[RISC-V International Logo] | *risc-v_logo.png*

RVA20 Profile Release

2023-04-03

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. RVA Profile Class 8
  2.1. RVA Description 8
  2.2. RVA Naming Scheme 10
  2.3. RVA Profile Releases 10
  2.4. Extension Presence 10
3. RVA20 Profile Release 13
  3.1. RVA20 Description 13
  3.2. RVA20U64 Profile 13
    3.2.1. Mandatory Extensions 13
    3.2.2. Optional Extensions 15
    3.2.3. Recommendations 16
    3.2.4. Implementation-dependencies 16
  3.3. RVA20S64 Profile 18
    3.3.1. Mandatory Extensions 18
    3.3.2. Optional Extensions 19
    3.3.3. Implementation-dependencies 20
Appendix A: Extension Details 24
  A.1. A Extension 25
    A.1.1. Synopsis 25
    A.1.2. Specifying Ordering of Atomic Instructions 25
    A.1.3. Instructions 26
    A.1.4. Parameters 26
  A.2. C Extension 28
    A.2.1. Synopsis 28
    A.2.2. Overview 28
    A.2.3. Compressed Instruction Formats 31
    A.2.4. Instructions 32
    A.2.5. Parameters 33
  A.3. D Extension 34

A.3.1. Synopsis . . . 34
A.3.2. D Register State . . . 34
A.3.3. NaN Boxing of Narrower Values . . . 34
A.3.4. Instructions . . . 35
A.3.5. Parameters . . . 37
A.4. F Extension . . . 38
A.4.1. Synopsis . . . 38
A.4.2. F Register State . . . 38
Floating-Point Control and Status Register . . . 40
A.4.3. NaN Generation and Propagation . . . 41
A.4.4. Subnormal Arithmetic . . . 42
A.4.5. Instructions . . . 42
A.4.6. Parameters . . . 43
A.5. I Extension . . . 45
A.5.1. Synopsis . . . 45
A.5.2. Instructions . . . 45
A.6. M Extension . . . 48
A.6.1. Synopsis . . . 48
A.6.2. Instructions . . . 48
A.6.3. Parameters . . . 49
A.7. S Extension . . . 50
A.7.1. Synopsis . . . 50
A.7.2. Instructions . . . 50
A.7.3. Parameters . . . 50
A.8. Ssccptr Extension . . . 54
A.8.1. Synopsis . . . 54
A.9. Sstvala Extension . . . 55
A.9.1. Synopsis . . . 55
A.10. Sstvecd Extension . . . 56
A.10.1. Synopsis . . . 56
A.11. Sv39 Extension . . . 57
A.11.1. Synopsis . . . 57
A.12. Sv48 Extension . . . 58
A.12.1. Synopsis . . . 58
A.13. Svade Extension . . . 59
A.13.1. Synopsis . . . 59
A.14. Svbare Extension . . . 60
A.15. U Extension . . . 61
A.15.1. Synopsis . . . 61
A.15.2. Parameters . . . 61
A.16. Za128rs Extension . . . 62

A.17. Ziccamoa Extension . . . 63
A.18. Ziccif Extension . . . 64
A.19. Zicclsm Extension . . . 65
A.20. Ziccrse Extension . . . 66
A.21. Zicntr Extension . . . 67
A.21.1. Synopsis . . . 67
A.21.2. Parameters . . . 67
A.22. Zifencei Extension . . . 68
A.22.1. Instructions . . . 69
A.23. Zihpm Extension . . . 70
A.23.1. Synopsis . . . 70
Appendix B: Instruction Details . . . 71
B.1. add . . . 72
B.1.1. Encoding . . . 72
B.1.2. Synopsis . . . 72
B.1.3. Access . . . 72
B.1.4. Decode Variables . . . 72
B.1.5. Execution . . . 72
B.1.6. Exceptions . . . 72
B.2. addi . . . 73
B.2.1. Encoding . . . 73
B.2.2. Synopsis . . . 73
B.2.3. Access . . . 73
B.2.4. Decode Variables . . . 73
B.2.5. Execution . . . 73
B.2.6. Exceptions . . . 73
B.3. addiw . . . 74
B.3.1. Encoding . . . 74
B.3.2. Synopsis . . . 74
B.3.3. Access . . . 74
B.3.4. Decode Variables . . . 74
B.3.5. Execution . . . 74
B.3.6. Exceptions . . . 74
B.4. addw . . . 75
B.4.1. Encoding . . . 75
B.4.2. Synopsis . . . 75
B.4.3. Access . . . 75
B.4.4. Decode Variables . . . 75
B.4.5. Execution . . . 75
B.4.6. Exceptions . . . 75
B.5. amoadd.d . . . 76

B.5.1. Encoding 76
B.5.2. Synopsis 76
B.5.3. Access 76
B.5.4. Decode Variables 76
B.5.5. Execution 76
B.5.6. Exceptions 77
B.6. amoadd.w 78
B.6.1. Encoding 78
B.6.2. Synopsis 78
B.6.3. Access 78
B.6.4. Decode Variables 78
B.6.5. Execution 78
B.6.6. Exceptions 79
B.7. amoand.d 80
B.7.1. Encoding 80
B.7.2. Synopsis 80
B.7.3. Access 80
B.7.4. Decode Variables 80
B.7.5. Execution 80
B.7.6. Exceptions 81
B.8. amoand.w 82
B.8.1. Encoding 82
B.8.2. Synopsis 82
B.8.3. Access 82
B.8.4. Decode Variables 82
B.8.5. Execution 82
B.8.6. Exceptions 83
B.9. amomax.d 84
B.9.1. Encoding 84
B.9.2. Synopsis 84
B.9.3. Access 84
B.9.4. Decode Variables 84
B.9.5. Execution 84
B.9.6. Exceptions 85
B.10. amomax.w 86
B.10.1. Encoding 86
B.10.2. Synopsis 86
B.10.3. Access 86
B.10.4. Decode Variables 86
B.10.5. Execution 86
B.10.6. Exceptions 87

B.11. amomaxu.d 88
B.11.1. Encoding 88
B.11.2. Synopsis 88
B.11.3. Access 88
B.11.4. Decode Variables 88
B.11.5. Execution 88
B.11.6. Exceptions 89
B.12. amomaxu.w 90
B.12.1. Encoding 90
B.12.2. Synopsis 90
B.12.3. Access 90
B.12.4. Decode Variables 90
B.12.5. Execution 90
B.12.6. Exceptions 91
B.13. amomin.d 92
B.13.1. Encoding 92
B.13.2. Synopsis 92
B.13.3. Access 92
B.13.4. Decode Variables 92
B.13.5. Execution 92
B.13.6. Exceptions 93
B.14. amomin.w 94
B.14.1. Encoding 94
B.14.2. Synopsis 94
B.14.3. Access 94
B.14.4. Decode Variables 94
B.14.5. Execution 94
B.14.6. Exceptions 95
B.15. amominu.d 96
B.15.1. Encoding 96
B.15.2. Synopsis 96
B.15.3. Access 96
B.15.4. Decode Variables 96
B.15.5. Execution 96
B.15.6. Exceptions 97
B.16. amominu.w 98
B.16.1. Encoding 98
B.16.2. Synopsis 98
B.16.3. Access 98
B.16.4. Decode Variables 98
B.16.5. Execution 98

B.16.6. Exceptions ... 99
B.17. amoor.d ... 100
B.17.1. Encoding ... 100
B.17.2. Synopsis ... 100
B.17.3. Access ... 100
B.17.4. Decode Variables ... 100
B.17.5. Execution ... 100
B.17.6. Exceptions ... 101
B.18. amoor.w ... 102
B.18.1. Encoding ... 102
B.18.2. Synopsis ... 102
B.18.3. Access ... 102
B.18.4. Decode Variables ... 102
B.18.5. Execution ... 102
B.18.6. Exceptions ... 103
B.19. amoswap.d ... 104
B.19.1. Encoding ... 104
B.19.2. Synopsis ... 104
B.19.3. Access ... 104
B.19.4. Decode Variables ... 104
B.19.5. Execution ... 104
B.19.6. Exceptions ... 105
B.20. amoswap.w ... 106
B.20.1. Encoding ... 106
B.20.2. Synopsis ... 106
B.20.3. Access ... 106
B.20.4. Decode Variables ... 106
B.20.5. Execution ... 106
B.20.6. Exceptions ... 107
B.21. amoxor.d ... 108
B.21.1. Encoding ... 108
B.21.2. Synopsis ... 108
B.21.3. Access ... 108
B.21.4. Decode Variables ... 108
B.21.5. Execution ... 108
B.21.6. Exceptions ... 109
B.22. amoxor.w ... 110
B.22.1. Encoding ... 110
B.22.2. Synopsis ... 110
B.22.3. Access ... 110
B.22.4. Decode Variables ... 110

B.22.5. Execution . . . 110
B.22.6. Exceptions . . . 111
B.23. and . . . 112
B.23.1. Encoding . . . 112
B.23.2. Synopsis . . . 112
B.23.3. Access . . . 112
B.23.4. Decode Variables . . . 112
B.23.5. Execution . . . 112
B.23.6. Exceptions . . . 112
B.24. andi . . . 113
B.24.1. Encoding . . . 113
B.24.2. Synopsis . . . 113
B.24.3. Access . . . 113
B.24.4. Decode Variables . . . 113
B.24.5. Execution . . . 113
B.24.6. Exceptions . . . 113
B.25. auipc . . . 114
B.25.1. Encoding . . . 114
B.25.2. Synopsis . . . 114
B.25.3. Access . . . 114
B.25.4. Decode Variables . . . 114
B.25.5. Execution . . . 114
B.25.6. Exceptions . . . 114
B.26. beq . . . 115
B.26.1. Encoding . . . 115
B.26.2. Synopsis . . . 115
B.26.3. Access . . . 115
B.26.4. Decode Variables . . . 115
B.26.5. Execution . . . 115
B.26.6. Exceptions . . . 115
B.27. bge . . . 116
B.27.1. Encoding . . . 116
B.27.2. Synopsis . . . 116
B.27.3. Access . . . 116
B.27.4. Decode Variables . . . 116
B.27.5. Execution . . . 116
B.27.6. Exceptions . . . 116
B.28. bgeu . . . 117
B.28.1. Encoding . . . 117
B.28.2. Synopsis . . . 117
B.28.3. Access . . . 117

B.28.4. Decode Variables ... 117
B.28.5. Execution ... 117
B.28.6. Exceptions ... 117
B.29. blt ... 118
B.29.1. Encoding ... 118
B.29.2. Synopsis ... 118
B.29.3. Access ... 118
B.29.4. Decode Variables ... 118
B.29.5. Execution ... 118
B.29.6. Exceptions ... 118
B.30. bltu ... 119
B.30.1. Encoding ... 119
B.30.2. Synopsis ... 119
B.30.3. Access ... 119
B.30.4. Decode Variables ... 119
B.30.5. Execution ... 119
B.30.6. Exceptions ... 119
B.31. bne ... 120
B.31.1. Encoding ... 120
B.31.2. Synopsis ... 120
B.31.3. Access ... 120
B.31.4. Decode Variables ... 120
B.31.5. Execution ... 120
B.31.6. Exceptions ... 120
B.32. c.add ... 121
B.32.1. Encoding ... 121
B.32.2. Synopsis ... 121
B.32.3. Access ... 121
B.32.4. Decode Variables ... 121
B.32.5. Execution ... 121
B.32.6. Exceptions ... 121
B.33. c.addi ... 122
B.33.1. Encoding ... 122
B.33.2. Synopsis ... 122
B.33.3. Access ... 122
B.33.4. Decode Variables ... 122
B.33.5. Execution ... 122
B.33.6. Exceptions ... 122
B.34. c.addi16sp ... 123
B.34.1. Encoding ... 123
B.34.2. Synopsis ... 123

B.34.3. Access . . . 123
B.34.4. Decode Variables . . . 123
B.34.5. Execution . . . 123
B.34.6. Exceptions . . . 123
B.35. c.addi4spn . . . 124
B.35.1. Encoding . . . 124
B.35.2. Synopsis . . . 124
B.35.3. Access . . . 124
B.35.4. Decode Variables . . . 124
B.35.5. Execution . . . 124
B.35.6. Exceptions . . . 124
B.36. c.addiw . . . 125
B.36.1. Encoding . . . 125
B.36.2. Synopsis . . . 125
B.36.3. Access . . . 125
B.36.4. Decode Variables . . . 125
B.36.5. Execution . . . 125
B.36.6. Exceptions . . . 125
B.37. c.addw . . . 126
B.37.1. Encoding . . . 126
B.37.2. Synopsis . . . 126
B.37.3. Access . . . 126
B.37.4. Decode Variables . . . 126
B.37.5. Execution . . . 126
B.37.6. Exceptions . . . 126
B.38. c.and . . . 127
B.38.1. Encoding . . . 127
B.38.2. Synopsis . . . 127
B.38.3. Access . . . 127
B.38.4. Decode Variables . . . 127
B.38.5. Execution . . . 127
B.38.6. Exceptions . . . 127
B.39. c.andi . . . 128
B.39.1. Encoding . . . 128
B.39.2. Synopsis . . . 128
B.39.3. Access . . . 128
B.39.4. Decode Variables . . . 128
B.39.5. Execution . . . 128
B.39.6. Exceptions . . . 128
B.40. c.beqz . . . 129
B.40.1. Encoding . . . 129

B.40.2. Synopsis . . . 129
B.40.3. Access . . . 129
B.40.4. Decode Variables . . . 129
B.40.5. Execution . . . 129
B.40.6. Exceptions . . . 130
B.41. c.bnez . . . 131
B.41.1. Encoding . . . 131
B.41.2. Synopsis . . . 131
B.41.3. Access . . . 131
B.41.4. Decode Variables . . . 131
B.41.5. Execution . . . 131
B.41.6. Exceptions . . . 132
B.42. c.ebreak . . . 133
B.42.1. Encoding . . . 133
B.42.2. Synopsis . . . 133
B.42.3. Access . . . 133
B.42.4. Decode Variables . . . 133
B.42.5. Execution . . . 133
B.42.6. Exceptions . . . 134
B.43. c.j . . . 135
B.43.1. Encoding . . . 135
B.43.2. Synopsis . . . 135
B.43.3. Access . . . 135
B.43.4. Decode Variables . . . 135
B.43.5. Execution . . . 135
B.43.6. Exceptions . . . 135
B.44. c.jal . . . 136
B.44.1. Encoding . . . 136
B.44.2. Synopsis . . . 136
B.44.3. Access . . . 136
B.44.4. Decode Variables . . . 136
B.44.5. Execution . . . 136
B.44.6. Exceptions . . . 136
B.45. c.jalr . . . 138
B.45.1. Encoding . . . 138
B.45.2. Synopsis . . . 138
B.45.3. Access . . . 138
B.45.4. Decode Variables . . . 138
B.45.5. Execution . . . 138
B.45.6. Exceptions . . . 138
B.46. c.jr . . . 140

B.46.1. Encoding 140
B.46.2. Synopsis 140
B.46.3. Access 140
B.46.4. Decode Variables 140
B.46.5. Execution 140
B.46.6. Exceptions 140
B.47. c.ld 141
B.47.1. Encoding 141
B.47.2. Synopsis 141
B.47.3. Access 141
B.47.4. Decode Variables 141
B.47.5. Execution 141
B.47.6. Exceptions 141
B.48. c.ldsp 143
B.48.1. Encoding 143
B.48.2. Synopsis 143
B.48.3. Access 143
B.48.4. Decode Variables 143
B.48.5. Execution 143
B.48.6. Exceptions 143
B.49. c.li 145
B.49.1. Encoding 145
B.49.2. Synopsis 145
B.49.3. Access 145
B.49.4. Decode Variables 145
B.49.5. Execution 145
B.49.6. Exceptions 145
B.50. c.lq 146
B.50.1. Encoding 146
B.50.2. Synopsis 146
B.50.3. Access 146
B.50.4. Decode Variables 146
B.50.5. Execution 146
B.50.6. Exceptions 146
B.51. c.lqsp 148
B.51.1. Encoding 148
B.51.2. Synopsis 148
B.51.3. Access 148
B.51.4. Decode Variables 148
B.51.5. Execution 148
B.51.6. Exceptions 148

B.52. c.lui 150
B.52.1. Encoding 150
B.52.2. Synopsis 150
B.52.3. Access 150
B.52.4. Decode Variables 150
B.52.5. Execution 150
B.52.6. Exceptions 150
B.53. c.lw 152
B.53.1. Encoding 152
B.53.2. Synopsis 152
B.53.3. Access 152
B.53.4. Decode Variables 152
B.53.5. Execution 152
B.53.6. Exceptions 152
B.54. c.lwsp 154
B.54.1. Encoding 154
B.54.2. Synopsis 154
B.54.3. Access 154
B.54.4. Decode Variables 154
B.54.5. Execution 154
B.54.6. Exceptions 154
B.55. c.mv 156
B.55.1. Encoding 156
B.55.2. Synopsis 156
B.55.3. Access 156
B.55.4. Decode Variables 156
B.55.5. Execution 156
B.55.6. Exceptions 156
B.56. c.nop 157
B.56.1. Encoding 157
B.56.2. Synopsis 157
B.56.3. Access 157
B.56.4. Decode Variables 157
B.56.5. Execution 157
B.56.6. Exceptions 157
B.57. c.or 158
B.57.1. Encoding 158
B.57.2. Synopsis 158
B.57.3. Access 158
B.57.4. Decode Variables 158
B.57.5. Execution 158

B.57.6. Exceptions 158
B.58. c.sd 159
B.58.1. Encoding 159
B.58.2. Synopsis 159
B.58.3. Access 159
B.58.4. Decode Variables 159
B.58.5. Execution 159
B.58.6. Exceptions 159
B.59. c.sdsp 161
B.59.1. Encoding 161
B.59.2. Synopsis 161
B.59.3. Access 161
B.59.4. Decode Variables 161
B.59.5. Execution 161
B.59.6. Exceptions 161
B.60. c.slli 163
B.60.1. Encoding 163
B.60.2. Synopsis 163
B.60.3. Access 163
B.60.4. Decode Variables 163
B.60.5. Execution 163
B.60.6. Exceptions 163
B.61. c.sq 164
B.61.1. Encoding 164
B.61.2. Synopsis 164
B.61.3. Access 164
B.61.4. Decode Variables 164
B.61.5. Execution 164
B.61.6. Exceptions 164
B.62. c.sqsp 166
B.62.1. Encoding 166
B.62.2. Synopsis 166
B.62.3. Access 166
B.62.4. Decode Variables 166
B.62.5. Execution 166
B.62.6. Exceptions 166
B.63. c.srai 168
B.63.1. Encoding 168
B.63.2. Synopsis 168
B.63.3. Access 168
B.63.4. Decode Variables 168

B.63.5. Execution 168
B.63.6. Exceptions 168
B.64. c.srli 169
B.64.1. Encoding 169
B.64.2. Synopsis 169
B.64.3. Access 169
B.64.4. Decode Variables 169
B.64.5. Execution 169
B.64.6. Exceptions 169
B.65. c.sub 170
B.65.1. Encoding 170
B.65.2. Synopsis 170
B.65.3. Access 170
B.65.4. Decode Variables 170
B.65.5. Execution 170
B.65.6. Exceptions 170
B.66. c.subw 171
B.66.1. Encoding 171
B.66.2. Synopsis 171
B.66.3. Access 171
B.66.4. Decode Variables 171
B.66.5. Execution 171
B.66.6. Exceptions 171
B.67. c.sw 172
B.67.1. Encoding 172
B.67.2. Synopsis 172
B.67.3. Access 172
B.67.4. Decode Variables 172
B.67.5. Execution 172
B.67.6. Exceptions 172
B.68. c.swsp 174
B.68.1. Encoding 174
B.68.2. Synopsis 174
B.68.3. Access 174
B.68.4. Decode Variables 174
B.68.5. Execution 174
B.68.6. Exceptions 174
B.69. c.xor 176
B.69.1. Encoding 176
B.69.2. Synopsis 176
B.69.3. Access 176

B.69.4. Decode Variables . . . 176
B.69.5. Execution . . . 176
B.69.6. Exceptions . . . 176
B.70. div . . . 177
B.70.1. Encoding . . . 177
B.70.2. Synopsis . . . 177
B.70.3. Access . . . 177
B.70.4. Decode Variables . . . 177
B.70.5. Execution . . . 177
B.70.6. Exceptions . . . 178
B.71. divu . . . 179
B.71.1. Encoding . . . 179
B.71.2. Synopsis . . . 179
B.71.3. Access . . . 179
B.71.4. Decode Variables . . . 179
B.71.5. Execution . . . 179
B.71.6. Exceptions . . . 180
B.72. divuw . . . 181
B.72.1. Encoding . . . 181
B.72.2. Synopsis . . . 181
B.72.3. Access . . . 181
B.72.4. Decode Variables . . . 181
B.72.5. Execution . . . 181
B.72.6. Exceptions . . . 182
B.73. divw . . . 183
B.73.1. Encoding . . . 183
B.73.2. Synopsis . . . 183
B.73.3. Access . . . 183
B.73.4. Decode Variables . . . 183
B.73.5. Execution . . . 183
B.73.6. Exceptions . . . 184
B.74. ebreak . . . 185
B.74.1. Encoding . . . 185
B.74.2. Synopsis . . . 185
B.74.3. Access . . . 185
B.74.4. Decode Variables . . . 185
B.74.5. Execution . . . 185
B.74.6. Exceptions . . . 185
B.75. ecall . . . 187
B.75.1. Encoding . . . 187
B.75.2. Synopsis . . . 187

B.75.3. Access 187
B.75.4. Decode Variables 187
B.75.5. Execution 187
B.75.6. Exceptions 188
B.76. fadd.d 189
B.76.1. Encoding 189
B.76.2. Synopsis 189
B.76.3. Access 189
B.76.4. Decode Variables 189
B.76.5. Execution 189
B.76.6. Exceptions 189
B.77. fadd.s 190
B.77.1. Encoding 190
B.77.2. Synopsis 190
B.77.3. Access 190
B.77.4. Decode Variables 190
B.77.5. Execution 190
B.77.6. Exceptions 190
B.78. fclass.d 191
B.78.1. Encoding 191
B.78.2. Synopsis 191
B.78.3. Access 191
B.78.4. Decode Variables 191
B.78.5. Execution 191
B.78.6. Exceptions 191
B.79. fclass.s 192
B.79.1. Encoding 192
B.79.2. Synopsis 192
B.79.3. Access 192
B.79.4. Decode Variables 192
B.79.5. Execution 193
B.79.6. Exceptions 193
B.80. fcvt.d.h 194
B.80.1. Encoding 194
B.80.2. Synopsis 194
B.80.3. Access 194
B.80.4. Decode Variables 194
B.80.5. Execution 194
B.80.6. Exceptions 194
B.81. fcvt.d.l 195
B.81.1. Encoding 195

B.81.2. Synopsis . . . 195
B.81.3. Access . . . 195
B.81.4. Decode Variables . . . 195
B.81.5. Execution . . . 195
B.81.6. Exceptions . . . 195
B.82. fcvt.d.lu . . . 196
B.82.1. Encoding . . . 196
B.82.2. Synopsis . . . 196
B.82.3. Access . . . 196
B.82.4. Decode Variables . . . 196
B.82.5. Execution . . . 196
B.82.6. Exceptions . . . 196
B.83. fcvt.d.s . . . 197
B.83.1. Encoding . . . 197
B.83.2. Synopsis . . . 197
B.83.3. Access . . . 197
B.83.4. Decode Variables . . . 197
B.83.5. Execution . . . 197
B.83.6. Exceptions . . . 197
B.84. fcvt.d.w . . . 198
B.84.1. Encoding . . . 198
B.84.2. Synopsis . . . 198
B.84.3. Access . . . 198
B.84.4. Decode Variables . . . 198
B.84.5. Execution . . . 198
B.84.6. Exceptions . . . 198
B.85. fcvt.d.wu . . . 199
B.85.1. Encoding . . . 199
B.85.2. Synopsis . . . 199
B.85.3. Access . . . 199
B.85.4. Decode Variables . . . 199
B.85.5. Execution . . . 199
B.85.6. Exceptions . . . 199
B.86. fcvt.h.d . . . 200
B.86.1. Encoding . . . 200
B.86.2. Synopsis . . . 200
B.86.3. Access . . . 200
B.86.4. Decode Variables . . . 200
B.86.5. Execution . . . 200
B.86.6. Exceptions . . . 200
B.87. fcvt.l.d . . . 201

B.87.1. Encoding . . . 201
B.87.2. Synopsis . . . 201
B.87.3. Access . . . 201
B.87.4. Decode Variables . . . 201
B.87.5. Execution . . . 201
B.87.6. Exceptions . . . 201
B.88. fcvt.l.s . . . 202
B.88.1. Encoding . . . 202
B.88.2. Synopsis . . . 202
B.88.3. Access . . . 202
B.88.4. Decode Variables . . . 202
B.88.5. Execution . . . 202
B.88.6. Exceptions . . . 202
B.89. fcvt.lu.d . . . 203
B.89.1. Encoding . . . 203
B.89.2. Synopsis . . . 203
B.89.3. Access . . . 203
B.89.4. Decode Variables . . . 203
B.89.5. Execution . . . 203
B.89.6. Exceptions . . . 203
B.90. fcvt.lu.s . . . 204
B.90.1. Encoding . . . 204
B.90.2. Synopsis . . . 204
B.90.3. Access . . . 204
B.90.4. Decode Variables . . . 204
B.90.5. Execution . . . 204
B.90.6. Exceptions . . . 204
B.91. fcvt.s.d . . . 205
B.91.1. Encoding . . . 205
B.91.2. Synopsis . . . 205
B.91.3. Access . . . 205
B.91.4. Decode Variables . . . 205
B.91.5. Execution . . . 205
B.91.6. Exceptions . . . 205
B.92. fcvt.s.l . . . 206
B.92.1. Encoding . . . 206
B.92.2. Synopsis . . . 206
B.92.3. Access . . . 206
B.92.4. Decode Variables . . . 206
B.92.5. Execution . . . 206
B.92.6. Exceptions . . . 206

B.93. fcvt.s.lu . . . 207
B.93.1. Encoding . . . 207
B.93.2. Synopsis . . . 207
B.93.3. Access . . . 207
B.93.4. Decode Variables . . . 207
B.93.5. Execution . . . 207
B.93.6. Exceptions . . . 207
B.94. fcvt.s.w . . . 208
B.94.1. Encoding . . . 208
B.94.2. Synopsis . . . 208
B.94.3. Access . . . 208
B.94.4. Decode Variables . . . 208
B.94.5. Execution . . . 208
B.94.6. Exceptions . . . 209
B.95. fcvt.s.wu . . . 210
B.95.1. Encoding . . . 210
B.95.2. Synopsis . . . 210
B.95.3. Access . . . 210
B.95.4. Decode Variables . . . 210
B.95.5. Execution . . . 210
B.95.6. Exceptions . . . 210
B.96. fcvt.w.d . . . 211
B.96.1. Encoding . . . 211
B.96.2. Synopsis . . . 211
B.96.3. Access . . . 211
B.96.4. Decode Variables . . . 211
B.96.5. Execution . . . 211
B.96.6. Exceptions . . . 211
B.97. fcvt.w.s . . . 212
B.97.1. Encoding . . . 212
B.97.2. Synopsis . . . 212
B.97.3. Access . . . 212
B.97.4. Decode Variables . . . 213
B.97.5. Execution . . . 213
B.97.6. Exceptions . . . 213
B.98. fcvt.wu.d . . . 214
B.98.1. Encoding . . . 214
B.98.2. Synopsis . . . 214
B.98.3. Access . . . 214
B.98.4. Decode Variables . . . 214
B.98.5. Execution . . . 214

B.98.6. Exceptions 214
B.99. fcvt.wu.s 215
B.99.1. Encoding 215
B.99.2. Synopsis 215
B.99.3. Access 215
B.99.4. Decode Variables 215
B.99.5. Execution 215
B.99.6. Exceptions 215
B.100. fcvtmod.w.d 216
B.100.1. Encoding 216
B.100.2. Synopsis 216
B.100.3. Access 216
B.100.4. Decode Variables 216
B.100.5. Execution 216
B.100.6. Exceptions 216
B.101. fdiv.d 217
B.101.1. Encoding 217
B.101.2. Synopsis 217
B.101.3. Access 217
B.101.4. Decode Variables 217
B.101.5. Execution 217
B.101.6. Exceptions 217
B.102. fdiv.s 218
B.102.1. Encoding 218
B.102.2. Synopsis 218
B.102.3. Access 218
B.102.4. Decode Variables 218
B.102.5. Execution 218
B.102.6. Exceptions 218
B.103. fence 219
B.103.1. Encoding 219
B.103.2. Synopsis 219
B.103.3. Access 220
B.103.4. Decode Variables 221
B.103.5. Execution 221
B.103.6. Exceptions 222
B.104. fence.i 223
B.104.1. Encoding 223
B.104.2. Synopsis 223
B.104.3. Access 223
B.104.4. Decode Variables 223

