

C.241. pmpaddr33

PMP Address 33

PMP entry address

C.241.1. Attributes

CSR Address	0x3d1
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.241.2. Format

This CSR format changes dynamically with XLEN.

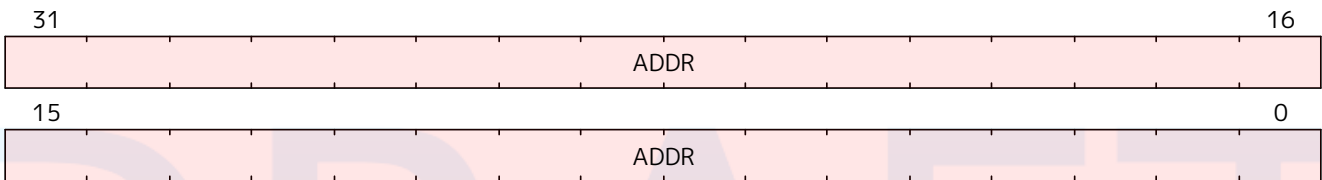


Figure 279. pmpaddr33 Format when CSR[misa].MXL == 0

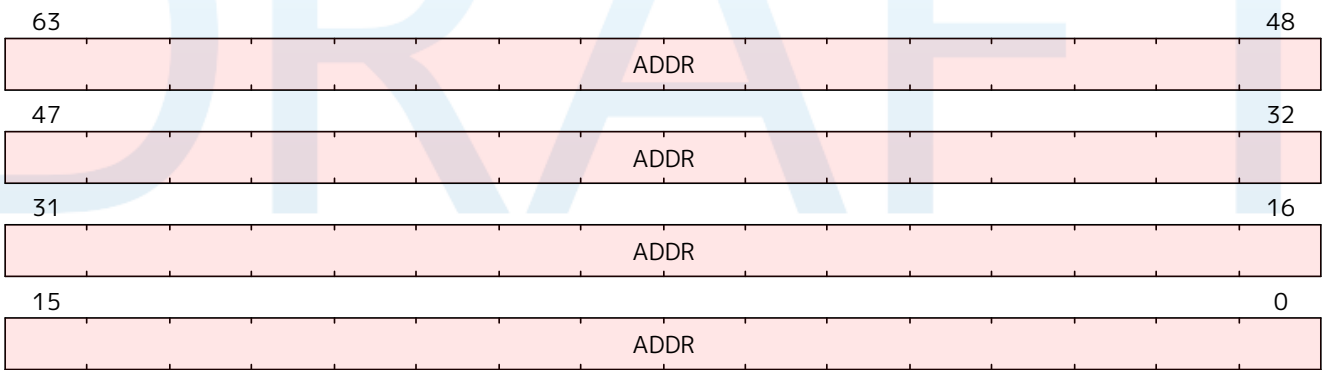


Figure 280. pmpaddr33 Format when CSR[misa].MXL == 1

C.242. pmpaddr34

PMP Address 34

PMP entry address

C.242.1. Attributes

CSR Address	0x3d2
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.242.2. Format

This CSR format changes dynamically with XLEN.

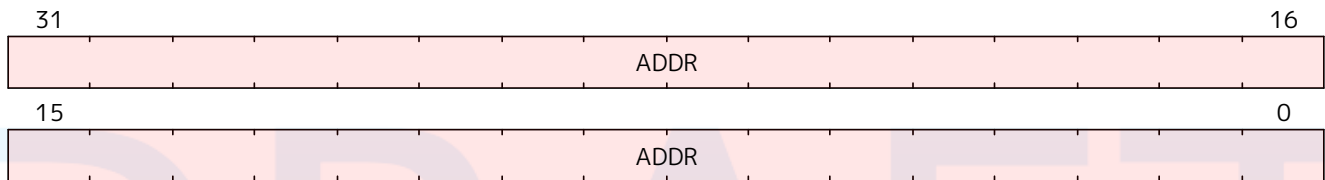


Figure 281. pmpaddr34 Format when CSR[misa].MXL == 0

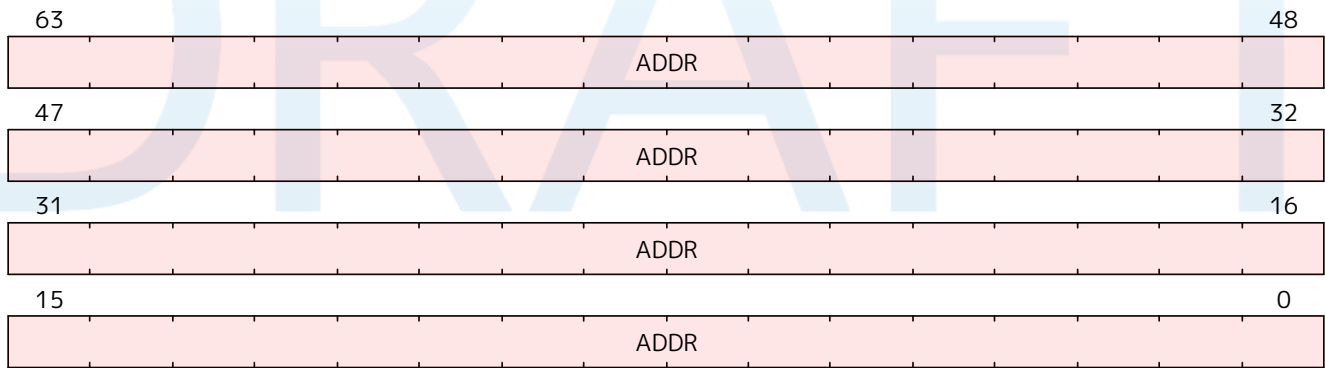


Figure 282. pmpaddr34 Format when CSR[misa].MXL == 1

C.243. pmpaddr35

PMP Address 35

PMP entry address

C.243.1. Attributes

CSR Address	0x3d3
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.243.2. Format

This CSR format changes dynamically with XLEN.

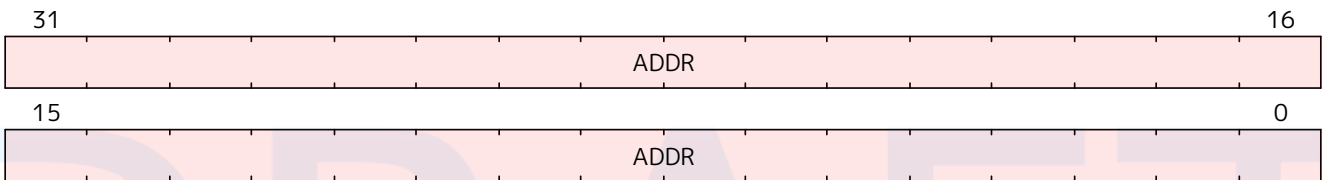


Figure 283. pmpaddr35 Format when CSR[misa].MXL == 0

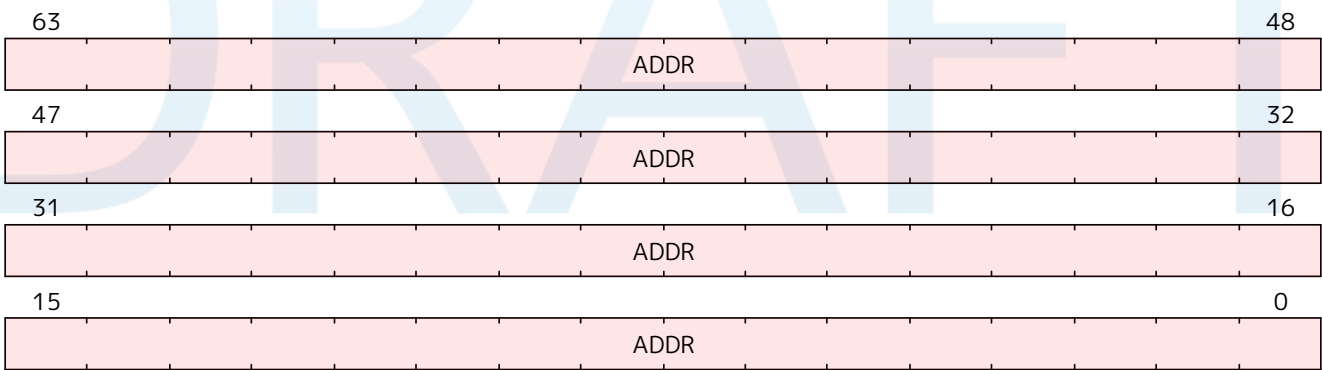


Figure 284. pmpaddr35 Format when CSR[misa].MXL == 1

C.244. pmpaddr36

PMP Address 36

PMP entry address

C.244.1. Attributes

CSR Address	0x3d4
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.244.2. Format

This CSR format changes dynamically with XLEN.

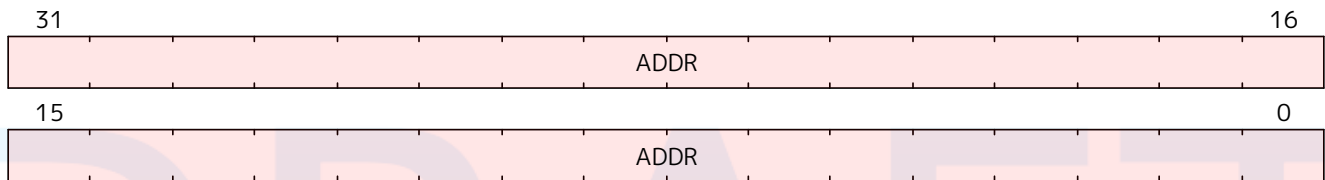


Figure 285. pmpaddr36 Format when CSR[misa].MXL == 0

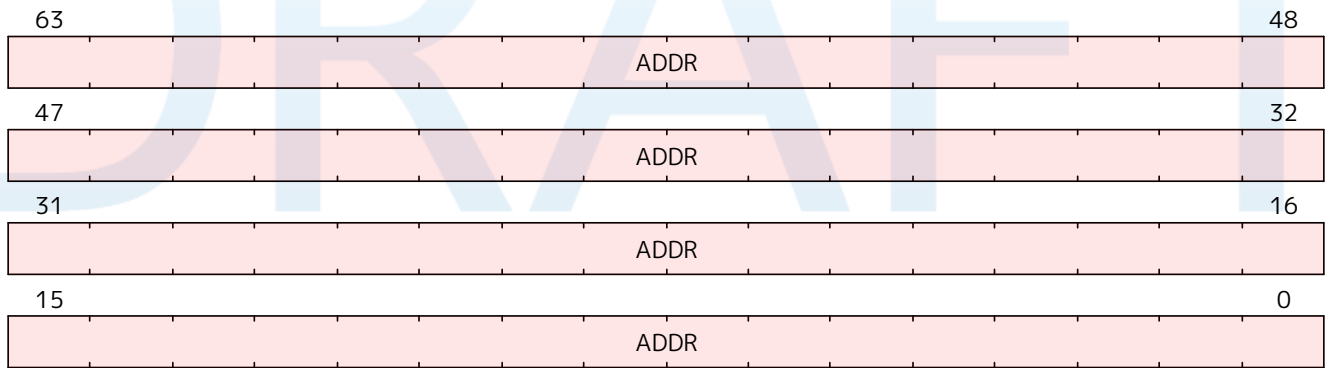


Figure 286. pmpaddr36 Format when CSR[misa].MXL == 1

C.245. pmpaddr37

PMP Address 37

PMP entry address

C.245.1. Attributes

CSR Address	0x3d5
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.245.2. Format

This CSR format changes dynamically with XLEN.

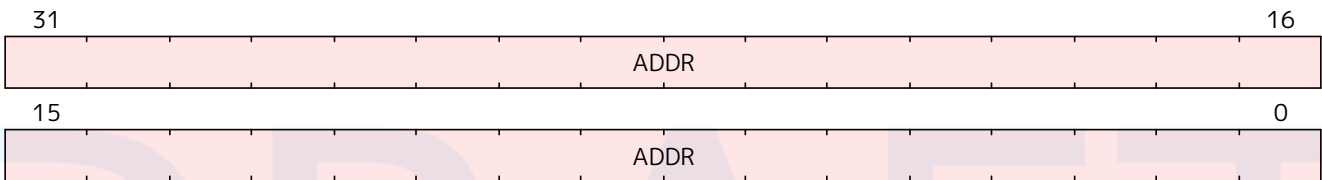


Figure 287. pmpaddr37 Format when CSR[misa].MXL == 0

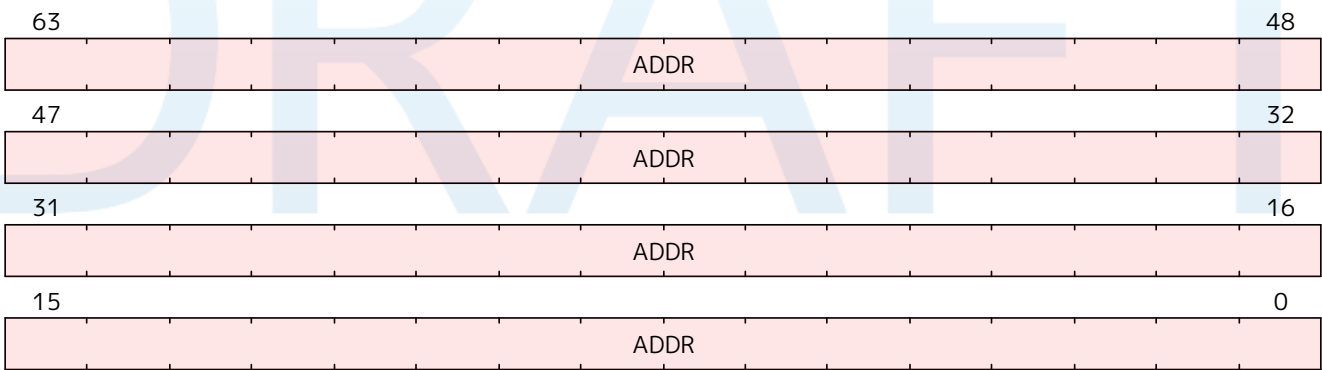


Figure 288. pmpaddr37 Format when CSR[misa].MXL == 1

C.246. pmpaddr38

PMP Address 38

PMP entry address

C.246.1. Attributes

CSR Address	0x3d6
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.246.2. Format

This CSR format changes dynamically with XLEN.

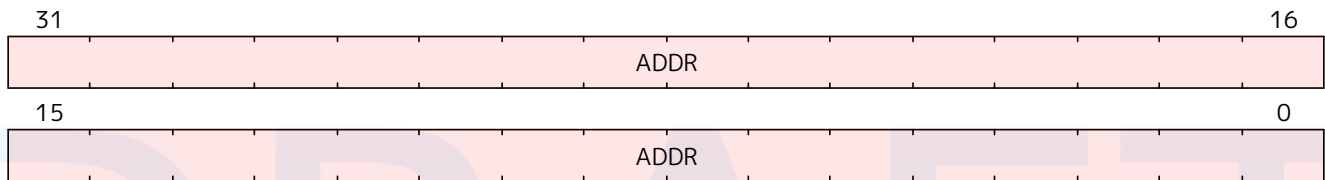


Figure 289. pmpaddr38 Format when CSR[misa].MXL == 0

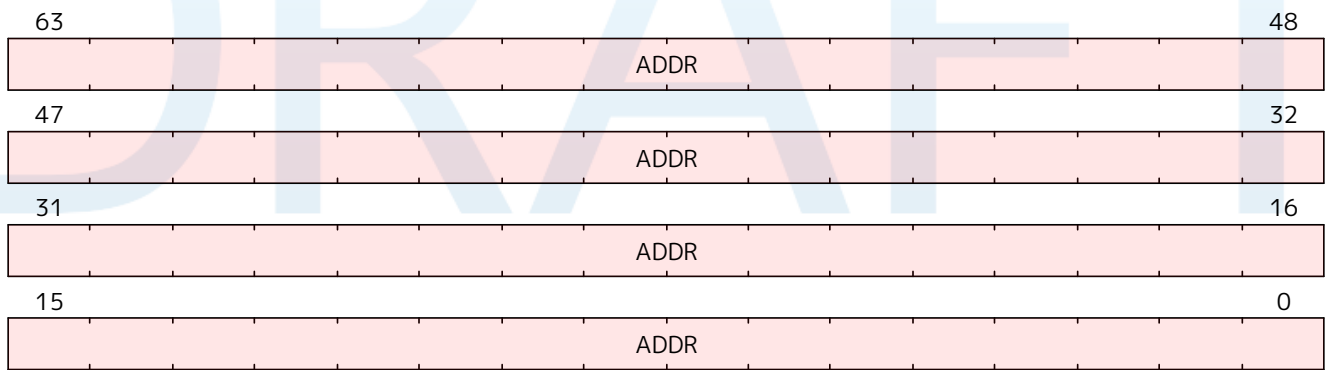


Figure 290. pmpaddr38 Format when CSR[misa].MXL == 1

C.247. pmpaddr39

PMP Address 39

PMP entry address

C.247.1. Attributes

CSR Address	0x3d7
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.247.2. Format

This CSR format changes dynamically with XLEN.

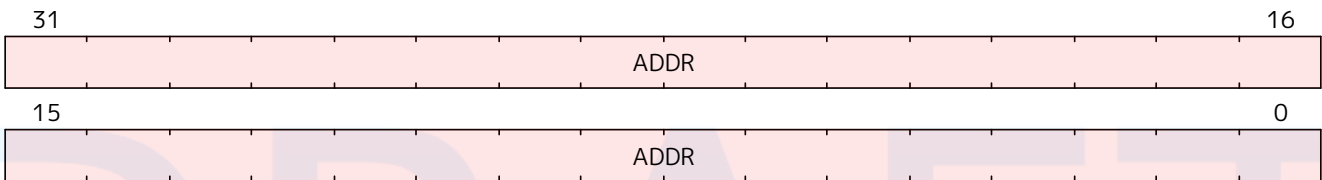


Figure 291. pmpaddr39 Format when CSR[misa].MXL == 0

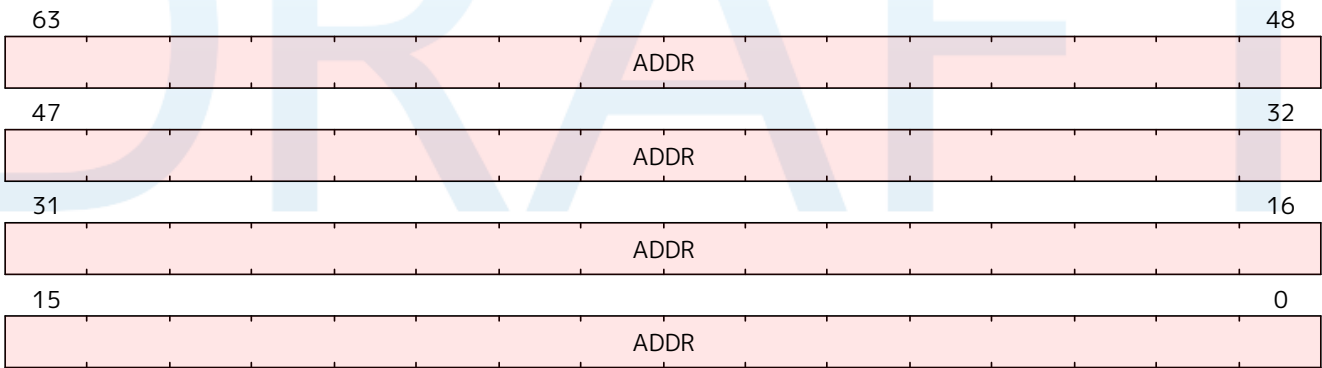


Figure 292. pmpaddr39 Format when CSR[misa].MXL == 1

C.248. pmpaddr4

PMP Address 4

PMP entry address

C.248.1. Attributes

CSR Address	0x3b4
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.248.2. Format

This CSR format changes dynamically with XLEN.

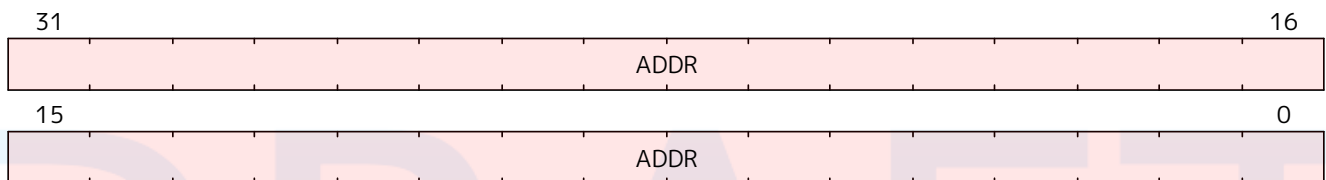


Figure 293. pmpaddr4 Format when CSR[misa].MXL == 0

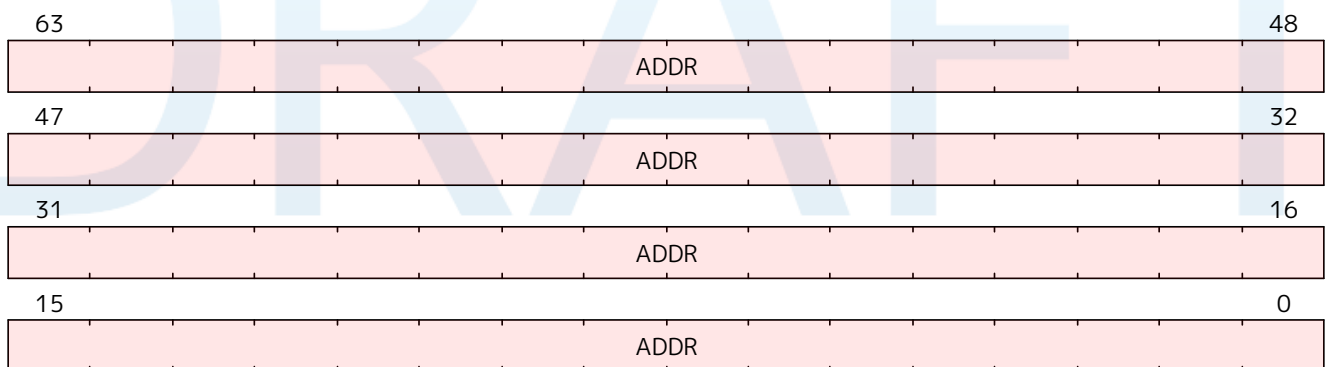


Figure 294. pmpaddr4 Format when CSR[misa].MXL == 1

C.249. pmpaddr40

PMP Address 40

PMP entry address

C.249.1. Attributes

CSR Address	0x3d8
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.249.2. Format

This CSR format changes dynamically with XLEN.