B.104.5. Execution 224
B.104.6. Exceptions 224
B.105. feq.d 225
B.105.1. Encoding 225
B.105.2. Synopsis 225
B.105.3. Access 225
B.105.4. Decode Variables 225
B.105.5. Execution 225
B.105.6. Exceptions 225
B.106. feq.s 226
B.106.1. Encoding 226
B.106.2. Synopsis 226
B.106.3. Access 226
B.106.4. Decode Variables 226
B.106.5. Execution 226
B.106.6. Exceptions 227
B.107. fld 228
B.107.1. Encoding 228
B.107.2. Synopsis 228
B.107.3. Access 228
B.107.4. Decode Variables 228
B.107.5. Execution 228
B.107.6. Exceptions 228
B.108. fle.d 229
B.108.1. Encoding 229
B.108.2. Synopsis 229
B.108.3. Access 229
B.108.4. Decode Variables 229
B.108.5. Execution 229
B.108.6. Exceptions 229
B.109. fle.s 230
B.109.1. Encoding 230
B.109.2. Synopsis 230
B.109.3. Access 230
B.109.4. Decode Variables 230
B.109.5. Execution 230
B.109.6. Exceptions 231
B.110. fleq.d 232
B.110.1. Encoding 232
B.110.2. Synopsis 232
B.110.3. Access 232

B.110.4. Decode Variables . . . 232
B.110.5. Execution . . . 232
B.110.6. Exceptions . . . 232
B.111. fli.d . . . 233
B.111.1. Encoding . . . 233
B.111.2. Synopsis . . . 233
B.111.3. Access . . . 233
B.111.4. Decode Variables . . . 233
B.111.5. Execution . . . 233
B.111.6. Exceptions . . . 233
B.112. flt.d . . . 234
B.112.1. Encoding . . . 234
B.112.2. Synopsis . . . 234
B.112.3. Access . . . 234
B.112.4. Decode Variables . . . 234
B.112.5. Execution . . . 234
B.112.6. Exceptions . . . 234
B.113. flt.s . . . 235
B.113.1. Encoding . . . 235
B.113.2. Synopsis . . . 235
B.113.3. Access . . . 235
B.113.4. Decode Variables . . . 235
B.113.5. Execution . . . 235
B.113.6. Exceptions . . . 236
B.114. fltq.d . . . 237
B.114.1. Encoding . . . 237
B.114.2. Synopsis . . . 237
B.114.3. Access . . . 237
B.114.4. Decode Variables . . . 237
B.114.5. Execution . . . 237
B.114.6. Exceptions . . . 237
B.115. flw . . . 238
B.115.1. Encoding . . . 238
B.115.2. Synopsis . . . 238
B.115.3. Access . . . 238
B.115.4. Decode Variables . . . 238
B.115.5. Execution . . . 238
B.115.6. Exceptions . . . 239
B.116. fmadd.d . . . 240
B.116.1. Encoding . . . 240
B.116.2. Synopsis . . . 240

B.116.3. Access . . . 240
B.116.4. Decode Variables . . . 240
B.116.5. Execution . . . 240
B.116.6. Exceptions . . . 240
B.117. fmadd.s . . . 241
B.117.1. Encoding . . . 241
B.117.2. Synopsis . . . 241
B.117.3. Access . . . 241
B.117.4. Decode Variables . . . 241
B.117.5. Execution . . . 241
B.117.6. Exceptions . . . 241
B.118. fmax.d . . . 242
B.118.1. Encoding . . . 242
B.118.2. Synopsis . . . 242
B.118.3. Access . . . 242
B.118.4. Decode Variables . . . 242
B.118.5. Execution . . . 242
B.118.6. Exceptions . . . 242
B.119. fmax.s . . . 243
B.119.1. Encoding . . . 243
B.119.2. Synopsis . . . 243
B.119.3. Access . . . 243
B.119.4. Decode Variables . . . 243
B.119.5. Execution . . . 243
B.119.6. Exceptions . . . 243
B.120. fmaxm.d . . . 244
B.120.1. Encoding . . . 244
B.120.2. Synopsis . . . 244
B.120.3. Access . . . 244
B.120.4. Decode Variables . . . 244
B.120.5. Execution . . . 244
B.120.6. Exceptions . . . 244
B.121. fmin.d . . . 245
B.121.1. Encoding . . . 245
B.121.2. Synopsis . . . 245
B.121.3. Access . . . 245
B.121.4. Decode Variables . . . 245
B.121.5. Execution . . . 245
B.121.6. Exceptions . . . 245
B.122. fmin.s . . . 246
B.122.1. Encoding . . . 246

B.122.2. Synopsis 246
B.122.3. Access 246
B.122.4. Decode Variables 246
B.122.5. Execution 246
B.122.6. Exceptions 246
B.123. fminm.d 247
B.123.1. Encoding 247
B.123.2. Synopsis 247
B.123.3. Access 247
B.123.4. Decode Variables 247
B.123.5. Execution 247
B.123.6. Exceptions 247
B.124. fmsub.d 248
B.124.1. Encoding 248
B.124.2. Synopsis 248
B.124.3. Access 248
B.124.4. Decode Variables 248
B.124.5. Execution 248
B.124.6. Exceptions 248
B.125. fmsub.s 249
B.125.1. Encoding 249
B.125.2. Synopsis 249
B.125.3. Access 249
B.125.4. Decode Variables 249
B.125.5. Execution 249
B.125.6. Exceptions 249
B.126. fmul.d 250
B.126.1. Encoding 250
B.126.2. Synopsis 250
B.126.3. Access 250
B.126.4. Decode Variables 250
B.126.5. Execution 250
B.126.6. Exceptions 250
B.127. fmul.s 251
B.127.1. Encoding 251
B.127.2. Synopsis 251
B.127.3. Access 251
B.127.4. Decode Variables 251
B.127.5. Execution 251
B.127.6. Exceptions 251
B.128. fmv.d.x 252

B.128.1. Encoding . . . 252
B.128.2. Synopsis . . . 252
B.128.3. Access . . . 252
B.128.4. Decode Variables . . . 252
B.128.5. Execution . . . 252
B.128.6. Exceptions . . . 252
B.129. fmv.h.x . . . 253
B.129.1. Encoding . . . 253
B.129.2. Synopsis . . . 253
B.129.3. Access . . . 253
B.129.4. Decode Variables . . . 253
B.129.5. Execution . . . 253
B.129.6. Exceptions . . . 253
B.130. fmv.w.x . . . 254
B.130.1. Encoding . . . 254
B.130.2. Synopsis . . . 254
B.130.3. Access . . . 254
B.130.4. Decode Variables . . . 254
B.130.5. Execution . . . 254
B.130.6. Exceptions . . . 254
B.131. fmv.x.d . . . 255
B.131.1. Encoding . . . 255
B.131.2. Synopsis . . . 255
B.131.3. Access . . . 255
B.131.4. Decode Variables . . . 255
B.131.5. Execution . . . 255
B.131.6. Exceptions . . . 255
B.132. fmv.x.w . . . 256
B.132.1. Encoding . . . 256
B.132.2. Synopsis . . . 256
B.132.3. Access . . . 256
B.132.4. Decode Variables . . . 256
B.132.5. Execution . . . 256
B.132.6. Exceptions . . . 256
B.133. fmvh.x.d . . . 257
B.133.1. Encoding . . . 257
B.133.2. Synopsis . . . 257
B.133.3. Access . . . 257
B.133.4. Decode Variables . . . 257
B.133.5. Execution . . . 257
B.133.6. Exceptions . . . 257

B.134. fmvp.d.x 258
B.134.1. Encoding 258
B.134.2. Synopsis 258
B.134.3. Access 258
B.134.4. Decode Variables 258
B.134.5. Execution 258
B.134.6. Exceptions 258
B.135. fnmadd.d 259
B.135.1. Encoding 259
B.135.2. Synopsis 259
B.135.3. Access 259
B.135.4. Decode Variables 259
B.135.5. Execution 259
B.135.6. Exceptions 259
B.136. fnmadd.s 260
B.136.1. Encoding 260
B.136.2. Synopsis 260
B.136.3. Access 260
B.136.4. Decode Variables 260
B.136.5. Execution 260
B.136.6. Exceptions 260
B.137. fnmsub.d 261
B.137.1. Encoding 261
B.137.2. Synopsis 261
B.137.3. Access 261
B.137.4. Decode Variables 261
B.137.5. Execution 261
B.137.6. Exceptions 261
B.138. fnmsub.s 262
B.138.1. Encoding 262
B.138.2. Synopsis 262
B.138.3. Access 262
B.138.4. Decode Variables 262
B.138.5. Execution 262
B.138.6. Exceptions 262
B.139. fround.d 263
B.139.1. Encoding 263
B.139.2. Synopsis 263
B.139.3. Access 263
B.139.4. Decode Variables 263
B.139.5. Execution 263

B.139.6. Exceptions . . . 263
B.140. froundnx.d . . . 264
B.140.1. Encoding . . . 264
B.140.2. Synopsis . . . 264
B.140.3. Access . . . 264
B.140.4. Decode Variables . . . 264
B.140.5. Execution . . . 264
B.140.6. Exceptions . . . 264
B.141. fsd . . . 265
B.141.1. Encoding . . . 265
B.141.2. Synopsis . . . 265
B.141.3. Access . . . 265
B.141.4. Decode Variables . . . 265
B.141.5. Execution . . . 265
B.141.6. Exceptions . . . 265
B.142. fsgnj.d . . . 266
B.142.1. Encoding . . . 266
B.142.2. Synopsis . . . 266
B.142.3. Access . . . 266
B.142.4. Decode Variables . . . 266
B.142.5. Execution . . . 266
B.142.6. Exceptions . . . 266
B.143. fsgnj.s . . . 267
B.143.1. Encoding . . . 267
B.143.2. Synopsis . . . 267
B.143.3. Access . . . 267
B.143.4. Decode Variables . . . 267
B.143.5. Execution . . . 267
B.143.6. Exceptions . . . 267
B.144. fsgnjn.d . . . 268
B.144.1. Encoding . . . 268
B.144.2. Synopsis . . . 268
B.144.3. Access . . . 268
B.144.4. Decode Variables . . . 268
B.144.5. Execution . . . 268
B.144.6. Exceptions . . . 268
B.145. fsgnjn.s . . . 269
B.145.1. Encoding . . . 269
B.145.2. Synopsis . . . 269
B.145.3. Access . . . 269
B.145.4. Decode Variables . . . 269

B.145.5. Execution . . . 269
B.145.6. Exceptions . . . 269
B.146. fsgnjx.d . . . 270
B.146.1. Encoding . . . 270
B.146.2. Synopsis . . . 270
B.146.3. Access . . . 270
B.146.4. Decode Variables . . . 270
B.146.5. Execution . . . 270
B.146.6. Exceptions . . . 270
B.147. fsgnjx.s . . . 271
B.147.1. Encoding . . . 271
B.147.2. Synopsis . . . 271
B.147.3. Access . . . 271
B.147.4. Decode Variables . . . 271
B.147.5. Execution . . . 271
B.147.6. Exceptions . . . 271
B.148. fsqrt.d . . . 272
B.148.1. Encoding . . . 272
B.148.2. Synopsis . . . 272
B.148.3. Access . . . 272
B.148.4. Decode Variables . . . 272
B.148.5. Execution . . . 272
B.148.6. Exceptions . . . 272
B.149. fsqrt.s . . . 273
B.149.1. Encoding . . . 273
B.149.2. Synopsis . . . 273
B.149.3. Access . . . 273
B.149.4. Decode Variables . . . 273
B.149.5. Execution . . . 273
B.149.6. Exceptions . . . 273
B.150. fsub.d . . . 274
B.150.1. Encoding . . . 274
B.150.2. Synopsis . . . 274
B.150.3. Access . . . 274
B.150.4. Decode Variables . . . 274
B.150.5. Execution . . . 274
B.150.6. Exceptions . . . 274
B.151. fsub.s . . . 275
B.151.1. Encoding . . . 275
B.151.2. Synopsis . . . 275
B.151.3. Access . . . 275

B.151.4. Decode Variables . . . 275
B.151.5. Execution . . . 275
B.151.6. Exceptions . . . 275
B.152. fsw . . . 276
B.152.1. Encoding . . . 276
B.152.2. Synopsis . . . 276
B.152.3. Access . . . 276
B.152.4. Decode Variables . . . 276
B.152.5. Execution . . . 276
B.152.6. Exceptions . . . 276
B.153. jal . . . 278
B.153.1. Encoding . . . 278
B.153.2. Synopsis . . . 278
B.153.3. Access . . . 278
B.153.4. Decode Variables . . . 278
B.153.5. Execution . . . 278
B.153.6. Exceptions . . . 278
B.154. jalr . . . 279
B.154.1. Encoding . . . 279
B.154.2. Synopsis . . . 279
B.154.3. Access . . . 279
B.154.4. Decode Variables . . . 279
B.154.5. Execution . . . 279
B.154.6. Exceptions . . . 279
B.155. lb . . . 280
B.155.1. Encoding . . . 280
B.155.2. Synopsis . . . 280
B.155.3. Access . . . 280
B.155.4. Decode Variables . . . 280
B.155.5. Execution . . . 280
B.155.6. Exceptions . . . 280
B.156. lbu . . . 281
B.156.1. Encoding . . . 281
B.156.2. Synopsis . . . 281
B.156.3. Access . . . 281
B.156.4. Decode Variables . . . 281
B.156.5. Execution . . . 281
B.156.6. Exceptions . . . 281
B.157. ld . . . 282
B.157.1. Encoding . . . 282
B.157.2. Synopsis . . . 282

B.157.3. Access 282
B.157.4. Decode Variables 282
B.157.5. Execution 282
B.157.6. Exceptions 282
B.158. lh 283
B.158.1. Encoding 283
B.158.2. Synopsis 283
B.158.3. Access 283
B.158.4. Decode Variables 283
B.158.5. Execution 283
B.158.6. Exceptions 283
B.159. lhu 284
B.159.1. Encoding 284
B.159.2. Synopsis 284
B.159.3. Access 284
B.159.4. Decode Variables 284
B.159.5. Execution 284
B.159.6. Exceptions 284
B.160. lr.d 285
B.160.1. Encoding 285
B.160.2. Synopsis 285
B.160.3. Access 286
B.160.4. Decode Variables 286
B.160.5. Execution 286
B.160.6. Exceptions 286
B.161. lr.w 287
B.161.1. Encoding 287
B.161.2. Synopsis 287
B.161.3. Access 288
B.161.4. Decode Variables 288
B.161.5. Execution 288
B.161.6. Exceptions 289
B.162. lui 290
B.162.1. Encoding 290
B.162.2. Synopsis 290
B.162.3. Access 290
B.162.4. Decode Variables 290
B.162.5. Execution 290
B.162.6. Exceptions 290
B.163. lw 291
B.163.1. Encoding 291

B.163.2. Synopsis 291
B.163.3. Access 291
B.163.4. Decode Variables 291
B.163.5. Execution 291
B.163.6. Exceptions 291
B.164. lwu 292
B.164.1. Encoding 292
B.164.2. Synopsis 292
B.164.3. Access 292
B.164.4. Decode Variables 292
B.164.5. Execution 292
B.164.6. Exceptions 292
B.165. mul 293
B.165.1. Encoding 293
B.165.2. Synopsis 293
B.165.3. Access 293
B.165.4. Decode Variables 293
B.165.5. Execution 293
B.165.6. Exceptions 294
B.166. mulh 295
B.166.1. Encoding 295
B.166.2. Synopsis 295
B.166.3. Access 295
B.166.4. Decode Variables 295
B.166.5. Execution 295
B.166.6. Exceptions 296
B.167. mulhsu 297
B.167.1. Encoding 297
B.167.2. Synopsis 297
B.167.3. Access 297
B.167.4. Decode Variables 297
B.167.5. Execution 297
B.167.6. Exceptions 298
B.168. mulhu 299
B.168.1. Encoding 299
B.168.2. Synopsis 299
B.168.3. Access 299
B.168.4. Decode Variables 299
B.168.5. Execution 299
B.168.6. Exceptions 300
B.169. mulw 301

B.169.1. Encoding 301
B.169.2. Synopsis 301
B.169.3. Access 301
B.169.4. Decode Variables 301
B.169.5. Execution 301
B.169.6. Exceptions 302
B.170. or 303
B.170.1. Encoding 303
B.170.2. Synopsis 303
B.170.3. Access 303
B.170.4. Decode Variables 303
B.170.5. Execution 303
B.170.6. Exceptions 303
B.171. ori 304
B.171.1. Encoding 304
B.171.2. Synopsis 304
B.171.3. Access 304
B.171.4. Decode Variables 304
B.171.5. Execution 304
B.171.6. Exceptions 305
B.172. rem 306
B.172.1. Encoding 306
B.172.2. Synopsis 306
B.172.3. Access 306
B.172.4. Decode Variables 306
B.172.5. Execution 306
B.172.6. Exceptions 307
B.173. remu 308
B.173.1. Encoding 308
B.173.2. Synopsis 308
B.173.3. Access 308
B.173.4. Decode Variables 308
B.173.5. Execution 308
B.173.6. Exceptions 308
B.174. remuw 309
B.174.1. Encoding 309
B.174.2. Synopsis 309
B.174.3. Access 309
B.174.4. Decode Variables 309
B.174.5. Execution 309
B.174.6. Exceptions 310

B.175. remw 311
B.175.1. Encoding 311
B.175.2. Synopsis 311
B.175.3. Access 311
B.175.4. Decode Variables 311
B.175.5. Execution 311
B.175.6. Exceptions 312
B.176. sb 313
B.176.1. Encoding 313
B.176.2. Synopsis 313
B.176.3. Access 313
B.176.4. Decode Variables 313
B.176.5. Execution 313
B.176.6. Exceptions 313
B.177. sc.d 314
B.177.1. Encoding 314
B.177.2. Synopsis 314
B.177.3. Access 316
B.177.4. Decode Variables 316
B.177.5. Execution 316
B.177.6. Exceptions 317
B.178. sc.w 318
B.178.1. Encoding 318
B.178.2. Synopsis 318
B.178.3. Access 320
B.178.4. Decode Variables 320
B.178.5. Execution 320
B.178.6. Exceptions 321
B.179. sd 322
B.179.1. Encoding 322
B.179.2. Synopsis 322
B.179.3. Access 322
B.179.4. Decode Variables 322
B.179.5. Execution 322
B.179.6. Exceptions 322
B.180. sfence.vma 323
B.180.1. Encoding 323
B.180.2. Synopsis 323
B.180.3. Access 326
B.180.4. Decode Variables 326
B.180.5. Execution 326

B.180.6. Exceptions . . . 328
B.181. sh . . . 329
B.181.1. Encoding . . . 329
B.181.2. Synopsis . . . 329
B.181.3. Access . . . 329
B.181.4. Decode Variables . . . 329
B.181.5. Execution . . . 329
B.181.6. Exceptions . . . 329
B.182. sll . . . 330
B.182.1. Encoding . . . 330
B.182.2. Synopsis . . . 330
B.182.3. Access . . . 330
B.182.4. Decode Variables . . . 330
B.182.5. Execution . . . 330
B.182.6. Exceptions . . . 330
B.183. slli . . . 331
B.183.1. Encoding . . . 331
B.183.2. Synopsis . . . 331
B.183.3. Access . . . 331
B.183.4. Decode Variables . . . 331
B.183.5. Execution . . . 332
B.183.6. Exceptions . . . 332
B.184. slliw . . . 333
B.184.1. Encoding . . . 333
B.184.2. Synopsis . . . 333
B.184.3. Access . . . 333
B.184.4. Decode Variables . . . 333
B.184.5. Execution . . . 333
B.184.6. Exceptions . . . 333
B.185. sllw . . . 334
B.185.1. Encoding . . . 334
B.185.2. Synopsis . . . 334
B.185.3. Access . . . 334
B.185.4. Decode Variables . . . 334
B.185.5. Execution . . . 334
B.185.6. Exceptions . . . 334
B.186. slt . . . 335
B.186.1. Encoding . . . 335
B.186.2. Synopsis . . . 335
B.186.3. Access . . . 335
B.186.4. Decode Variables . . . 335

B.186.5. Execution . . . 335
B.186.6. Exceptions . . . 335
B.187. slti . . . 336
B.187.1. Encoding . . . 336
B.187.2. Synopsis . . . 336
B.187.3. Access . . . 336
B.187.4. Decode Variables . . . 336
B.187.5. Execution . . . 336
B.187.6. Exceptions . . . 336
B.188. sltiu . . . 337
B.188.1. Encoding . . . 337
B.188.2. Synopsis . . . 337
B.188.3. Access . . . 337
B.188.4. Decode Variables . . . 337
B.188.5. Execution . . . 337
B.188.6. Exceptions . . . 337
B.189. sltu . . . 338
B.189.1. Encoding . . . 338
B.189.2. Synopsis . . . 338
B.189.3. Access . . . 338
B.189.4. Decode Variables . . . 338
B.189.5. Execution . . . 338
B.189.6. Exceptions . . . 338
B.190. sra . . . 339
B.190.1. Encoding . . . 339
B.190.2. Synopsis . . . 339
B.190.3. Access . . . 339
B.190.4. Decode Variables . . . 339
B.190.5. Execution . . . 339
B.190.6. Exceptions . . . 339
B.191. srai . . . 340
B.191.1. Encoding . . . 340
B.191.2. Synopsis . . . 340
B.191.3. Access . . . 340
B.191.4. Decode Variables . . . 340
B.191.5. Execution . . . 341
B.191.6. Exceptions . . . 341
B.192. sraiw . . . 342
B.192.1. Encoding . . . 342
B.192.2. Synopsis . . . 342
B.192.3. Access . . . 342

B.192.4. Decode Variables . . . 342
B.192.5. Execution . . . 342
B.192.6. Exceptions . . . 342
B.193. sraw . . . 343
B.193.1. Encoding . . . 343
B.193.2. Synopsis . . . 343
B.193.3. Access . . . 343
B.193.4. Decode Variables . . . 343
B.193.5. Execution . . . 343
B.193.6. Exceptions . . . 343
B.194. sret . . . 344
B.194.1. Encoding . . . 344
B.194.2. Synopsis . . . 344
B.194.3. Access . . . 345
B.194.4. Decode Variables . . . 345
B.194.5. Execution . . . 345
B.194.6. Exceptions . . . 346
B.195. srl . . . 347
B.195.1. Encoding . . . 347
B.195.2. Synopsis . . . 347
B.195.3. Access . . . 347
B.195.4. Decode Variables . . . 347
B.195.5. Execution . . . 347
B.195.6. Exceptions . . . 347
B.196. srli . . . 348
B.196.1. Encoding . . . 348
B.196.2. Synopsis . . . 348
B.196.3. Access . . . 348
B.196.4. Decode Variables . . . 348
B.196.5. Execution . . . 349
B.196.6. Exceptions . . . 349
B.197. srliw . . . 350
B.197.1. Encoding . . . 350
B.197.2. Synopsis . . . 350
B.197.3. Access . . . 350
B.197.4. Decode Variables . . . 350
B.197.5. Execution . . . 350
B.197.6. Exceptions . . . 350
B.198. srlw . . . 351
B.198.1. Encoding . . . 351
B.198.2. Synopsis . . . 351

B.198.3. Access 351
B.198.4. Decode Variables 351
B.198.5. Execution 351
B.198.6. Exceptions 351
B.199. sub 352
B.199.1. Encoding 352
B.199.2. Synopsis 352
B.199.3. Access 352
B.199.4. Decode Variables 352
B.199.5. Execution 352
B.199.6. Exceptions 352
B.200. subw 353
B.200.1. Encoding 353
B.200.2. Synopsis 353
B.200.3. Access 353
B.200.4. Decode Variables 353
B.200.5. Execution 353
B.200.6. Exceptions 353
B.201. sw 354
B.201.1. Encoding 354
B.201.2. Synopsis 354
B.201.3. Access 354
B.201.4. Decode Variables 354
B.201.5. Execution 354
B.201.6. Exceptions 354
B.202. xor 355
B.202.1. Encoding 355
B.202.2. Synopsis 355
B.202.3. Access 355
B.202.4. Decode Variables 355
B.202.5. Execution 355
B.202.6. Exceptions 355
B.203. xori 356
B.203.1. Encoding 356
B.203.2. Synopsis 356
B.203.3. Access 356
B.203.4. Decode Variables 356
B.203.5. Execution 356
B.203.6. Exceptions 356
Appendix C: CSR Details 357
C.1. cycle 358

C.1.1. Attributes . . . 358
C.1.2. Format . . . 358
C.2. cycleh . . . 359
C.2.1. Attributes . . . 359
C.2.2. Format . . . 359
C.3. fcsr . . . 360
C.3.1. Attributes . . . 361
C.3.2. Format . . . 361
C.4. hpmcounter10 . . . 362
C.4.1. Attributes . . . 362
C.4.2. Format . . . 363
C.5. hpmcounter11 . . . 364
C.5.1. Attributes . . . 364
C.5.2. Format . . . 365
C.6. hpmcounter12 . . . 366
C.6.1. Attributes . . . 366
C.6.2. Format . . . 367
C.7. hpmcounter13 . . . 368
C.7.1. Attributes . . . 368
C.7.2. Format . . . 369
C.8. hpmcounter14 . . . 370
C.8.1. Attributes . . . 370
C.8.2. Format . . . 371
C.9. hpmcounter15 . . . 372
C.9.1. Attributes . . . 372
C.9.2. Format . . . 373
C.10. hpmcounter16 . . . 374
C.10.1. Attributes . . . 374
C.10.2. Format . . . 375
C.11. hpmcounter17 . . . 376
C.11.1. Attributes . . . 376
C.11.2. Format . . . 377
C.12. hpmcounter18 . . . 378
C.12.1. Attributes . . . 378
C.12.2. Format . . . 379
C.13. hpmcounter19 . . . 380
C.13.1. Attributes . . . 380
C.13.2. Format . . . 381
C.14. hpmcounter20 . . . 382
C.14.1. Attributes . . . 382
C.14.2. Format . . . 383

C.15. hpmcounter21 384
C.15.1. Attributes 384
C.15.2. Format 385
C.16. hpmcounter22 386
C.16.1. Attributes 386
C.16.2. Format 387
C.17. hpmcounter23 388
C.17.1. Attributes 388
C.17.2. Format 389
C.18. hpmcounter24 390
C.18.1. Attributes 390
C.18.2. Format 391
C.19. hpmcounter25 392
C.19.1. Attributes 392
C.19.2. Format 393
C.20. hpmcounter26 394
C.20.1. Attributes 394
C.20.2. Format 395
C.21. hpmcounter27 396
C.21.1. Attributes 396
C.21.2. Format 397
C.22. hpmcounter28 398
C.22.1. Attributes 398
C.22.2. Format 399
C.23. hpmcounter29 400
C.23.1. Attributes 400
C.23.2. Format 401
C.24. hpmcounter3 402
C.24.1. Attributes 402
C.24.2. Format 403
C.25. hpmcounter30 404
C.25.1. Attributes 404
C.25.2. Format 405
C.26. hpmcounter31 406
C.26.1. Attributes 406
C.26.2. Format 407
C.27. hpmcounter4 408
C.27.1. Attributes 408
C.27.2. Format 409
C.28. hpmcounter5 410
C.28.1. Attributes 410

C.28.2. Format 411
C.29. hpmcounter6 412
C.29.1. Attributes 412
C.29.2. Format 413
C.30. hpmcounter7 414
C.30.1. Attributes 414
C.30.2. Format 415
C.31. hpmcounter8 416
C.31.1. Attributes 416
C.31.2. Format 417
C.32. hpmcounter9 418
C.32.1. Attributes 418
C.32.2. Format 419
C.33. instret 420
C.33.1. Attributes 420
C.33.2. Format 420
C.34. instreth 421
C.34.1. Attributes 421
C.34.2. Format 421
C.35. mcounteren 422
C.35.1. Attributes 423
C.35.2. Format 423
C.36. mcycle 424
C.36.1. Attributes 424
C.36.2. Format 424
C.37. mcycleh 425
C.37.1. Attributes 425
C.37.2. Format 425
C.38. medeleg 426
C.38.1. Attributes 426
C.38.2. Format 426
C.39. minstret 427
C.39.1. Attributes 427
C.39.2. Format 427
C.40. minstreth 428
C.40.1. Attributes 428
C.40.2. Format 428
C.41. satp 429
C.41.1. Attributes 429
C.41.2. Format 429
C.42. scause 430

C.42.1. Attributes 430
C.42.2. Format 430
C.43. scounteren 431
C.43.1. Attributes 431
C.43.2. Format 431
C.44. sepc 432
C.44.1. Attributes 432
C.44.2. Format 432
C.45. sip 433
C.45.1. Attributes 433
C.45.2. Format 433
C.46. sscratch 434
C.46.1. Attributes 434
C.46.2. Format 434
C.47. sstatus 435
C.47.1. Attributes 435
C.47.2. Format 435
C.48. stval 436
C.48.1. Attributes 436
C.48.2. Format 436
C.49. stvec 437
C.49.1. Attributes 437
C.49.2. Format 437
C.50. time 438
C.50.1. Attributes 438
C.50.2. Format 438
C.51. timeh 439
C.51.1. Attributes 439
C.51.2. Format 439

Licensing and Acknowledgements

TODO: revmark

Copyright and license information

This document is released under the Creative Commons Attribution 4.0 International License.

Copyright 2023 by RISC-V International.

Acknowledgements

Contributors to the RVA20 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.

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 harts 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.

Chapter 2. RVA Profile Class

The RVA profile class targets application processors for markets requiring a high-degree of binary compatibility between compliant implementations.