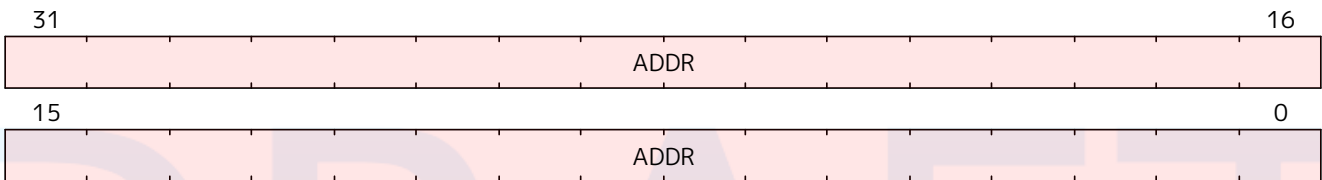


Figure 295. pmpaddr40 Format when CSR[misa].MXL == 0

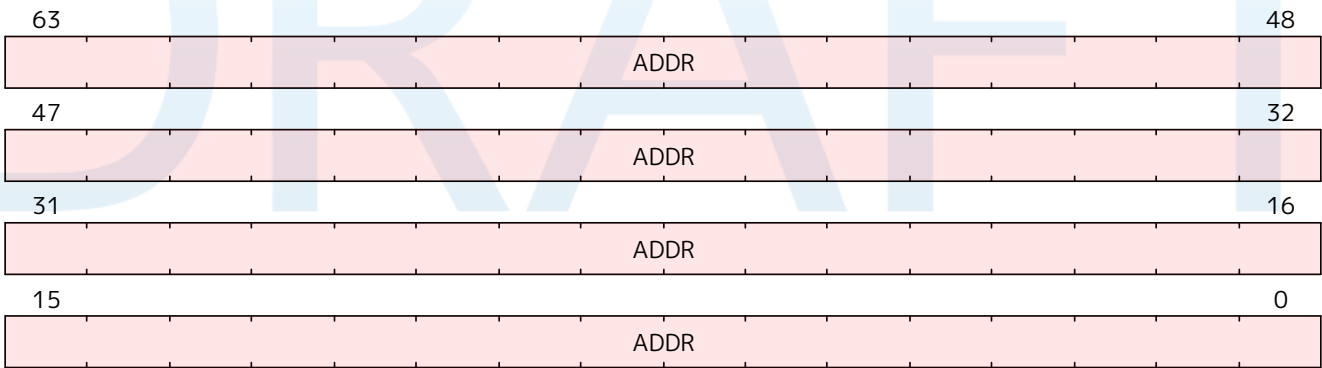


Figure 296. pmpaddr40 Format when CSR[misa].MXL == 1

C.250. pmpaddr41

PMP Address 41

PMP entry address

C.250.1. Attributes

CSR Address	0x3d9
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.250.2. Format

This CSR format changes dynamically with XLEN.

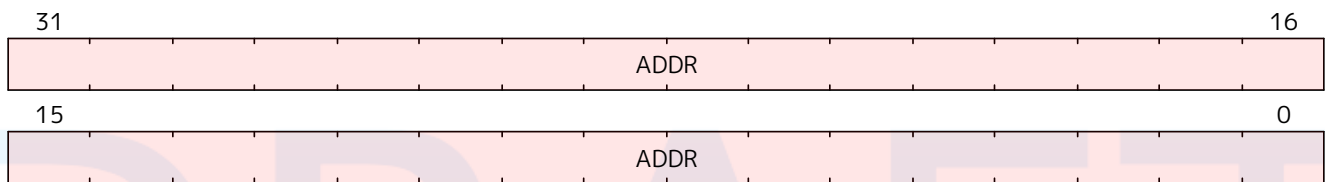


Figure 297. pmpaddr41 Format when CSR[misa].MXL == 0

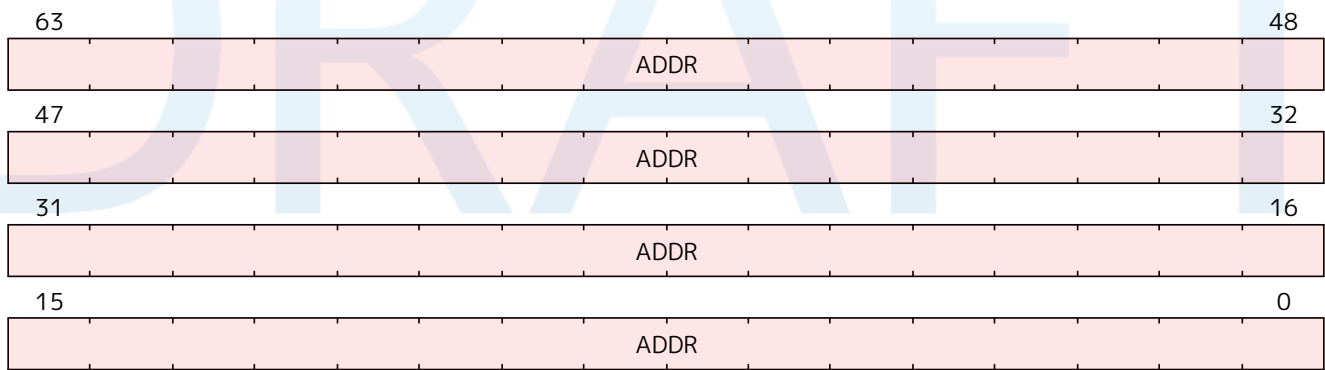


Figure 298. pmpaddr41 Format when CSR[misa].MXL == 1

C.251. pmpaddr42

PMP Address 42

PMP entry address

C.251.1. Attributes

CSR Address	0x3da
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.251.2. Format

This CSR format changes dynamically with XLEN.

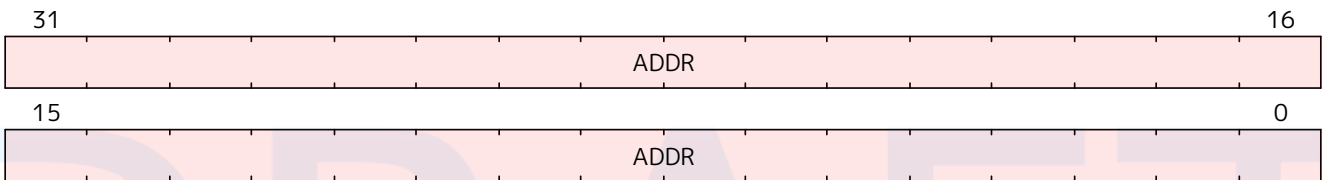


Figure 299. pmpaddr42 Format when CSR[misa].MXL == 0

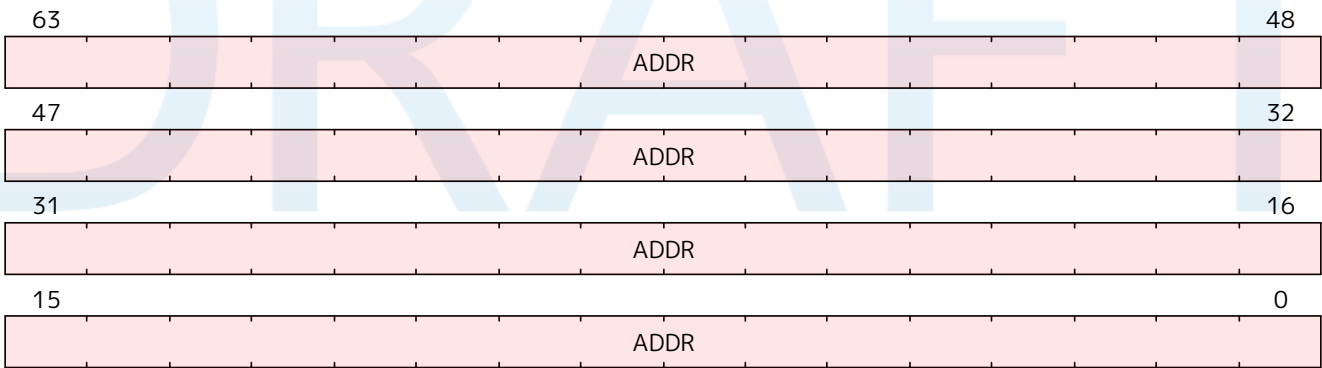


Figure 300. pmpaddr42 Format when CSR[misa].MXL == 1

C.252. pmpaddr43

PMP Address 43

PMP entry address

C.252.1. Attributes

CSR Address	0x3db
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.252.2. Format

This CSR format changes dynamically with XLEN.

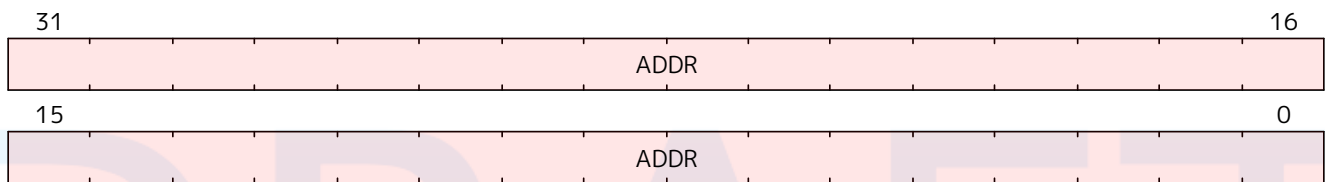


Figure 301. pmpaddr43 Format when CSR[misa].MXL == 0

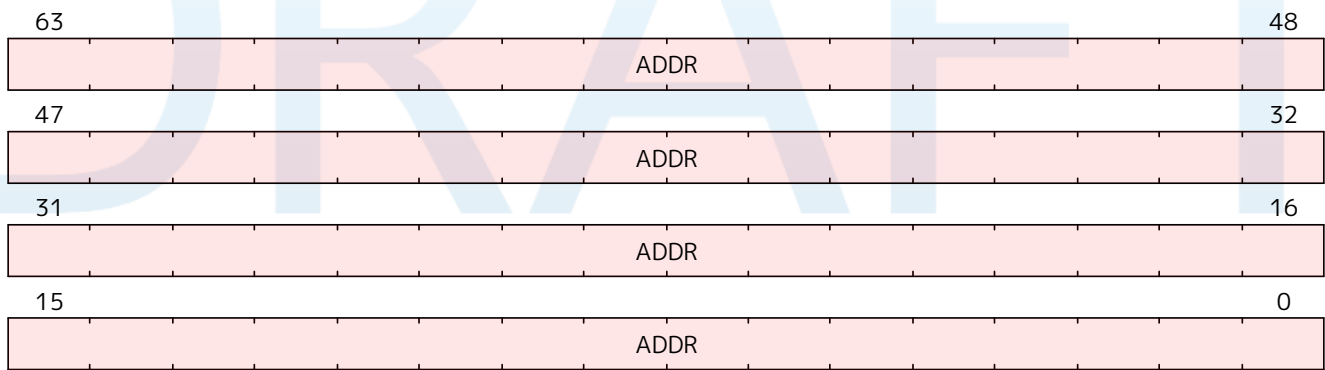


Figure 302. pmpaddr43 Format when CSR[misa].MXL == 1

C.253. pmpaddr44

PMP Address 44

PMP entry address

C.253.1. Attributes

CSR Address	0x3dc
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.253.2. Format

This CSR format changes dynamically with XLEN.

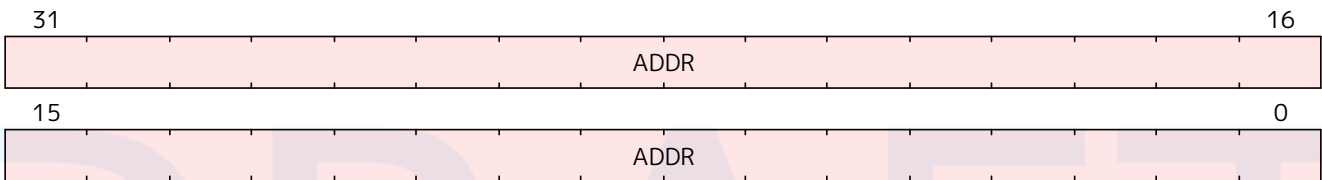


Figure 303. pmpaddr44 Format when CSR[misa].MXL == 0

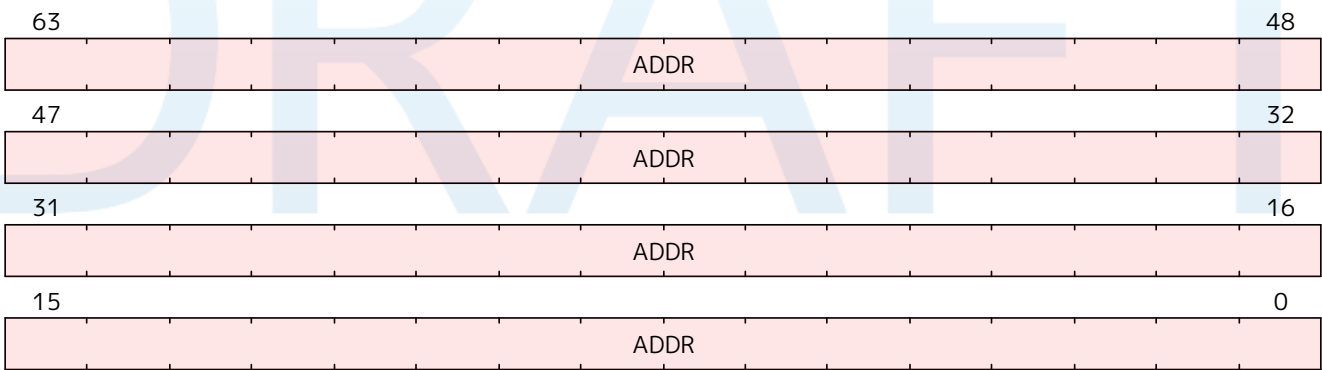


Figure 304. pmpaddr44 Format when CSR[misa].MXL == 1

C.254. pmpaddr45

PMP Address 45

PMP entry address

C.254.1. Attributes

CSR Address	0x3dd
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.254.2. Format

This CSR format changes dynamically with XLEN.

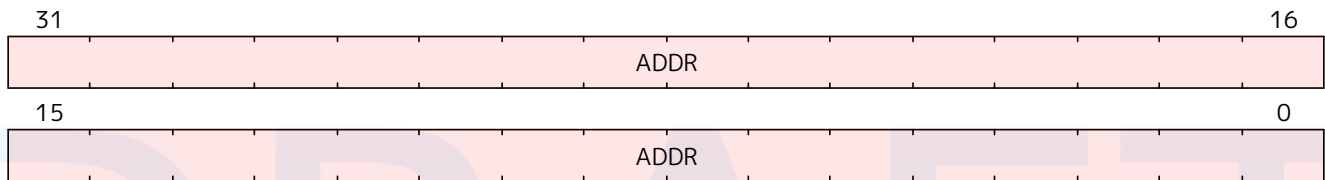


Figure 305. pmpaddr45 Format when CSR[misa].MXL == 0

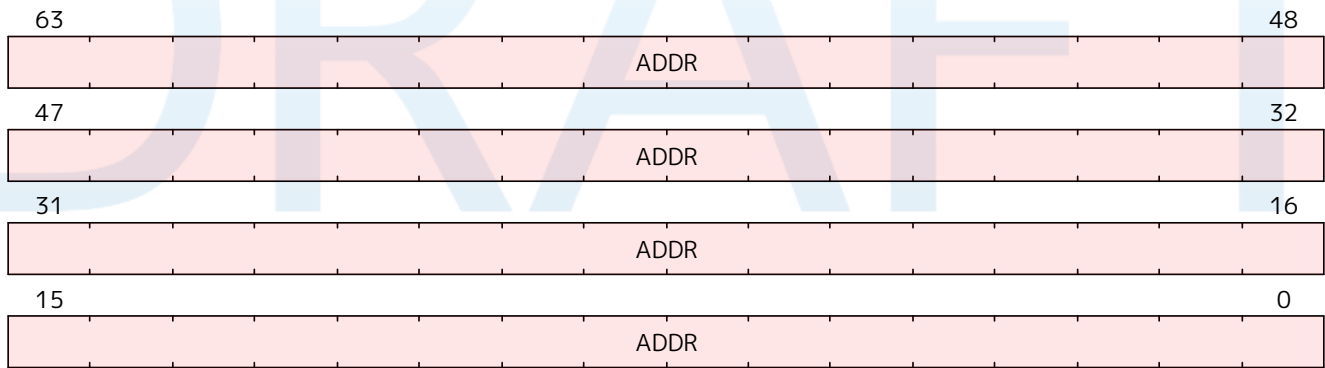


Figure 306. pmpaddr45 Format when CSR[misa].MXL == 1

C.255. pmpaddr46

PMP Address 46

PMP entry address

C.255.1. Attributes

CSR Address	0x3de
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.255.2. Format

This CSR format changes dynamically with XLEN.

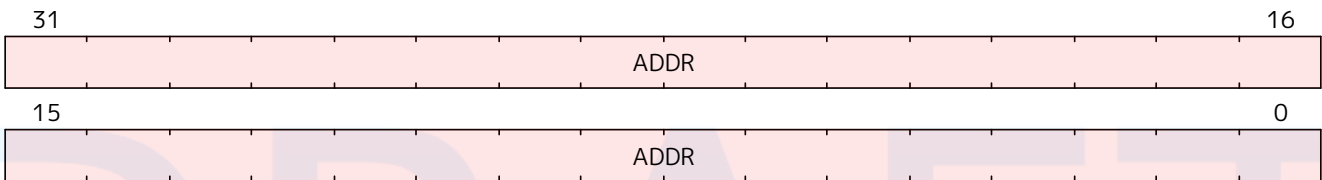


Figure 307. pmpaddr46 Format when CSR[misa].MXL == 0

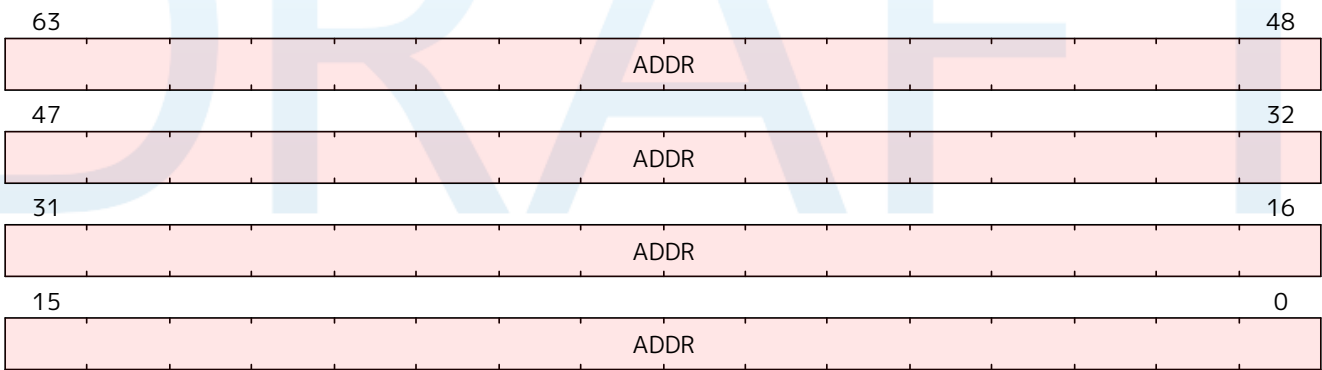


Figure 308. pmpaddr46 Format when CSR[misa].MXL == 1

C.256. pmpaddr47

PMP Address 47

PMP entry address

C.256.1. Attributes

CSR Address	0x3df
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.256.2. Format

This CSR format changes dynamically with XLEN.