2.1. RVA Description

RISC-V was designed to provide a highly modular and extensible instruction set and includes a large and growing set of standard extensions, where each standard extension is a bundle of instruction-set features. This is no different than other industry ISAs that continue to add new ISA features. Unlike other ISAs, however, RISC-V has a broad set of contributors and implementers, and also allows users to add their own custom extensions. For some deep embedded markets, highly customized processor configurations are desirable for efficiency, and all software is compiled, ported, and/or developed in-house by the same organization for that specific processor configuration. However, for other markets that expect a substantial fraction of software to be delivered to end-customers in binary form, compatibility across multiple implementations from different RISC-V vendors is required.

The RVIA ISA extension ratification process ensures that all processor vendors have agreed to the specification of a standard extension if present. However, by themselves, the ISA extension specifications do not guarantee that a certain set of standard extensions will be present in all implementations.

The primary goal of the RVA profiles is to align processor vendors targeting binary software markets, so software can rely on the existence of a certain set of ISA features in a particular generation of RISC-V implementations.

Alignment is not only for compatibility, but also to ensure RISC-V is competitive in these markets. The binary app markets are also generally those with the most competitive performance requirements (e.g., mobile, client, server). RVIA cannot mandate the ISA features that a RISC-V binary software ecosystem should use, as each ecosystem will typically select the lowest-common denominator they empirically observe in the deployed devices in their target markets. But RVIA can align hardware vendors to support a common set of features in each generation through the RVA profiles. Without proactive alignment through RVA profiles, RISC-V will be uncompetitive, as even if a particular vendor implements a certain feature, if other vendors do not, then binary distributions will not generally use that feature and all implementations will suffer. While certain features may be discoverable, and alternate code provided in case of presence/absence of a feature, the added cost to support such options is only justified for certain limited cases, and binary app markets will not support a wide range of optional features, particularly for the nascent RISC-V binary app ecosystems.

To maintain alignment and increase RISC-V competitiveness over time, the mandatory set of extensions must increase over time in successive generations of RVA profile. (RVA profiles may eventually have to deprecate previously mandatory instructions, but that is unlikely in the near future.) Note that the RISC-V ISA will continue to evolve, regardless of whether a given software ecosystem settles on a certain generation of profile as the baseline for their ecosystem for many years or even decades. There are many existing binary software ecosystems, which will migrate to RISC-V and evolve at different rates, and more new ones will doubtless be created over the

hopefully long lifetime of RISC-V. High-performance application processors require considerable investment, and no single binary app ecosystem can justify the development costs of these processors, especially for RISC-V in its early stage of adoption.

While the heart of the profile is the set of mandatory extensions, there are several kinds of optional extension that serve important roles in the profile.

The first kind are *localized options*, whose presence or use necessarily differs along geo-political and/or jurisdictional boundaries, with crypto being the obvious example. These will always be optional. At least for crypto, discovery has been found to be perfectly acceptable to handle this optionality on other architectures, as the use of the extensions is well contained in certain libraries.

The second kind of optional extension is a *development option*, which represents a new ISA extension in an early part of its lifecycle but which is intended to become mandatory in a later generation of the RVA profile. Processor vendors and software toolchain providers will have varying development schedules, and providing an optional phase in a new extension's lifecycle provides some flexibility while maintaining overall alignment, and is particularly appropriate when hardware or software development for the extension is complex. Denoting an extension as a *development option* signals to the community that development should be prioritized for such extensions as they will become mandatory.

The third kind of optional extension are *expansion options*, which are those that may have a large implementation cost but are not always needed in a particular platform, and which can be readily handled by discovery. These are also intended to remain available as expansion options in future versions of the profile. Several supervisor-mode extensions fall into this category, e.g., Sv57, which has a notable PPA impact over Sv48 and is not needed on smaller platforms. Some unprivileged extensions that may fall into this category are possible future matrix extensions. These have large implementation costs, and use of matrix instructions can be readily supported with discovery and alternate math libraries.

The fourth kind of optional extensions are *transitory options*, where it is not clear if the extension will change to a mandatory, localized, or expansion option, or be possibly dropped over time. Cryptography provides some examples where earlier cyphers have been broken and are now deprecated. RVIA used this mechanism to enable scalar crypto until vector crypto was ready. Software security features may also be in this category, with examples of deprecated security features occuring in other architectures. As another example, the recent avalanche of new numeric datatypes for AI/ML may eventually subside with a few survivors actually being used longer term. Denoting an option as transitory signals to the community that this extension may be removed in a future profile, though the time scale may span many years.

Except for the localized options, it could be argued that other three kinds of option could be left out of profiles. Binary distributions of applications willing to invest in discovery can use an optional extension, and customers compiling their own applications can take advantage of the feature on a particular implementation, even when that system is mostly running binary distributions that ignore the new extension. However, there is value in providing guidance to align hardware vendors and software developers around what extensions are worth implementing and worth discovering, by designating only a few important features as profile options and limiting their granularity.

2.2. RVA Naming Scheme

The profile class name is RVA (RISC-V Apps processor). A profile release name is an integer (currently 2 digits, could grow in the future). A full profile name is comprised of, in order:

- Prefix **RVA** for RISC-V Applications
- Profile release
- Privilege mode:
  - **U** Unprivileged (available to any privilege mode, **U** is **not** User-mode)
  - **S** Supervisor mode (note that Hypervisor support is treated as an option)
  - **M** Machine mode
- A base ISA XLEN specifier (**32**, **64**)

Profile names are embeddable into RISC-V ISA naming strings. This implies that there will be no standard ISA extension with a name that matches the profile naming convention. This allows tools that process the RISC-V ISA naming string to parse and/or process a combined string.

2.3. RVA Profile Releases

The following profile releases are defined in this profile class:

Name
RVA20

State
ratified

Ratification date
2023-04-03

Name
RVA22

State
ratified

Ratification date
2023-04-03

2.4. Extension Presence

The RVA Profile Class references 42 extensions.

Table 1. Status

Extension	RVA20U64	RVA20S64	RVA22U64	RVA22S64
A	mandatory	-	mandatory	-
C	mandatory	-	mandatory	-
D	mandatory	-	mandatory	-
F	mandatory	-	mandatory	-
H	-	-	-	optional
I	mandatory	-	mandatory	-
M	mandatory	-	mandatory	-
S	-	mandatory	-	mandatory
Ssccptr	-	mandatory	-	mandatory
Sscofpmf	-	-	-	optional
Sscounterenw	-	-	-	mandatory
Sstc	-	-	-	optional
Sstvala	-	mandatory	-	mandatory
Sstvecd	-	mandatory	-	mandatory
Sv39	-	mandatory	-	mandatory
Sv48	-	optional	-	optional
Sv57	-	-	-	optional
Svade	-	mandatory	-	mandatory
Svbare	-	mandatory	-	mandatory
Svinval	-	-	-	mandatory
Svnapot	-	-	-	optional
Svpbmt	-	-	-	mandatory
U	mandatory	-	mandatory	-
V	-	-	optional	-
Za128rs	mandatory	-	mandatory	-
Zba	-	-	mandatory	-
Zbb	-	-	mandatory	-
Zbs	-	-	mandatory	-
Zfhmin	-	-	mandatory	-
Zic64b	-	-	mandatory	-
Zicbom	-	-	mandatory	-
Zicbop	-	-	mandatory	-
Zicboz	-	-	mandatory	-

Ziccamoa	mandatory	-	mandatory	-
Ziccif	mandatory	-	mandatory	-
Zicclsm	mandatory	-	mandatory	-
Ziccrse	mandatory	-	mandatory	-
Zicntr	mandatory	-	mandatory	-
Zifencei	-	mandatory	-	mandatory
Zihintpause	-	-	mandatory	-
Zihpm	optional	-	mandatory	-
Zkt	-	-	mandatory	-

Chapter 3. RVA20 Profile Release

This profile release targets 64-bit application processors for markets requiring a high-degree of binary compatibility between compliant implementations.

RVA20 has 45 associated implementation-defined parameters across all its defined profiles.

3.1. RVA20 Description

This profile release is intended to be used for 64-bit application processors running rich OS stacks. Only user-mode and supervisor-mode profiles are specified in this release.

There is no machine-mode profile currently defined for this release. 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 this release.

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

3.2. 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 application processors in terms of the amount of software that targets this profile.

3.2.1. Mandatory Extensions

The RVA20U64 Profile has 13 mandatory extensions.

- **A** Atomic instructions

  Version ~> 2.1

- **C** Compressed instructions

  Version ~> 2.2

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

  Version ~> 2.2

The rationale to not include Q as a profile option is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect software to

expend effort optimizing use of Q instructions in case they are present.

- **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 profile-defined extension 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 profile-defined extension introduced with RVA20.

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

  Version ~> 1.0

  Ziccamo is a profile-defined extension introduced with RVA20.

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

  Version ~> 1.0

  Za128rs is a profile-defined extension 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 profile-defined extension 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.

- **I** Base integer ISA (RV32I or RV64I)

  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.

Misaligned loads and stores might not be supported.

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.

3.2.2. Optional Extensions

The RVA20U64 Profile has 1 optional extension.

- **Zihpm** Programmable hardware performance counters

  Version ~> 2.0

The number of counters is platform-specific.

The rationale to not make Q an optional extension is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect A-profile software to expend effort optimizing use of Q instructions in case they are present.

Zifencei is not classed as a supported option in the user-mode profile because it is not sufficient by itself to produce the desired effect in a multiprogrammed

multiprocessor environment without OS support, and so the instruction cache flush should always be performed using an OS call rather than using the fence.i instruction. fence.i semantics can be expensive to implement for some hardware memory hierarchy designs, and so alternative non-standard instruction-cache coherence mechanisms can be used behind the OS abstraction. A separate extension is being developed for more general and efficient instruction cache coherence.

The execution environment must provide a means to synchronize writes to instruction memory with instruction fetches, the implementation of which likely relies on the Zifencei extension. For example, RISC-V Linux supplies the `__riscv_flush_icache` system call and a corresponding vDSO call.

3.2.3. Recommendations

Recommendations are not strictly mandated but are included to guide implementers making design choices.

- Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

3.2.4. Implementation-dependencies

RVA20U64 has 17 associated implementation-defined parameters.

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-explainitory
- 'always raise access fault': self-explainitory
- '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.

HW_MSTATUS_FS_DIRTY_UPDATE

Indicates whether or not hardware will write to mstatus.FS

Values are:

never	Hardware never writes mstatus.FS
precise	Hardware writes mstatus.FS to the Dirty (3) state precisely when F registers are modified
imprecise	Hardware writes mstatus.FS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write FP state.

MSTATUS_FS_LEGAL_VALUES

The set of values that mstatus.FS will accept from a software write.

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.

TRAP_ON_ECALL_FROM_U

Whether or not an ECALL-from-U-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

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

TIME_CSR_IMPLEMENTED

Whether or not a real hardware time CSR exists. Implementations can either provide a real CSR or emulate access at M-mode.

Possible values:

true

time/timeh exists, and accessing it will not cause an IllegalInstruction trap

false

time/timeh does not exist. Accessing the CSR will cause an IllegalInstruction trap or enter an unpredictable state, depending on TRAP_ON_UNIMPLEMENTED_CSR. Privileged software may emulate the time CSR, or may pass the exception to a lower level.

3.3. 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.

3.3.1. Mandatory Extensions

The RVA20S64 Profile has 8 mandatory extensions.

- **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.

3.3.2. Optional Extensions

The RVA20S64 Profile has 2 optional extensions.

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

  Version ~> 1.11

- **Ssu64xl**

  Version ~> 1.0

  Ssu64xl is a new extension name introduced with RVA20.

3.3.3. Implementation-dependencies

RVA20S64 has 28 associated implementation-defined parameters.

ASID_WIDTH

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

MSTATUS_FS_LEGAL_VALUES

The set of values that mstatus.FS will accept from a software write.

MSTATUS_FS_WRITEABLE

When S is enabled but F is not, mstatus.FS is optionally writeable.

This parameter only has an effect when both S and F mode are disabled.

MSTATUS_TVM_IMPLEMENTED

Whether or not mstatus.TVM is implemented.

When not implemented mstatus.TVM will be read-only-zero.

MSTATUS_VS_LEGAL_VALUES

The set of values that mstatus.VS will accept from a software write.

MSTATUS_VS_WRITEABLE

When S is enabled but V is not, mstatus.VS is optionally writeable.

This parameter only has an effect when both S and V mode are disabled.

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 instructino 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 instructino 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`.

SATP_MODE_BARE

Whether or not satp.MODE == Bare is supported.

SCOUNTENABLE_EN

Indicates which counters can 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*(`PHYS_ADDR_WIDTH`, `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_ECALL_FROM_S

Whether or not an ECALL-from-S-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

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.

Appendix A: Extension Details

A.1. A Extension

Atomic instructions

Table 2. Status

Profile	v2.1.0
RVA20U64	mandatory
RVA20S64	-

2.1.0

Ratification date

2019-12

A.1.1. 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.1.2. 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.1.3. 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.1.4. 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-explainitory
- 'always raise access fault': self-explainitory
- '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.2. C Extension

Compressed instructions

Table 3. Status

Profile	v2.2.0
RVA20U64	mandatory
RVA20S64	-

2.2.0

Ratification date

2019-12

A.2.1. Synopsis

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.2.2. 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.2.3. Compressed Instruction Formats

Table 4 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 5 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 4. Compressed 16-bit RVC instruction formats

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

Table 5. 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.2.4. Instructions

The following instructions are added by this extension:

c.add	**Add**
c.addi	**Add a sign-extended non-zero immediate**
c.addi16sp	**Add a sign-extended non-zero immediate**
c.addi4spn	**Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer**
c.addiw	**Add a sign-extended non-zero immediate**
c.addw	**Add word**
c.and	**And**
c.andi	**And immediate**
c.beqz	**Branch if Equal Zero**
c.bnez	**Branch if NOT Equal Zero**
c.ebreak	**Breakpoint exception.**
c.j	**Jump**
c.jal	**Jump and Link**

c.jalr	**Jump and Link Register.**
c.jr	**Jump Register**
c.ld	**Load double**
c.ldsp	**Load doubleword from stack pointer**
c.li	**Load the sign-extended 6-bit immediate**
c.lq	**Load quadruple**
c.lqsp	**Load quadruple word from stack pointer**
c.lui	**Load the non-zero 6-bit immediate field into bits 17-12 of the destination register**
c.lw	**Load word**
c.lwsp	**Load word from stack pointer**
c.mv	**Move Register**
c.nop	**Non-operation**
c.or	**Or**
c.sd	**Store double**
c.sdsp	**Store doubleword to stack**
c.slli	**Shift left logical immediate**
c.sq	**Store quadruple**
c.sqsp	**Store quadruple word to stack**
c.srai	**Shift right arithmetical immediate**
c.srli	**Shift right logical immediate**
c.sub	**Subtract**
c.subw	**Subtract word**
c.sw	**Store word**
c.swsp	**Store word to stack**
c.xor	**Exclusive Or**

A.2.5. 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.

A.3. D Extension

Double-precision floating-point

Table 6. Status

Profile	v2.2.0
RVA20U64	mandatory
RVA20S64	-

2.2.0

Ratification date

2019-12

Changes

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

A.3.1. 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.3.2. D Register State

The D extension widens the 32 floating-point registers, `f0-f31`, to 64 bits (FLEN=64 in Table 8. The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section A.3.3.

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.3.3. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower *n*-bit types, *n*<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, $n < m \leq$ 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-boxedvalue.

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*<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*<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.3.4. Instructions

The following instructions are added by this extension:

fadd.d	**No synopsis available.**
fclass.d	**No synopsis available.**

fcvt.d.h	**No synopsis available.**
fcvt.d.l	**No synopsis available.**
fcvt.d.lu	**No synopsis available.**
fcvt.d.s	**No synopsis available.**
fcvt.d.w	**No synopsis available.**
fcvt.d.wu	**No synopsis available.**
fcvt.h.d	**No synopsis available.**
fcvt.l.d	**No synopsis available.**
fcvt.lu.d	**No synopsis available.**
fcvt.s.d	**No synopsis available.**
fcvt.w.d	**No synopsis available.**
fcvt.wu.d	**No synopsis available.**
fcvtmod.w.d	**No synopsis available.**
fdiv.d	**No synopsis available.**
feq.d	**No synopsis available.**
fld	**No synopsis available.**
fle.d	**No synopsis available.**
fleq.d	**No synopsis available.**
fli.d	**No synopsis available.**
flt.d	**No synopsis available.**
fltq.d	**No synopsis available.**
fmadd.d	**No synopsis available.**
fmax.d	**No synopsis available.**
fmaxm.d	**No synopsis available.**
fmin.d	**No synopsis available.**
fminm.d	**No synopsis available.**
fmsub.d	**No synopsis available.**
fmul.d	**No synopsis available.**
fmv.d.x	**No synopsis available.**
fmv.x.d	**No synopsis available.**
fmvh.x.d	**No synopsis available.**
fmvp.d.x	**No synopsis available.**
fnmadd.d	**No synopsis available.**
fnmsub.d	**No synopsis available.**

fround.d	**No synopsis available.**
froundnx.d	**No synopsis available.**
fsd	**No synopsis available.**
fsgnj.d	**No synopsis available.**
fsgnjn.d	**No synopsis available.**
fsgnjx.d	**No synopsis available.**
fsqrt.d	**No synopsis available.**
fsub.d	**No synopsis available.**

A.3.5. 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.4. F Extension

Single-precision floating-point

Table 7. Status

Profile	v2.2.0
RVA20U64	mandatory
RVA20S64	-

2.2.0

Ratification date

2019-12

Changes

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

A.4.1. Synopsis

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.4.2. 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 8. 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 8. RISC-V standard F extension single-precision floating-point state

FLEN-1	0
f0	
f1	
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 RVA20.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 9. 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 9. 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		*Reserved for future use.*

Rounding Mode	Mnemonic	Meaning
110		*Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.

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 10. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 10. 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.4.3. 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.4.4. 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.4.5. Instructions

The following instructions are added by this extension:

fadd.s	**No synopsis available.**
fclass.s	**Single-precision floating-point classify.**
fcvt.l.s	**No synopsis available.**
fcvt.lu.s	**No synopsis available.**
fcvt.s.l	**No synopsis available.**
fcvt.s.lu	**No synopsis available.**
fcvt.s.w	**Convert signed 32-bit integer to single-precision float**
fcvt.s.wu	**No synopsis available.**
fcvt.w.s	**Convert single-precision float to integer word to signed 32-bit integer.**
fcvt.wu.s	**No synopsis available.**
fdiv.s	**No synopsis available.**
feq.s	**Single-precision floating-point equal**

fle.s	**Single-precision floating-point less than or equal**
flt.s	**Single-precision floating-point less than**
flw	**Single-precision floating-point load**
fmadd.s	**No synopsis available.**
fmax.s	**No synopsis available.**
fmin.s	**No synopsis available.**
fmsub.s	**No synopsis available.**
fmul.s	**No synopsis available.**
fmv.h.x	**Half-precision floating-point move from integer**
fmv.w.x	**Single-precision floating-point move from integer**
fmv.x.w	**Move single-precision value from floating-point to integer register**
fnmadd.s	**No synopsis available.**
fnmsub.s	**No synopsis available.**
fsgnj.s	**Single-precision sign inject**
fsgnjn.s	**Single-precision sign inject negate**
fsgnjx.s	**Single-precision sign inject exclusive or**
fsqrt.s	**No synopsis available.**
fsub.s	**No synopsis available.**
fsw	**Single-precision floating-point store**

A.4.6. Parameters

This extension has the following implementation options:

HW_MSTATUS_FS_DIRTY_UPDATE

Indicates whether or not hardware will write to mstatus.FS

Values are:

never	Hardware never writes mstatus.FS
precise	Hardware writes mstatus.FS to the Dirty (3) state precisely when F registers are modified
imprecise	Hardware writes mstatus.FS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write FP state.

MSTATUS_FS_LEGAL_VALUES

The set of values that mstatus.FS will accept from a software write.

MUTABLE_MISA_F

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

A.5. I Extension

Base integer ISA (RV32I or RV64I)

Table 11. Status

Profile	v2.1.0
RVA20U64	mandatory
RVA20S64	-

2.1.0

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.5.1. Synopsis

Base integer instructions — TODO

A.5.2. Instructions

The following instructions are added by this extension:

add	**Integer add**
addi	**Add immediate**
addiw	**Add immediate word**
addw	**Add word**
and	**And**
andi	**And immediate**
auipc	**Add upper immediate to pc**
beq	**Branch if equal**
bge	**Branch if greater than or equal**
bgeu	**Branch if greater than or equal unsigned**
blt	**Branch if less than**
bltu	**Branch if less than unsigned**
bne	**Branch if not equal**
ebreak	**Breakpoint exception**
ecall	**Environment call**

fence	**Memory ordering fence**
jal	**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**
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**
xor	**Exclusive Or**
xori	**Exclusive Or immediate**

A.6. M Extension

Integer multiply and divide instructions

Table 12. Status

Profile	v2.0.0
RVA20U64	mandatory
RVA20S64	-

2.0.0

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.

A.7. S Extension

Supervisor mode

Table 13. Status

Profile	v1.12.0
RVA20U64	-
RVA20S64	mandatory

1.12.0

Ratification date

2021-12

A.7.1. 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.7.2. Instructions

The following instructions are added by this extension:

sfence.vma	**Supervisor memory-management fence**
sret	**Supervisor Exception Return**

A.7.3. 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

MSTATUS_FS_LEGAL_VALUES

The set of values that mstatus.FS will accept from a software write.

MSTATUS_FS_WRITEABLE

When S is enabled but F is not, mstatus.FS is optionally writeable.

This parameter only has an effect when both S and F mode are disabled.

MSTATUS_TVM_IMPLEMENTED

Whether or not mstatus.TVM is implemented.

When not implemented mstatus.TVM will be read-only-zero.

MSTATUS_VS_LEGAL_VALUES

The set of values that mstatus.VS will accept from a software write.

MSTATUS_VS_WRITEABLE

When S is enabled but V is not, mstatus.VS is optionally writeable.

This parameter only has an effect when both S and V mode are disabled.

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 instructino 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 instructino 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.

SATP_MODE_BARE

Whether or not satp.MODE == Bare is supported.

SCOUNTENABLE_EN

Indicates which counters can 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*(`PHYS_ADDR_WIDTH`, `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_ECALL_FROM_S

Whether or not an ECALL-from-S-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

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.

A.8. Ssccptr Extension

Cacheable and coherent main memory page table reads

Table 14. Status

Profile	v1.0.0
RVA20U64	-
RVA20S64	mandatory

1.0.0

Ratification date

Ratification document

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

A.8.1. Synopsis

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.9. Sstvala Extension

stval requirements for RVA profiles

Table 15. Status

Profile	v1.0.0
RVA20U64	-
RVA20S64	mandatory

1.0.0

Ratification date

Ratification document

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

A.9.1. 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.10. Sstvecd Extension

Direct exception vectoring

Table 16. Status

Profile	**v1.0.0**
RVA20U64	-
RVA20S64	mandatory

1.0.0

Ratification date

Ratification document

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

A.10.1. Synopsis

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.11. Sv39 Extension

39-bit virtual address translation (3 level)

Table 17. Status

Profile	v1.12.0
RVA20U64	-
RVA20S64	mandatory

1.12.0

Ratification date

unknown

Ratification document

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

A.11.1. Synopsis

39-bit virtual address translation (3 level)

A.12. Sv48 Extension

48-bit virtual address translation (4 level)

Table 18. Status

Profile	v1.12.0
RVA20U64	-
RVA20S64	optional

1.12.0

Ratification date

unknown

Ratification document

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

A.12.1. Synopsis

48-bit virtual address translation (4 level)

A.13. Svade Extension

Exception on PTE A/D Bits

Table 19. Status

Profile	v1.0.0
RVA20U64	-
RVA20S64	mandatory

1.0.0

Ratification date

2023-11

Ratification document

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

A.13.1. Synopsis

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.14. Svbare Extension

Bare virtual addressing

Table 20. Status

Profile	v1.0.0
RVA20U64	-
RVA20S64	mandatory

1.0.0

Ratification date

==== Synopsis

This extension mandates that the satp mode Bare must be supported.

This extension was ratified as part of the RVA22 profile.

A.15. U Extension

User-level privilege mode

Table 21. Status

Profile	v1.12.0
RVA20U64	-
RVA20S64	-

1.12.0

Ratification date

2019-12

A.15.1. Synopsis

User-level privilege mode

A.15.2. 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.

TRAP_ON_ECALL_FROM_U

Whether or not an ECALL-from-U-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

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.16. Za128rs Extension

Reservation set requirement for RVA profiles

Table 22. Status

Profile	v1.0.0
RVA20U64	mandatory
RVA20S64	-

1.0.0

Ratification date

==== Synopsis

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.17. Ziccamoa Extension

Main memory atomicity requirement for RVA profiles

Table 23. Status

Profile	**v1.0.0**
RVA20U64	mandatory
RVA20S64	-

1.0.0

Ratification date

==== Synopsis

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

This extension was ratified as part of the RVA20 profile.

A.18. Ziccif Extension

Main memory fetch requirement for RVA profiles

Table 24. Status

Profile	**v1.0.0**
RVA20U64	mandatory
RVA20S64	-

1.0.0

Ratification date

==== Synopsis

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.

A.19. Ziccslm Extension

Main memory misaligned requirement for RVA profiles

Table 25. Status

Profile	v1.0.0
RVA20U64	mandatory
RVA20S64	-

1.0.0

Ratification date

==== Synopsis

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.20. Ziccrse Extension

Main memory reservability requirement for RVA profiles

Table 26. Status

Profile	v1.0.0
RVA20U64	mandatory
RVA20S64	-

1.0.0

Ratification date

==== Synopsis

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

This extension was ratified as part of the RVA20 profile.

A.21. Zicntr Extension

Architectural performance counters

Table 27. Status

Profile	v2.0.0
RVA20U64	mandatory
RVA20S64	-

2.0.0

Ratification date

2019-12

A.21.1. Synopsis

Architectural performance counters

A.21.2. Parameters

This extension has the following implementation options:

TIME_CSR_IMPLEMENTED

Whether or not a real hardware time CSR exists. Implementations can either provide a real CSR or emulate access at M-mode.

Possible values:

true

time/timeh exists, and accessing it will not cause an IllegalInstruction trap

false

time/timeh does not exist. Accessing the CSR will cause an IllegalInstruction trap or enter an unpredictable state, depending on TRAP_ON_UNIMPLEMENTED_CSR. Privileged software may emulate the time CSR, or may pass the exception to a lower level.

A.22. Zifencei Extension

Instruction fence

Table 28. Status

Profile	v2.0.0
RVA20U64	-
RVA20S64	mandatory

2.0.0

Ratification date

==== Synopsis

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.22.1. Instructions

The following instructions are added by this extension:

fence.i	**Instruction fence**

A.23. Zihpm Extension

Programmable hardware performance counters

Table 29. Status

Profile	v2.0.0
RVA20U64	optional
RVA20S64	-

2.0.0

Ratification date

unknown

A.23.1. Synopsis

Programmable hardware performance counters

Appendix B: Instruction Details

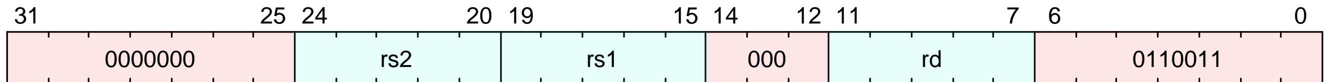
B.1. add

Integer add

This instruction is defined by:

- I, version >= 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	S	U
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

This instruction does not generate synchronous exceptions.

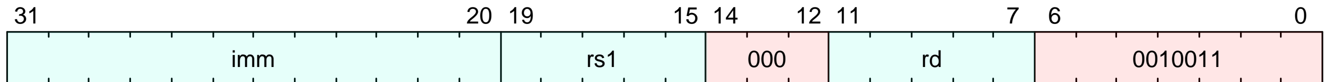
B.2. addi

Add immediate

This instruction is defined by:

- I, version >= 0

B.2.1. Encoding



B.2.2. Synopsis

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

B.2.3. Access

M	S	U
Always	Always	Always

B.2.4. Decode Variables

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

B.2.5. Execution

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

B.2.6. Exceptions

This instruction does not generate synchronous exceptions.

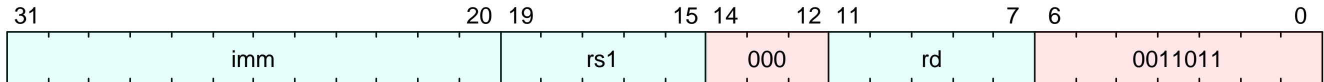
B.3. addiw

Add immediate word

This instruction is defined by:

- I, version >= 0

B.3.1. Encoding



B.3.2. Synopsis

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

B.3.3. Access

M	S	U
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

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

B.3.6. Exceptions

This instruction does not generate synchronous exceptions.

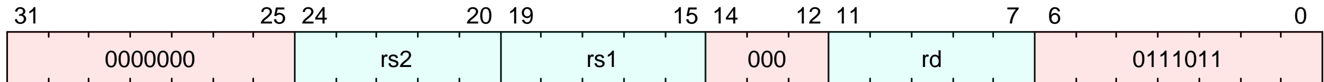
B.4. addw

Add word

This instruction is defined by:

- I, version >= 0

B.4.1. Encoding



B.4.2. Synopsis

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

B.4.3. Access

M	S	U
Always	Always	Always

B.4.4. Decode Variables

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

B.4.5. Execution

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

B.4.6. Exceptions

This instruction does not generate synchronous exceptions.