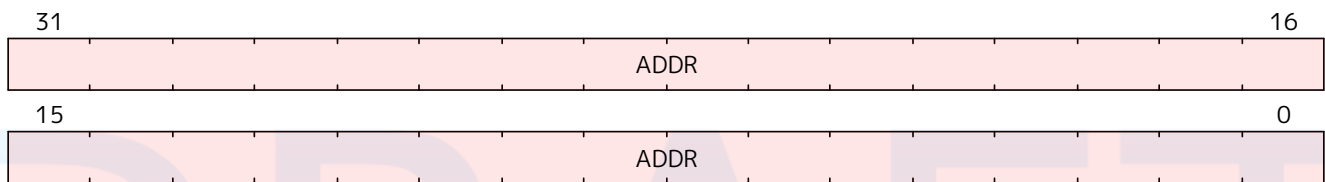


Figure 309. pmpaddr47 Format when CSR[misa].MXL == 0

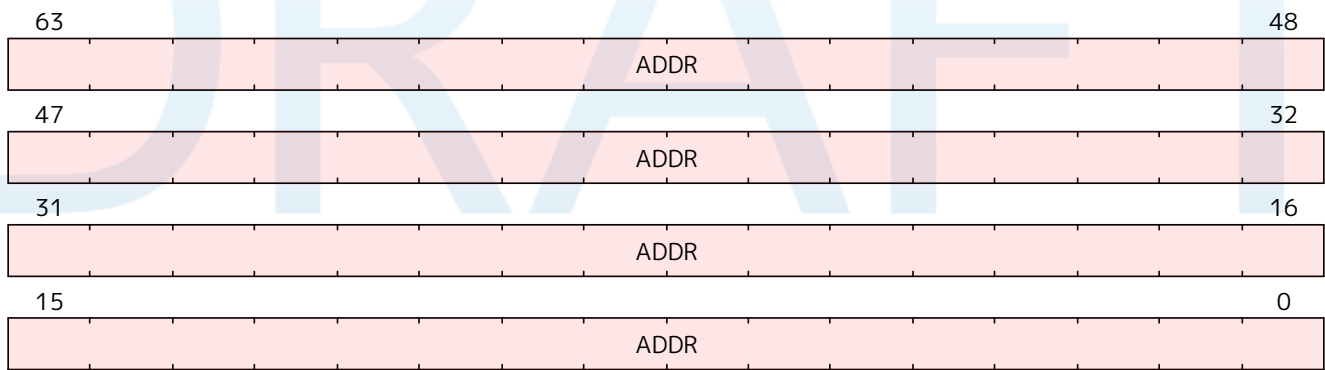


Figure 310. pmpaddr47 Format when CSR[misa].MXL == 1

C.257. pmpaddr48

PMP Address 48

PMP entry address

C.257.1. Attributes

CSR Address	0x3e0
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.257.2. Format

This CSR format changes dynamically with XLEN.

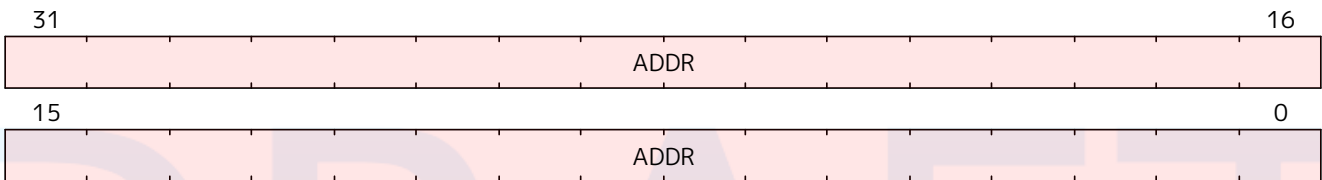


Figure 311. pmpaddr48 Format when CSR[misa].MXL == 0

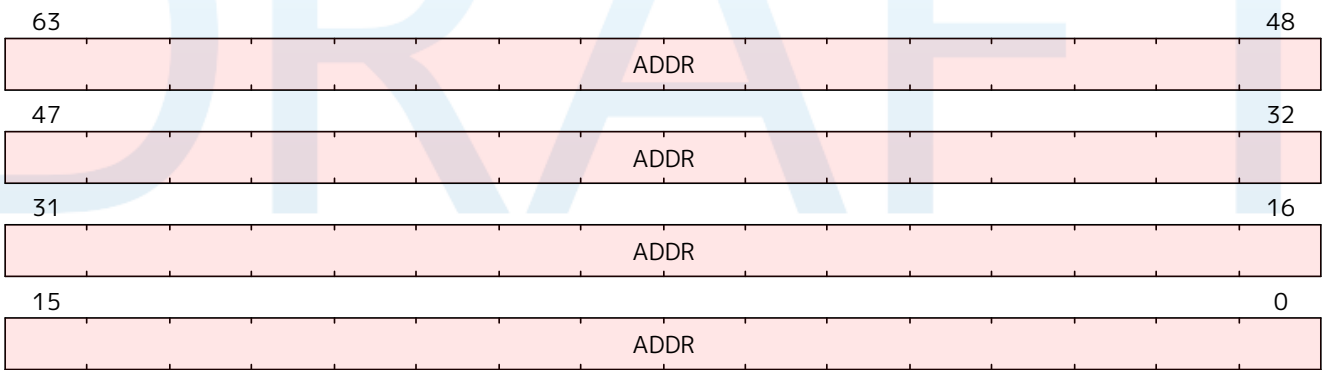


Figure 312. pmpaddr48 Format when CSR[misa].MXL == 1

C.258. pmpaddr49

PMP Address 49

PMP entry address

C.258.1. Attributes

CSR Address	0x3e1
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.258.2. Format

This CSR format changes dynamically with XLEN.

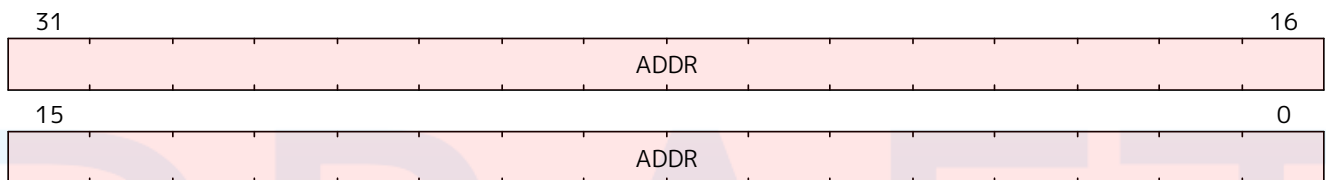


Figure 313. pmpaddr49 Format when CSR[misa].MXL == 0

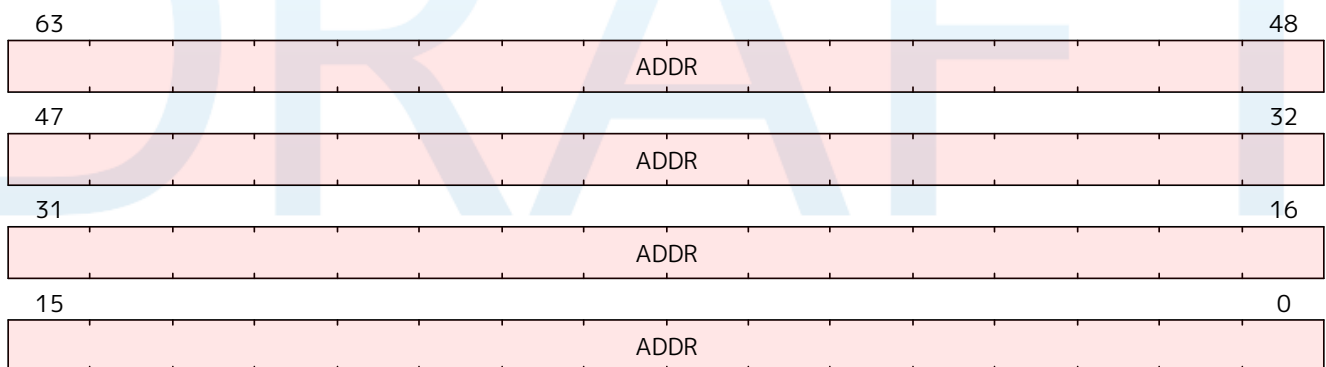


Figure 314. pmpaddr49 Format when CSR[misa].MXL == 1

C.259. pmpaddr5

PMP Address 5

PMP entry address

C.259.1. Attributes

CSR Address	0x3b5
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.259.2. Format

This CSR format changes dynamically with XLEN.

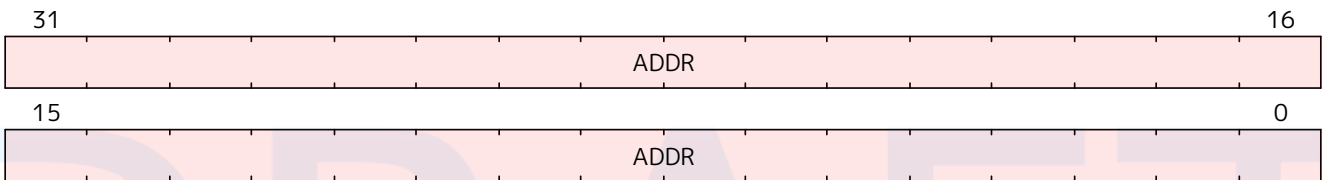


Figure 315. pmpaddr5 Format when CSR[misa].MXL == 0

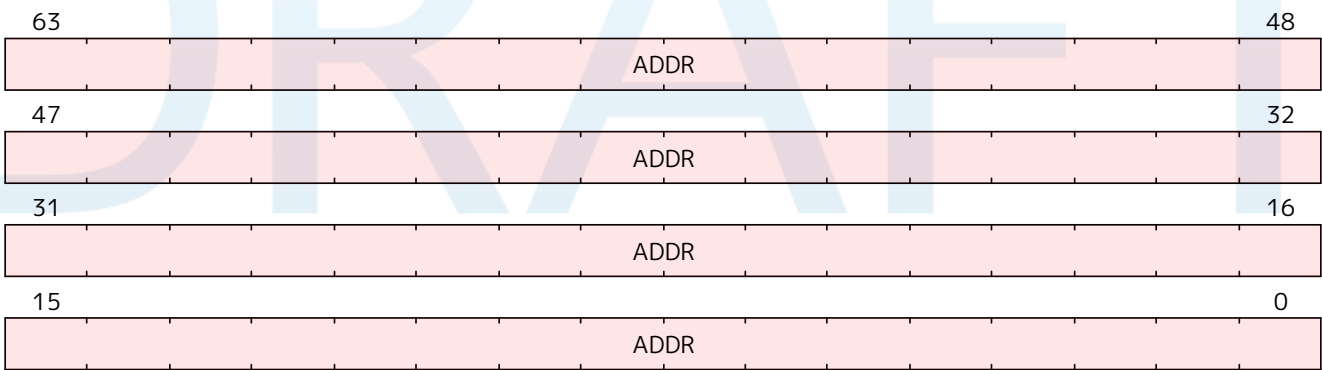


Figure 316. pmpaddr5 Format when CSR[misa].MXL == 1

C.260. pmpaddr50

PMP Address 50

PMP entry address

C.260.1. Attributes

CSR Address	0x3e2
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.260.2. Format

This CSR format changes dynamically with XLEN.

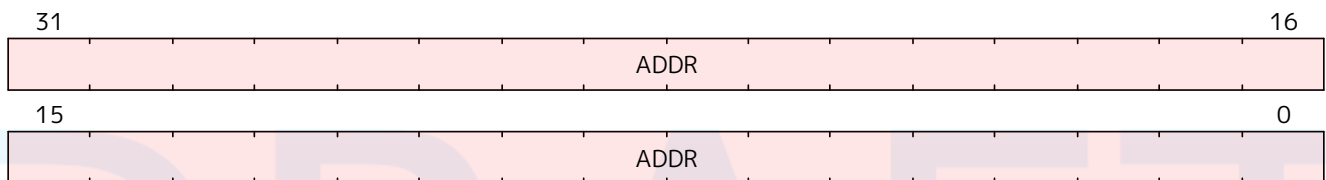


Figure 317. pmpaddr50 Format when CSR[misa].MXL == 0

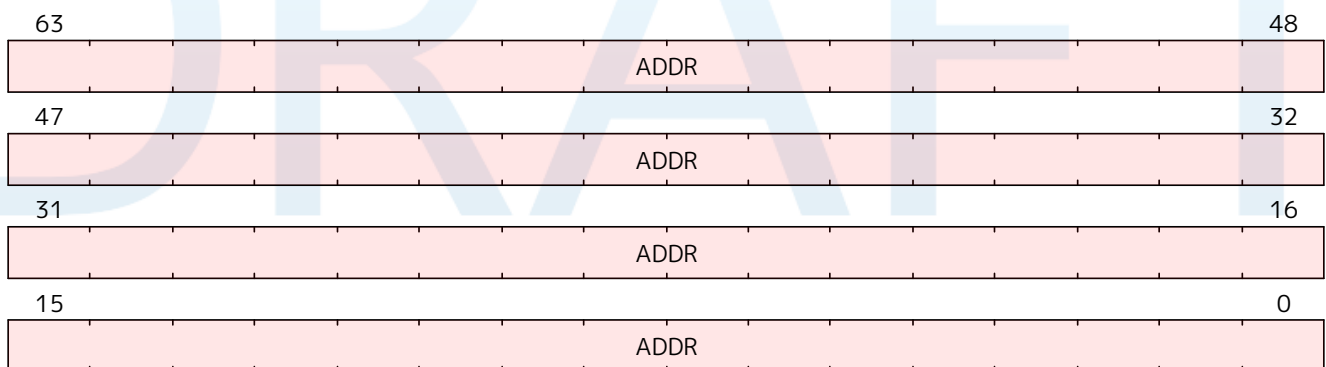


Figure 318. pmpaddr50 Format when CSR[misa].MXL == 1

C.261. pmpaddr51

PMP Address 51

PMP entry address

C.261.1. Attributes

CSR Address	0x3e3
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.261.2. Format

This CSR format changes dynamically with XLEN.

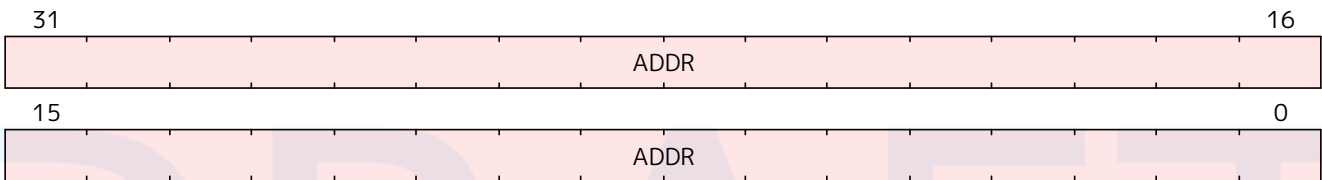


Figure 319. pmpaddr51 Format when CSR[misa].MXL == 0

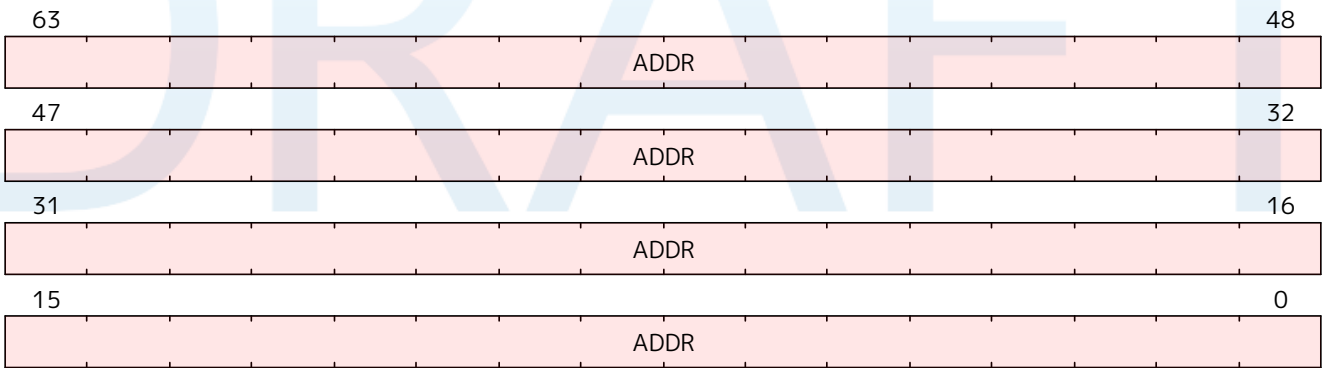


Figure 320. pmpaddr51 Format when CSR[misa].MXL == 1

C.262. pmpaddr52

PMP Address 52

PMP entry address

C.262.1. Attributes

CSR Address	0x3e4
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.262.2. Format

This CSR format changes dynamically with XLEN.

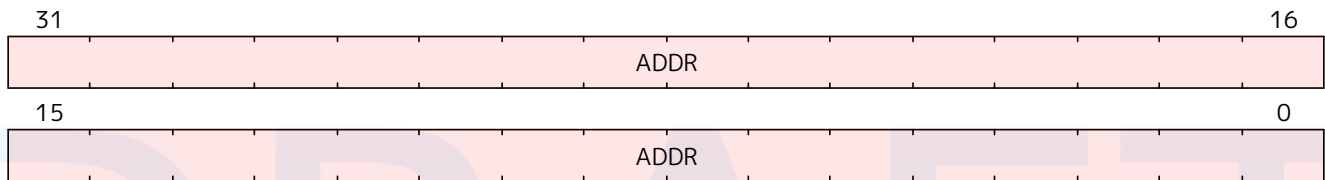


Figure 321. pmpaddr52 Format when CSR[misa].MXL == 0

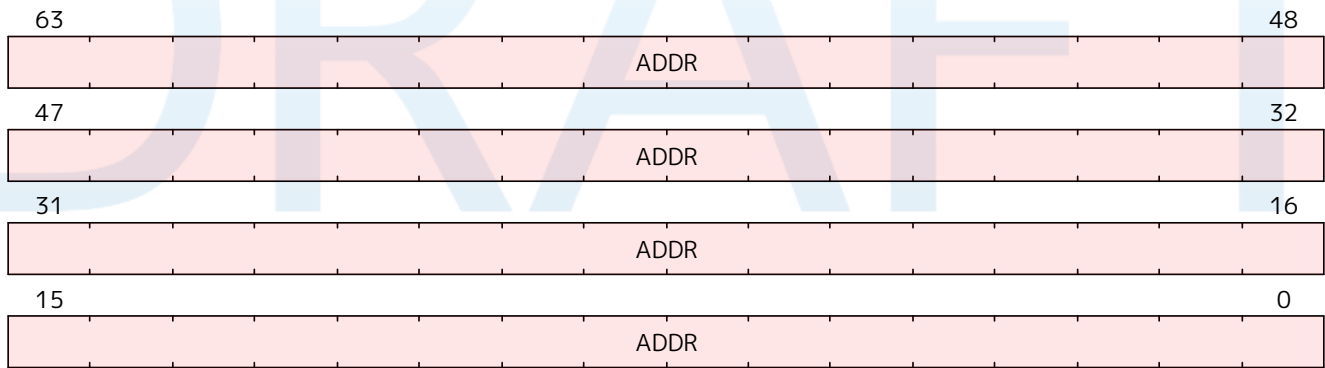


Figure 322. pmpaddr52 Format when CSR[misa].MXL == 1

C.263. pmpaddr53

PMP Address 53

PMP entry address

C.263.1. Attributes

CSR Address	0x3e5
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.263.2. Format

This CSR format changes dynamically with XLEN.

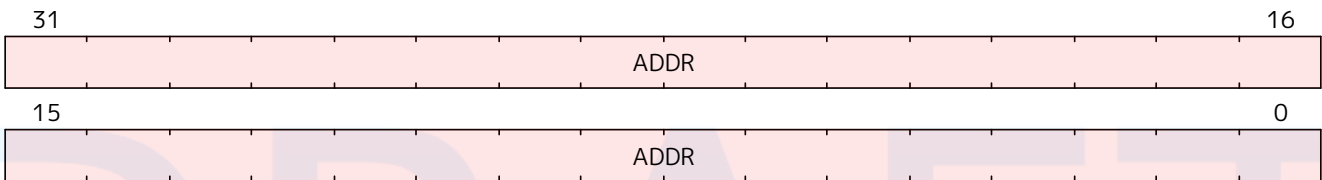


Figure 323. pmpaddr53 Format when CSR[misa].MXL == 0

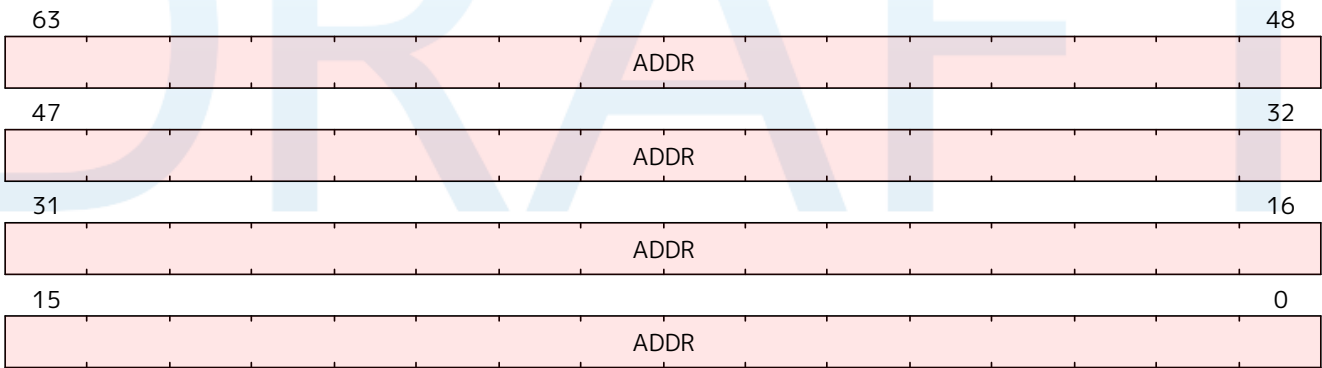


Figure 324. pmpaddr53 Format when CSR[misa].MXL == 1

C.264. pmpaddr54

PMP Address 54

PMP entry address

C.264.1. Attributes

CSR Address	0x3e6
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.264.2. Format

This CSR format changes dynamically with XLEN.

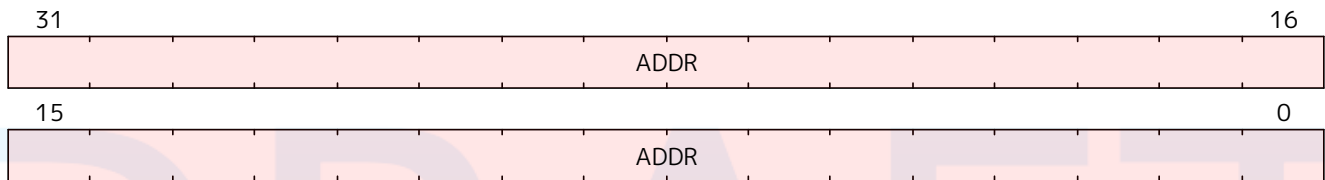


Figure 325. pmpaddr54 Format when CSR[misa].MXL == 0

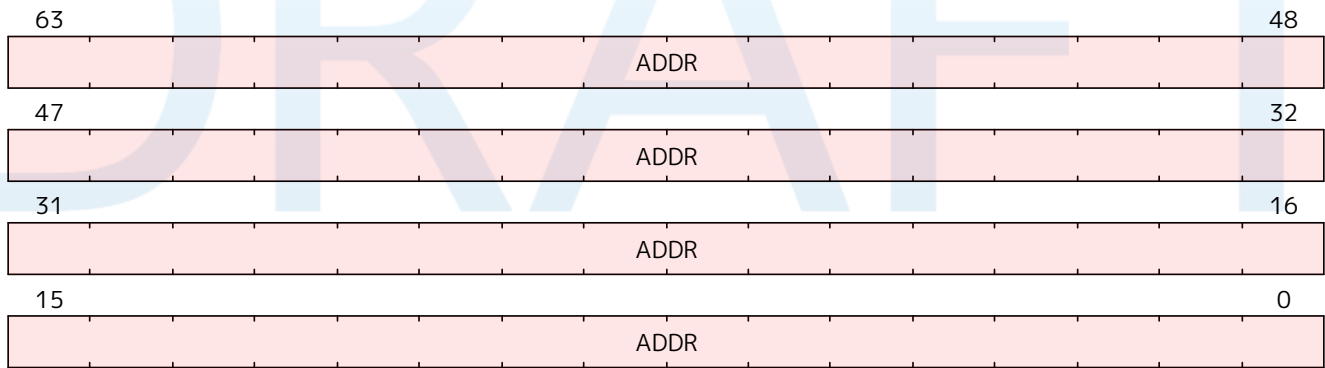


Figure 326. pmpaddr54 Format when CSR[misa].MXL == 1

C.265. pmpaddr55

PMP Address 55

PMP entry address

C.265.1. Attributes

CSR Address	0x3e7
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.265.2. Format

This CSR format changes dynamically with XLEN.

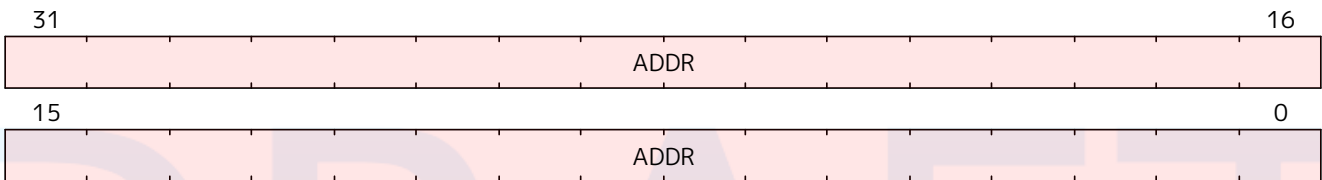


Figure 327. pmpaddr55 Format when CSR[misa].MXL == 0

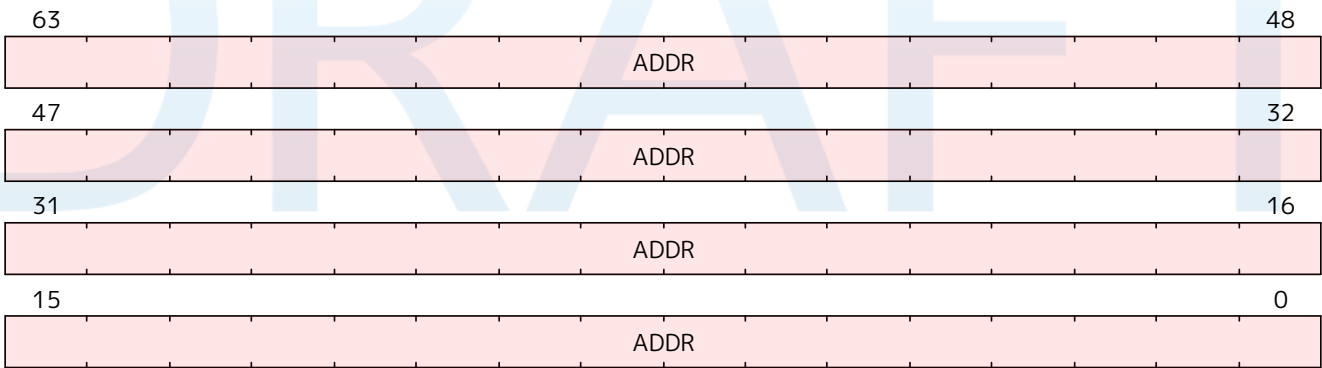


Figure 328. pmpaddr55 Format when CSR[misa].MXL == 1

C.266. pmpaddr56

PMP Address 56

PMP entry address

C.266.1. Attributes

CSR Address	0x3e8
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.266.2. Format

This CSR format changes dynamically with XLEN.

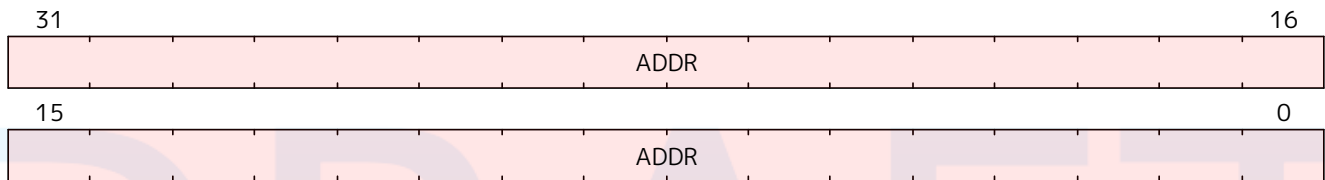


Figure 329. pmpaddr56 Format when CSR[misa].MXL == 0

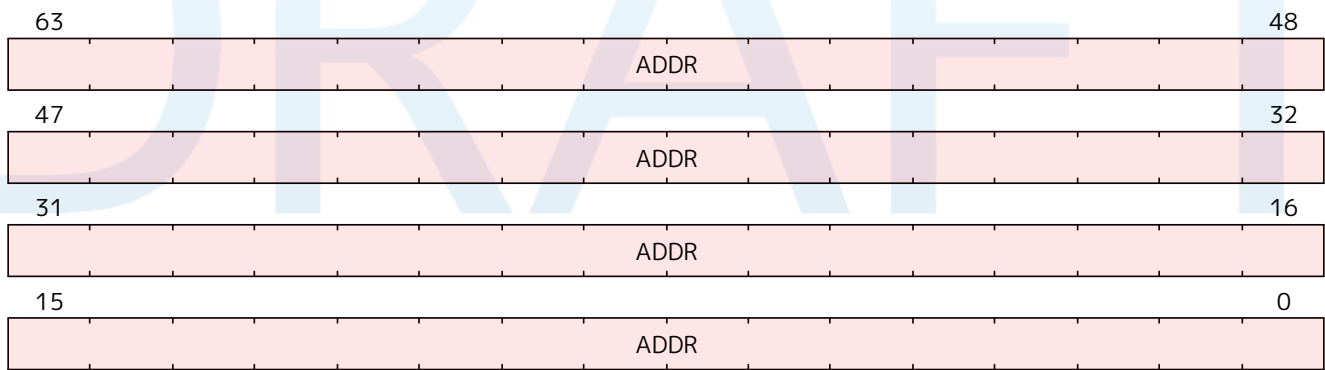


Figure 330. pmpaddr56 Format when CSR[misa].MXL == 1

C.267. pmpaddr57

PMP Address 57

PMP entry address

C.267.1. Attributes

CSR Address	0x3e9
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.267.2. Format

This CSR format changes dynamically with XLEN.

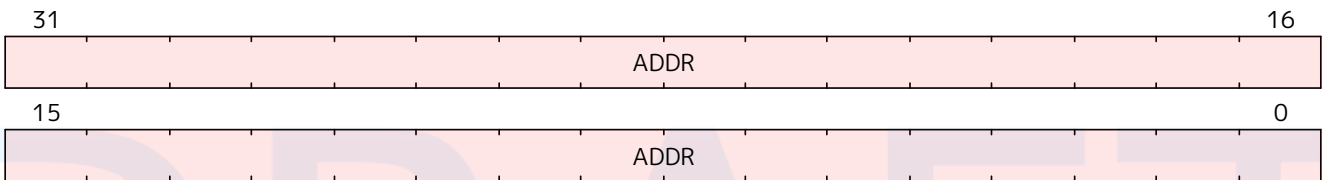


Figure 331. pmpaddr57 Format when CSR[misa].MXL == 0

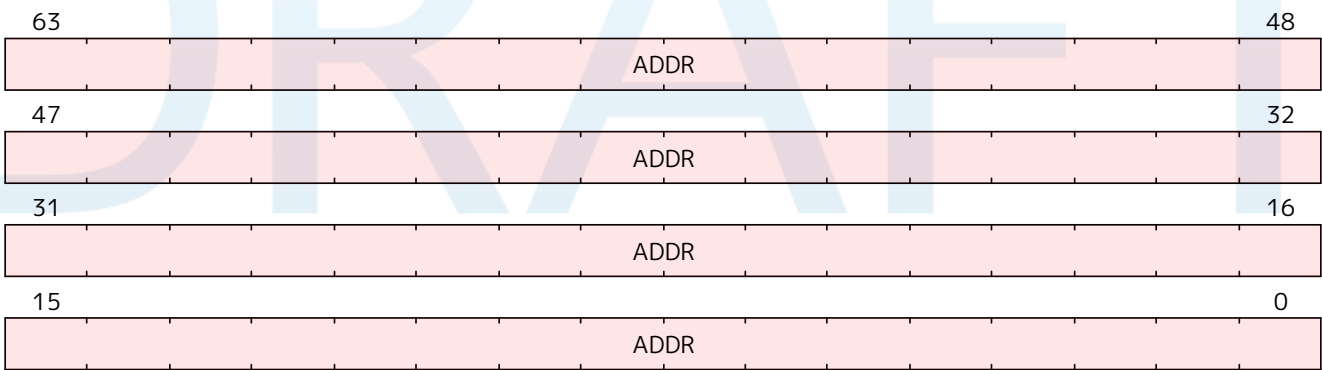


Figure 332. pmpaddr57 Format when CSR[misa].MXL == 1

C.268. pmpaddr58

PMP Address 58

PMP entry address

C.268.1. Attributes

CSR Address	0x3ea
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.268.2. Format

This CSR format changes dynamically with XLEN.

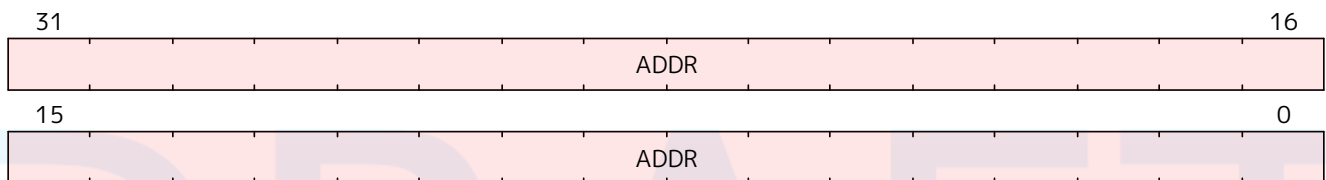


Figure 333. pmpaddr58 Format when CSR[misa].MXL == 0

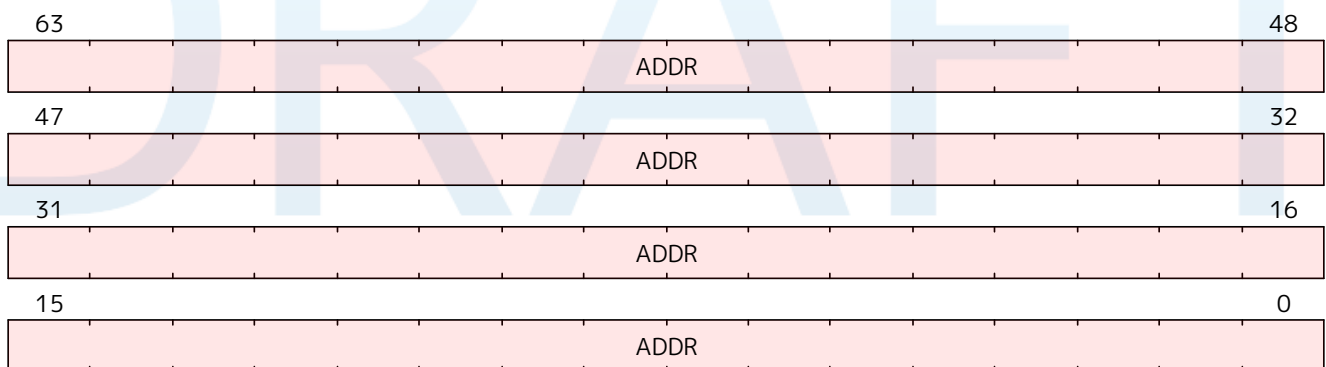


Figure 334. pmpaddr58 Format when CSR[misa].MXL == 1

C.269. pmpaddr59

PMP Address 59

PMP entry address

C.269.1. Attributes

CSR Address	0x3eb
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.269.2. Format

This CSR format changes dynamically with XLEN.

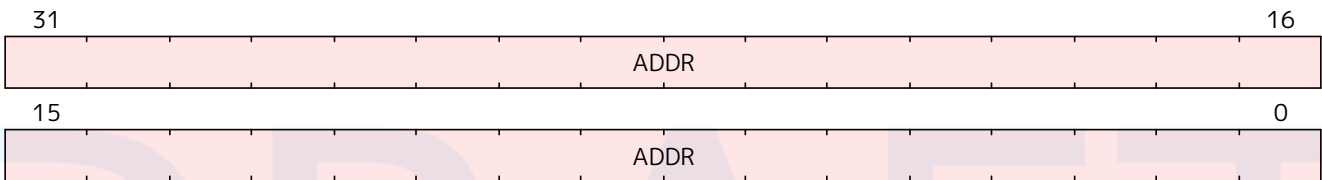


Figure 335. pmpaddr59 Format when CSR[misa].MXL == 0

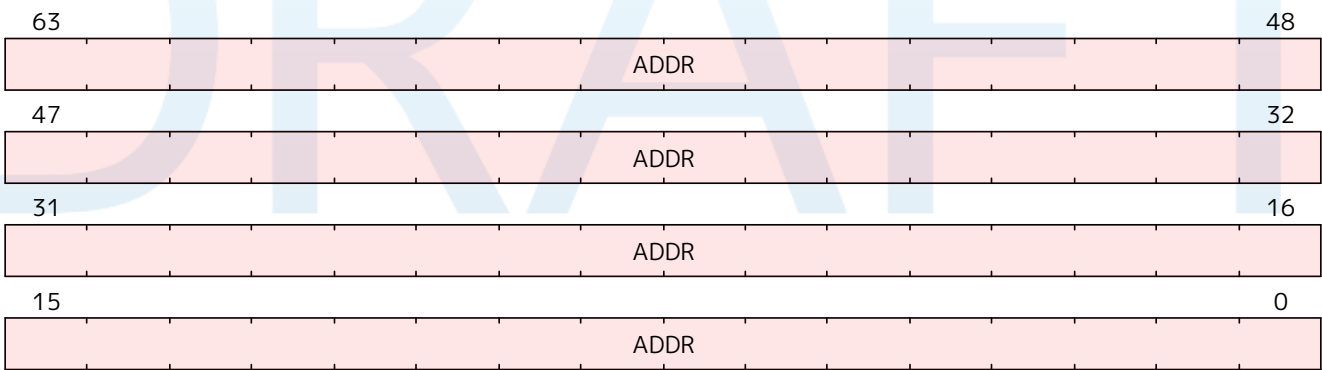


Figure 336. pmpaddr59 Format when CSR[misa].MXL == 1

C.270. pmpaddr6

PMP Address 6

PMP entry address

C.270.1. Attributes

CSR Address	0x3b6
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.270.2. Format

This CSR format changes dynamically with XLEN.

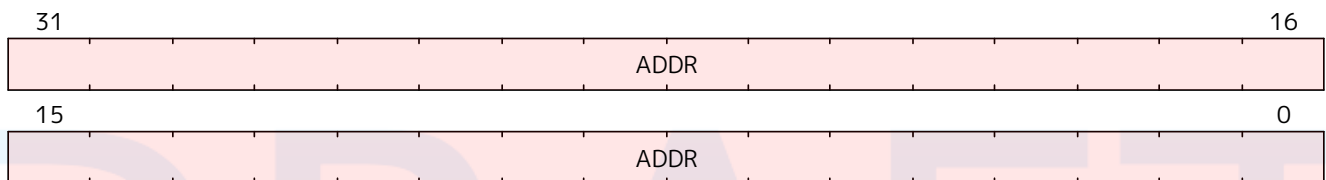


Figure 337. pmpaddr6 Format when CSR[misa].MXL == 0

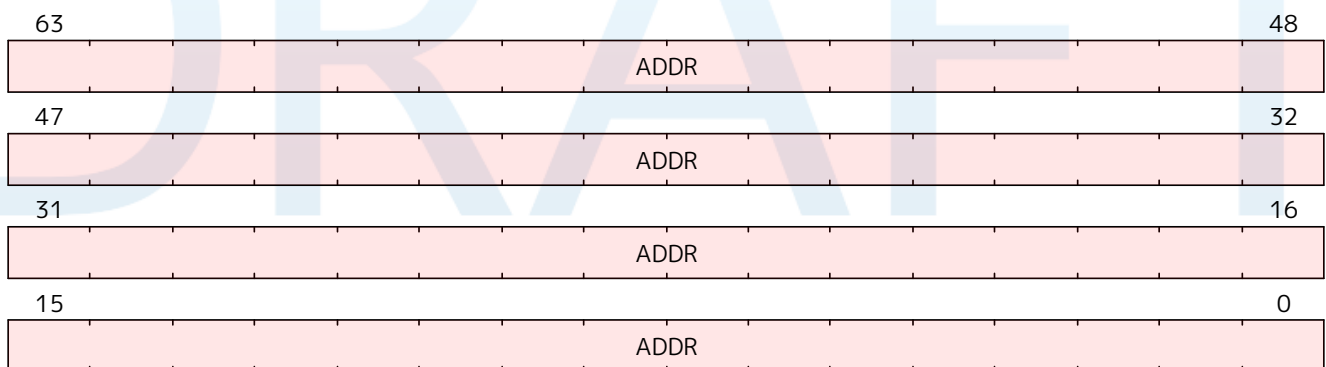


Figure 338. pmpaddr6 Format when CSR[misa].MXL == 1

C.271. pmpaddr60

PMP Address 60

PMP entry address

C.271.1. Attributes

CSR Address	0x3ec
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.271.2. Format

This CSR format changes dynamically with XLEN.