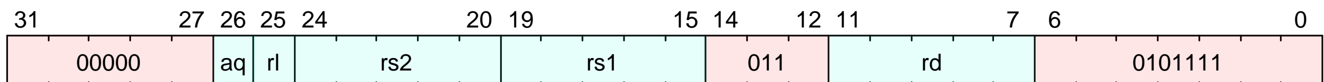
B.5. amoadd.d

Atomic fetch-and-add doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.5.1. Encoding



B.5.2. 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.5.3. Access

M	S	U
Always	Always	Always

B.5.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.5.5. Execution

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

B.5.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

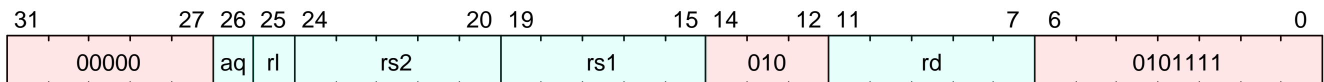
B.6. amoadd.w

Atomic fetch-and-add word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.6.1. Encoding



B.6.2. 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.6.3. Access

M	S	U
Always	Always	Always

B.6.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Add, aq, rl, $encoding);
```

B.6.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

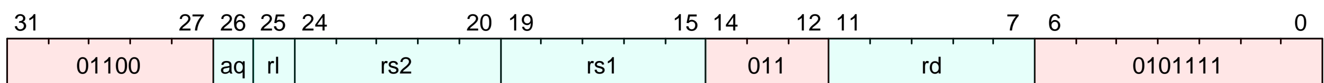
B.7. amoand.d

Atomic fetch-and-and doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.7.1. Encoding



B.7.2. 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.7.3. Access

M	S	U
Always	Always	Always

B.7.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::And, aq, rl, $encoding);
```

B.7.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

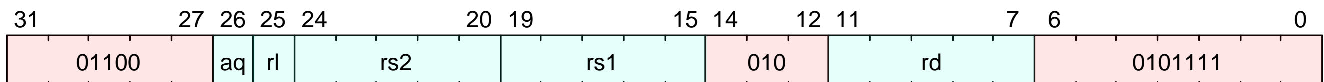
B.8. amoand.w

Atomic fetch-and-and word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.8.1. Encoding



B.8.2. 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.8.3. Access

M	S	U
Always	Always	Always

B.8.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::And, aq, rl, $encoding);
```

B.8.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

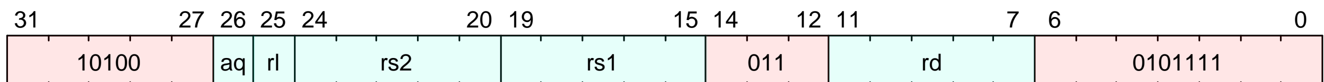
B.9. amomax.d

Atomic MAX doubleword

This instruction is defined by:

- anyOf:
    - A, version >= 0
    - Zaamo, version >= 0

B.9.1. Encoding



B.9.2. 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.9.3. Access

M	S	U
Always	Always	Always

B.9.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Max, aq, rl, $encoding);
```

B.9.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

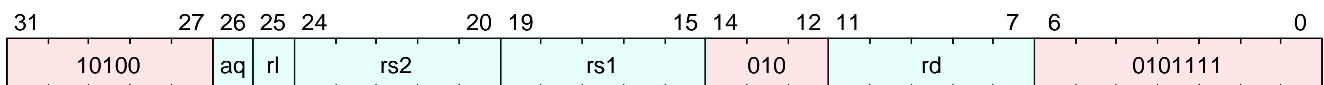
B.10. amomax.w

Atomic MAX word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.10.1. Encoding



B.10.2. 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.10.3. Access

M	S	U
Always	Always	Always

B.10.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
```

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

B.10.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

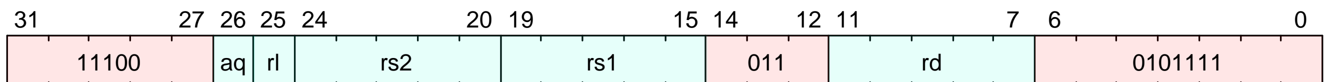
B.11. amomaxu.d

Atomic MAX unsigned doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.11.1. Encoding



B.11.2. 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.11.3. Access

M	S	U
Always	Always	Always

B.11.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Maxu, aq, rl, $encoding);
```

B.11.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

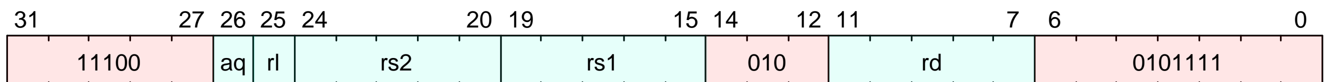
B.12. amomaxu.w

Atomic MAX unsigned word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.12.1. Encoding



B.12.2. 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.12.3. Access

M	S	U
Always	Always	Always

B.12.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
```

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

B.12.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

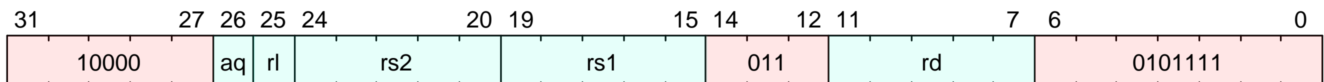
B.13. amomin.d

Atomic MIN doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.13.1. Encoding



B.13.2. 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 mimimum value
- Write the minimum to the address in *rs1*

B.13.3. Access

M	S	U
Always	Always	Always

B.13.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Min, aq, rl, $encoding);
```

B.13.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

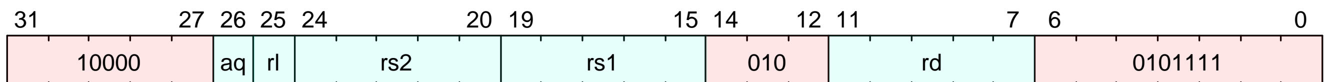
B.14. amomin.w

Atomic MIN word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.14.1. Encoding



B.14.2. 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 mimimum value
- Write the result to the address in *rs1*

B.14.3. Access

M	S	U
Always	Always	Always

B.14.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
```

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

B.14.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

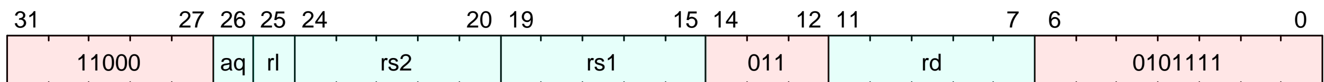
B.15. amominu.d

Atomic MIN unsigned doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.15.1. Encoding



B.15.2. 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 mimimum value
- Write the minimum to the address in *rs1*

B.15.3. Access

M	S	U
Always	Always	Always

B.15.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Minu, aq, rl, $encoding);
```

B.15.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

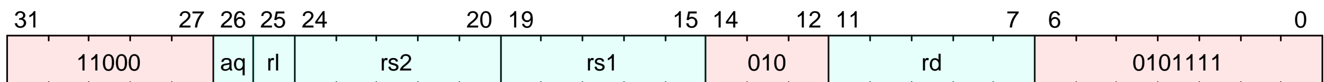
B.16. amominu.w

Atomic MIN unsigned word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.16.1. Encoding



B.16.2. 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 mimimum value
- Write the result to the address in *rs1*

B.16.3. Access

M	S	U
Always	Always	Always

B.16.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
```

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

B.16.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

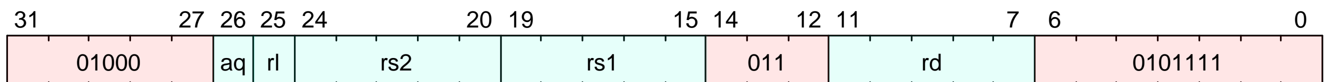
B.17. amoor.d

Atomic fetch-and-or doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.17.1. Encoding



B.17.2. 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.17.3. Access

M	S	U
Always	Always	Always

B.17.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Or, aq, rl, $encoding);
```

B.17.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

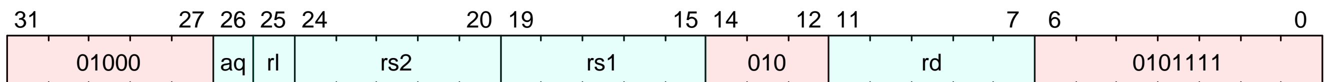
B.18. amoor.w

Atomic fetch-and-or word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.18.1. Encoding



B.18.2. 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.18.3. Access

M	S	U
Always	Always	Always

B.18.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Or, aq, rl, $encoding);
```

B.18.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

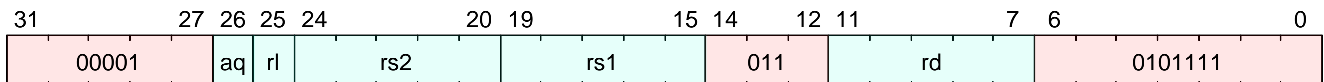
B.19. amoswap.d

Atomic SWAP doubleword

This instruction is defined by:

- anyOf:
    - A, version >= 0
    - Zaamo, version >= 0

B.19.1. Encoding



B.19.2. 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.19.3. Access

M	S	U
Always	Always	Always

B.19.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Swap, aq, rl, $encoding);
```

B.19.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

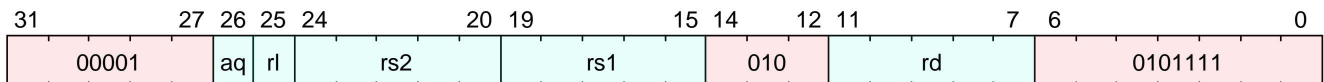
B.20. amoswap.w

Atomic SWAP word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.20.1. Encoding



B.20.2. 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.20.3. Access

M	S	U
Always	Always	Always

B.20.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Swap, aq, rl, $encoding);
```

B.20.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

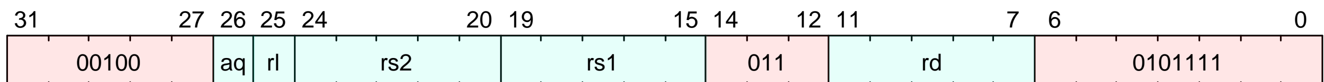
B.21. amoxor.d

Atomic fetch-and-xor doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.21.1. Encoding



B.21.2. 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.21.3. Access

M	S	U
Always	Always	Always

B.21.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Xor, aq, rl, $encoding);
```

B.21.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

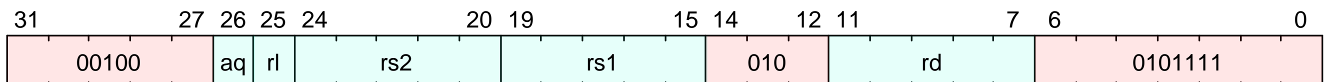
B.22. amoxor.w

Atomic fetch-and-xor word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zaamo, version >= 0

B.22.1. Encoding



B.22.2. 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.22.3. Access

M	S	U
Always	Always	Always

B.22.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
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, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Xor, aq, rl, $encoding);
```

B.22.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

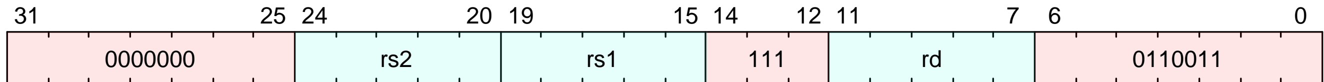
B.23. and

And

This instruction is defined by:

- I, version >= 0

B.23.1. Encoding



B.23.2. Synopsis

And rs1 with rs2, and store the result in rd

B.23.3. Access

M	S	U
Always	Always	Always

B.23.4. Decode Variables

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

B.23.5. Execution

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

B.23.6. Exceptions

This instruction does not generate synchronous exceptions.

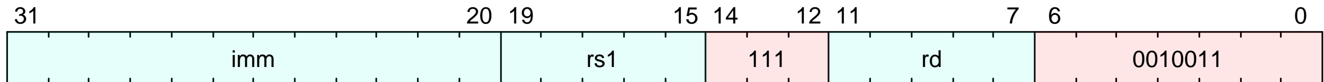
B.24. andi

And immediate

This instruction is defined by:

- I, version >= 0

B.24.1. Encoding



B.24.2. Synopsis

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

B.24.3. Access

M	S	U
Always	Always	Always

B.24.4. Decode Variables

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

B.24.5. Execution

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

B.24.6. Exceptions

This instruction does not generate synchronous exceptions.

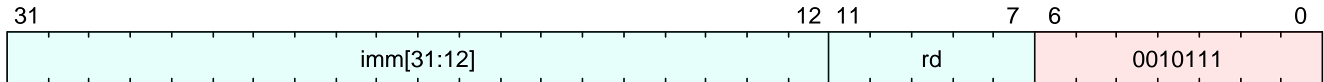
B.25. auipc

Add upper immediate to pc

This instruction is defined by:

- I, version >= 0

B.25.1. Encoding



B.25.2. Synopsis

Add an immediate to the current PC.

B.25.3. Access

M	S	U
Always	Always	Always

B.25.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.25.5. Execution

```
X[rd] = $pc + imm;
```

B.25.6. Exceptions

This instruction does not generate synchronous exceptions.

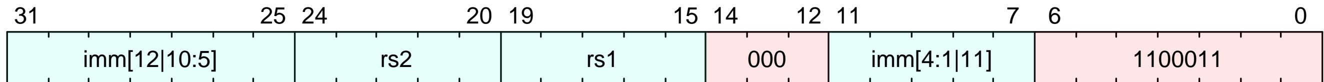
B.26. beq

Branch if equal

This instruction is defined by:

- I, version >= 0

B.26.1. Encoding



B.26.2. 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.26.3. Access

M	S	U
Always	Always	Always

B.26.4. 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.26.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs == rhs) {
  jump_halfword($pc + imm);
}
```

B.26.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

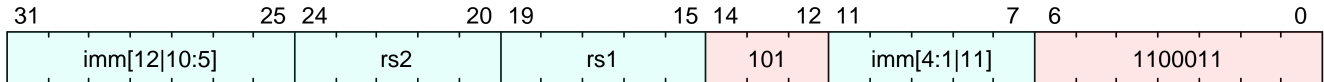
B.27. bge

Branch if greater than or equal

This instruction is defined by:

- I, version >= 0

B.27.1. Encoding



B.27.2. 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.27.3. Access

M	S	U
Always	Always	Always

B.27.4. 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.27.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if ($signed(lhs) >= $signed(rhs)) {
  jump_halfword($pc + imm);
}
```

B.27.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

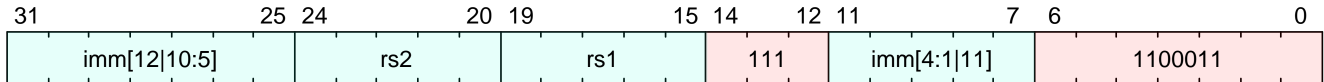
B.28. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

- I, version >= 0

B.28.1. Encoding



B.28.2. 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.28.3. Access

M	S	U
Always	Always	Always

B.28.4. 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.28.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs >= rhs) {
  jump_halfword($pc + imm);
}
```

B.28.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

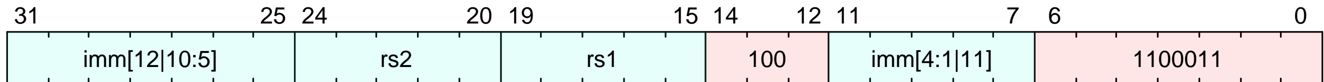
B.29. blt

Branch if less than

This instruction is defined by:

- I, version >= 0

B.29.1. Encoding



B.29.2. 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.29.3. Access

M	S	U
Always	Always	Always

B.29.4. 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.29.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if ($signed(lhs) < $signed(rhs)) {
  jump_halfword($pc + imm);
}
```

B.29.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

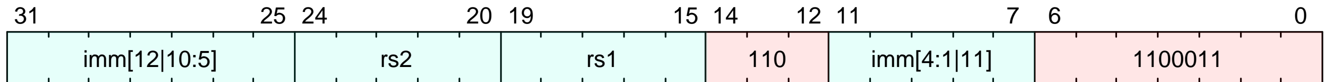
B.30. bltu

Branch if less than unsigned

This instruction is defined by:

- I, version >= 0

B.30.1. Encoding



B.30.2. 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.30.3. Access

M	S	U
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

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

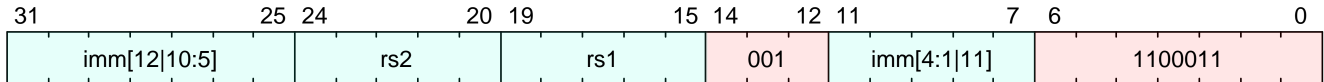
B.31. bne

Branch if not equal

This instruction is defined by:

- I, version >= 0

B.31.1. Encoding



B.31.2. 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.31.3. Access

M	S	U
Always	Always	Always

B.31.4. 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.31.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs != rhs) {
  jump_halfword($pc + imm);
}
```

B.31.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

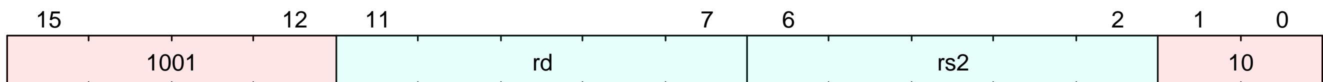
B.32. c.add

Add

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.32.1. Encoding



B.32.2. Synopsis

Add the value in rs2 to rd, and store the result in rd. C.ADD expands into `add rd, rd, rs2`.

B.32.3. Access

M	S	U
Always	Always	Always

B.32.4. Decode Variables

```
Bits<5> rs2 = $encoding[6:2];
Bits<5> rd = $encoding[11:7];
```

B.32.5. Execution

```
XReg t0 = X[rd];
XReg t1 = X[rs2];
X[rd] = t0 + t1;
```

B.32.6. Exceptions

This instruction does not generate synchronous exceptions.

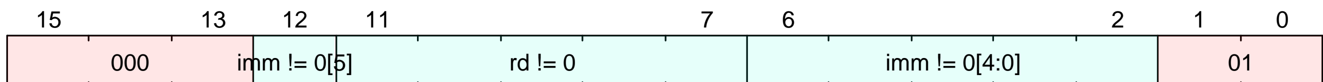
B.33. c.addi

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.33.1. Encoding



B.33.2. Synopsis

C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd. C.ADDI expands into <code>addi rd, rd, imm</code>. C.ADDI is only valid when rd ≠ x0 and imm ≠ 0. The code points with rd=x0 encode the C.NOP instruction; the remaining code points with imm=0 encode HINTs.

B.33.3. Access

M	S	U
Always	Always	Always

B.33.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.33.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rd] + imm;
```

B.33.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

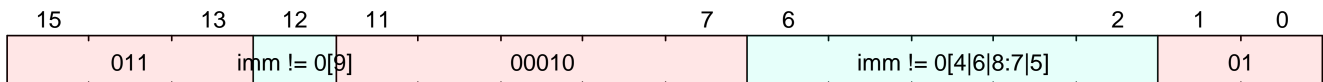
B.34. c.addi16sp

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.34.1. Encoding



B.34.2. Synopsis

C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into <code>addi x2, x2, nzimm[9:4]</code>. C.ADDI16SP is only valid when nzimm ≠ 0; the code point with nzimm=0 is reserved.

B.34.3. Access

M	S	U
Always	Always	Always

B.34.4. Decode Variables

```
Bits<10> imm = {$encoding[12], $encoding[4:3], $encoding[5], $encoding[2], $encoding[
6], 4'd0};
```

B.34.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[2] = X[2] + imm;
```

B.34.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

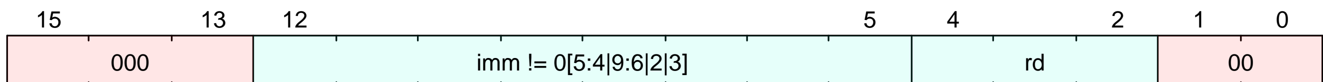
B.35. c.addi4spn

Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.35.1. Encoding



B.35.2. Synopsis

Adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. This instruction is used to generate pointers to stack-allocated variables. It expands to <code>addi rd', x2, nzuimm[9:2]</code>. C.ADDI4SPN is only valid when nzuimm ≠ 0; the code points with nzuimm=0 are reserved.

B.35.3. Access

M	S	U
Always	Always	Always

B.35.4. Decode Variables

```
Bits<10> imm = {$encoding[10:7], $encoding[12:11], $encoding[5], $encoding[6], 2'd0};
Bits<3> rd = $encoding[4:2];
```

B.35.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd + 8] = X[2] + imm;
```

B.35.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

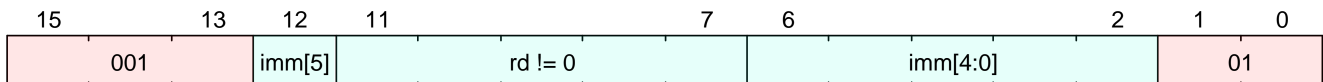
B.36. c.addiw

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.36.1. Encoding



B.36.2. Synopsis

C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation as C.ADDI but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into <code>addiw rd, rd, imm</code>. The immediate can be zero for C.ADDIW, where this corresponds to <code>sext.w rd</code>. C.ADDIW is only valid when rd ≠ x0; the code points with rd=x0 are reserved.

B.36.3. Access

M	S	U
Always	Always	Always

B.36.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.36.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = sext((X[rd] + imm), 32);
```

B.36.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

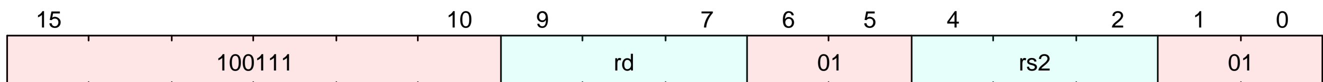
B.37. c.addw

Add word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.37.1. Encoding



B.37.2. Synopsis

Add the 32-bit values in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.ADDW expands into `addw rd, rd, rs2`.

B.37.3. Access

M	S	U
Always	Always	Always

B.37.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.37.5. Execution

```
Bits<32> t0 = X[rd + 8][31:0];
Bits<32> t1 = X[rs2 + 8][31:0];
X[rd + 8] = sext(t0 + t1, 31);
```

B.37.6. Exceptions

This instruction does not generate synchronous exceptions.

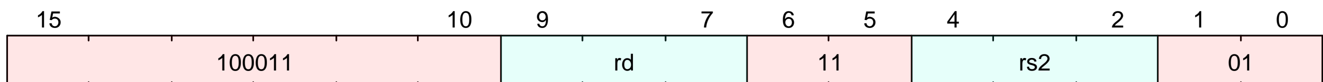
B.38. c.and

And

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.38.1. Encoding



B.38.2. Synopsis

And rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.AND expands into `and rd, rd, rs2`.

B.38.3. Access

M	S	U
Always	Always	Always

B.38.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.38.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 & t1;
```

B.38.6. Exceptions

This instruction does not generate synchronous exceptions.

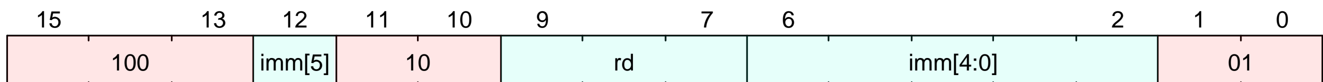
B.39. c.andi

And immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.39.1. Encoding



B.39.2. Synopsis

And an immediate to the value in rd, and store the result in rd. The rd register index should be used as rd+8 (registers x8-x15). C.ANDI expands into `andi rd, rd, imm`.

B.39.3. Access

M	S	U
Always	Always	Always

B.39.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.39.5. Execution

```
X[rd + 8] = X[rd + 8] & imm;
```

B.39.6. Exceptions

This instruction does not generate synchronous exceptions.

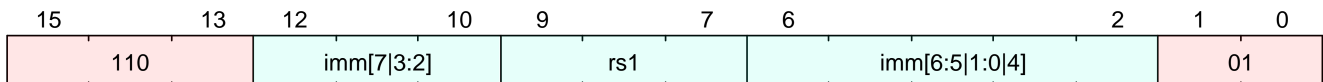
B.40. c.beqz

Branch if Equal Zero

This instruction is defined by:

- anyOf:
    - C, version >= 0
    - Zca, version >= 0

B.40.1. Encoding



B.40.2. Synopsis

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register rs1' is zero. It expands to beq <code>rs1, x0, offset</code>.

B.40.3. Access

M	S	U
Always	Always	Always

B.40.4. Decode Variables

```
Bits<8> imm = {$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10],
$encoding[4:3]};
Bits<3> rs1 = $encoding[9:7];
```

B.40.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[rs1] != 0) {
  jump($pc + imm);
}
```

B.40.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

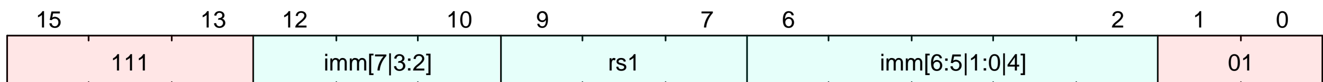
B.41. c.bnez

Branch if NOT Equal Zero

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.41.1. Encoding



B.41.2. Synopsis

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register rs1' is NOT zero. It expands to beq <code>rs1, x0, offset</code>.

B.41.3. Access

M	S	U
Always	Always	Always

B.41.4. Decode Variables

```
Bits<8> imm = {$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10],
$encoding[4:3]};
Bits<3> rs1 = $encoding[9:7];
```

B.41.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[rs1] != 0) {
  jump($pc + imm);
}
```

B.41.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

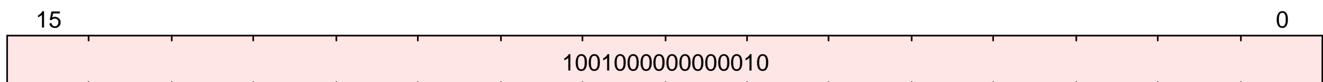
B.42. c.ebreak

Breakpoint exception.

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.42.1. Encoding



B.42.2. Synopsis

The C.EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, C.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.42.3. Access

M	S	U
Always	Always	Always

B.42.4. Decode Variables

```
```

B.42.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (TRAP_ON_EBREAK) {
  raise_precise(ExceptionCode::Breakpoint, mode(), $pc);
} else {
  eei_ebreak();
```

```
}
```

B.42.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint
- IllegalInstruction

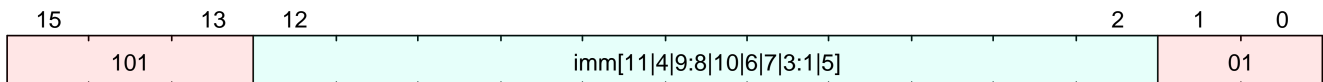
B.43. c.j

Jump

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.43.1. Encoding



B.43.2. Synopsis

C.J performs an unconditional control transfer. The offset is sign-extended and added to the pc to form the jump target address. C.J can therefore target a ±2 KiB range. It expands to jal <code>x0, offset</code>.

B.43.3. Access

M	S	U
Always	Always	Always

B.43.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[12], $encoding[8], $encoding[10:9], $encoding[
6], $encoding[7], $encoding[2], $encoding[11], $encoding[5:3], 1'd0});
```

B.43.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
jump($pc + imm);
```

B.43.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

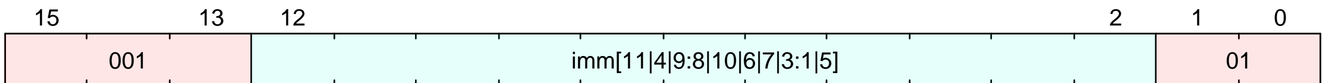
B.44. c.jal

Jump and Link

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.44.1. Encoding



B.44.2. Synopsis

C.JAL is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. It expands to `jal x1, offset`.

B.44.3. Access

M	S	U
Always	Always	Always

B.44.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[12], $encoding[8], $encoding[10:9], $encoding[
6], $encoding[7], $encoding[2], $encoding[11], $encoding[5:3], 1'd0});
```

B.44.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg retrun_addr = $pc + 2;
jump_halfword($pc + imm);
X[1] = retrun_addr;
```

B.44.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- InstructionAddressMisaligned

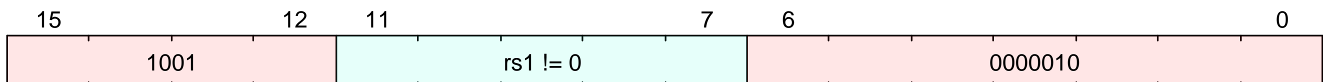
B.45. c.jalr

Jump and Link Register.

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.45.1. Encoding



B.45.2. Synopsis

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. C.JALR expands to jalr x1, 0(rs1).