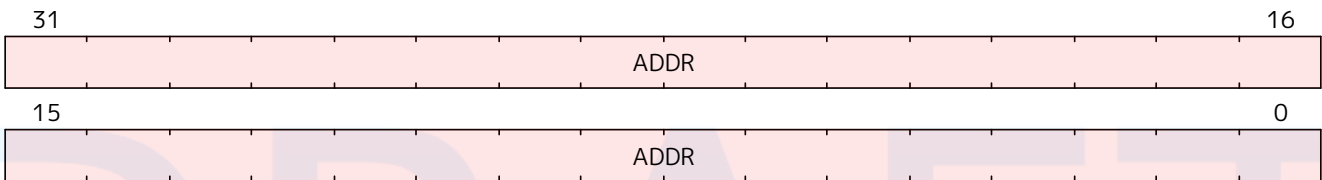


Figure 339. pmpaddr60 Format when CSR[misa].MXL == 0

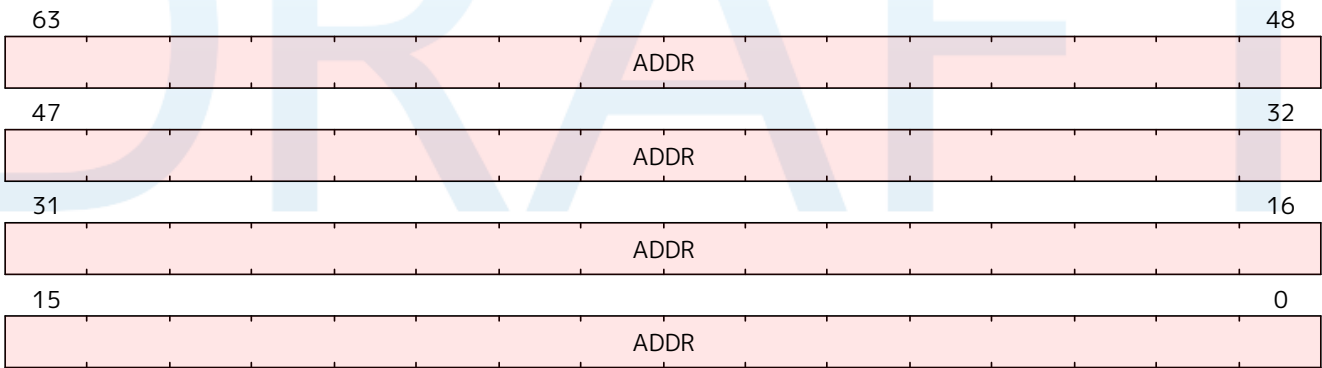


Figure 340. pmpaddr60 Format when CSR[misa].MXL == 1

C.272. pmpaddr61

PMP Address 61

PMP entry address

C.272.1. Attributes

CSR Address	0x3ed
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.272.2. Format

This CSR format changes dynamically with XLEN.

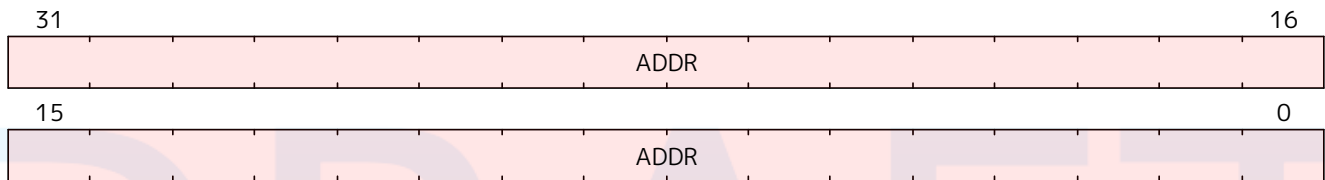


Figure 341. pmpaddr61 Format when CSR[misa].MXL == 0

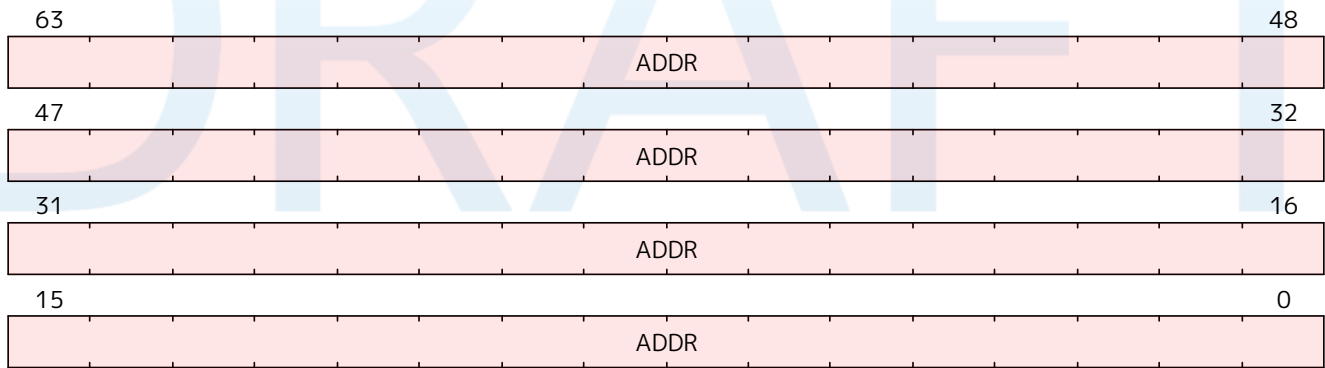


Figure 342. pmpaddr61 Format when CSR[misa].MXL == 1

C.273. pmpaddr62

PMP Address 62

PMP entry address

C.273.1. Attributes

CSR Address	0x3ee
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.273.2. Format

This CSR format changes dynamically with XLEN.

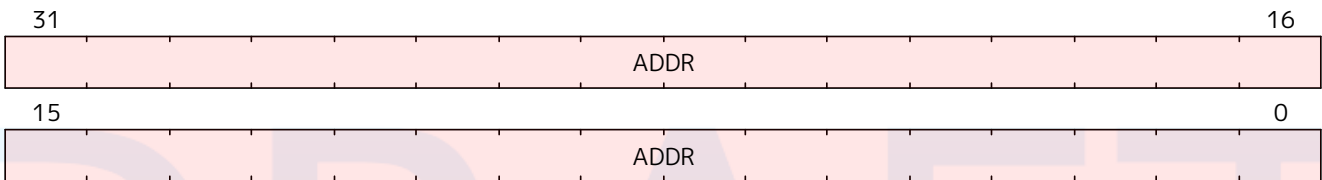


Figure 343. pmpaddr62 Format when CSR[misa].MXL == 0

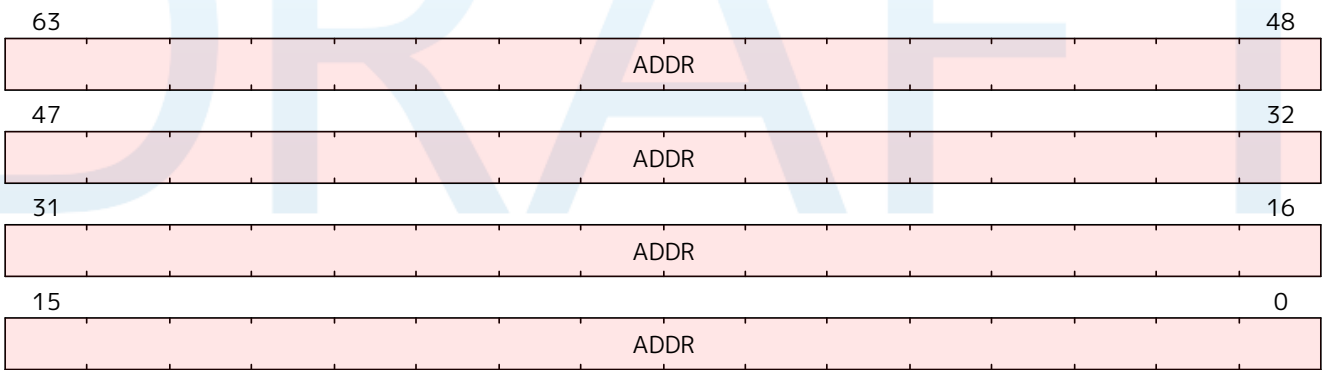


Figure 344. pmpaddr62 Format when CSR[misa].MXL == 1

C.274. pmpaddr63

PMP Address 63

PMP entry address

C.274.1. Attributes

CSR Address	0x3ef
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.274.2. Format

This CSR format changes dynamically with XLEN.

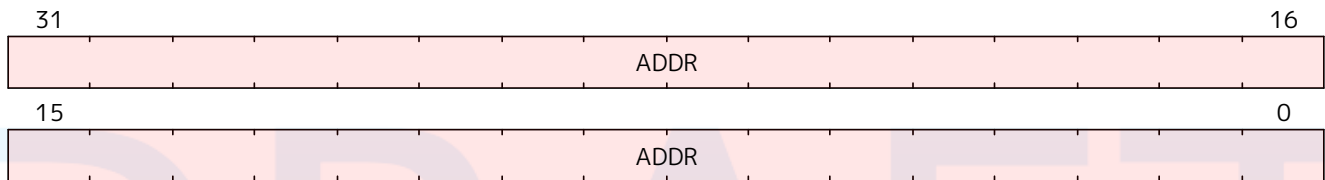


Figure 345. pmpaddr63 Format when CSR[misa].MXL == 0

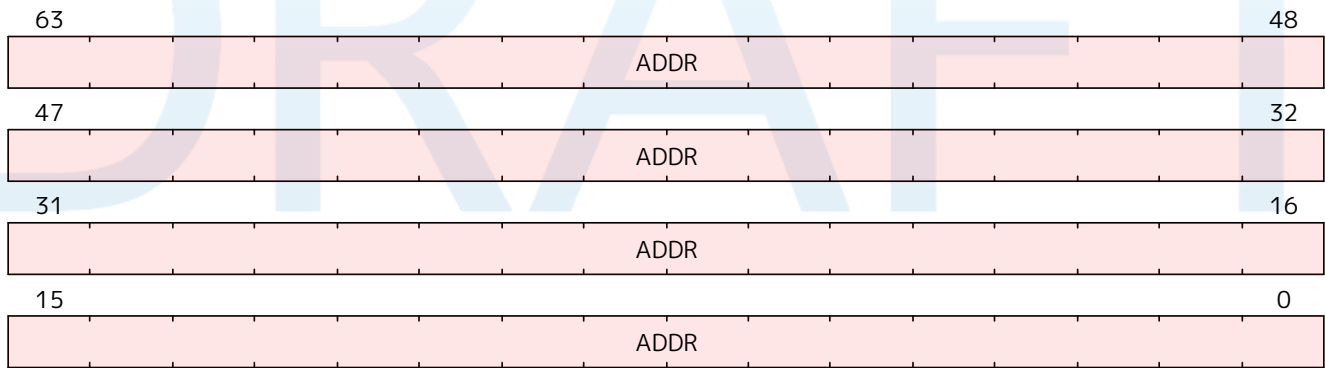


Figure 346. pmpaddr63 Format when CSR[misa].MXL == 1

C.275. pmpaddr7

PMP Address 7

PMP entry address

C.275.1. Attributes

CSR Address	0x3b7
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.275.2. Format

This CSR format changes dynamically with XLEN.

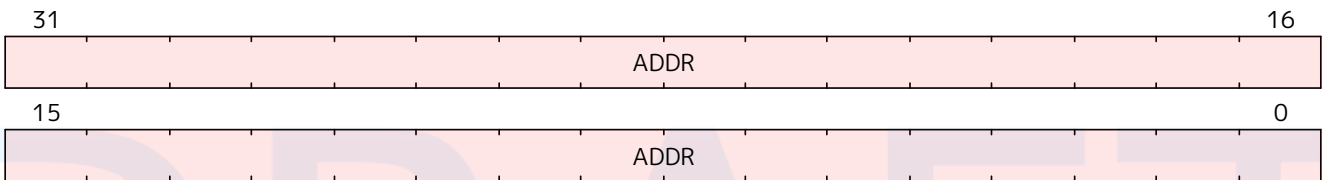


Figure 347. pmpaddr7 Format when CSR[misa].MXL == 0

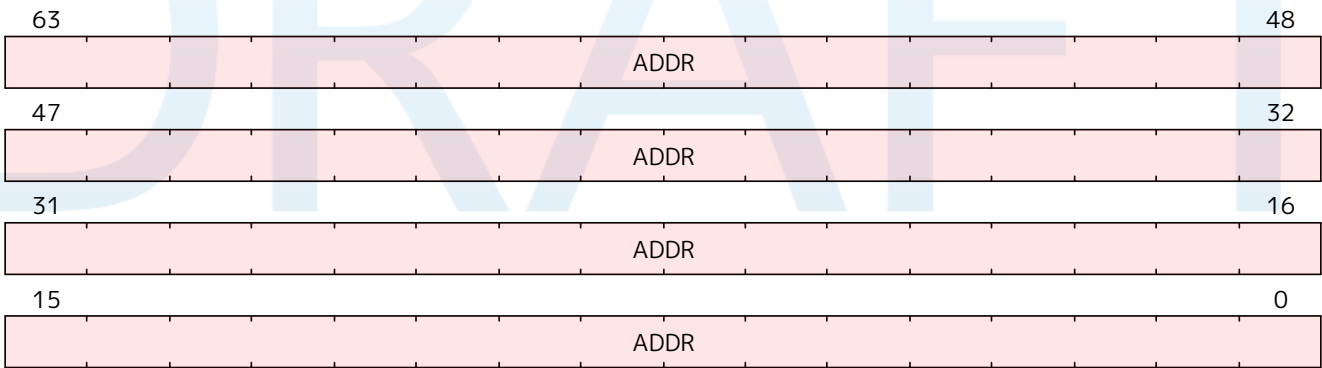


Figure 348. pmpaddr7 Format when CSR[misa].MXL == 1

C.276. pmpaddr8

PMP Address 8

PMP entry address

C.276.1. Attributes

CSR Address	0x3b8
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.276.2. Format

This CSR format changes dynamically with XLEN.

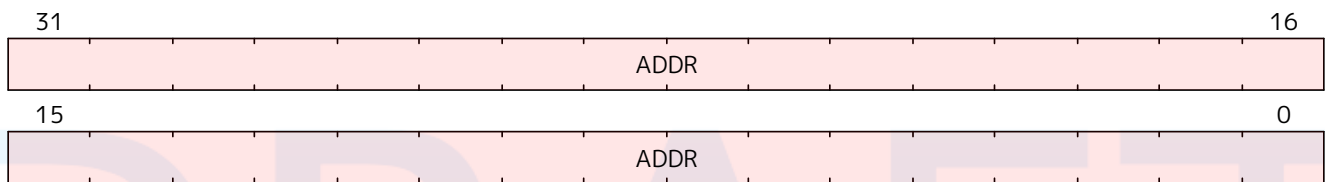


Figure 349. pmpaddr8 Format when CSR[misa].MXL == 0

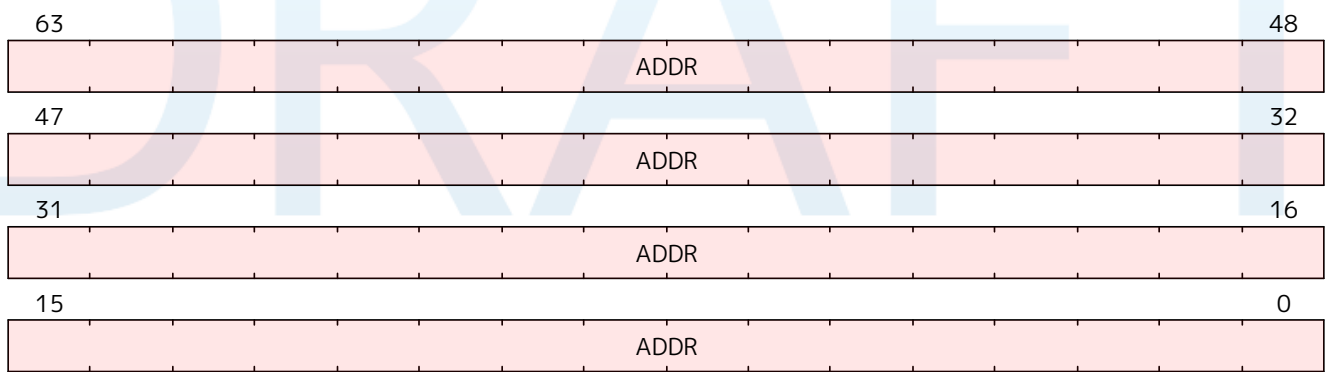


Figure 350. pmpaddr8 Format when CSR[misa].MXL == 1

C.277. pmpaddr9

PMP Address 9

PMP entry address

C.277.1. Attributes

CSR Address	0x3b9
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.277.2. Format

This CSR format changes dynamically with XLEN.

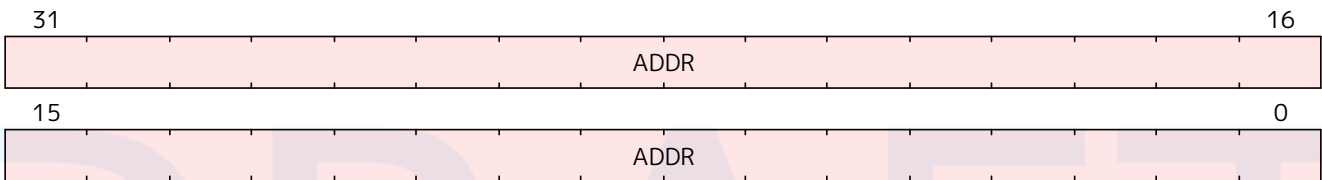


Figure 351. pmpaddr9 Format when CSR[misa].MXL == 0

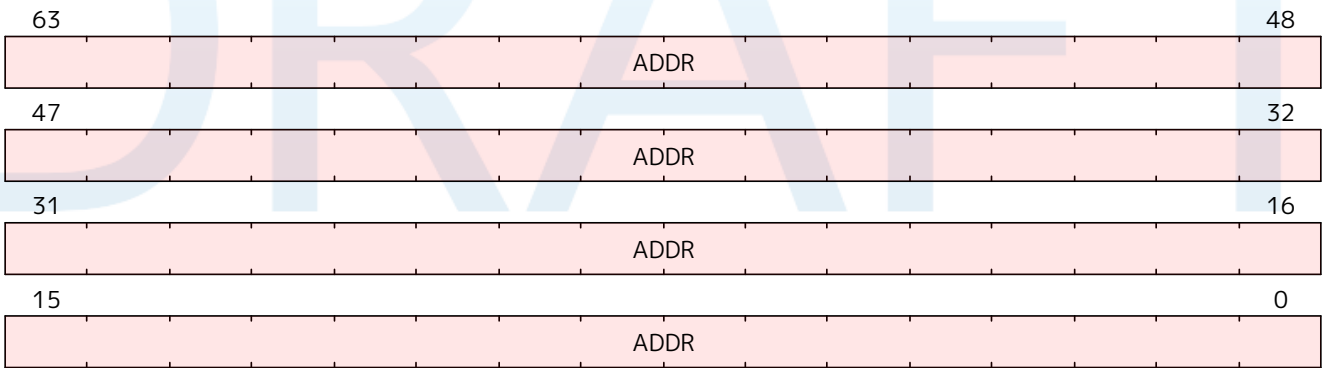


Figure 352. pmpaddr9 Format when CSR[misa].MXL == 1

C.278. pmpcfg0

PMP Configuration Register 0

PMP entry configuration

C.278.1. Attributes

CSR Address	0x3a0
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.278.2. Format

This CSR format changes dynamically with XLEN.

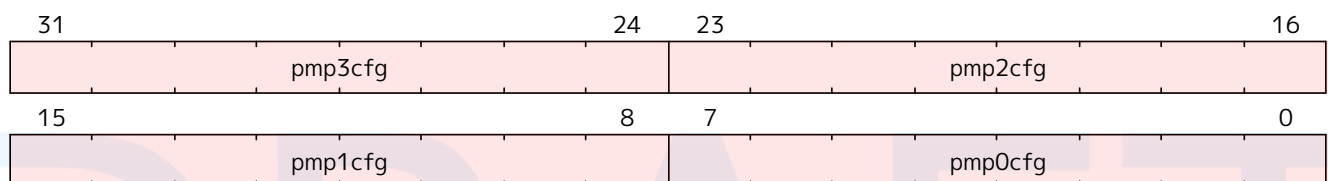


Figure 353. pmpcfg0 Format when CSR[misa].MXL == 0

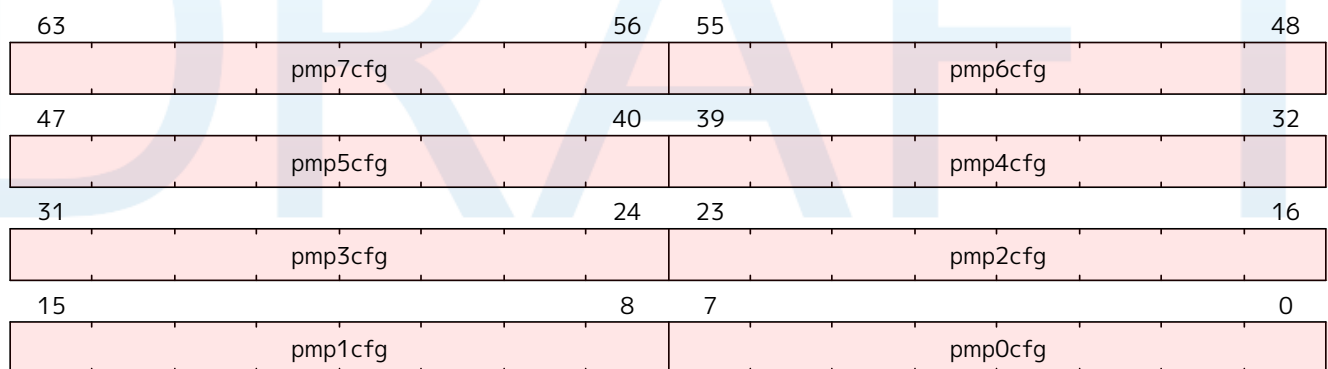



Figure 354. pmpcfg0 Format when CSR[misa].MXL == 1

C.279. pmpcfg1

PMP Configuration Register 1



pmpcfg1 is only defined in RV32.

PMP entry configuration

C.279.1. Attributes

CSR Address	0x3a1
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.279.2. Format

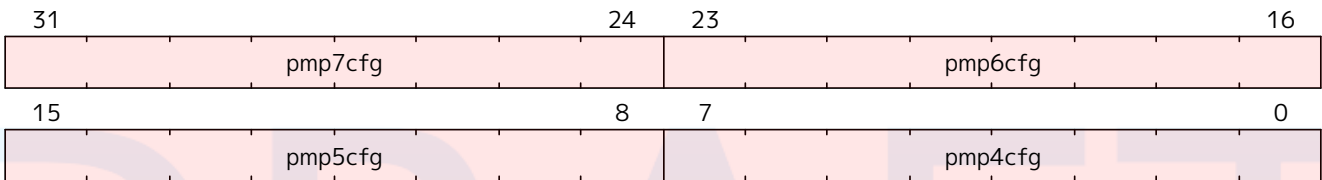


Figure 355. pmpcfg1 format

C.280. pmpcfg10

PMP Configuration Register 10

PMP entry configuration

C.280.1. Attributes

CSR Address	0x3aa
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.280.2. Format

This CSR format changes dynamically with XLEN.

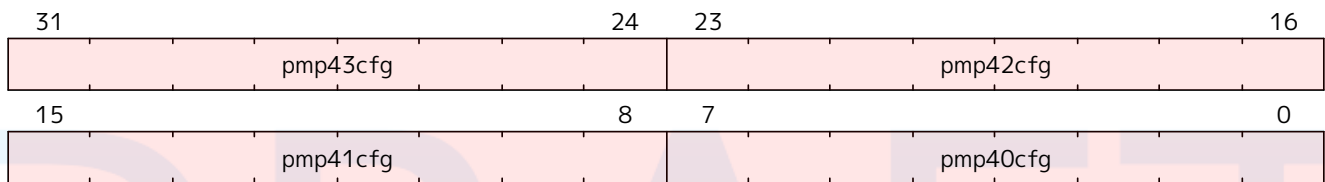


Figure 356. pmpcfg10 Format when CSR[misa].MXL == 0

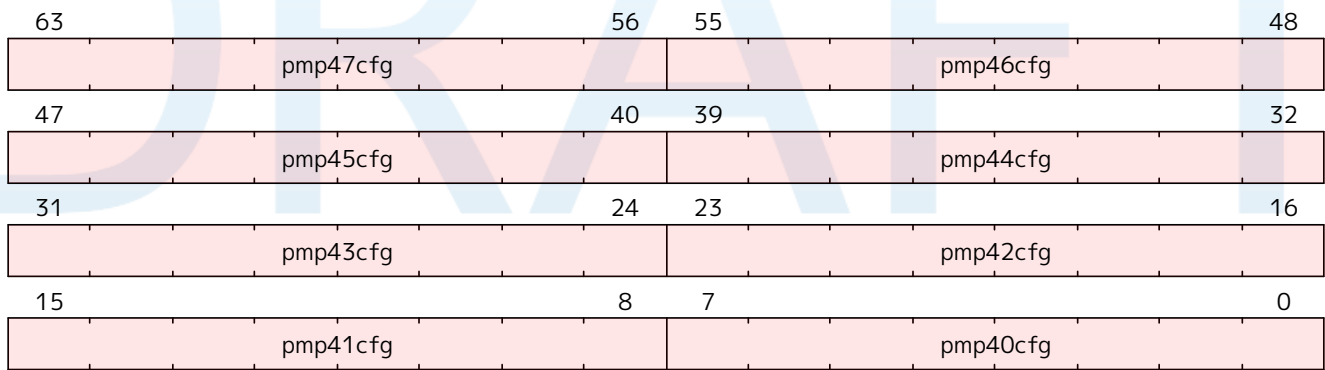



Figure 357. pmpcfg10 Format when CSR[misa].MXL == 1

C.281. pmpcfg11

PMP Configuration Register 11



pmpcfg11 is only defined in RV32.

PMP entry configuration

C.281.1. Attributes

CSR Address	0x3ab
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.281.2. Format

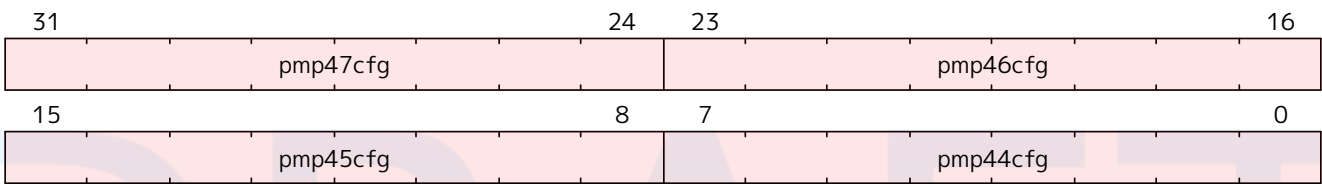


Figure 358. *pmpcfg11* format

C.282. pmpcfg12

PMP Configuration Register 12

PMP entry configuration

C.282.1. Attributes

CSR Address	0x3ac
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.282.2. Format

This CSR format changes dynamically with XLEN.

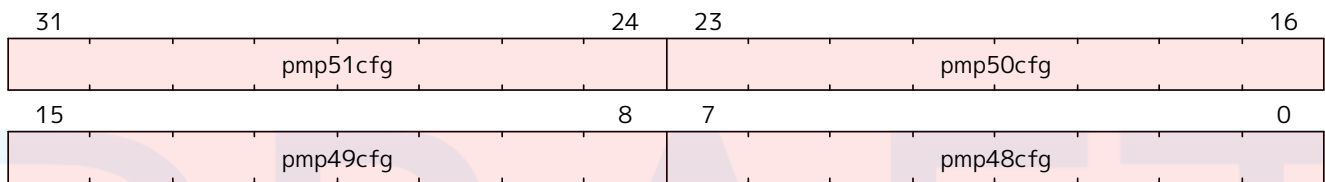


Figure 359. pmpcfg12 Format when CSR[misa].MXL == 0

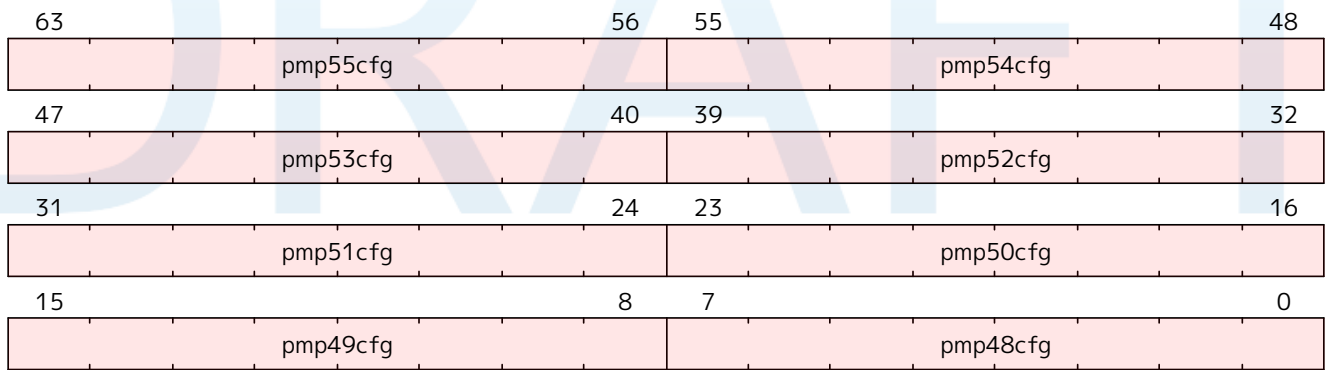



Figure 360. pmpcfg12 Format when CSR[misa].MXL == 1

C.283. pmpcfg13

PMP Configuration Register 13

 *pmpcfg13* is only defined in RV32.

PMP entry configuration

C.283.1. Attributes

CSR Address	0x3ad
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.283.2. Format

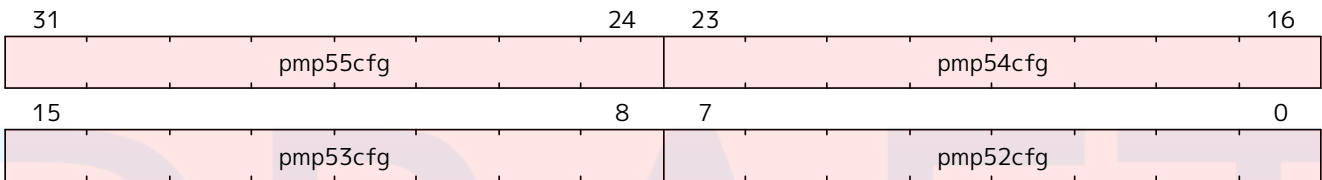


Figure 361. pmpcfg13 format

C.284. pmpcfg14

PMP Configuration Register 14

PMP entry configuration

C.284.1. Attributes

CSR Address	0x3ae
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.284.2. Format

This CSR format changes dynamically with XLEN.

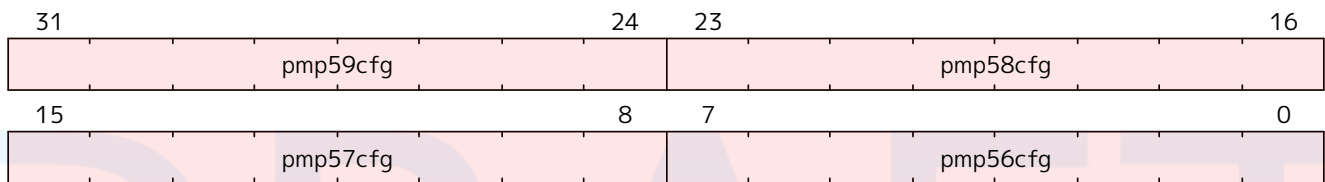


Figure 362. pmpcfg14 Format when CSR[misa].MXL == 0

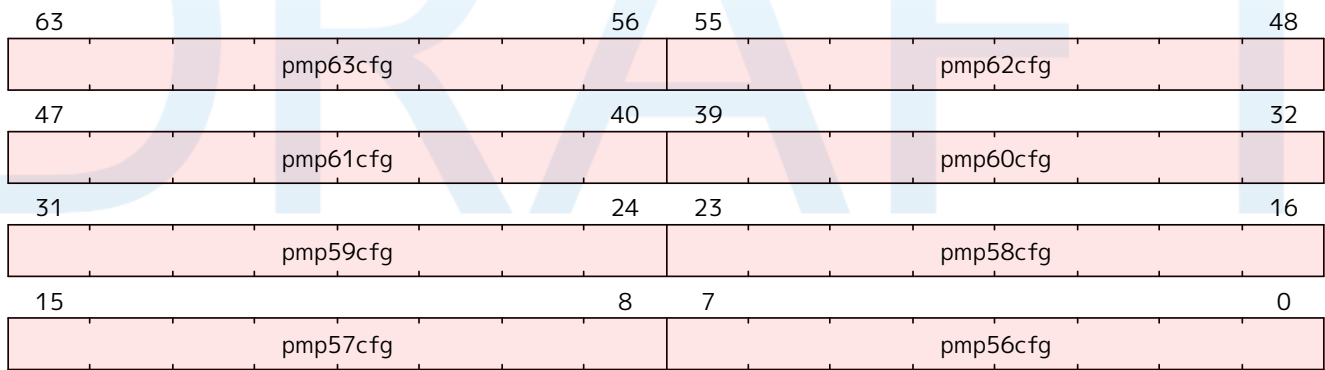



Figure 363. pmpcfg14 Format when CSR[misa].MXL == 1

C.285. pmpcfg15

PMP Configuration Register 15

 *pmpcfg15* is only defined in RV32.

PMP entry configuration

C.285.1. Attributes

CSR Address	0x3af
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.285.2. Format

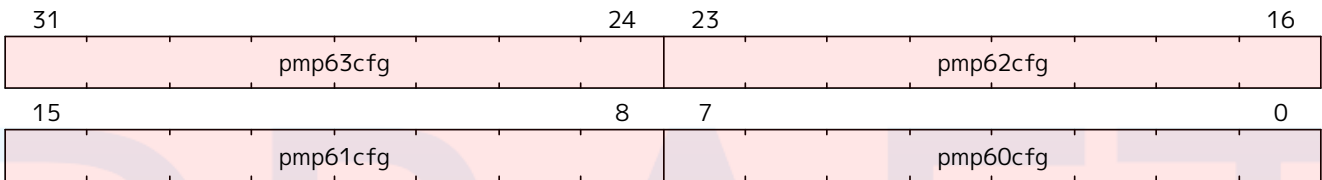


Figure 364. *pmpcfg15* format

C.286. pmpcfg2

PMP Configuration Register 2

PMP entry configuration

C.286.1. Attributes

CSR Address	0x3a2
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.286.2. Format

This CSR format changes dynamically with XLEN.

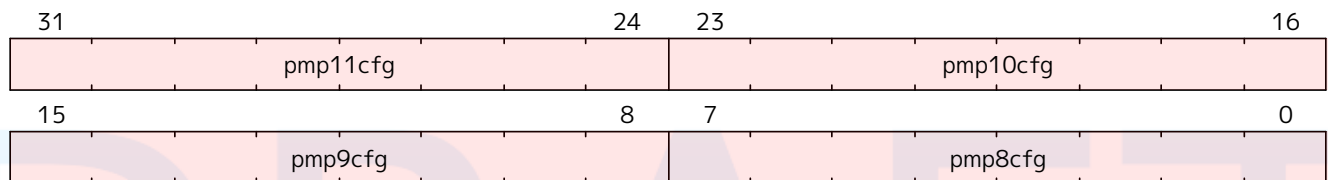


Figure 365. pmpcfg2 Format when CSR[misa].MXL == 0

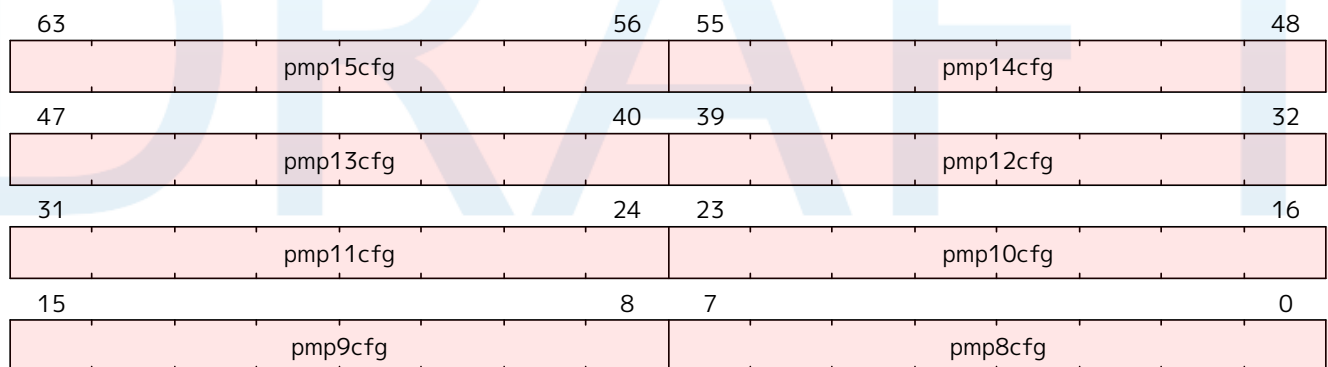



Figure 366. pmpcfg2 Format when CSR[misa].MXL == 1

C.287. pmpcfg3

PMP Configuration Register 3



pmpcfg3 is only defined in RV32.

PMP entry configuration

C.287.1. Attributes

CSR Address	0x3a3
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.287.2. Format

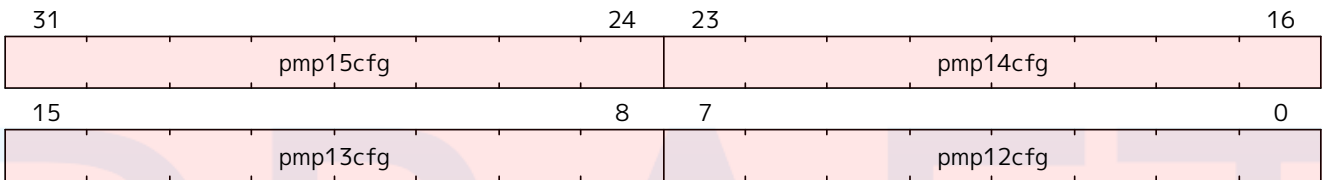


Figure 367. pmpcfg3 format

C.288. pmpcfg4

PMP Configuration Register 4

PMP entry configuration

C.288.1. Attributes

CSR Address	0x3a4
Defining extension	I \geq 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.288.2. Format

This CSR format changes dynamically with XLEN.

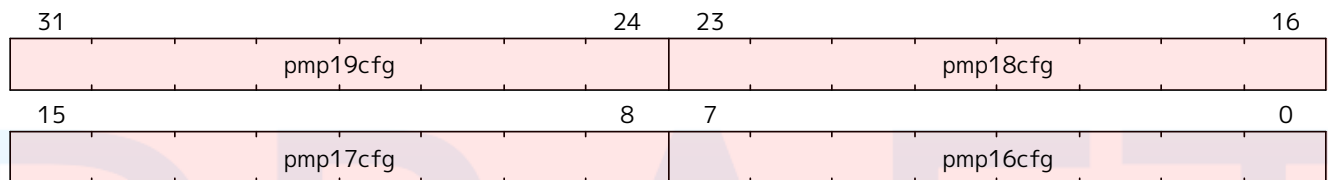


Figure 368. pmpcfg4 Format when CSR[misa].MXL == 0

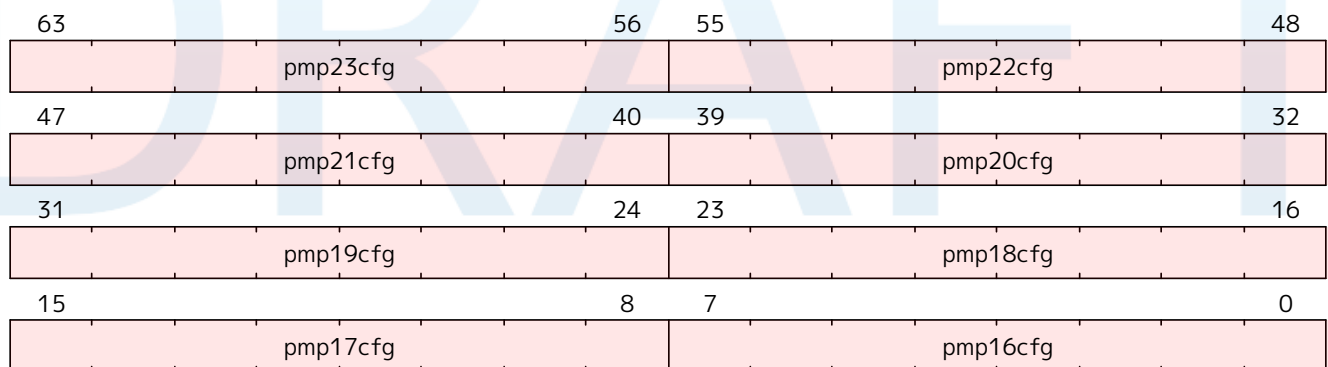



Figure 369. pmpcfg4 Format when CSR[misa].MXL == 1

C.289. pmpcfg5

PMP Configuration Register 5



pmpcfg5 is only defined in RV32.

PMP entry configuration

C.289.1. Attributes

CSR Address	0x3a5
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.289.2. Format

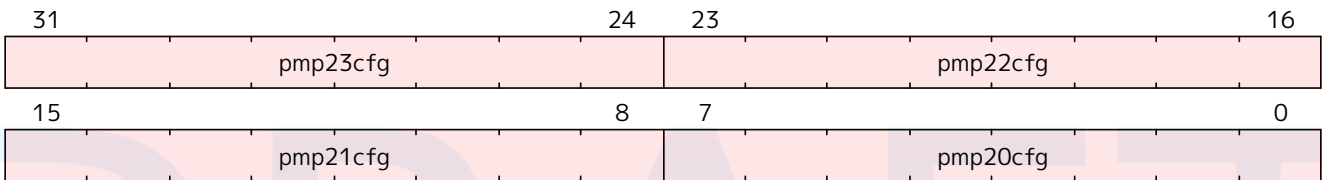


Figure 370. *pmpcfg5* format

C.290. pmpcfg6

PMP Configuration Register 6

PMP entry configuration

C.290.1. Attributes

CSR Address	0x3a6
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.290.2. Format

This CSR format changes dynamically with XLEN.