B.45.3. Access

M	S	U
Always	Always	Always

B.45.4. Decode Variables

```
Bits<5> rs1 = $encoding[11:7];
```

B.45.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg returnaddr;
returnaddr = $pc + 2;
jump(X[rs1]);
X[1] = returnaddr;
```

B.45.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- InstructionAddressMisaligned

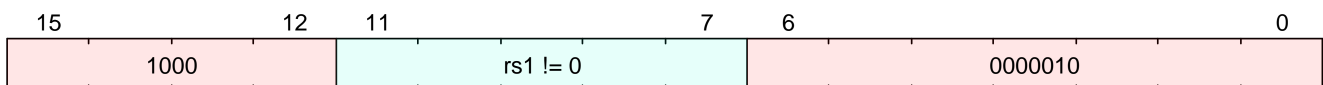
B.46. c.jr

Jump Register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.46.1. Encoding



B.46.2. Synopsis

C.JR (jump register) performs an unconditional control transfer to the address in register rs1. C.JR expands to jalr x0, 0(rs1).

B.46.3. Access

M	S	U
Always	Always	Always

B.46.4. Decode Variables

```
Bits<5> rs1 = $encoding[11:7];
```

B.46.5. Execution

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

B.46.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

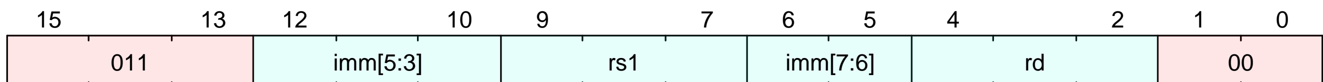
B.47. c.ld

Load double

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.47.1. Encoding



B.47.2. Synopsis

Loads a 64-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to `ld rd, offset(rs1)`.

B.47.3. Access

M	S	U
Always	Always	Always

B.47.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> rd = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.47.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<64>(virtual_address, $encoding), 64);
```

B.47.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

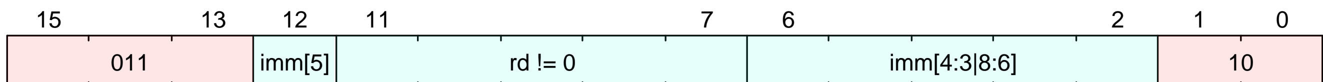
B.48. c.ldsp

Load doubleword from stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.48.1. Encoding



B.48.2. Synopsis

C.LDSP is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register rd. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to ld <code>rd, offset(x2)</code>. C.LDSP is only valid when rd ≠ x0 the code points with rd=x0 are reserved.

B.48.3. Access

M	S	U
Always	Always	Always

B.48.4. Decode Variables

```
Bits<9> imm = {$encoding[4:2], $encoding[12], $encoding[6:5], 3'd0};
Bits<5> rd = $encoding[11:7];
```

B.48.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
X[rd] = read_memory<64>(virtual_address, $encoding);
```

B.48.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

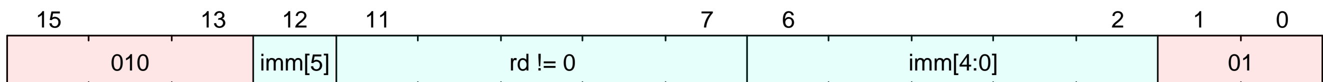
B.49. c.li

Load the sign-extended 6-bit immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.49.1. Encoding



B.49.2. Synopsis

C.LI loads the sign-extended 6-bit immediate, imm, into register rd. C.LI expands into <code>addi rd, x0, imm</code>. C.LI is only valid when rd ≠ x0; the code points with rd=x0 encode HINTs.

B.49.3. Access

M	S	U
Always	Always	Always

B.49.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.49.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = imm;
```

B.49.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

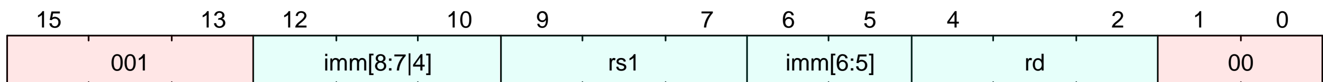
B.50. c.lq

Load quadruple

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.50.1. Encoding



B.50.2. Synopsis

Loads a 128-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 16, to the base address in register rs1. It expands to `lq rd, offset(rs1)`.

B.50.3. Access

M	S	U
Always	Always	Always

B.50.4. Decode Variables

```
Bits<9> imm = {$encoding[12:11], $encoding[6:5], $encoding[10], 4'd0};
Bits<3> rd = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.50.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<128>(virtual_address, $encoding), 128);
```

B.50.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

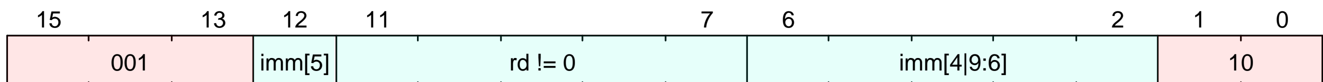
B.51. c.lqsp

Load quadruple word from stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.51.1. Encoding



B.51.2. Synopsis

C.LQSP is an RV128C-only instruction that loads a 128-bit value from memory into register rd. It computes its effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, x2. It expands to <code>lq</code> <code>rd, offset(x2)</code>. C.LQSP is only valid when rd ≠ x0 the code points with rd=x0 are reserved.

B.51.3. Access

M	S	U
Always	Always	Always

B.51.4. Decode Variables

```
Bits<10> imm = {$encoding[5:2], $encoding[12], $encoding[6], 4'd0};
Bits<5> rd = $encoding[11:7];
```

B.51.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
X[rd] = read_memory<128>(virtual_address, $encoding);
```

B.51.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

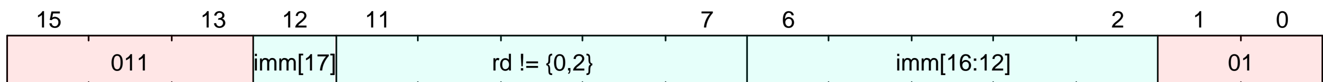
B.52. c.lui

Load the non-zero 6-bit immediate field into bits 17-12 of the destination register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.52.1. Encoding



B.52.2. Synopsis

C.LUI loads the non-zero 6-bit immediate field into bits 17-12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI expands into <code>lui rd, imm</code>. C.LUI is only valid when rd≠x0 and rd≠x2, and when the immediate is not equal to zero. The code points with imm=0 are reserved; the remaining code points with rd=x0 are HINTs; and the remaining code points with rd=x2 correspond to the C.ADDI16SP instruction

B.52.3. Access

M	S	U
Always	Always	Always

B.52.4. Decode Variables

```
Bits<18> imm = {$encoding[12], $encoding[6:2], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.52.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = imm;
```

B.52.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

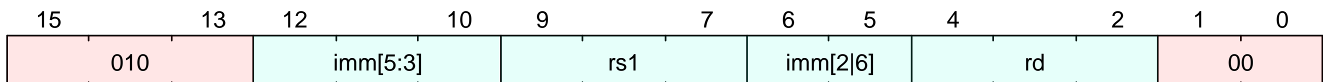
B.53. c.lw

Load word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.53.1. Encoding



B.53.2. Synopsis

Loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1. It expands to `lw rd, offset(rs1)`.

B.53.3. Access

M	S	U
Always	Always	Always

B.53.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> rd = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.53.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

B.53.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

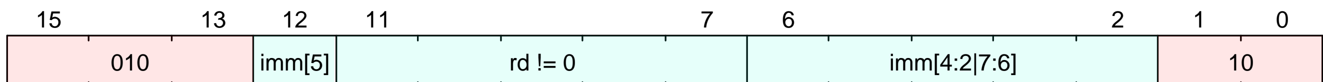
B.54. c.lwsp

Load word from stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.54.1. Encoding



B.54.2. Synopsis

Loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to lw <code>rd, offset(x2)</code>. C.LWSP is only valid when rd ≠ x0. The code points with rd=x0 are reserved.

B.54.3. Access

M	S	U
Always	Always	Always

B.54.4. Decode Variables

```
Bits<8> imm = {$encoding[3:2], $encoding[12], $encoding[6:4], 2'd0};
Bits<5> rd = $encoding[11:7];
```

B.54.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
X[rd] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

B.54.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

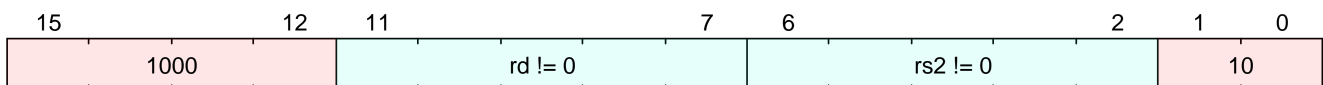
B.55. c.mv

Move Register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.55.1. Encoding



B.55.2. Synopsis

C.MV (move register) performs copy of the data in register rs2 to register rd C.MV expands to addi rd, x0, rs2.

B.55.3. Access

M	S	U
Always	Always	Always

B.55.4. Decode Variables

```
Bits<5> rd = $encoding[11:7];
Bits<5> rs2 = $encoding[6:2];
```

B.55.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2];
```

B.55.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

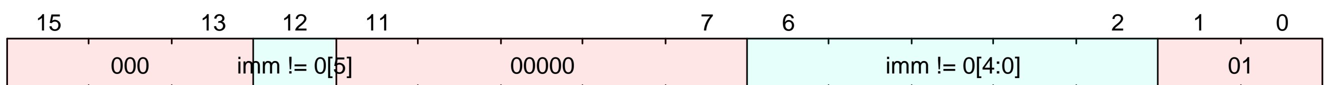
B.56. c.nop

Non-operation

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.56.1. Encoding



B.56.2. Synopsis

C.NOP expands into `addi x0, x0, imm`.

B.56.3. Access

M	S	U
Always	Always	Always

B.56.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
```

B.56.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
```

B.56.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

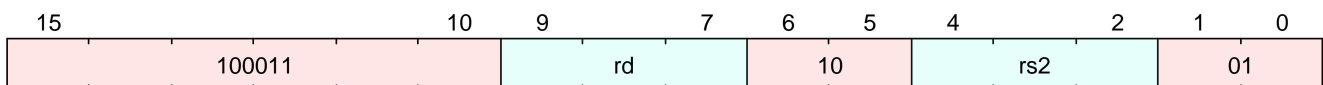
B.57. c.or

Or

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.57.1. Encoding



B.57.2. Synopsis

Or rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.OR expands into `or rd, rd, rs2`.

B.57.3. Access

M	S	U
Always	Always	Always

B.57.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.57.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 | t1;
```

B.57.6. Exceptions

This instruction does not generate synchronous exceptions.

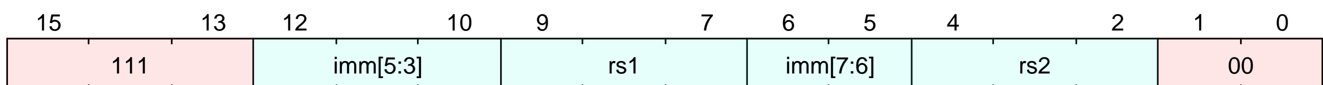
B.58. c.sd

Store double

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.58.1. Encoding



B.58.2. Synopsis

Stores a 64-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to `sd rs2, offset(rs1)`.

B.58.3. Access

M	S	U
Always	Always	Always

B.58.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> rs2 = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.58.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.58.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

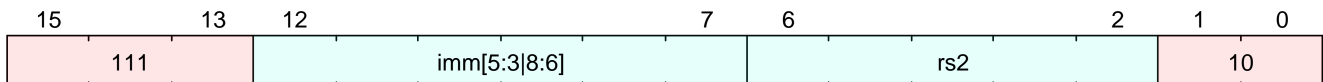
B.59. c.sdsp

Store doubleword to stack

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.59.1. Encoding



B.59.2. Synopsis

Stores a 64-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to `sd rs2, offset(x2)`.

B.59.3. Access

M	S	U
Always	Always	Always

B.59.4. Decode Variables

```
Bits<9> imm = {$encoding[9:7], $encoding[12:10], 3'd0};
Bits<5> rs2 = $encoding[6:2];
```

B.59.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.59.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

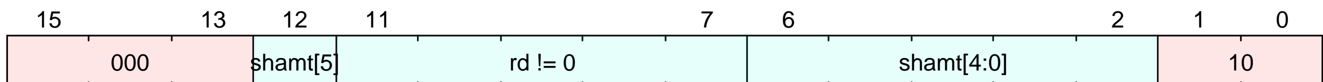
B.60. c.slli

Shift left logical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.60.1. Encoding



B.60.2. Synopsis

Shift the value in rd left by shamt, and store the result back in rd. C.SLLI expands into `slli rd, rd, shamt`.

B.60.3. Access

M	S	U
Always	Always	Always

B.60.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.60.5. Execution

```
X[rd] = X[rd] << shamt;
```

B.60.6. Exceptions

This instruction does not generate synchronous exceptions.

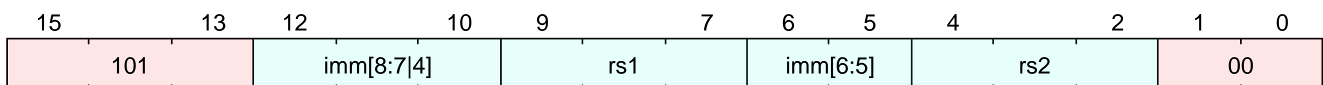
B.61. c.sq

Store quadruple

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.61.1. Encoding



B.61.2. Synopsis

Stores a 128-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 16, to the base address in register rs1. It expands to `sq rs2, offset(rs1)`.

B.61.3. Access

M	S	U
Always	Always	Always

B.61.4. Decode Variables

```
Bits<9> imm = {$encoding[12:11], $encoding[6:5], $encoding[10], 4'd0};
Bits<3> rs2 = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.61.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
write_memory<128>(virtual_address, X[rs2], $encoding);
```

B.61.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

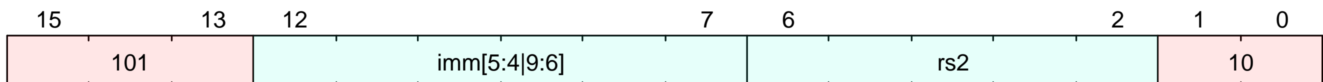
B.62. c.sqsp

Store quadruple word to stack

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.62.1. Encoding



B.62.2. Synopsis

Stores a 128-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, x2. It expands to `sq rs2, offset(x2)`.

B.62.3. Access

M	S	U
Always	Always	Always

B.62.4. Decode Variables

```
Bits<10> imm = {$encoding[10:7], $encoding[12:11], 4'd0};
Bits<5> rs2 = $encoding[6:2];
```

B.62.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<128>(virtual_address, X[rs2], $encoding);
```

B.62.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

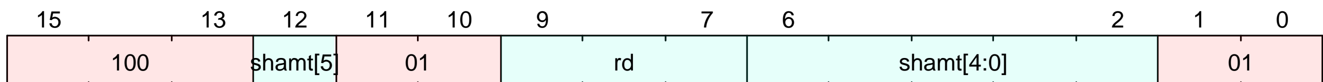
B.63. c.srai

Shift right arithmetical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.63.1. Encoding



B.63.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the value in rd right by shamt, and store the result in rd. The rd register index should be used as rd+8 (registers x8-x15). C.SRAI expands into `srai rd, rd, shamt`.

B.63.3. Access

M	S	U
Always	Always	Always

B.63.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.63.5. Execution

```
X[rd + 8] = X[rd + 8] >>> shamt;
```

B.63.6. Exceptions

This instruction does not generate synchronous exceptions.

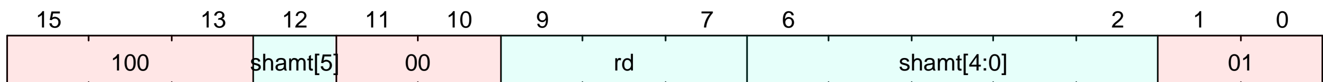
B.64. c.srli

Shift right logical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.64.1. Encoding



B.64.2. Synopsis

Shift the value in rd right by shamt, and store the result back in rd. The rd register index should be used as rd+8 (registers x8-x15). C.SRLI expands into `srli rd, rd, shamt`.

B.64.3. Access

M	S	U
Always	Always	Always

B.64.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.64.5. Execution

```
X[rd + 8] = X[rd + 8] >> shamt;
```

B.64.6. Exceptions

This instruction does not generate synchronous exceptions.

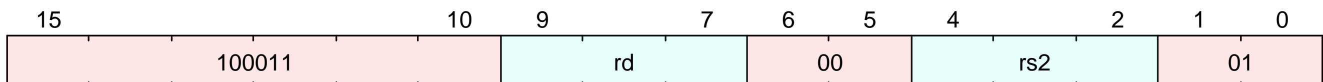
B.65. c.sub

Subtract

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.65.1. Encoding



B.65.2. Synopsis

Subtract the value in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.SUB expands into `sub rd, rd, rs2`.

B.65.3. Access

M	S	U
Always	Always	Always

B.65.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.65.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 - t1;
```

B.65.6. Exceptions

This instruction does not generate synchronous exceptions.

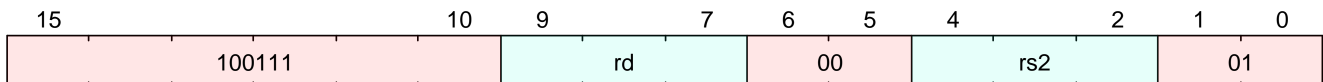
B.66. c.subw

Subtract word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.66.1. Encoding



B.66.2. Synopsis

Subtract the 32-bit values in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.SUBW expands into `subw rd, rd, rs2`.

B.66.3. Access

M	S	U
Always	Always	Always

B.66.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.66.5. Execution

```
Bits<32> t0 = X[rd + 8][31:0];
Bits<32> t1 = X[rs2 + 8][31:0];
X[rd + 8] = sext(t0 - t1, 31);
```

B.66.6. Exceptions

This instruction does not generate synchronous exceptions.

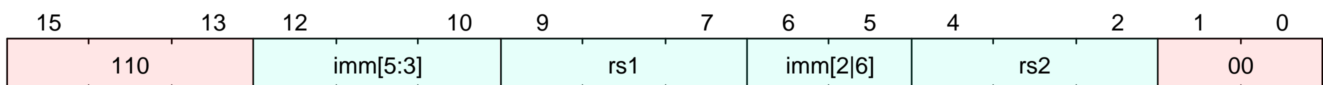
B.67. c.sw

Store word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.67.1. Encoding



B.67.2. Synopsis

Stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1. It expands to `sw rs2, offset(rs1)`.

B.67.3. Access

M	S	U
Always	Always	Always

B.67.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> rs2 = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.67.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.67.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

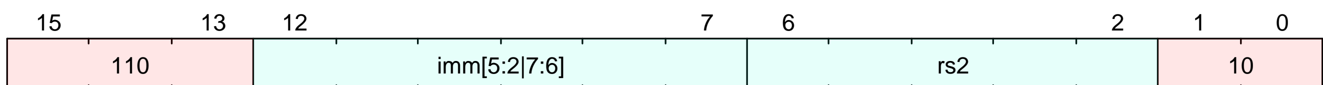
B.68. c.swsp

Store word to stack

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.68.1. Encoding



B.68.2. Synopsis

Stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to `sw rs2, offset(x2)`.

B.68.3. Access

M	S	U
Always	Always	Always

B.68.4. Decode Variables

```
Bits<8> imm = {$encoding[8:7], $encoding[12:9], 2'd0};
Bits<5> rs2 = $encoding[6:2];
```

B.68.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.68.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

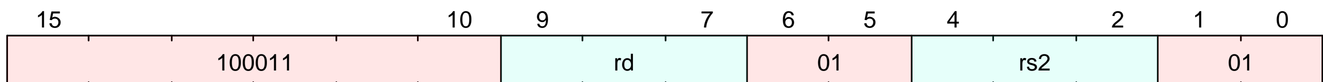
B.69. c.xor

Exclusive Or

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.69.1. Encoding



B.69.2. Synopsis

Exclusive or rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.XOR expands into `xor rd, rd, rs2`.

B.69.3. Access

M	S	U
Always	Always	Always

B.69.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.69.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 ^ t1;
```

B.69.6. Exceptions

This instruction does not generate synchronous exceptions.

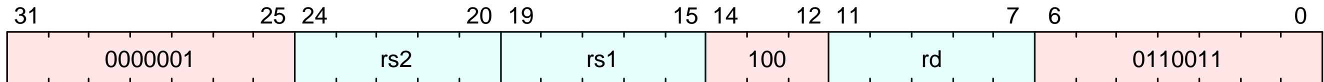
B.70. div

Signed division

This instruction is defined by:

- M, version >= 0

B.70.1. Encoding



B.70.2. 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.70.3. Access

M	S	U
Always	Always	Always

B.70.4. Decode Variables

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

B.70.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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);
```

```
}
```

B.70.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

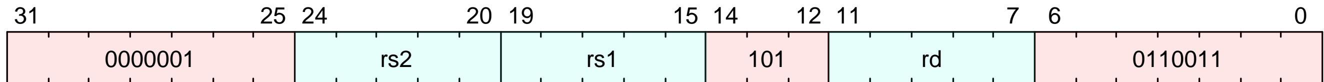
B.71. divu

Unsigned division

This instruction is defined by:

- M, version >= 0

B.71.1. Encoding



B.71.2. 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.71.3. Access

M	S	U
Always	Always	Always

B.71.4. Decode Variables

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

B.71.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = {XLEN{1'b1}};
} else {
  X[rd] = src1 / src2;
}
```

B.71.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

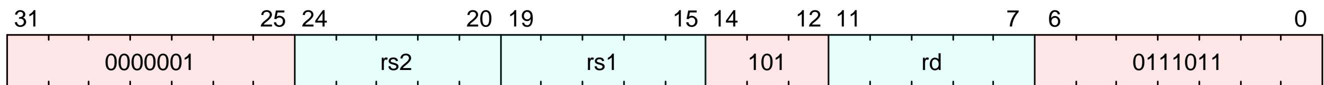
B.72. divuw

Unsigned 32-bit division

This instruction is defined by:

- M, version >= 0

B.72.1. Encoding



B.72.2. 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.72.3. Access

M	S	U
Always	Always	Always

B.72.4. Decode Variables

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

B.72.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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};
}
```

B.72.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

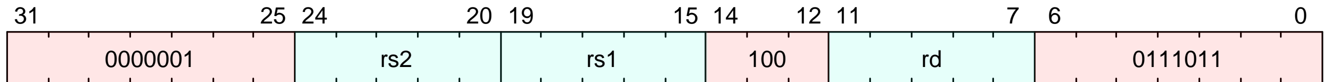
B.73. divw

Signed 32-bit division

This instruction is defined by:

- M, version >= 0

B.73.1. Encoding



B.73.2. 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.73.3. Access

M	S	U
Always	Always	Always

B.73.4. Decode Variables

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

B.73.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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};
}
```

B.73.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

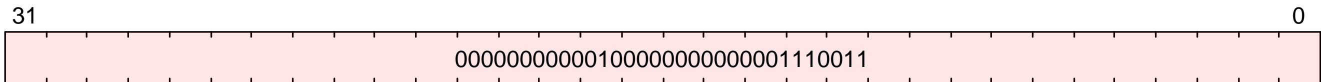
B.74. ebreak

Breakpoint exception

This instruction is defined by:

- I, version >= 0

B.74.1. Encoding



B.74.2. 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.74.3. Access

M	S	U
Always	Always	Always

B.74.4. Decode Variables

```
```

B.74.5. Execution

```
if (TRAP_ON_EBREAK) {
  raise_precise(ExceptionCode::Breakpoint, mode(), $pc);
} else {
  eei_ebreak();
}
```

B.74.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

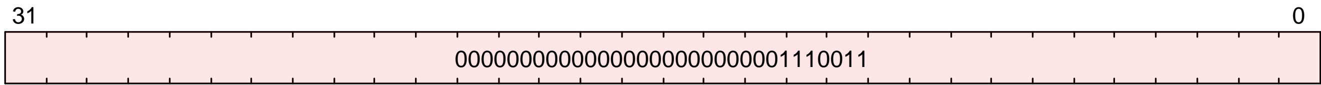
B.75. ecall

Environment call

This instruction is defined by:

- I, version >= 0

B.75.1. Encoding



B.75.2. 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.75.3. Access

M	S	U
Always	Always	Always

B.75.4. Decode Variables

```
```

B.75.5. Execution

```
if (mode() == PrivilegeMode::M) {
  if (TRAP_ON_ECALL_FROM_M) {
    raise_precise(ExceptionCode::Mcall, PrivilegeMode::M, 0);
  } else {
    eei_ecall_from_m();
  }
} else if (mode() == PrivilegeMode::S) {
```

```
  if (TRAP_ON_ECALL_FROM_S) {
    raise_precise(ExceptionCode::Scall, PrivilegeMode::S, 0);
  } else {
    eei_ecall_from_s();
  }
} else if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::VU) {
  if (TRAP_ON_ECALL_FROM_U) {
    raise_precise(ExceptionCode::Ucall, mode(), 0);
  } else {
    eei_ecall_from_u();
  }
} else if (mode() == PrivilegeMode::VS) {
  if (TRAP_ON_ECALL_FROM_VS) {
    raise_precise(ExceptionCode::VScall, PrivilegeMode::VS, 0);
  } else {
    eei_ecall_from_vs();
  }
}
```

B.75.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

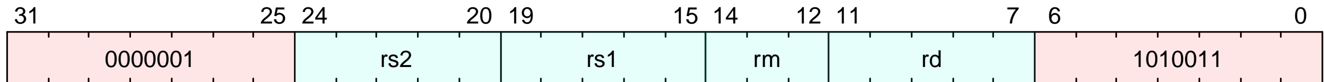
B.76. fadd.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.76.1. Encoding



B.76.2. Synopsis

No description available.

B.76.3. Access

M	S	U
Always	Always	Always

B.76.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.76.5. Execution

```

```

B.76.6. Exceptions

This instruction does not generate synchronous exceptions.

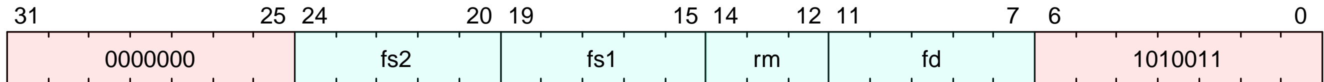
B.77. fadd.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.77.1. Encoding



B.77.2. Synopsis

No description available.

B.77.3. Access

M	S	U
Always	Always	Always

B.77.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.77.5. Execution

```

```

B.77.6. Exceptions

This instruction does not generate synchronous exceptions.

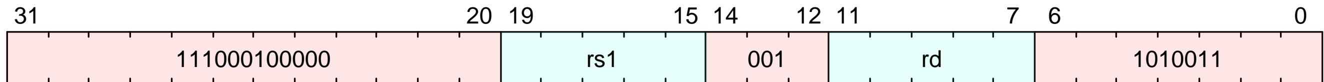
B.78. fclass.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.78.1. Encoding



B.78.2. Synopsis

No description available.

B.78.3. Access

M	S	U
Always	Always	Always

B.78.4. Decode Variables

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

B.78.5. Execution

```

```

B.78.6. Exceptions

This instruction does not generate synchronous exceptions.

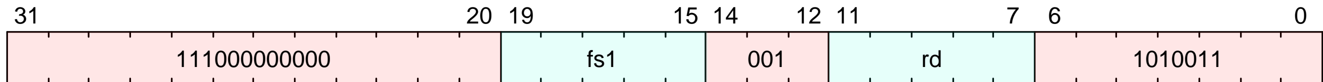
B.79. fclass.s

Single-precision floating-point classify.

This instruction is defined by:

- F, version >= 0

B.79.1. Encoding



B.79.2. Synopsis

The fclass.s instruction examines the value in floating-point register *fs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in the table below. The corresponding bit in *rd* will be set if the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in rd will be set. fclass.s does not set the floating-point exception flags.

Table 30. Format of result of `fclass` instruction.

rd **bit**	**Meaning**
0	*rs1* is `-\infty`.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is `-0`.
4	*rs1* is `+0`.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is `+\infty`.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

B.79.3. Access

M	S	U
Always	Always	Always

B.79.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
```

```
Bits<5> rd = $encoding[11:7];
```

B.79.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = f[fs1][31:0];
if (is_sp_neg_inf?(sp_value)) {
  X[rd] = 1 << 0;
} else if (is_sp_neg_norm?(sp_value)) {
  X[rd] = 1 << 1;
} else if (is_sp_neg_subnorm?(sp_value)) {
  X[rd] = 1 << 2;
} else if (is_sp_neg_zero?(sp_value)) {
  X[rd] = 1 << 3;
} else if (is_sp_pos_zero?(sp_value)) {
  X[rd] = 1 << 4;
} else if (is_sp_pos_subnorm?(sp_value)) {
  X[rd] = 1 << 5;
} else if (is_sp_pos_norm?(sp_value)) {
  X[rd] = 1 << 6;
} else if (is_sp_pos_inf?(sp_value)) {
  X[rd] = 1 << 7;
} else if (is_sp_signaling_nan?(sp_value)) {
  X[rd] = 1 << 8;
} else {
  assert(is_sp_quiet_nan?(sp_value), "Unexpected SP value");
  X[rd] = 1 << 9;
}
```

B.79.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

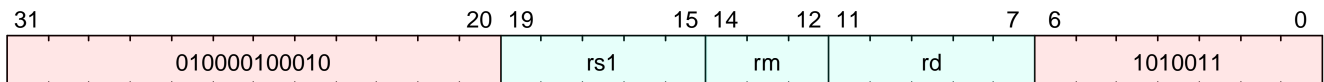
B.80. fcvt.d.h

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfh, version >= 0

B.80.1. Encoding



B.80.2. Synopsis

No description available.

B.80.3. Access

M	S	U
Always	Always	Always

B.80.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.80.5. Execution

```
```

B.80.6. Exceptions

This instruction does not generate synchronous exceptions.

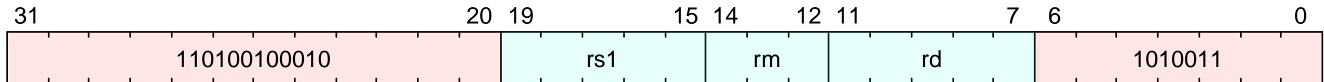
B.81. fcvt.d.l

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.81.1. Encoding



B.81.2. Synopsis

No description available.

B.81.3. Access

M	S	U
Always	Always	Always

B.81.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.81.5. Execution

```
```

B.81.6. Exceptions

This instruction does not generate synchronous exceptions.

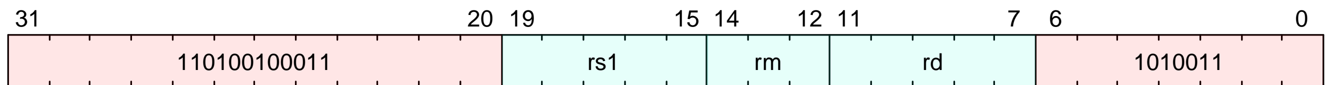
B.82. fcvt.d.lu

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.82.1. Encoding



B.82.2. Synopsis

No description available.

B.82.3. Access

M	S	U
Always	Always	Always

B.82.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.82.5. Execution

```
```

B.82.6. Exceptions

This instruction does not generate synchronous exceptions.

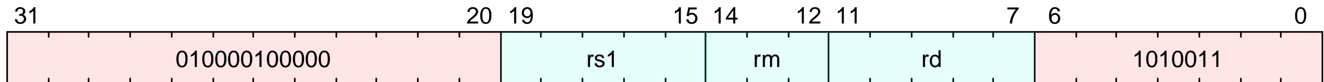
B.83. fcvt.d.s

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.83.1. Encoding



B.83.2. Synopsis

No description available.

B.83.3. Access

M	S	U
Always	Always	Always

B.83.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.83.5. Execution

```
```

B.83.6. Exceptions

This instruction does not generate synchronous exceptions.

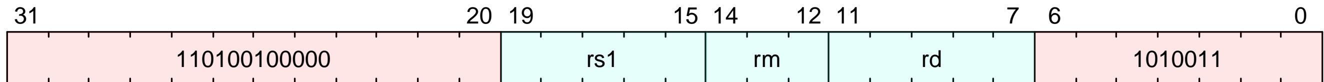
B.84. fcvt.d.w

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.84.1. Encoding



B.84.2. Synopsis

No description available.

B.84.3. Access

M	S	U
Always	Always	Always

B.84.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.84.5. Execution

```
```

B.84.6. Exceptions

This instruction does not generate synchronous exceptions.

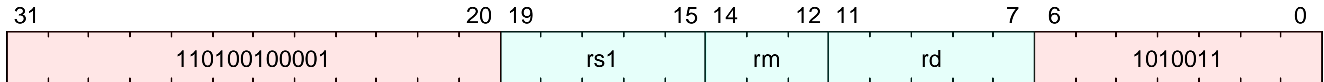
B.85. fcvt.d.wu

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.85.1. Encoding



B.85.2. Synopsis

No description available.

B.85.3. Access

M	S	U
Always	Always	Always

B.85.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.85.5. Execution

```

```

B.85.6. Exceptions

This instruction does not generate synchronous exceptions.

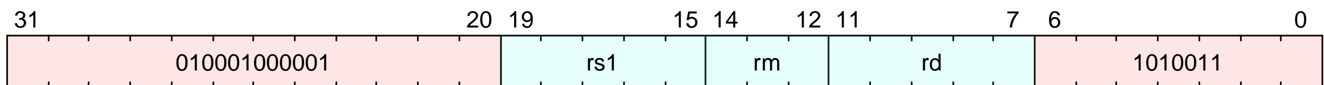
B.86. fcvt.h.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfh, version >= 0

B.86.1. Encoding



B.86.2. Synopsis

No description available.

B.86.3. Access

M	S	U
Always	Always	Always

B.86.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.86.5. Execution

```
```

B.86.6. Exceptions

This instruction does not generate synchronous exceptions.

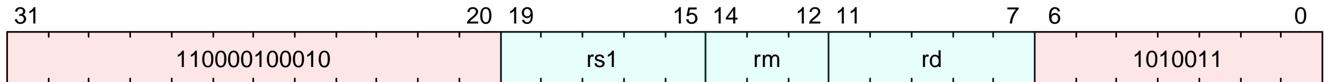
B.87. fcvt.l.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.87.1. Encoding



B.87.2. Synopsis

No description available.

B.87.3. Access

M	S	U
Always	Always	Always

B.87.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.87.5. Execution

```

```

B.87.6. Exceptions

This instruction does not generate synchronous exceptions.

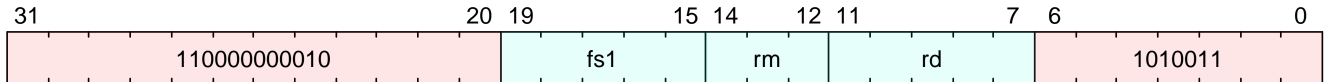
B.88. fcvt.l.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.88.1. Encoding



B.88.2. Synopsis

No description available.

B.88.3. Access

M	S	U
Always	Always	Always

B.88.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.88.5. Execution

```
```

B.88.6. Exceptions

This instruction does not generate synchronous exceptions.

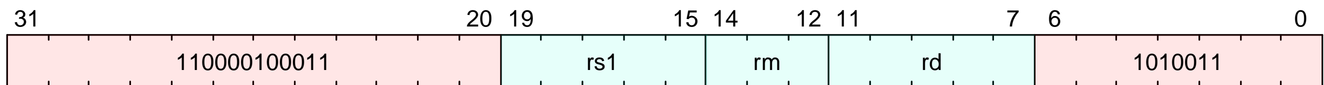
B.89. fcvt.lu.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.89.1. Encoding



B.89.2. Synopsis

No description available.

B.89.3. Access

M	S	U
Always	Always	Always

B.89.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.89.5. Execution

```

```

B.89.6. Exceptions

This instruction does not generate synchronous exceptions.

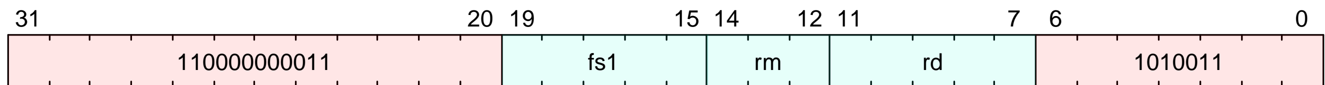
B.90. fcvt.lu.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.90.1. Encoding



B.90.2. Synopsis

No description available.

B.90.3. Access

M	S	U
Always	Always	Always

B.90.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.90.5. Execution

```

```

B.90.6. Exceptions

This instruction does not generate synchronous exceptions.

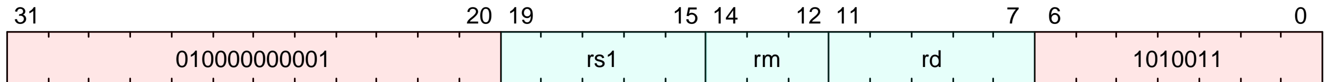
B.91. fcvt.s.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.91.1. Encoding



B.91.2. Synopsis

No description available.

B.91.3. Access

M	S	U
Always	Always	Always

B.91.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.91.5. Execution

```

```

B.91.6. Exceptions

This instruction does not generate synchronous exceptions.

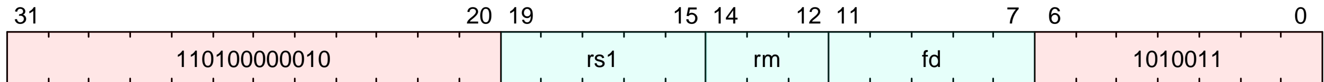
B.92. fcvt.s.l

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.92.1. Encoding



B.92.2. Synopsis

No description available.

B.92.3. Access

M	S	U
Always	Always	Always

B.92.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.92.5. Execution

```
```

B.92.6. Exceptions

This instruction does not generate synchronous exceptions.

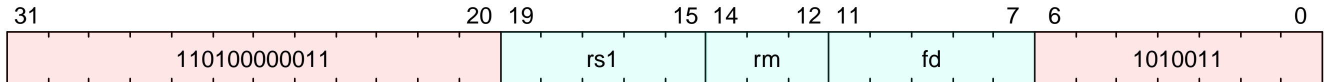
B.93. fcvt.s.lu

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.93.1. Encoding



B.93.2. Synopsis

No description available.

B.93.3. Access

M	S	U
Always	Always	Always

B.93.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.93.5. Execution

```
```

B.93.6. Exceptions

This instruction does not generate synchronous exceptions.

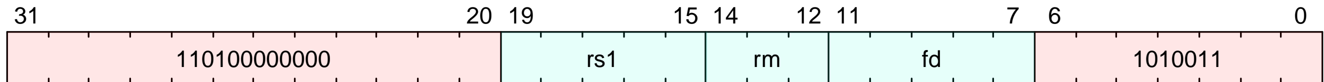
B.94. fcvt.s.w

Convert signed 32-bit integer to single-precision float

This instruction is defined by:

- F, version >= 0

B.94.1. Encoding



B.94.2. Synopsis

Converts a 32-bit signed integer in integer register *rs1* into a floating-point number in floating-point register *fd*.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

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.94.3. Access

M	S	U
Always	Always	Always

B.94.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.94.5. Execution

```
check_f_ok($encoding);
Bits<32> int_value = X[rs1];
Bits<1> sign = int_value[31];
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
if ((int_value & 32'h7fff_ffff) == 0) {
  X[fd] = (sign == 1) ? packToF32UI(1, 0x9E, 0) : 0;
} else {
  Bits<32> absA = (sign == 1) ? -int_value : int_value;
  X[fd] = softfloat_normRoundPackToF32(sign, 0x9C, absA, rounding_mode);
```

```
}
mark_f_state_dirty();
```

B.94.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

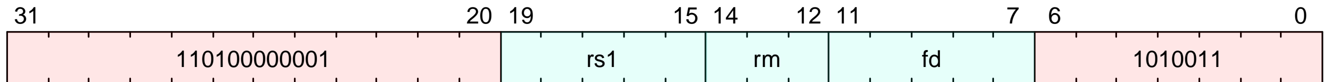
B.95. fcvt.s.wu

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.95.1. Encoding



B.95.2. Synopsis

No description available.

B.95.3. Access

M	S	U
Always	Always	Always

B.95.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.95.5. Execution

```
```

B.95.6. Exceptions

This instruction does not generate synchronous exceptions.

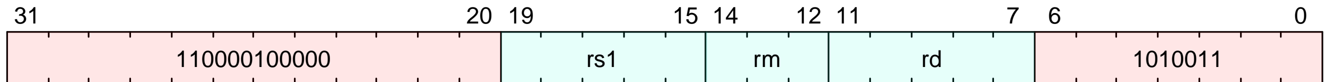
B.96. fcvt.w.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.96.1. Encoding



B.96.2. Synopsis

No description available.

B.96.3. Access

M	S	U
Always	Always	Always

B.96.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.96.5. Execution

```

```

B.96.6. Exceptions

This instruction does not generate synchronous exceptions.

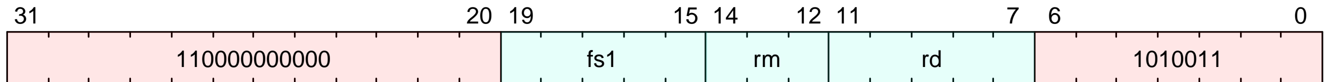
B.97. fcvt.w.s

Convert single-precision float to integer word to signed 32-bit integer.

This instruction is defined by:

- F, version >= 0

B.97.1. Encoding



B.97.2. Synopsis

Converts a floating-point number in floating-point register *fs1* to a signed 32-bit integer indicates integer register *rd*.

For XLEN >32, fcvt.w.s sign-extends the 32-bit result to the destination register width.

If the rounded result is not representable as a 32-bit signed integer, it is clipped to the nearest value and the invalid flag is set.

The range of valid inputs and behavior for invalid inputs are:

	Value
Minimum valid input (after rounding)	`-2^31`
Maximum valid input (after rounding)	`2^31 - 1`
Output for out-of-range negative input	`-2^31`
Output for <code>-∞</code>	`-2^31`
Output for out-of-range positive input	`2^31 - 1`
Output for <code>+∞</code> for <code>NaN</code>	`2^31 - 1`

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

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.97.3. Access

M	S	U
Always	Always	Always

B.97.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.97.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = f[fs1][31:0];
Bits<1> sign = sp_value[31];
Bits<8> exp = sp_value[30:23];
Bits<23> sig = sp_value[22:0];
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
if ((exp == 0xff) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  X[rd] = SP_CANONICAL_NAN;
} else {
  if (exp != 0) {
    sig = sig | 0x00800000;
  }
  Bits<64> sig64 = sig << 32;
  Bits<16> shift_dist = 0xAA - exp;
  if (0 < shift_dist) {
    sig64 = softfloat_shiftRightJam64(sig64, shift_dist);
  }
  X[rd] = softfloat_roundToI32(sign, sig64, rounding_mode);
}
```

B.97.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

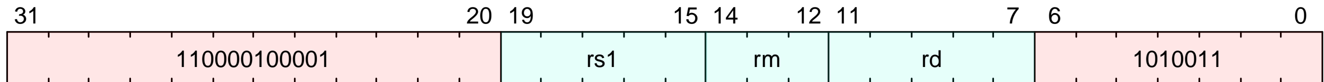
B.98. fcvt.wu.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.98.1. Encoding



B.98.2. Synopsis

No description available.

B.98.3. Access

M	S	U
Always	Always	Always

B.98.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.98.5. Execution

```
```

B.98.6. Exceptions

This instruction does not generate synchronous exceptions.

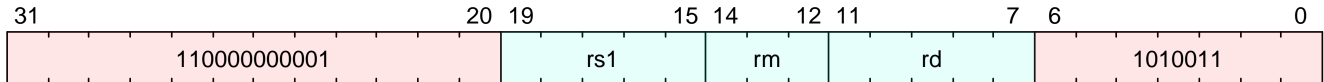
B.99. fcvt.wu.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.99.1. Encoding



B.99.2. Synopsis

No description available.

B.99.3. Access

M	S	U
Always	Always	Always

B.99.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.99.5. Execution

```

```

B.99.6. Exceptions

This instruction does not generate synchronous exceptions.

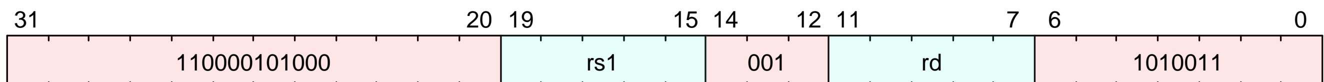
B.100. fcvtmod.w.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.100.1. Encoding



B.100.2. Synopsis

No description available.

B.100.3. Access

M	S	U
Always	Always	Always

B.100.4. Decode Variables

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

B.100.5. Execution

```
```

B.100.6. Exceptions

This instruction does not generate synchronous exceptions.

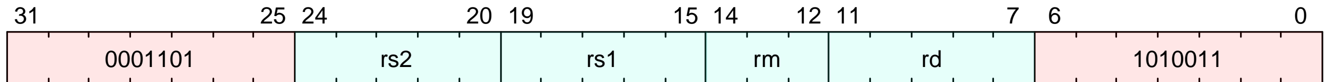
B.101. fdiv.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.101.1. Encoding



B.101.2. Synopsis

No description available.

B.101.3. Access

M	S	U
Always	Always	Always

B.101.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.101.5. Execution

```
```

B.101.6. Exceptions

This instruction does not generate synchronous exceptions.

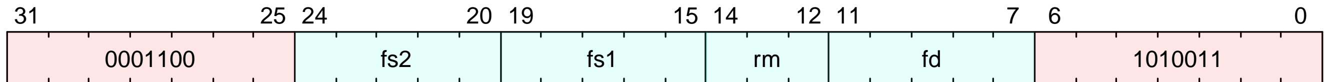
B.102. fdiv.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.102.1. Encoding



B.102.2. Synopsis

No description available.

B.102.3. Access

M	S	U
Always	Always	Always

B.102.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.102.5. Execution

```
```

B.102.6. Exceptions

This instruction does not generate synchronous exceptions.

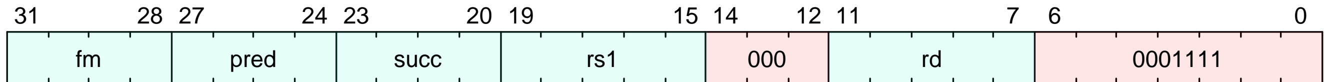
B.103. fence

Memory ordering fence

This instruction is defined by:

- I, version >= 0

B.103.1. Encoding



B.103.2. 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 31. Fence mode encoding

fm field	Mnemonic	Meaning
0000	*none*	Normal Fence
1000	TSO	With `FENCE RW,RW`: exclude write-to-read ordering; otherwise: *Reserved for future use.*
other		*Reserved for future use.*

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 32. Effective PR/PW/SR/SW in (H)S-mode

menvcfg.FIOM	`pred.PI`
	`pred.PO`
	`succ.SI`
	`succ.SO`
		→
		→
		→
		→
			effective `PR`
			effective `PW`
			effective `SR`
			effective `SW`

0	-		from encoding
1	0		from encoding
1	1		1

Table 33. Effective PR/PW/SR/SW in U-mode

menvcfg.FIOM	**senvcfg.FIOM**	`pred.PI`
		`pred.PO`
		`succ.SI`
		`succ.SO`
			→
			→
			→
			→
				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.PO`
		`succ.SI`
		`succ.SO`
			→
			→
			→
			→
				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.103.3. Access

M	S	U

Always	Always	Always

B.103.4. 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.103.5. 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] == 1;
Boolean pred_o = pred[2] == 1;
Boolean pred_r = pred[1] == 1;
Boolean pred_w = pred[0] == 1;
Boolean succ_i = succ[3] == 1;
Boolean succ_o = succ[2] == 1;
Boolean succ_r = succ[1] == 1;
Boolean succ_w = succ[0] == 1;
if (is_fence_tso) {
  fence_tso();
} else if (is_pause) {
  pause();
} else {
  if (mode() == PrivilegeMode::S) {
    if (CSR[menvcfg].FIOM == 1) {
      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) == 1) {
      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) == 1) {
      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.103.6. Exceptions

This instruction does not generate synchronous exceptions.

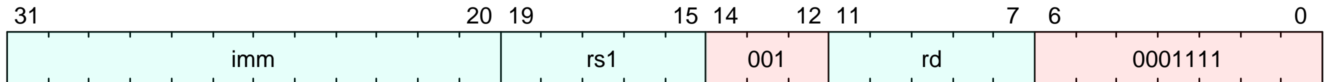
B.104. fence.i

Instruction fence

This instruction is defined by:

- Zifencei, version >= 0

B.104.1. Encoding



B.104.2. 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.104.3. Access

M	S	U
Always	Always	Always

B.104.4. Decode Variables

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

B.104.5. Execution

```
ifence();
```

B.104.6. Exceptions

This instruction does not generate synchronous exceptions.

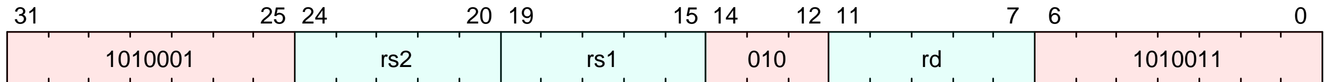
B.105. feq.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.105.1. Encoding



B.105.2. Synopsis

No description available.

B.105.3. Access

M	S	U
Always	Always	Always

B.105.4. Decode Variables

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

B.105.5. Execution

```

```

B.105.6. Exceptions

This instruction does not generate synchronous exceptions.

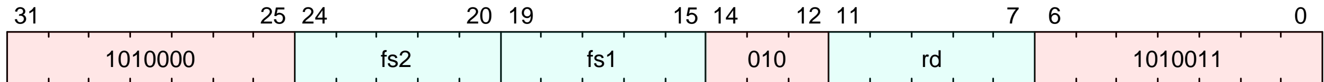
B.106. feq.s

Single-precision floating-point equal

This instruction is defined by:

- F, version >= 0

B.106.1. Encoding



B.106.2. Synopsis

Writes 1 to *rd* if *fs1* and *fs2* are equal, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a signaling NaN, the invalid flag is set.

Positive zero is considered equal to negative zero.

B.106.3. Access

M	S	U
Always	Always	Always

B.106.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.106.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs1][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[rd] = 0;
} else {
  X[rd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

B.106.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

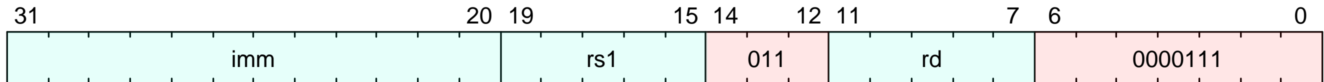
B.107. fld

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.107.1. Encoding



B.107.2. Synopsis

No description available.

B.107.3. Access

M	S	U
Always	Always	Always

B.107.4. Decode Variables

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

B.107.5. Execution

```
```

B.107.6. Exceptions

This instruction does not generate synchronous exceptions.

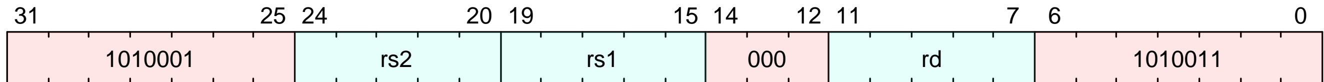
B.108. fle.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.108.1. Encoding



B.108.2. Synopsis

No description available.

B.108.3. Access

M	S	U
Always	Always	Always

B.108.4. Decode Variables

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

B.108.5. Execution

```
```

B.108.6. Exceptions

This instruction does not generate synchronous exceptions.

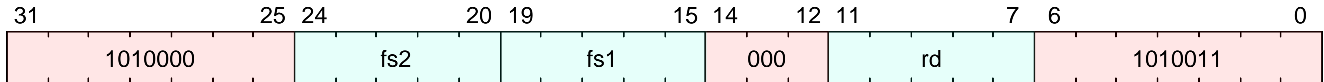
B.109. fle.s

Single-precision floating-point less than or equal

This instruction is defined by:

- F, version >= 0

B.109.1. Encoding



B.109.2. Synopsis

Writes 1 to *rd* if *fs1* is less than or equal to *fs2*, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

Positive zero and negative zero are considered equal.

B.109.3. Access

M	S	U
Always	Always	Always

B.109.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.109.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[rd] = 0;
} else {
  X[rd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

B.109.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

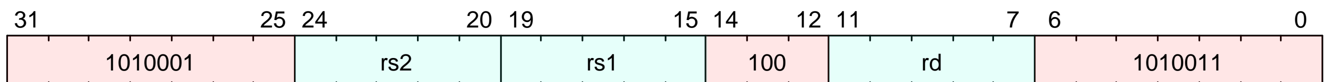
B.110. fleq.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.110.1. Encoding



B.110.2. Synopsis

No description available.

B.110.3. Access

M	S	U
Always	Always	Always

B.110.4. Decode Variables

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

B.110.5. Execution

```
```

B.110.6. Exceptions

This instruction does not generate synchronous exceptions.

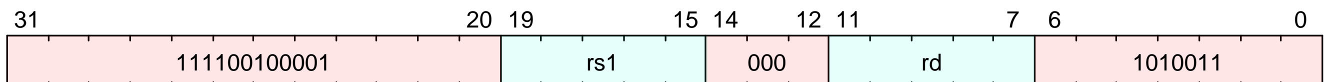
B.111. fli.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.111.1. Encoding



B.111.2. Synopsis

No description available.

B.111.3. Access

M	S	U
Always	Always	Always

B.111.4. Decode Variables

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

B.111.5. Execution

```

```

B.111.6. Exceptions

This instruction does not generate synchronous exceptions.

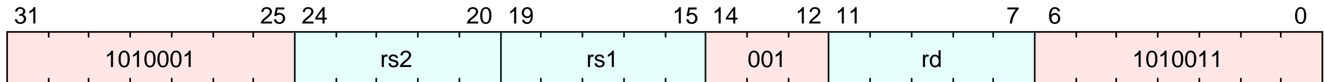
B.112. flt.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.112.1. Encoding



B.112.2. Synopsis

No description available.

B.112.3. Access

M	S	U
Always	Always	Always

B.112.4. Decode Variables

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

B.112.5. Execution

```
```

B.112.6. Exceptions

This instruction does not generate synchronous exceptions.

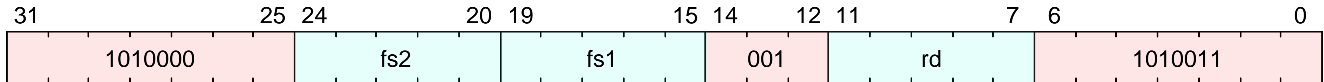
B.113. flt.s

Single-precision floating-point less than

This instruction is defined by:

- F, version >= 0

B.113.1. Encoding



B.113.2. Synopsis

Writes 1 to *rd* if *fs1* is less than *fs2*, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

B.113.3. Access

M	S	U
Always	Always	Always

B.113.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.113.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  set_fp_flag(FpFlag::NV);
  X[rd] = 0;
} else {
  Boolean sign_a = sp_value_a[31] == 1;
  Boolean sign_b = sp_value_b[31] == 1;
  Boolean a_lt_b = (sign_a != sign_b) ? (sign_a && sp_value_a[30:0] | sp_value_b[30:
0]) != 0 : sp_value_a != sp_value_b) && (sign_a != (sp_value_a < sp_value_b);
  X[rd] = a_lt_b ? 1 : 0;
}
```

B.113.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

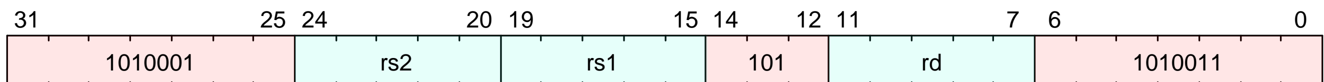
B.114. fltq.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.114.1. Encoding



B.114.2. Synopsis

No description available.

B.114.3. Access

M	S	U
Always	Always	Always

B.114.4. Decode Variables

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

B.114.5. Execution

```

```

B.114.6. Exceptions

This instruction does not generate synchronous exceptions.

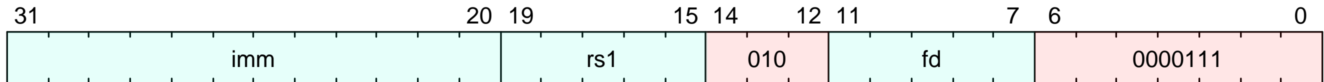
B.115. flw

Single-precision floating-point load

This instruction is defined by:

- F, version >= 0

B.115.1. Encoding



B.115.2. Synopsis

The flw instruction loads a single-precision floating-point value from memory at address *rs1* + *imm* into floating-point register *fd*.

flw does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.115.3. Access

M	S	U
Always	Always	Always

B.115.4. Decode Variables

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

B.115.5. Execution

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
Bits<32> sp_value = read_memory<32>(virtual_address, $encoding);
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.115.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

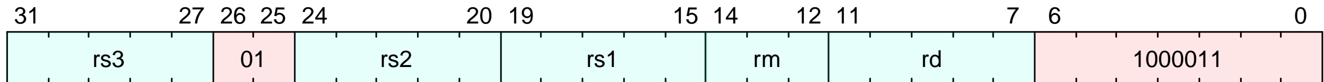
B.116. fmadd.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.116.1. Encoding



B.116.2. Synopsis

No description available.

B.116.3. Access

M	S	U
Always	Always	Always

B.116.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.116.5. Execution

```
```

B.116.6. Exceptions

This instruction does not generate synchronous exceptions.

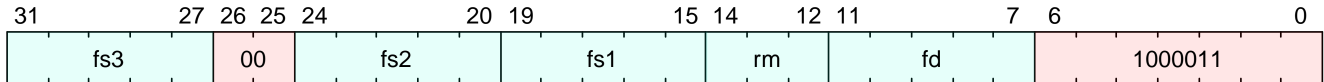
B.117. fmadd.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.117.1. Encoding



B.117.2. Synopsis

No description available.

B.117.3. Access

M	S	U
Always	Always	Always

B.117.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.117.5. Execution

```
```

B.117.6. Exceptions

This instruction does not generate synchronous exceptions.

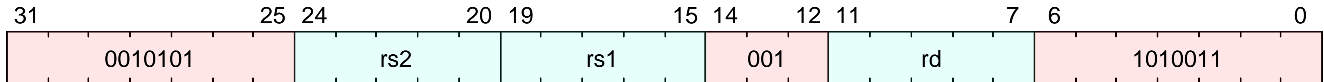
B.118. fmax.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.118.1. Encoding



B.118.2. Synopsis

No description available.

B.118.3. Access

M	S	U
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

```
```

B.118.6. Exceptions

This instruction does not generate synchronous exceptions.

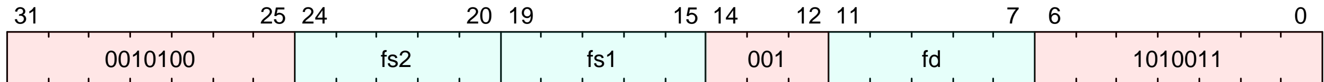
B.119. fmax.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.119.1. Encoding



B.119.2. Synopsis

No description available.

B.119.3. Access

M	S	U
Always	Always	Always

B.119.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.119.5. Execution

```
```

B.119.6. Exceptions

This instruction does not generate synchronous exceptions.

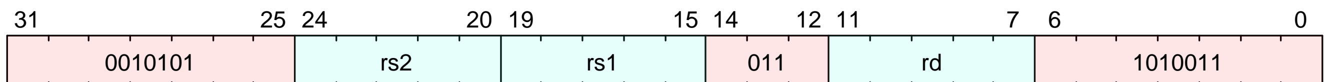
B.120. fmaxm.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.120.1. Encoding



B.120.2. Synopsis

No description available.