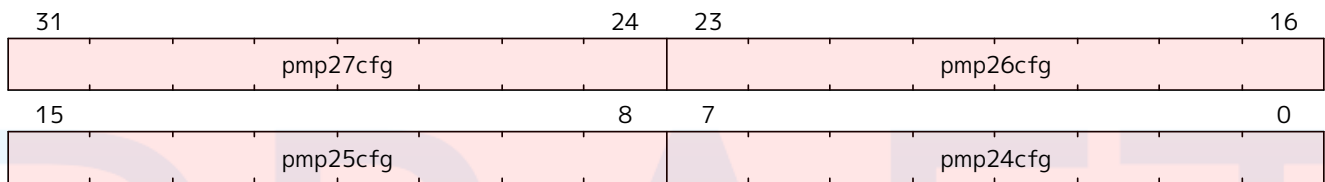


Figure 371. pmpcfg6 Format when CSR[misa].MXL == 0

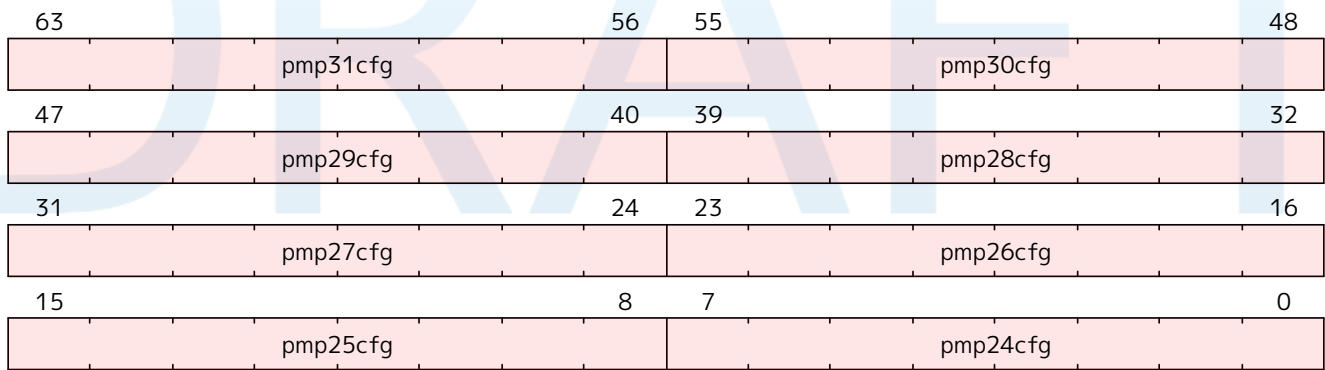



Figure 372. pmpcfg6 Format when CSR[misa].MXL == 1

C.291. pmpcfg7

PMP Configuration Register 7



pmpcfg7 is only defined in RV32.

PMP entry configuration

C.291.1. Attributes

CSR Address	0x3a7
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.291.2. Format

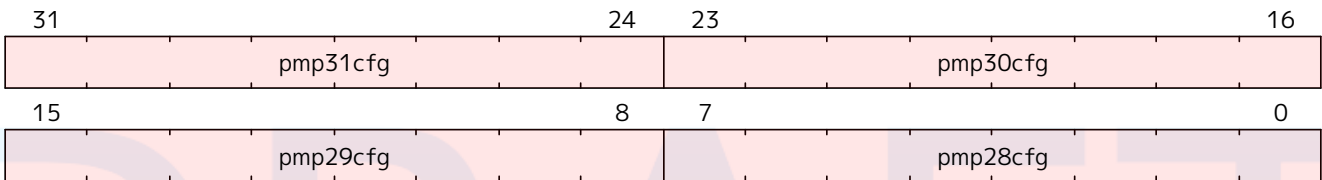


Figure 373. *pmpcfg7* format

C.292. pmpcfg8

PMP Configuration Register 8

PMP entry configuration

C.292.1. Attributes

CSR Address	0x3a8
Defining extension	I >= 0
Length	32 when CSR[misa].MXL == 0 64 when CSR[misa].MXL == 1
Privilege Mode	M

C.292.2. Format

This CSR format changes dynamically with XLEN.

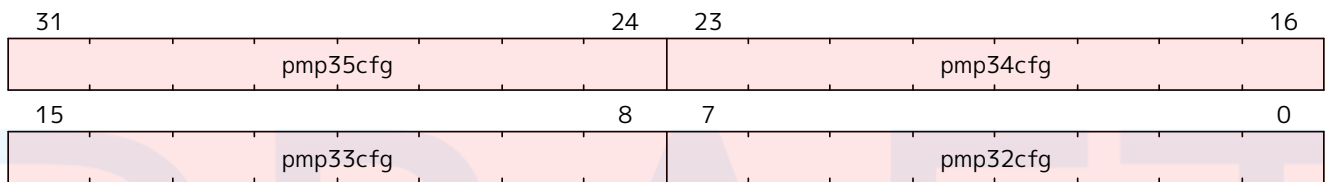


Figure 374. pmpcfg8 Format when CSR[misa].MXL == 0

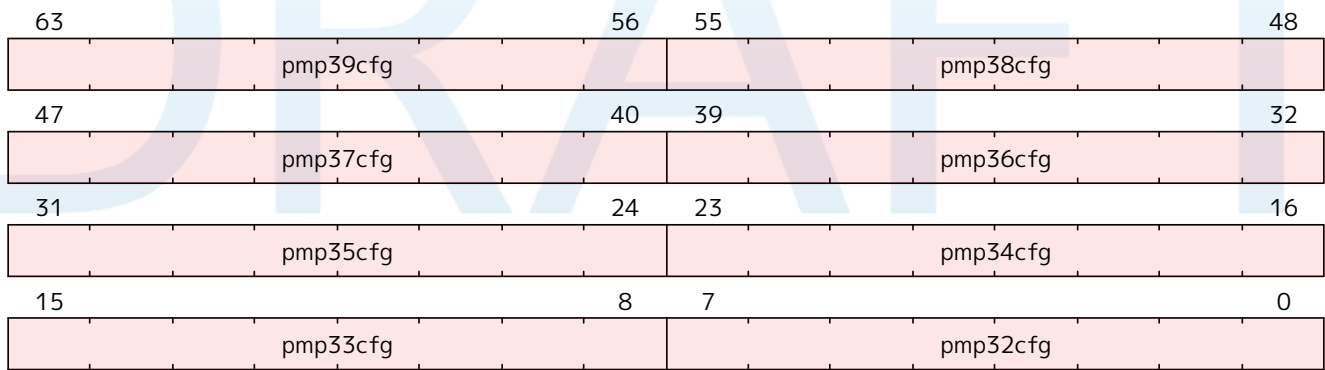



Figure 375. pmpcfg8 Format when CSR[misa].MXL == 1

C.293. pmpcfg9

PMP Configuration Register 9



pmpcfg9 is only defined in RV32.

PMP entry configuration

C.293.1. Attributes

CSR Address	0x3a9
Defining extension	I >= 0
Length	32-bit
Privilege Mode	M

C.293.2. Format

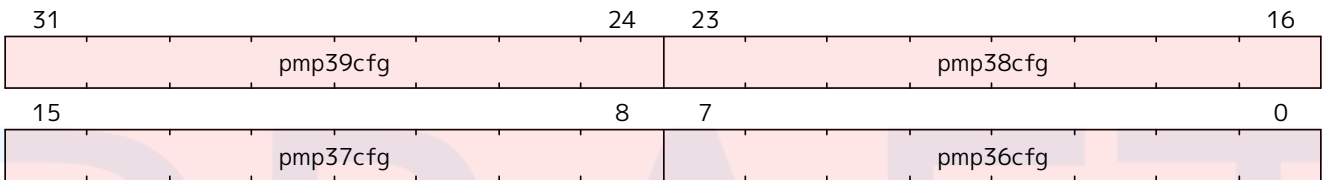


Figure 376. *pmpcfg9* format

C.294. satp

Supervisor Address Translation and Protection

Controls the translation mode in (H)S-mode and U-mode, and holds the current ASID and page table base pointer.

C.294.1. Attributes

CSR Address	0x180
Defining extension	S >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.294.2. Format

This CSR format changes dynamically with XLEN.

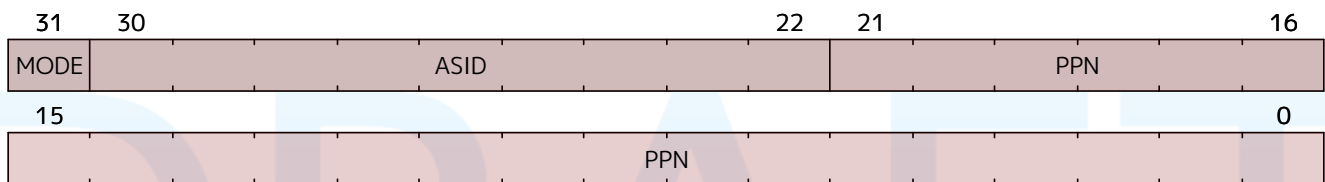


Figure 377. satp Format when CSR[mstatus].SXL == 0

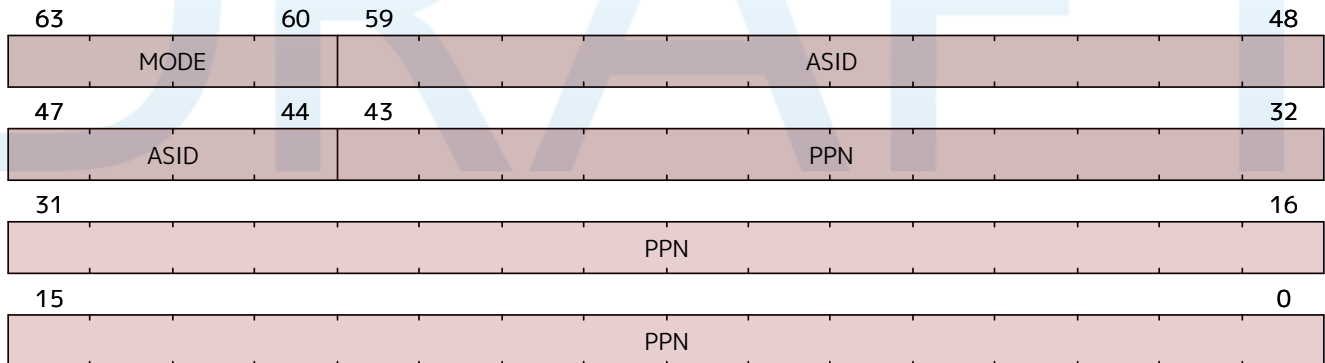


Figure 378. satp Format when CSR[mstatus].SXL == 1

C.295. scause

Supervisor Cause

Reports the cause of the latest exception.

C.295.1. Attributes

CSR Address	0x142
Defining extension	S >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.295.2. Format

This CSR format changes dynamically with XLEN.

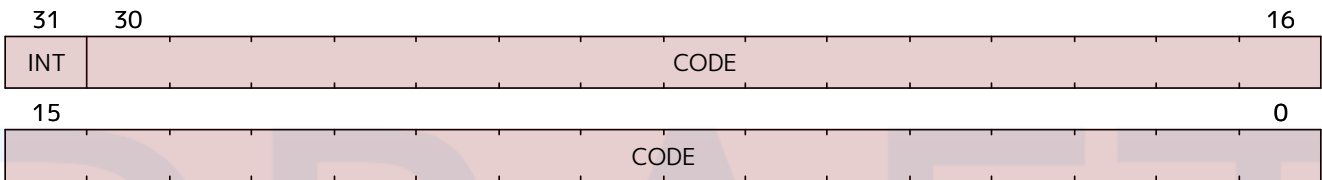


Figure 379. scause Format when CSR[mstatus].SXL == 0

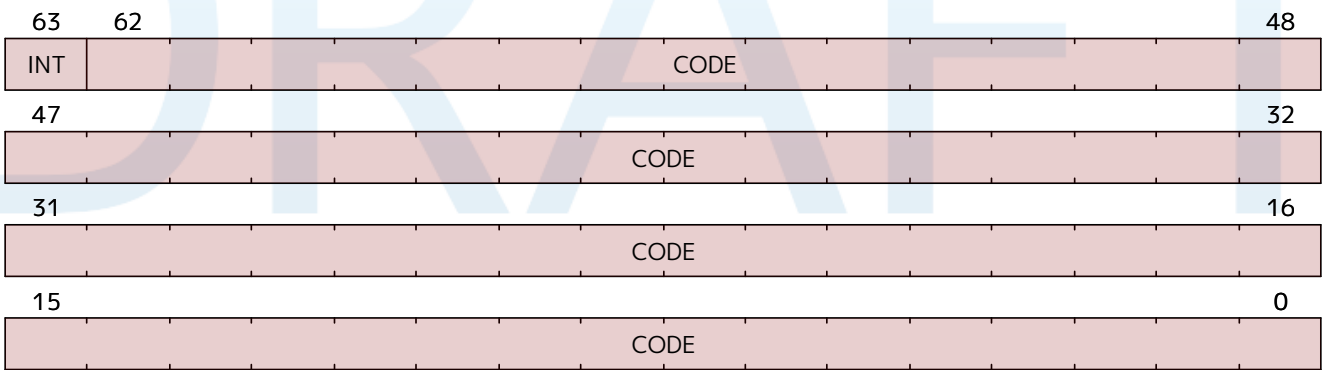


Figure 380. scause Format when CSR[mstatus].SXL == 1

C.296. scounteren

Supervisor Counter Enable

Delegates control of the hardware performance-monitoring counters to U-mode

C.296.1. Attributes

CSR Address	0x106
Defining extension	S >= 0
Length	32-bit
Privilege Mode	S

C.296.2. Format

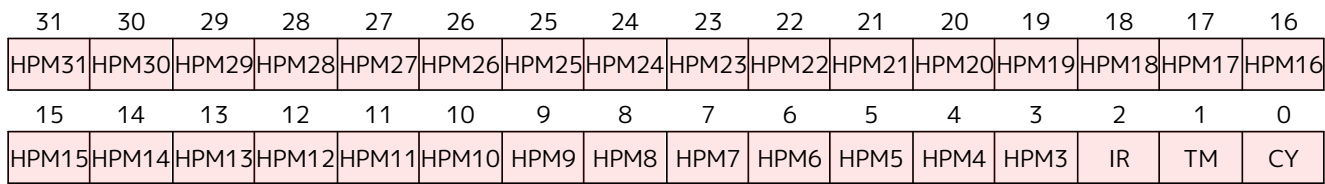


Figure 381. scounteren format

C.297. senvcfg

Supervisor Environment Configuration

Contains bits to enable/disable extensions

C.297.1. Attributes

CSR Address	0x10a
Defining extension	S \geq 0
Length	64-bit
Privilege Mode	S

C.297.2. Format

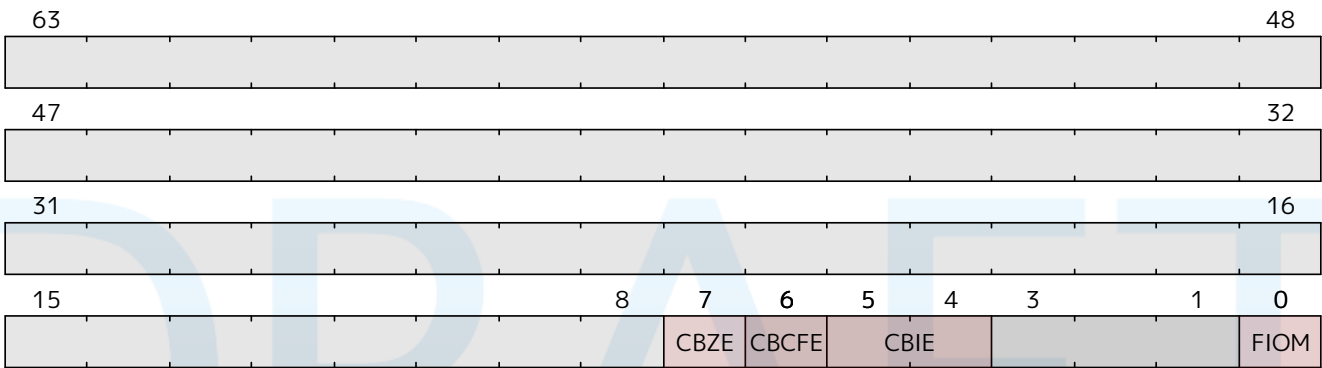


Figure 382. senvcfg format

C.298. sepc

Supervisor Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in (H)S-mode.

Also controls where the hart jumps on an exception return from (H)S-mode.

C.298.1. Attributes

CSR Address	0x141
Defining extension	S >= 0
Length	64-bit
Privilege Mode	S

C.298.2. Format

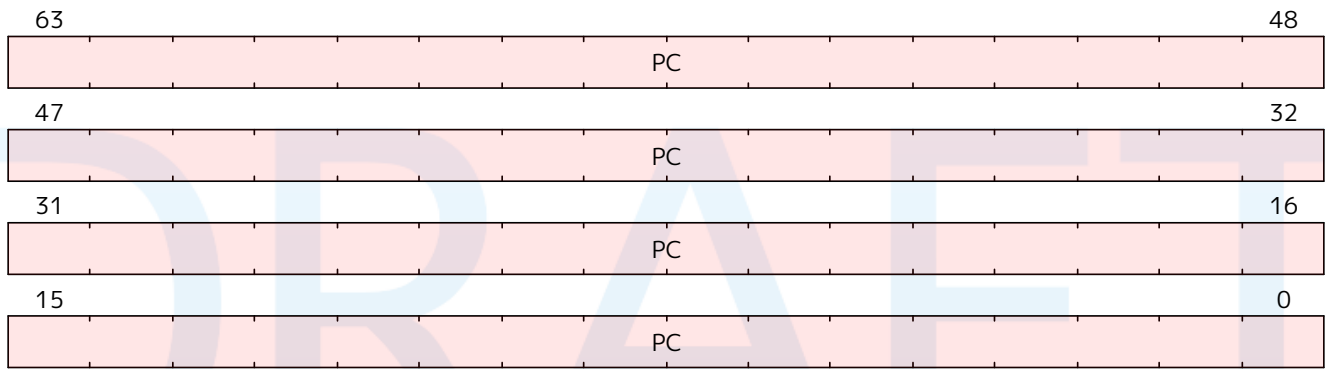


Figure 383. sepc format

C.299. sip

Supervisor Interrupt Pending

A restricted view of the interrupt pending bits in [mip](#).

Hypervisor-related interrupts (VS-mode interrupts and Supervisor Guest interrupts) are not reflected in [sip](#) even though those interrupts can be taken in HS-mode. Instead, they are reported through **hip**.

C.299.1. Attributes

CSR Address	0x144
Defining extension	I >= 0
Length	64-bit
Privilege Mode	S

C.299.2. Format

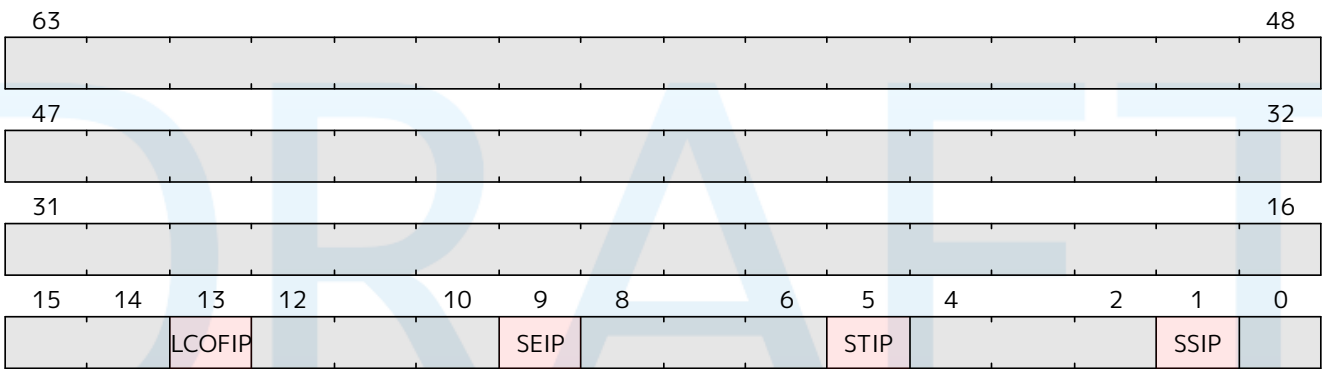


Figure 384. sip format

C.300. sscratch

Supervisor Scratch Register

Scratch register for software use. Bits are not interpreted by hardware.

C.300.1. Attributes

CSR Address	0x140
Defining extension	S \geq 0
Length	64-bit
Privilege Mode	S

C.300.2. Format

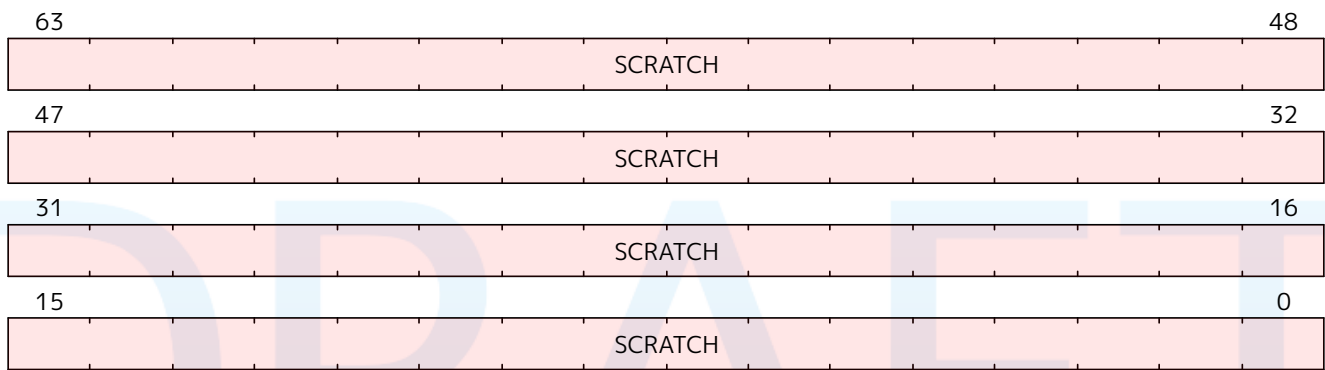


Figure 385. sscratch format

C.301. sstatus

Supervisor Status

The sstatus register tracks and controls the hart’s current operating state.

All fields in sstatus are aliases of the same field in mstatus.

C.301.1. Attributes

CSR Address	0x100
Defining extension	S >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.301.2. Format

This CSR format changes dynamically with XLEN.

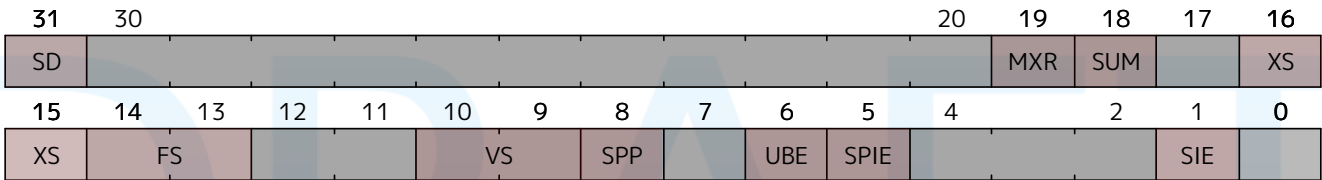


Figure 386. sstatus Format when CSR[mstatus].SXL == 0

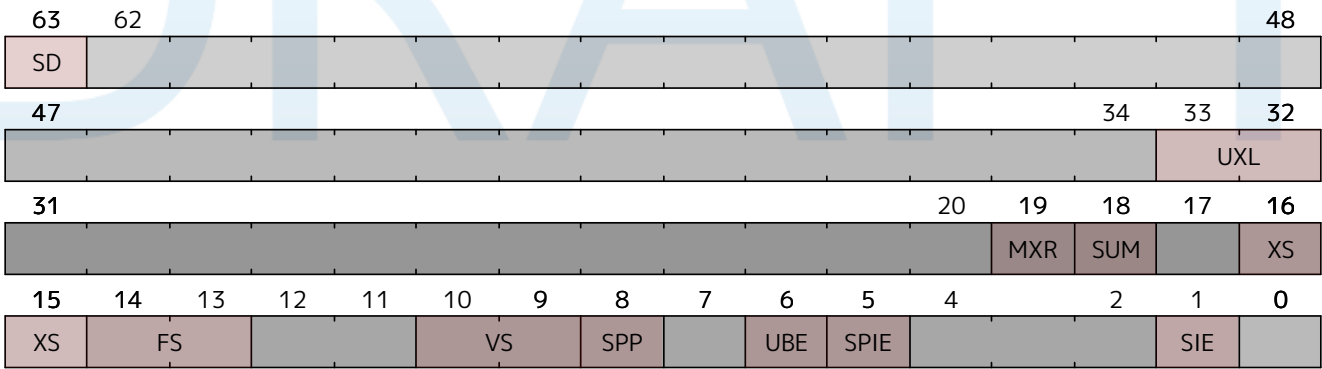


Figure 387. sstatus Format when CSR[mstatus].SXL == 1

C.302. stval

Supervisor Trap Value

Holds trap-specific information

C.302.1. Attributes

CSR Address	0x143
Defining extension	S \geq 0
Length	64-bit
Privilege Mode	S

C.302.2. Format

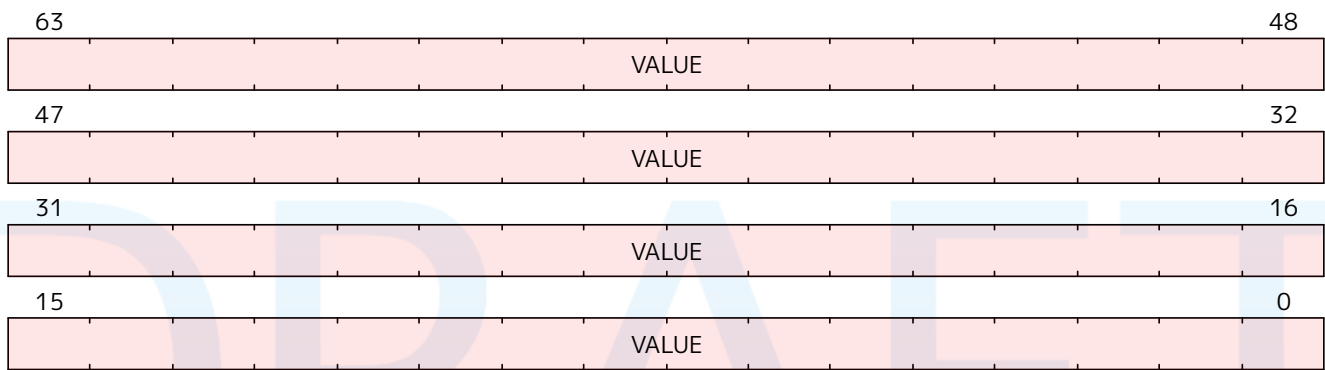


Figure 388. stval format

C.303. stvec

Supervisor Trap Vector

Controls where traps jump.

C.303.1. Attributes

CSR Address	0x105
Defining extension	I >= 0
Length	64-bit
Privilege Mode	S

C.303.2. Format

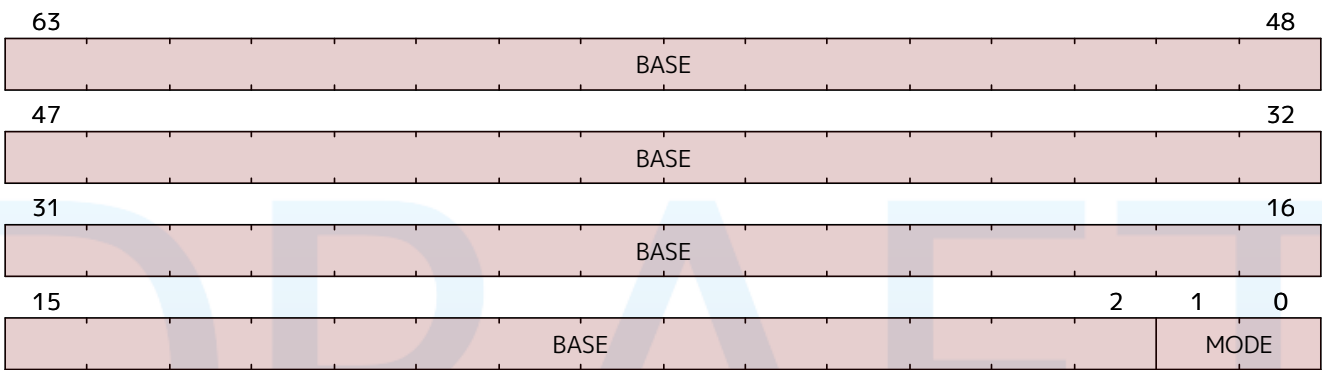


Figure 389. stvec format

C.304. time

Timer for RDTIME Instruction

Alias for the memory-mapped M-mode CSR **mtime**.

Privilege mode access is controlled with `mcounteren.TM`, `scounteren.TM`, and `hcounteren.TM` as follows:

mcounteren.TM	scounteren.TM	scounteren.TM	time behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	0	0	read-only	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	1	0	read-only	read-only	Illegal Instruction	Illegal Instruction
1	0	1	read-only	Illegal Instruction	read-only	Illegal Instruction
1	1	1	read-only	read-only	read-only	read-only

C.304.1. Attributes

CSR Address	0xc01
Defining extension	I >= 0
Length	64-bit
Privilege Mode	U

C.304.2. Format

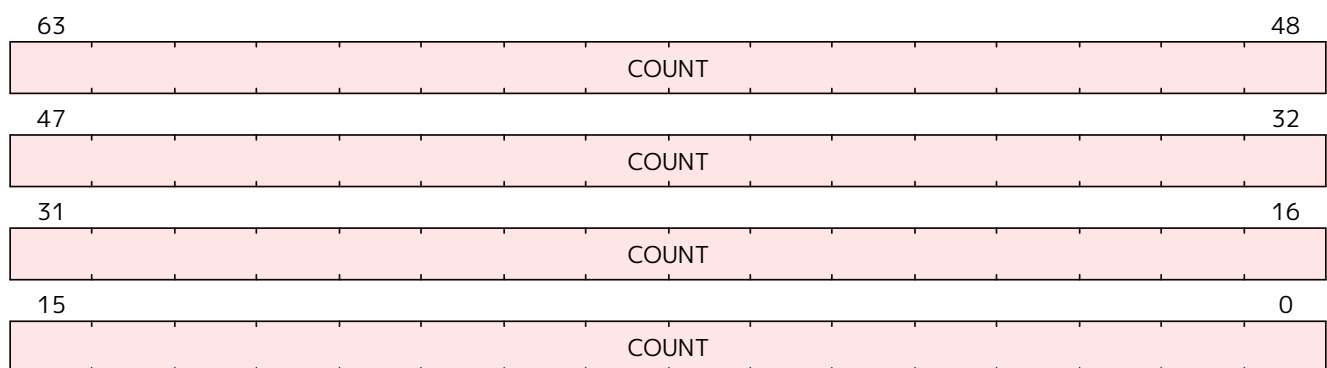


Figure 390. time format

C.305. vscause

Virtual Supervisor Cause

Reports the cause of the latest exception taken in VS-mode.

C.305.1. Attributes

CSR Address	0x242
Virtual CSR Address	0x142
Defining extension	H >= 0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	VS

C.305.2. Format

This CSR format changes dynamically with XLEN.

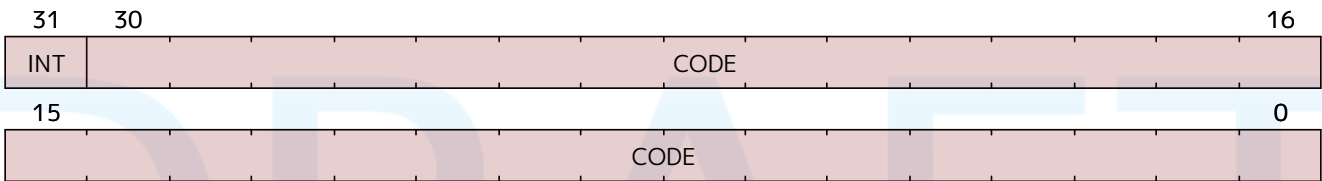


Figure 391. vscause Format when CSR[hstatus].VSXL == 0

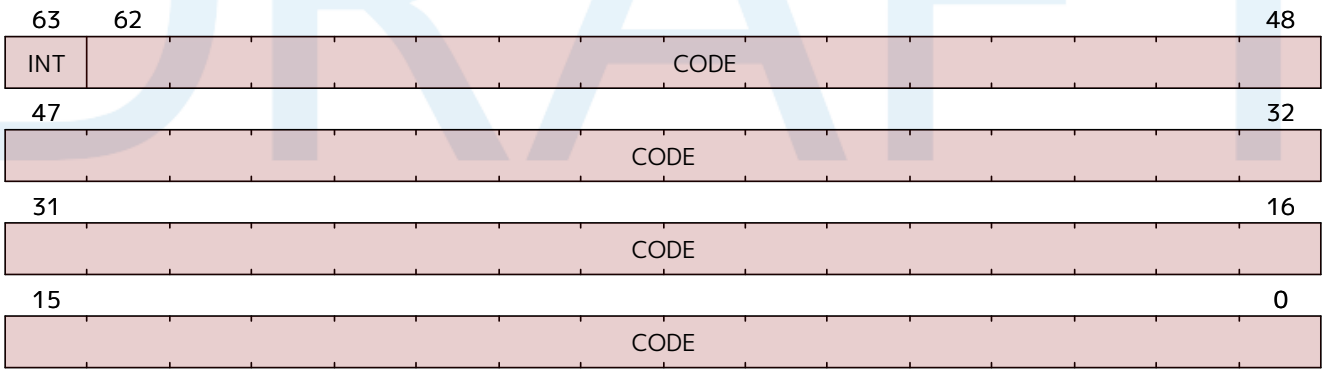


Figure 392. vscause Format when CSR[hstatus].VSXL == 1

C.306. vsepc

Virtual Supervisor Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in VS-mode.

Also controls where the hart jumps on an exception return from VS-mode.

C.306.1. Attributes

CSR Address	0x241
Virtual CSR Address	0x141
Defining extension	H >= 0
Length	64-bit
Privilege Mode	VS

C.306.2. Format

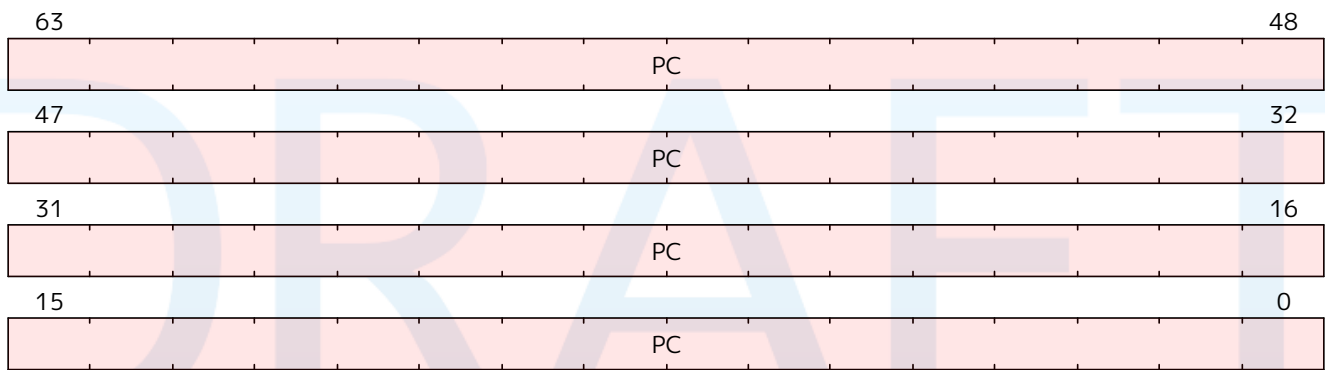


Figure 393. vsepc format

C.307. vsstatus

Virtual Supervisor Status

The vsstatus register tracks and controls the hart’s current operating state.

It is VS-mode’s version of `sstatus`, and substitutes for it when in VS-mode (*i.e.*, in VS-mode CSR address 0x100 is `vsstatus`, not `sstatus`).

Unlike the relationship between `sstatus` and `mstatus`, none of the bits in `vsstatus` are aliases of another field.

C.307.1. Attributes

CSR Address	0x200
Virtual CSR Address	0x100
Defining extension	H >= 0
Length	64-bit
Privilege Mode	VS

C.307.2. Format

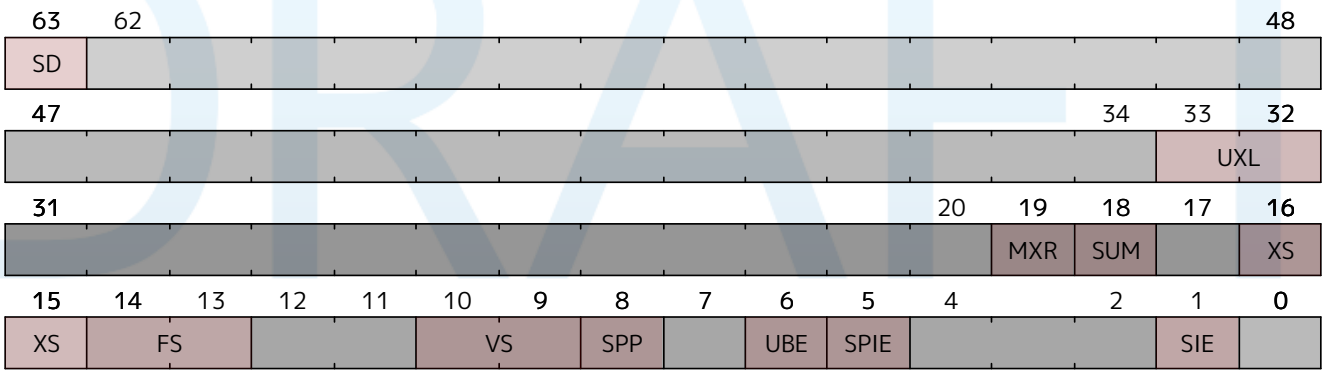


Figure 394. vsstatus format

C.308. vstval

Virtual supervisor Trap Value

Holds trap-specific information

C.308.1. Attributes

CSR Address	0x243
Defining extension	H >= 0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	S

C.308.2. Format

This CSR format changes dynamically with XLEN.

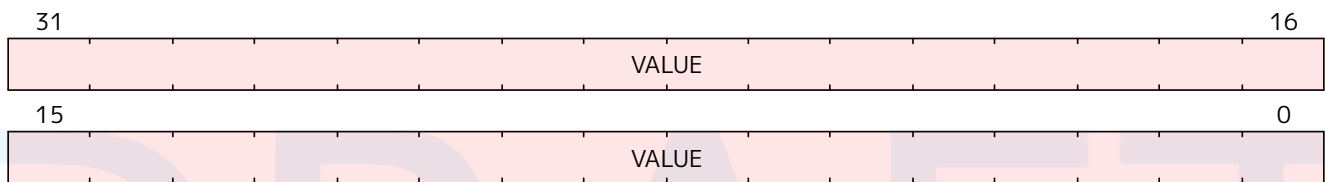


Figure 395. vstval Format when CSR[hstatus].VSXL == 0

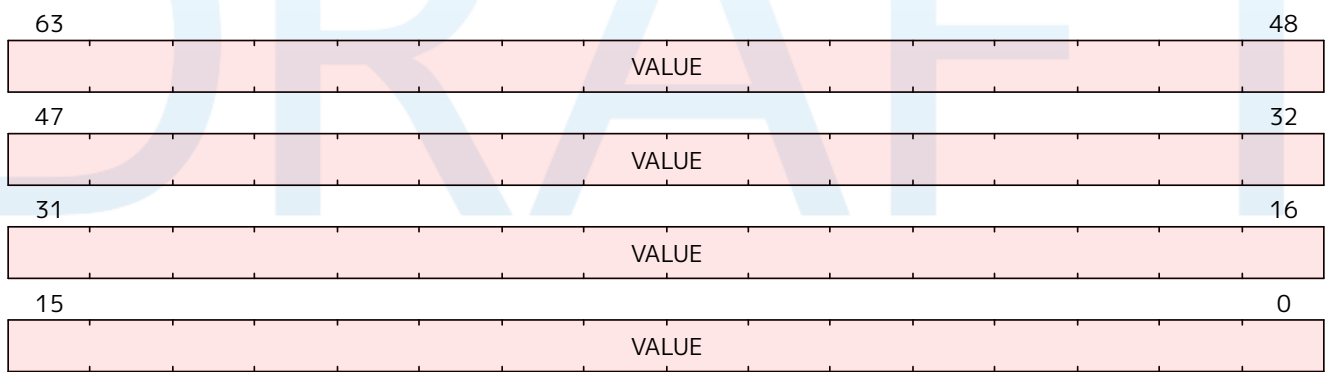


Figure 396. vstval Format when CSR[hstatus].VSXL == 1

C.309. vstvec

Supervisor Trap Vector

Controls where traps jump.

C.309.1. Attributes

CSR Address	0x205
Defining extension	H >= 0
Length	64-bit
Privilege Mode	S

C.309.2. Format

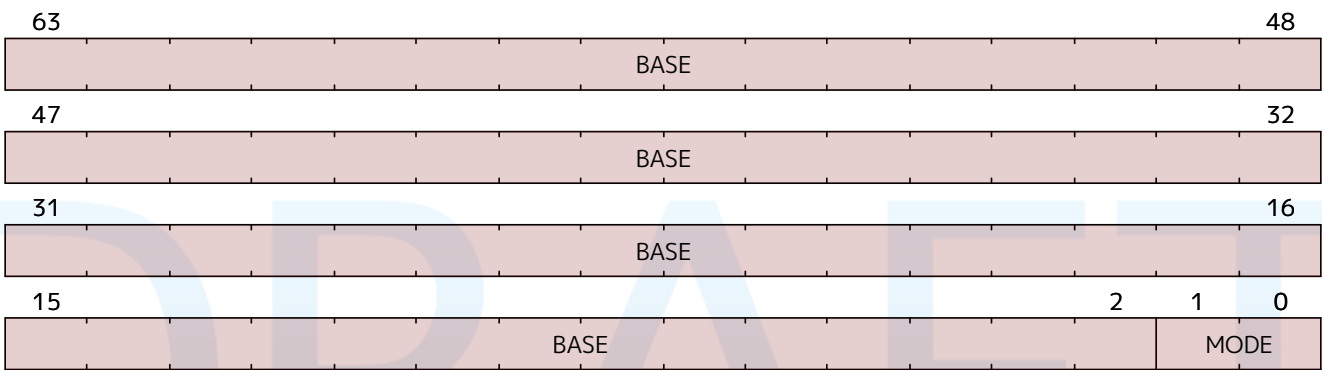


Figure 397. vstvec format