B.120.3. Access

M	S	U
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

```

```

B.120.6. Exceptions

This instruction does not generate synchronous exceptions.

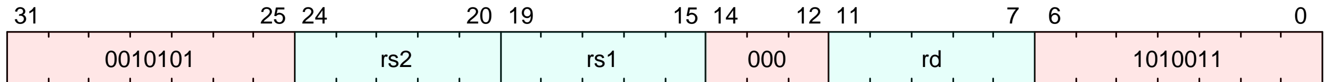
B.121. fmin.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.121.1. Encoding



B.121.2. Synopsis

No description available.

B.121.3. Access

M	S	U
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

```

```

B.121.6. Exceptions

This instruction does not generate synchronous exceptions.

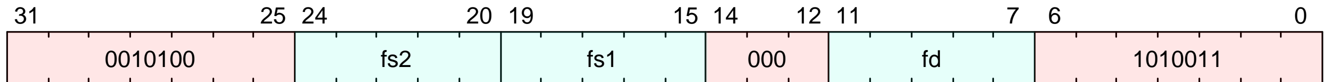
B.122. fmin.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.122.1. Encoding



B.122.2. Synopsis

No description available.

B.122.3. Access

M	S	U
Always	Always	Always

B.122.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.122.5. Execution

```
```

B.122.6. Exceptions

This instruction does not generate synchronous exceptions.

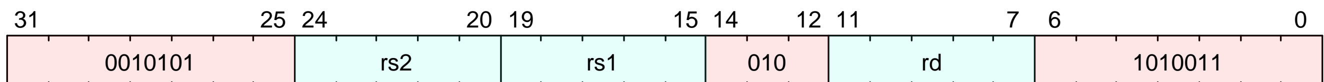
B.123. fminm.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.123.1. Encoding



B.123.2. Synopsis

No description available.

B.123.3. Access

M	S	U
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

```

```

B.123.6. Exceptions

This instruction does not generate synchronous exceptions.

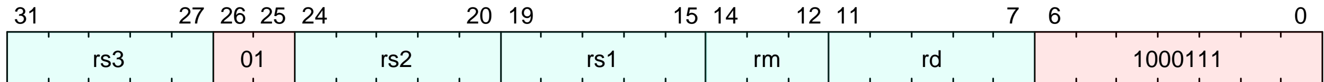
B.124. fmsub.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.124.1. Encoding



B.124.2. Synopsis

No description available.

B.124.3. Access

M	S	U
Always	Always	Always

B.124.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.124.5. Execution

```
```

B.124.6. Exceptions

This instruction does not generate synchronous exceptions.

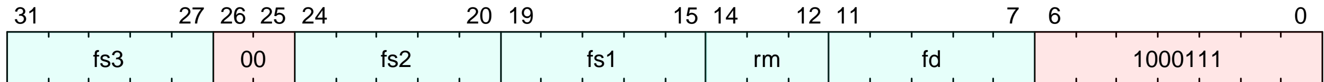
B.125. fmsub.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.125.1. Encoding



B.125.2. Synopsis

No description available.

B.125.3. Access

M	S	U
Always	Always	Always

B.125.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.125.5. Execution

```
```

B.125.6. Exceptions

This instruction does not generate synchronous exceptions.

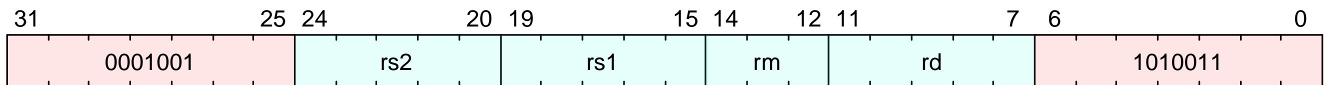
B.126. fmul.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.126.1. Encoding



B.126.2. Synopsis

No description available.

B.126.3. Access

M	S	U
Always	Always	Always

B.126.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.126.5. Execution

```
```

B.126.6. Exceptions

This instruction does not generate synchronous exceptions.

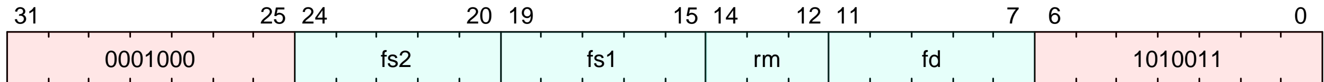
B.127. fmul.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.127.1. Encoding



B.127.2. Synopsis

No description available.

B.127.3. Access

M	S	U
Always	Always	Always

B.127.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.127.5. Execution

```
```

B.127.6. Exceptions

This instruction does not generate synchronous exceptions.

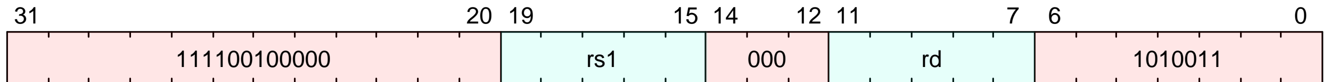
B.128. fmv.d.x

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.128.1. Encoding



B.128.2. Synopsis

No description available.

B.128.3. Access

M	S	U
Always	Always	Always

B.128.4. Decode Variables

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

B.128.5. Execution

```
```

B.128.6. Exceptions

This instruction does not generate synchronous exceptions.

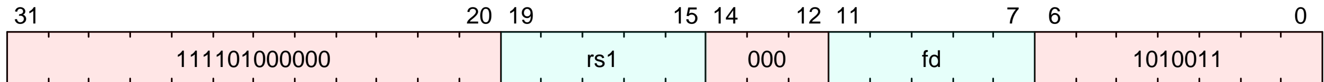
B.129. fmv.h.x

Half-precision floating-point move from integer

This instruction is defined by:

- F, version >= 0

B.129.1. Encoding



B.129.2. 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.129.3. Access

M	S	U
Always	Always	Always

B.129.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.129.5. Execution

```
check_f_ok($encoding);
Bits<16> hp_value = X[rs1][15:0];
f[fd] = nan_box<16, FLEN>(hp_value);
mark_f_state_dirty();
```

B.129.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

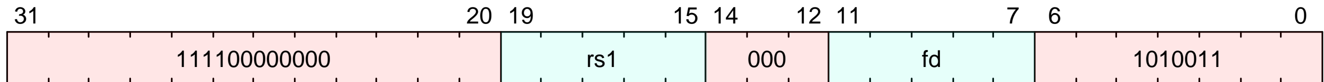
B.130. fmv.w.x

Single-precision floating-point move from integer

This instruction is defined by:

- F, version >= 0

B.130.1. Encoding



B.130.2. 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 `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

B.130.3. Access

M	S	U
Always	Always	Always

B.130.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.130.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = X[rs1][31:0];
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.130.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

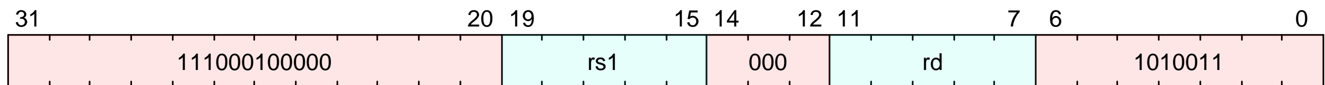
B.131. fmv.x.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.131.1. Encoding



B.131.2. Synopsis

No description available.

B.131.3. Access

M	S	U
Always	Always	Always

B.131.4. Decode Variables

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

B.131.5. Execution

```
```

B.131.6. Exceptions

This instruction does not generate synchronous exceptions.

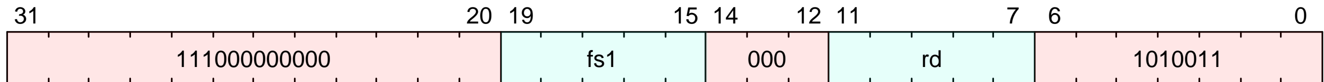
B.132. fmv.x.w

Move single-precision value from floating-point to integer register

This instruction is defined by:

- F, version >= 0

B.132.1. Encoding



B.132.2. Synopsis

Moves the single-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

B.132.3. Access

M	S	U
Always	Always	Always

B.132.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.132.5. Execution

```
check_f_ok($encoding);
X[rd] = sext(f[fs1][31:0], 32);
```

B.132.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

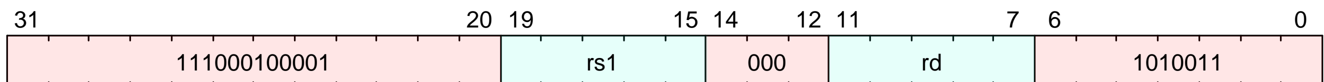
B.133. fmvh.x.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.133.1. Encoding



B.133.2. Synopsis

No description available.

B.133.3. Access

M	S	U
Always	Always	Always

B.133.4. Decode Variables

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

B.133.5. Execution

```
```

B.133.6. Exceptions

This instruction does not generate synchronous exceptions.

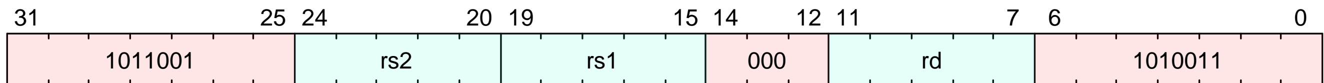
B.134. fmvp.d.x

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.134.1. Encoding



B.134.2. Synopsis

No description available.

B.134.3. Access

M	S	U
Always	Always	Always

B.134.4. Decode Variables

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

B.134.5. Execution

```
```

B.134.6. Exceptions

This instruction does not generate synchronous exceptions.

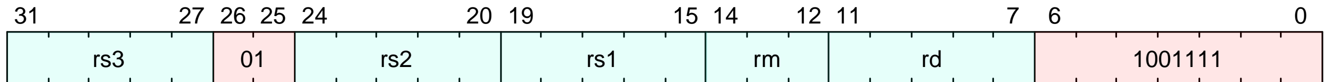
B.135. fnmadd.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.135.1. Encoding



B.135.2. Synopsis

No description available.

B.135.3. Access

M	S	U
Always	Always	Always

B.135.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.135.5. Execution

```
```

B.135.6. Exceptions

This instruction does not generate synchronous exceptions.

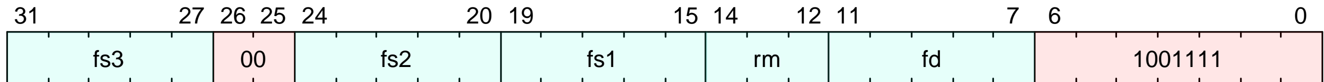
B.136. fnmadd.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.136.1. Encoding



B.136.2. Synopsis

No description available.

B.136.3. Access

M	S	U
Always	Always	Always

B.136.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.136.5. Execution

```
```

B.136.6. Exceptions

This instruction does not generate synchronous exceptions.

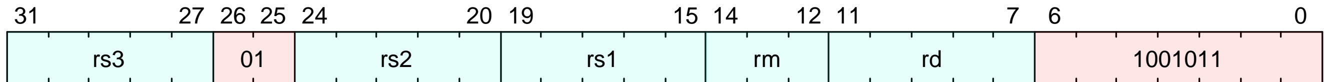
B.137. fnmsub.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.137.1. Encoding



B.137.2. Synopsis

No description available.

B.137.3. Access

M	S	U
Always	Always	Always

B.137.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.137.5. Execution

```
```

B.137.6. Exceptions

This instruction does not generate synchronous exceptions.

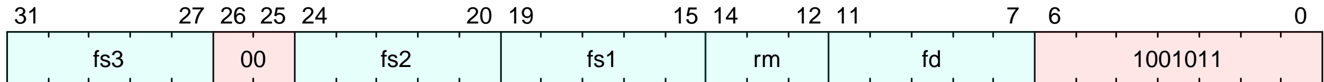
B.138. fnmsub.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.138.1. Encoding



B.138.2. Synopsis

No description available.

B.138.3. Access

M	S	U
Always	Always	Always

B.138.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.138.5. Execution

```
```

B.138.6. Exceptions

This instruction does not generate synchronous exceptions.

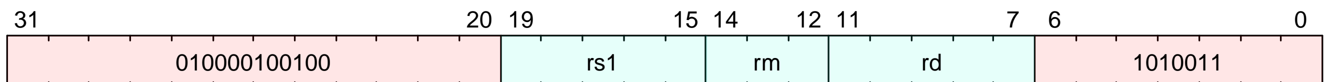
B.139. fround.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.139.1. Encoding



B.139.2. Synopsis

No description available.

B.139.3. Access

M	S	U
Always	Always	Always

B.139.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.139.5. Execution

```

```

B.139.6. Exceptions

This instruction does not generate synchronous exceptions.

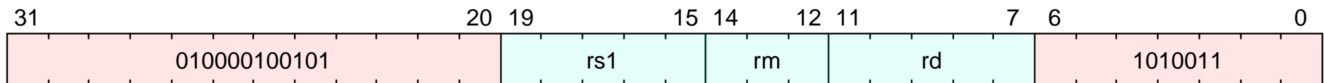
B.140. froundnx.d

No synopsis available.

This instruction is defined by:

- anyOf:
  - D, version >= 0
  - Zfa, version >= 0

B.140.1. Encoding



B.140.2. Synopsis

No description available.

B.140.3. Access

M	S	U
Always	Always	Always

B.140.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.140.5. Execution

```
```

B.140.6. Exceptions

This instruction does not generate synchronous exceptions.

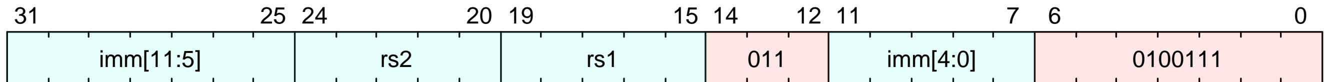
B.141. fsd

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.141.1. Encoding



B.141.2. Synopsis

No description available.

B.141.3. Access

M	S	U
Always	Always	Always

B.141.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.141.5. Execution

```

```

B.141.6. Exceptions

This instruction does not generate synchronous exceptions.

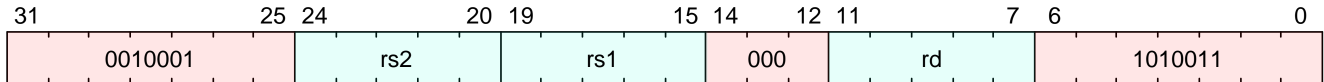
B.142. fsgnj.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.142.1. Encoding



B.142.2. Synopsis

No description available.

B.142.3. Access

M	S	U
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

```
```

B.142.6. Exceptions

This instruction does not generate synchronous exceptions.

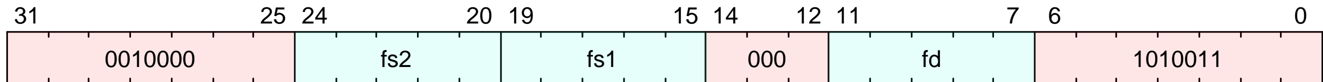
B.143. fsgnj.s

Single-precision sign inject

This instruction is defined by:

- F, version >= 0

B.143.1. Encoding



B.143.2. Synopsis

Writes *fd* with sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.143.3. Access

M	S	U
Always	Always	Always

B.143.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.143.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.143.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

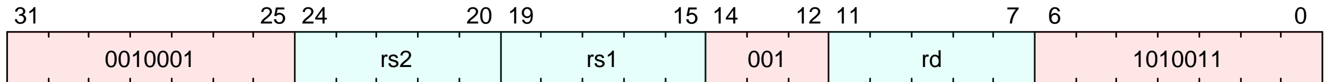
B.144. fsgnjn.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.144.1. Encoding



B.144.2. Synopsis

No description available.

B.144.3. Access

M	S	U
Always	Always	Always

B.144.4. Decode Variables

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

B.144.5. Execution

```

```

B.144.6. Exceptions

This instruction does not generate synchronous exceptions.

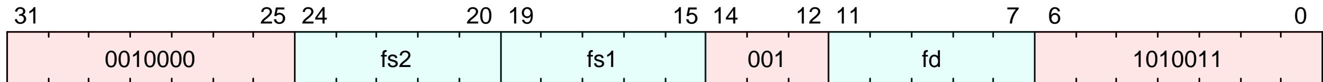
B.145. fsgnjn.s

Single-precision sign inject negate

This instruction is defined by:

- F, version >= 0

B.145.1. Encoding



B.145.2. Synopsis

Writes *fd* with the opposite of the sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.145.3. Access

M	S	U
Always	Always	Always

B.145.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.145.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = {~f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.145.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

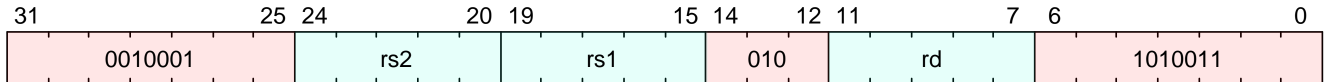
B.146. fsgnjx.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.146.1. Encoding



B.146.2. Synopsis

No description available.

B.146.3. Access

M	S	U
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

```
```

B.146.6. Exceptions

This instruction does not generate synchronous exceptions.

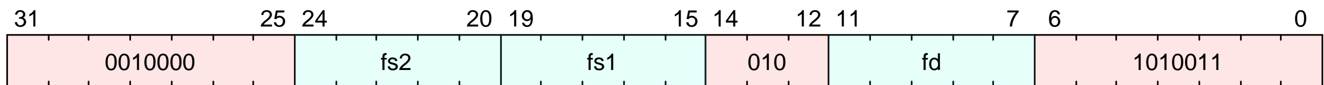
B.147. fsgnjx.s

Single-precision sign inject exclusive or

This instruction is defined by:

- F, version >= 0

B.147.1. Encoding



B.147.2. Synopsis

Writes *fd* with the xor of the sign bits of *fs2* and *fs1* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.147.3. Access

M	S	U
Always	Always	Always

B.147.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.147.5. Execution

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs1][31] ^ f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.147.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

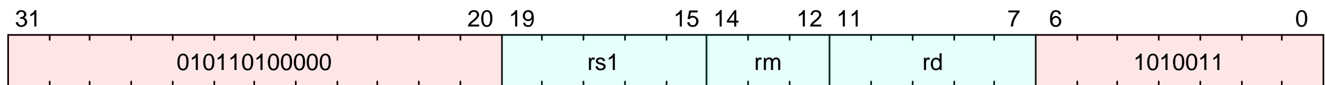
B.148. fsqrt.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.148.1. Encoding



B.148.2. Synopsis

No description available.

B.148.3. Access

M	S	U
Always	Always	Always

B.148.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.148.5. Execution

```
```

B.148.6. Exceptions

This instruction does not generate synchronous exceptions.

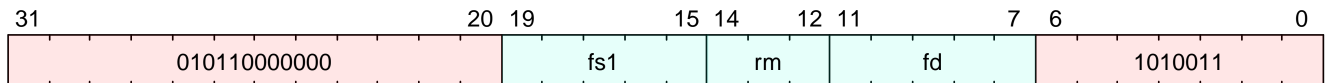
B.149. fsqrt.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.149.1. Encoding



B.149.2. Synopsis

No description available.

B.149.3. Access

M	S	U
Always	Always	Always

B.149.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.149.5. Execution

```
```

B.149.6. Exceptions

This instruction does not generate synchronous exceptions.

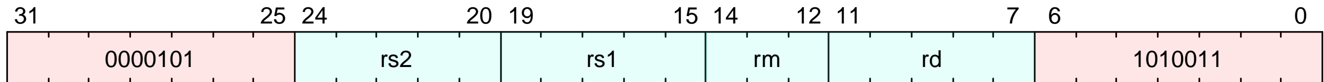
B.150. fsub.d

No synopsis available.

This instruction is defined by:

- D, version >= 0

B.150.1. Encoding



B.150.2. Synopsis

No description available.

B.150.3. Access

M	S	U
Always	Always	Always

B.150.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

B.150.5. Execution

```
```

B.150.6. Exceptions

This instruction does not generate synchronous exceptions.

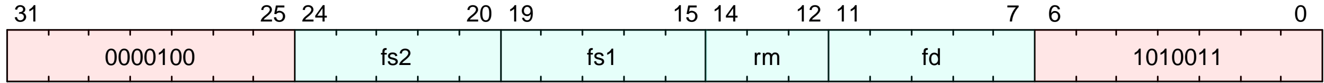
B.151. fsub.s

No synopsis available.

This instruction is defined by:

- F, version >= 0

B.151.1. Encoding



B.151.2. Synopsis

No description available.

B.151.3. Access

M	S	U
Always	Always	Always

B.151.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.151.5. Execution

```
```

B.151.6. Exceptions

This instruction does not generate synchronous exceptions.

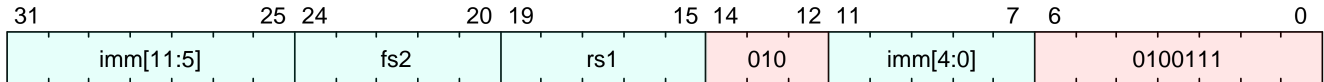
B.152. fsw

Single-precision floating-point store

This instruction is defined by:

- F, version >= 0

B.152.1. Encoding



B.152.2. Synopsis

The fsw instruction stores a single-precision floating-point value in *fs2* to memory at address *rs1* + *imm*.

fsw does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.152.3. Access

M	S	U
Always	Always	Always

B.152.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.152.5. Execution

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

B.152.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

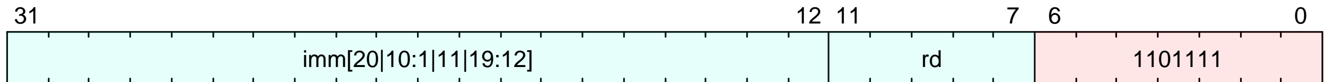
B.153. jal

Jump and link

This instruction is defined by:

- I, version >= 0

B.153.1. Encoding



B.153.2. Synopsis

Jump to a PC-relative offset and store the return address in rd.

B.153.3. Access

M	S	U
Always	Always	Always

B.153.4. 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.153.5. Execution

```
XReg retrun_addr = $pc + 4;
jump_halfword($pc + imm);
X[rd] = retrun_addr;
```

B.153.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

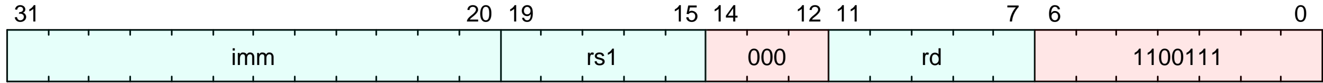
B.154. jalr

Jump and link register

This instruction is defined by:

- I, version >= 0

B.154.1. Encoding



B.154.2. Synopsis

Jump to an address formed by adding rs1 to a signed offset, and store the return address in rd.

B.154.3. Access

M	S	U
Always	Always	Always

B.154.4. Decode Variables

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

B.154.5. Execution

```
XReg returnaddr;
returnaddr = $pc + 4;
jump(X[rs1] + imm);
X[rd] = returnaddr;
```

B.154.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

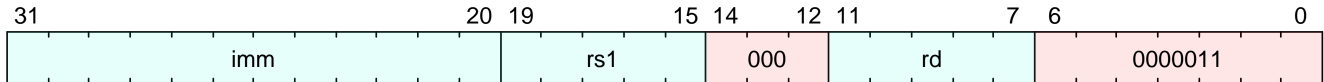
B.155. lb

Load byte

This instruction is defined by:

- I, version >= 0

B.155.1. Encoding



B.155.2. 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.155.3. Access

M	S	U
Always	Always	Always

B.155.4. Decode Variables

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

B.155.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<8>(virtual_address, $encoding), 8);
```

B.155.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

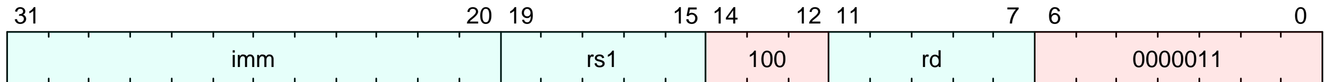
B.156. lbu

Load byte unsigned

This instruction is defined by:

- I, version >= 0

B.156.1. Encoding



B.156.2. 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.156.3. Access

M	S	U
Always	Always	Always

B.156.4. Decode Variables

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

B.156.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<8>(virtual_address, $encoding);
```

B.156.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

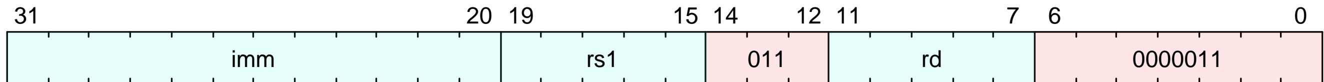
B.157. ld

Load doubleword

This instruction is defined by:

- I, version >= 0

B.157.1. Encoding



B.157.2. Synopsis

Load 64 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset.

B.157.3. Access

M	S	U
Always	Always	Always

B.157.4. Decode Variables

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

B.157.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<64>(virtual_address, $encoding);
```

B.157.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

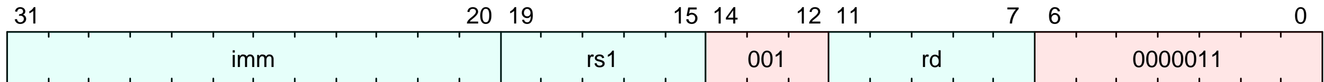
B.158. lh

Load halfword

This instruction is defined by:

- I, version >= 0

B.158.1. Encoding



B.158.2. 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.158.3. Access

M	S	U
Always	Always	Always

B.158.4. Decode Variables

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

B.158.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<16>(virtual_address, $encoding), 16);
```

B.158.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

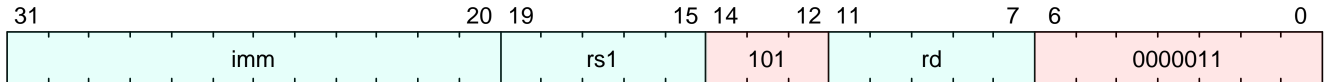
B.159. lhu

Load halfword unsigned

This instruction is defined by:

- I, version >= 0

B.159.1. Encoding



B.159.2. 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.159.3. Access

M	S	U
Always	Always	Always

B.159.4. Decode Variables

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

B.159.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<16>(virtual_address, $encoding);
```

B.159.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

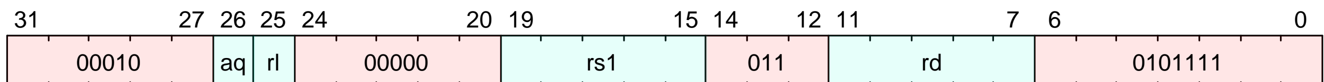
B.160. lr.d

Load reserved doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zalrsc, version >= 0

B.160.1. Encoding



B.160.2. 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.160.3. Access

M	S	U
Always	Always	Always

B.160.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.160.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(),
virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), 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, $encoding);
```

B.160.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

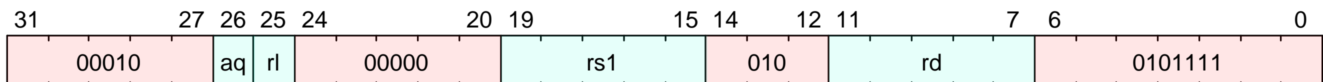
B.161. lr.w

Load reserved word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zalrsc, version >= 0

B.161.1. Encoding



B.161.2. 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.161.3. Access

M	S	U
Always	Always	Always

B.161.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.161.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(),
virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), 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, $encoding);
if (xlen() == 64) {
  X[rd] = load_value;
} else {
  X[rd] = sext(load_value[31:0], 32);
}
```

B.161.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

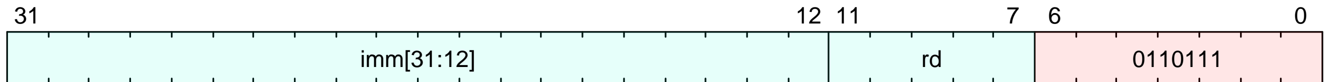
B.162. lui

Load upper immediate

This instruction is defined by:

- I, version >= 0

B.162.1. Encoding



B.162.2. Synopsis

Load the zero-extended imm into rd.

B.162.3. Access

M	S	U
Always	Always	Always

B.162.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.162.5. Execution

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

B.162.6. Exceptions

This instruction does not generate synchronous exceptions.

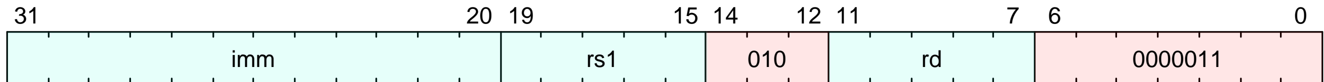
B.163. lw

Load word

This instruction is defined by:

- I, version >= 0

B.163.1. Encoding



B.163.2. 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.163.3. Access

M	S	U
Always	Always	Always

B.163.4. Decode Variables

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

B.163.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address, $encoding);
```

B.163.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

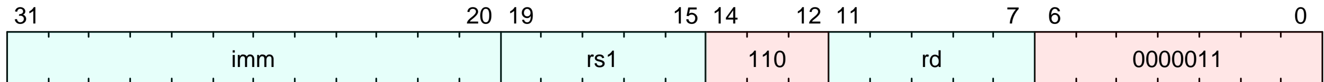
B.164. lwu

Load word unsigned

This instruction is defined by:

- I, version >= 0

B.164.1. Encoding



B.164.2. 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.164.3. Access

M	S	U
Always	Always	Always

B.164.4. Decode Variables

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

B.164.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address, $encoding);
```

B.164.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

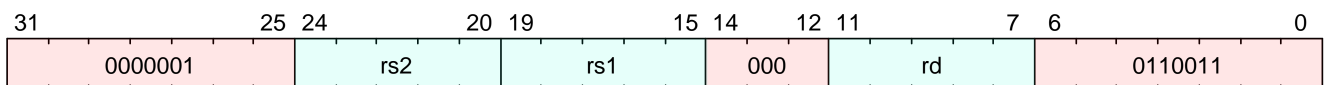
B.165. mul

Signed multiply

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.165.1. Encoding



B.165.2. 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.165.3. Access

M	S	U
Always	Always	Always

B.165.4. Decode Variables

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

B.165.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
```

```
X[rd] = (src1 * src2)[XLEN - 1:0];
```

B.165.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

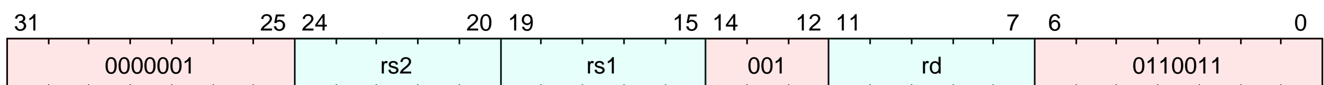
B.166. mulh

Signed multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.166.1. Encoding



B.166.2. 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.166.3. Access

M	S	U
Always	Always	Always

B.166.4. Decode Variables

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

B.166.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
Bits<1> rs1_sign_bit = X[rs1][xlen() - 1];
Bits<XLEN * 2> src1 = {{xlen(){rs1_sign_bit}}, X[rs1]};
Bits<1> rs2_sign_bit = X[rs2][xlen() - 1];
Bits<XLEN * 2> src2 = {{xlen(){rs2_sign_bit}}, X[rs2]};
X[rd] = (src1 * src2)[(xlen() * 8'd2) - 1:xlen()];
```

B.166.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

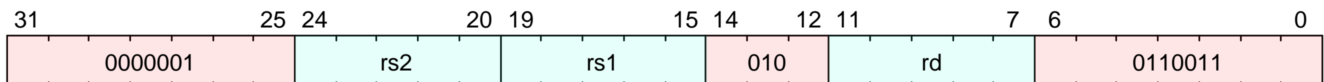
B.167. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.167.1. Encoding



B.167.2. 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.167.3. Access

M	S	U
Always	Always	Always

B.167.4. Decode Variables

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

B.167.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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];
```

B.167.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

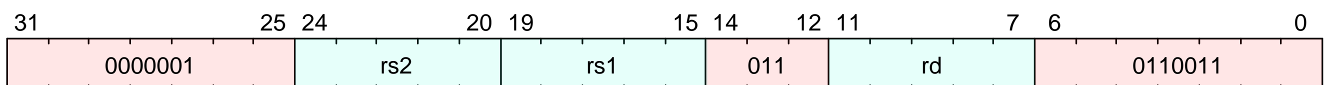
B.168. mulhu

Unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.168.1. Encoding



B.168.2. 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.168.3. Access

M	S	U
Always	Always	Always

B.168.4. Decode Variables

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

B.168.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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];
```

B.168.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

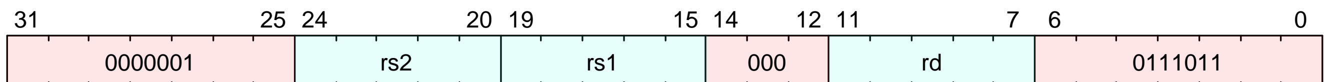
B.169. mulw

Signed 32-bit multiply

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.169.1. Encoding



B.169.2. 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.169.3. Access

M	S	U
Always	Always	Always

B.169.4. Decode Variables

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

B.169.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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};
```

B.169.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

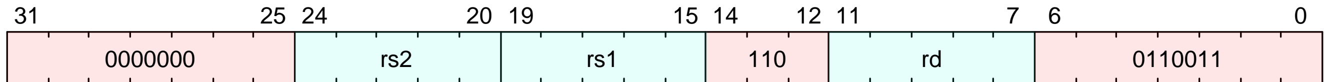
B.170. or

Or

This instruction is defined by:

- I, version >= 0

B.170.1. Encoding



B.170.2. Synopsis

Or rs1 with rs2, and store the result in rd

B.170.3. Access

M	S	U
Always	Always	Always

B.170.4. Decode Variables

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

B.170.5. Execution

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

B.170.6. Exceptions

This instruction does not generate synchronous exceptions.

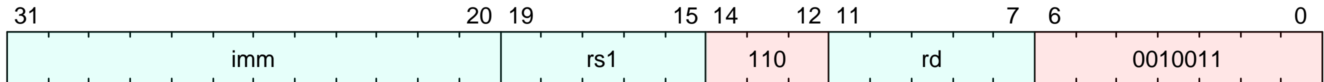
B.171. ori

Or immediate

This instruction is defined by:

- I, version >= 0

B.171.1. Encoding



B.171.2. Synopsis

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

B.171.3. Access

M	S	U
Always	Always	Always

B.171.4. Decode Variables

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

B.171.5. Execution

```
if (implemented?(ExtensionName::Zicbop)) {
  if (rd == 0) {
    if (imm[4:0] == 0) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_instruction(offset);
    } else if (imm[4:0] == 1) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_read(offset);
    } else if (imm[4:0] == 3) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_write(offset);
    }
  }
}
X[rd] = X[rs1] | imm;
```

B.171.6. Exceptions

This instruction does not generate synchronous exceptions.

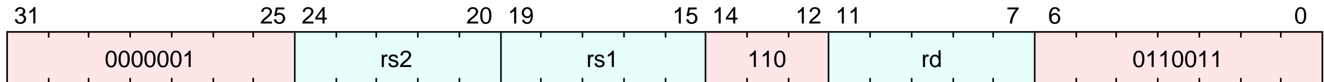
B.172. rem

Signed remainder

This instruction is defined by:

- M, version >= 0

B.172.1. Encoding



B.172.2. 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.172.3. Access

M	S	U
Always	Always	Always

B.172.4. Decode Variables

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

B.172.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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);
}
```

B.172.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

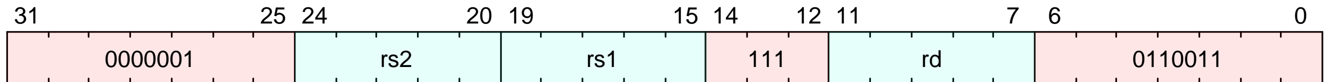
B.173. remu

Unsigned remainder

This instruction is defined by:

- M, version >= 0

B.173.1. Encoding



B.173.2. Synopsis

Calculate the remainder of unsigned division of rs1 by rs2, and store the result in rd.

B.173.3. Access

M	S	U
Always	Always	Always

B.173.4. Decode Variables

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

B.173.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = src1;
} else {
  X[rd] = src1 % src2;
}
```

B.173.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

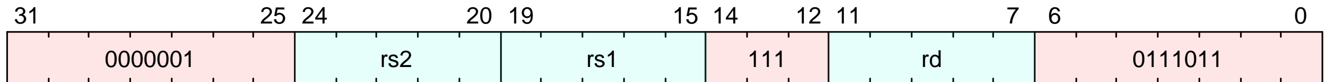
B.174. remuw

Unsigned 32-bit remainder

This instruction is defined by:

- M, version >= 0

B.174.1. Encoding



B.174.2. 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.174.3. Access

M	S	U
Always	Always	Always

B.174.4. Decode Variables

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

B.174.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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};
}
```

B.174.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

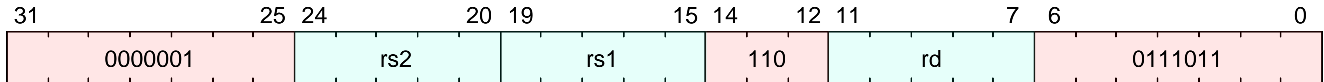
B.175. remw

Signed 32-bit remainder

This instruction is defined by:

- M, version >= 0

B.175.1. Encoding



B.175.2. 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.175.3. Access

M	S	U
Always	Always	Always

B.175.4. Decode Variables

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

B.175.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $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};
}
```

B.175.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

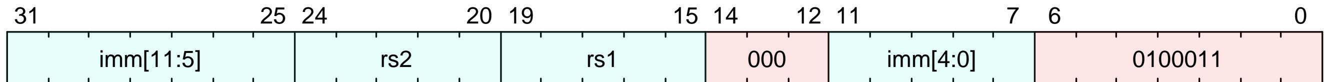
B.176. sb

Store byte

This instruction is defined by:

- I, version >= 0

B.176.1. Encoding



B.176.2. Synopsis

Store 8 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.176.3. Access

M	S	U
Always	Always	Always

B.176.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.176.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<8>(virtual_address, X[rs2][7:0], $encoding);
```

B.176.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

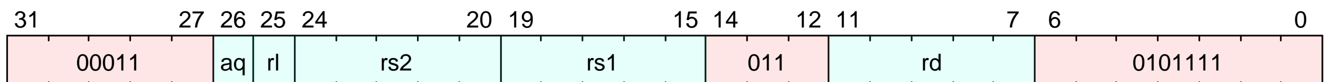
B.177. sc.d

Store conditional doubleword

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zalrsc, version >= 0

B.177.1. Encoding



B.177.2. 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.177.3. Access

M	S	U
Always	Always	Always

B.177.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.177.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(),
virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), 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, $encoding);
```

```
X[rd] = success ? 0 : 1;
```

B.177.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

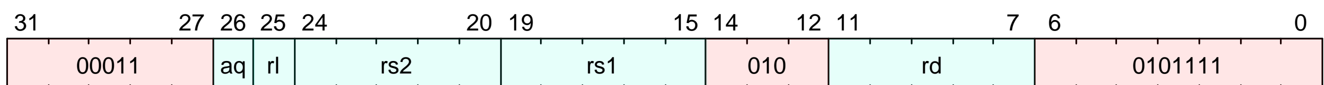
B.178. sc.w

Store conditional word

This instruction is defined by:

- anyOf:
  - A, version >= 0
  - Zalrsc, version >= 0

B.178.1. Encoding



B.178.2. 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.178.3. Access

M	S	U
Always	Always	Always

B.178.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.178.5. Execution

```
if (implemented?(ExtensionName::A) && (CSR[misa].A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(),
virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), 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, $encoding);
X[rd] = success ? 0 : 1;
```

B.178.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

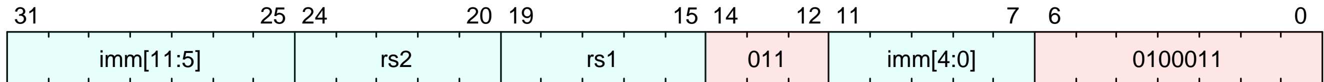
B.179. sd

Store doubleword

This instruction is defined by:

- I, version >= 0

B.179.1. Encoding



B.179.2. Synopsis

Store 64 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.179.3. Access

M	S	U
Always	Always	Always

B.179.4. 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.179.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.179.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

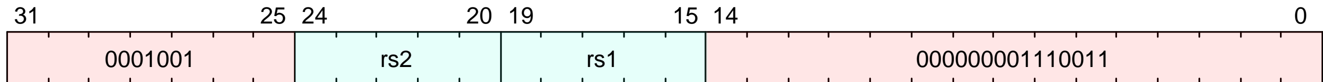
B.180. sfence.vma

Supervisor memory-management fence

This instruction is defined by:

- S, version >= 0

B.180.1. Encoding



B.180.2. 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 shootdown.

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*≠`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*≠`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*≠`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 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*≠`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 ASIDLEN<ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN 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 shootdown. However, as OSes have evolved to significantly reduce the scope of TLB shootdowns 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.180.3. Access

M	S	U
Always	Always	Never

B.180.4. Decode Variables

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

B.180.5. Execution

```
XReg vaddr = X[rs1];
Bits<16> asid = X[rs2][ASID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
```

```
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) || (mode() == PrivilegeMode
::VS) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && CSR[hstatus].VTVM == 1 && mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::VirtualInstruction, mode(), $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, mode(), $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;
  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.180.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

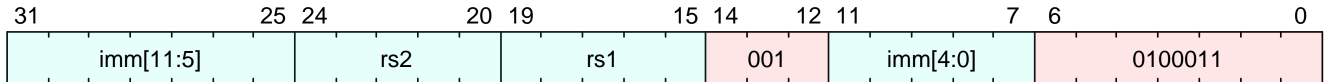
B.181. sh

Store halfword

This instruction is defined by:

- I, version >= 0

B.181.1. Encoding



B.181.2. Synopsis

Store 16 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.181.3. Access

M	S	U
Always	Always	Always

B.181.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.181.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<16>(virtual_address, X[rs2][15:0], $encoding);
```

B.181.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

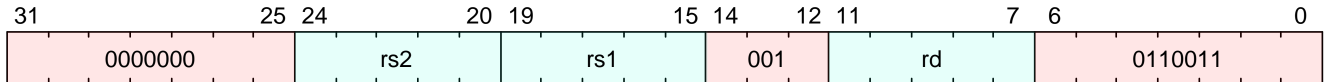
B.182. sll

Shift left logical

This instruction is defined by:

- I, version >= 0

B.182.1. Encoding



B.182.2. Synopsis

Shift the value in `rs1` left by the value in the lower 6 bits of `rs2`, and store the result in `rd`.

B.182.3. Access

M	S	U
Always	Always	Always

B.182.4. Decode Variables

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

B.182.5. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] << X[rs2][5:0];
} else {
  X[rd] = X[rs1] << X[rs2][4:0];
}
```

B.182.6. Exceptions

This instruction does not generate synchronous exceptions.

B.183. slli

Shift left logical immediate

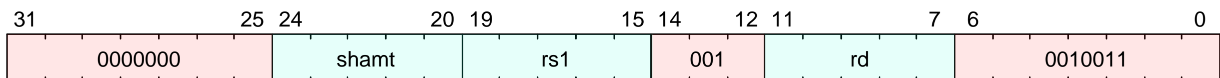
This instruction is defined by:

- I, version >= 0

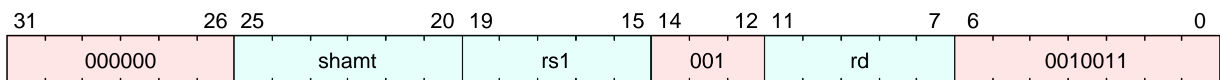
B.183.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.183.2. Synopsis

Shift the value in rs1 left by shamt, and store the result in rd

B.183.3. Access

M	S	U
Always	Always	Always

B.183.4. 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.183.5. Execution

```
X[rd] = X[rs1] << shamt;
```

B.183.6. Exceptions

This instruction does not generate synchronous exceptions.

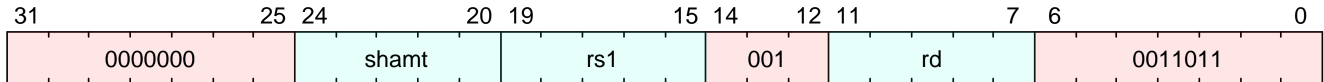
B.184. slliw

Shift left logical immediate word

This instruction is defined by:

- I, version >= 0

B.184.1. Encoding



B.184.2. Synopsis

Shift the 32-bit value in rs1 left by shamt, and store the sign-extended result in rd

B.184.3. Access

M	S	U
Always	Always	Always

B.184.4. Decode Variables

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

B.184.5. Execution

```
X[rd] = sext(X[rs1] << shamt, 31);
```

B.184.6. Exceptions

This instruction does not generate synchronous exceptions.

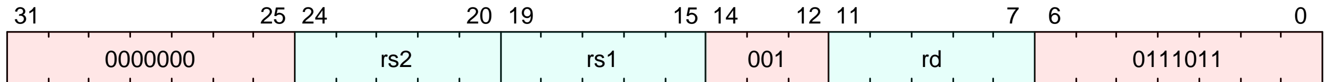
B.185. sllw

Shift left logical word

This instruction is defined by:

- I, version >= 0

B.185.1. Encoding



B.185.2. 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.185.3. Access

M	S	U
Always	Always	Always

B.185.4. Decode Variables

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

B.185.5. Execution

```
X[rd] = sext(X[rs1] << X[rs2][4:0], 31);
```

B.185.6. Exceptions

This instruction does not generate synchronous exceptions.

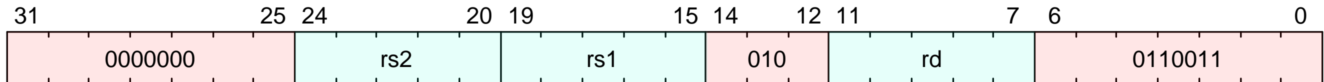
B.186. slt

Set on less than

This instruction is defined by:

- I, version >= 0

B.186.1. Encoding



B.186.2. 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.186.3. Access

M	S	U
Always	Always	Always

B.186.4. Decode Variables

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

B.186.5. Execution

```
XReg src1 = X[rs1];
XReg src2 = X[rs2];
X[rd] = ($signed(src1) < $signed(src2)) ? '1 : '0;
```

B.186.6. Exceptions

This instruction does not generate synchronous exceptions.

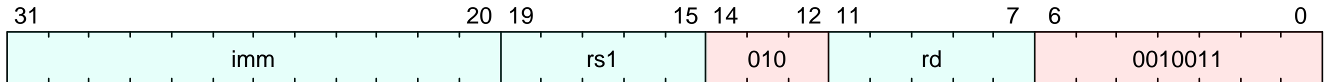
B.187. slti

Set on less than immediate

This instruction is defined by:

- I, version >= 0

B.187.1. Encoding



B.187.2. 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.187.3. Access

M	S	U
Always	Always	Always

B.187.4. Decode Variables

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

B.187.5. Execution

```
X[rd] = ($signed(X[rs1]) < $signed(imm)) ? '1 : '0;
```

B.187.6. Exceptions

This instruction does not generate synchronous exceptions.

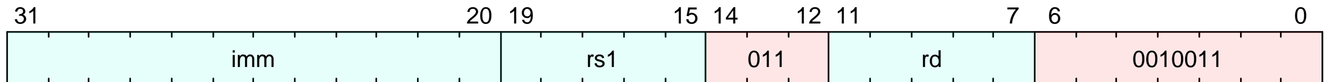
B.188. sltiu

Set on less than immediate unsigned

This instruction is defined by:

- I, version >= 0

B.188.1. Encoding



B.188.2. 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.188.3. Access

M	S	U
Always	Always	Always

B.188.4. Decode Variables

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

B.188.5. Execution

```
X[rd] = (X[rs1] < imm) ? 1 : 0;
```

B.188.6. Exceptions

This instruction does not generate synchronous exceptions.

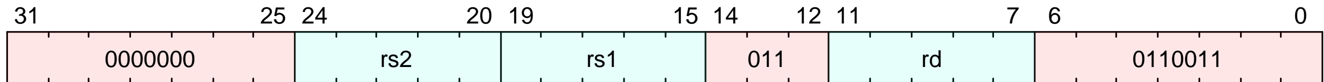
B.189. sltu

Set on less than unsigned

This instruction is defined by:

- I, version >= 0

B.189.1. Encoding



B.189.2. 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.189.3. Access

M	S	U
Always	Always	Always

B.189.4. Decode Variables

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

B.189.5. Execution

```
X[rd] = (X[rs1] < X[rs2]) ? 1 : 0;
```

B.189.6. Exceptions

This instruction does not generate synchronous exceptions.

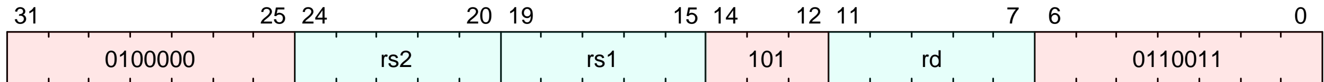
B.190. sra

Shift right arithmetic

This instruction is defined by:

- I, version >= 0

B.190.1. Encoding



B.190.2. 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.190.3. Access

M	S	U
Always	Always	Always

B.190.4. Decode Variables

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

B.190.5. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] >>> X[rs2][5:0];
} else {
  X[rd] = X[rs1] >>> X[rs2][4:0];
}
```

B.190.6. Exceptions

This instruction does not generate synchronous exceptions.

B.191. srai

Shift right arithmetic immediate

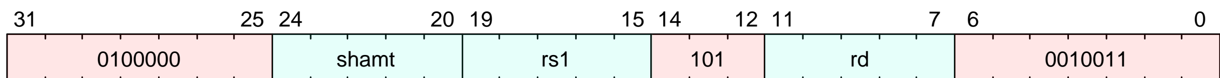
This instruction is defined by:

- I, version >= 0

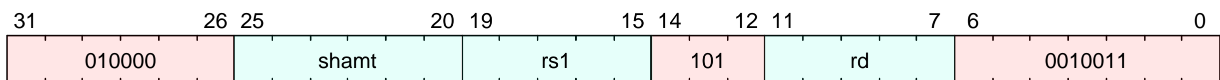
B.191.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.191.2. 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.191.3. Access

M	S	U
Always	Always	Always

B.191.4. 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.191.5. Execution

```
X[rd] = X[rs1] >>> shamt;
```

B.191.6. Exceptions

This instruction does not generate synchronous exceptions.

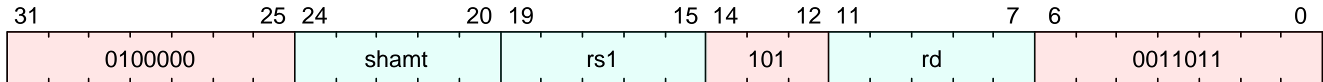
B.192. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

- I, version >= 0

B.192.1. Encoding



B.192.2. 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.192.3. Access

M	S	U
Always	Always	Always

B.192.4. Decode Variables

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

B.192.5. Execution

```
XReg operand = sext(X[rs1], 31);
X[rd] = sext(operand >>> shamt, 31);
```

B.192.6. Exceptions

This instruction does not generate synchronous exceptions.

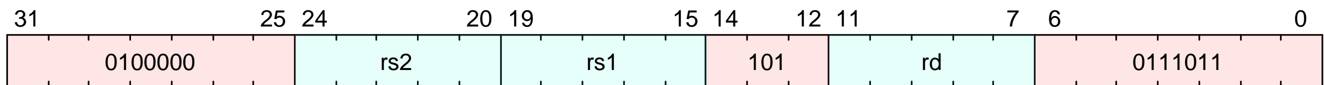
B.193. sraw

Shift right arithmetic word

This instruction is defined by:

- I, version >= 0

B.193.1. Encoding



B.193.2. 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.193.3. Access

M	S	U
Always	Always	Always

B.193.4. Decode Variables

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

B.193.5. Execution

```
XReg operand1 = sext(X[rs1], 31);
X[rd] = sext(operand1 >>> X[rs2][4:0], 31);
```

B.193.6. Exceptions

This instruction does not generate synchronous exceptions.

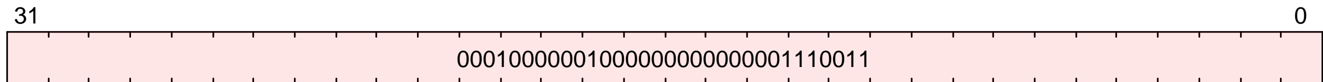
B.194. sret

Supervisor Exception Return

This instruction is defined by:

- S, version >= 0

B.194.1. Encoding



B.194.2. 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 privlege 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 privlege mode according to the table below, and then jumps to the address in sepc.

Table 34. 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 privlege mode is VS-mode

sret sets vsstatus.SPP = 0, vsstatus.SIE = `vstatus.SPIE`, and vsstatus.SPIE = 1, changes the privlege mode according to the table below, and then jumps to the address in vsepc.

Table 35. 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.194.3. Access

M	S	U
Always	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.194.4. Decode Variables

```
```

B.194.5. Execution

```
if (implemented?(ExtensionName::H)) {
  if (CSR[mstatus].TSR == 1'b0 && CSR[hstatus].VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b0 && CSR[hstatus].VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b1 && CSR[hstatus].VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (CSR[mstatus].TSR == 1'b1 && CSR[hstatus].VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
```

```
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  }
} else {
  if (mode() != PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $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, mode(), $encoding);
  }
  CSR[vsstatus].SPP = 0;
  CSR[vsstatus].SIE = CSR[vsstatus].SPIE;
  CSR[vsstatus].SPIE = 1;
  $pc = $bits(CSR[vsepc]);
}
```

B.194.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

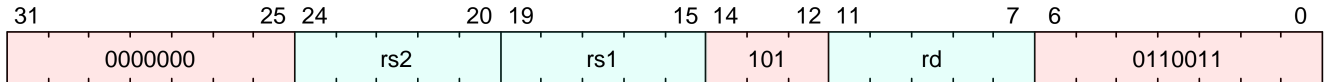
B.195. srl

Shift right logical

This instruction is defined by:

- I, version >= 0

B.195.1. Encoding



B.195.2. Synopsis

Logical shift the value in `rs1` right by the value in the lower bits of `rs2`, and store the result in `rd`.

B.195.3. Access

M	S	U
Always	Always	Always

B.195.4. Decode Variables

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

B.195.5. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] >> X[rs2][5:0];
} else {
  X[rd] = X[rs1] >> X[rs2][4:0];
}
```

B.195.6. Exceptions

This instruction does not generate synchronous exceptions.

B.196. srli

Shift right logical immediate

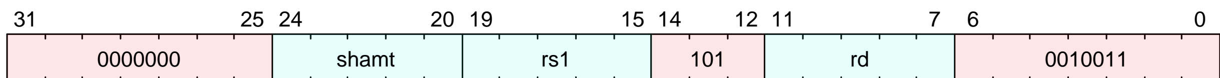
This instruction is defined by:

- I, version >= 0

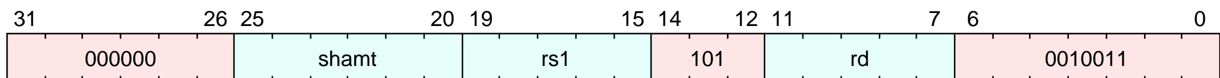
B.196.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.196.2. Synopsis

Shift the value in rs1 right by shamt, and store the result in rd

B.196.3. Access

M	S	U
Always	Always	Always

B.196.4. 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.196.5. Execution

```
X[rd] = X[rs1] >> shamt;
```

B.196.6. Exceptions

This instruction does not generate synchronous exceptions.

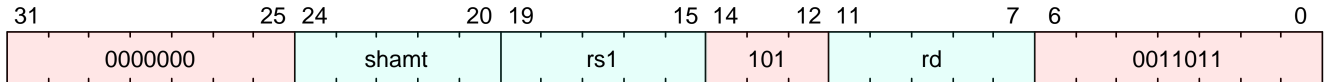
B.197. srliw

Shift right logical immediate word

This instruction is defined by:

- I, version >= 0

B.197.1. Encoding



B.197.2. Synopsis

Shift the 32-bit value in rs1 right by shamt, and store the sign-extended result in rd

B.197.3. Access

M	S	U
Always	Always	Always

B.197.4. Decode Variables

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

B.197.5. Execution

```
XReg operand = X[rs1][31:0];
X[rd] = sext(operand >> shamt, 31);
```

B.197.6. Exceptions

This instruction does not generate synchronous exceptions.

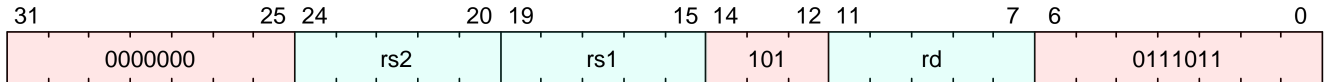
B.198. srlw

Shift right logical word

This instruction is defined by:

- I, version >= 0

B.198.1. Encoding



B.198.2. 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.198.3. Access

M	S	U
Always	Always	Always

B.198.4. Decode Variables

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

B.198.5. Execution

```
X[rd] = sext(X[rs1][31:0] >> X[rs2][4:0], 31);
```

B.198.6. Exceptions

This instruction does not generate synchronous exceptions.

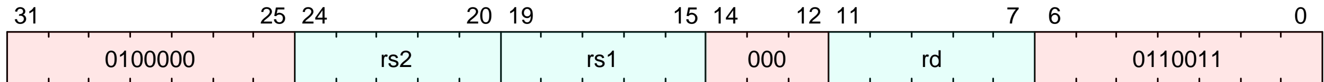
B.199. sub

Subtract

This instruction is defined by:

- I, version >= 0

B.199.1. Encoding



B.199.2. Synopsis

Subtract the value in rs2 from rs1, and store the result in rd

B.199.3. Access

M	S	U
Always	Always	Always

B.199.4. Decode Variables

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

B.199.5. Execution

```
XReg t0 = X[rs1];
XReg t1 = X[rs2];
X[rd] = t0 - t1;
```

B.199.6. Exceptions

This instruction does not generate synchronous exceptions.

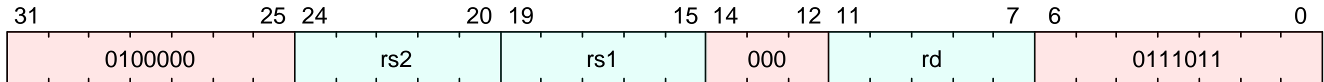
B.200. subw

Subtract word

This instruction is defined by:

- I, version >= 0

B.200.1. Encoding



B.200.2. Synopsis

Subtract the 32-bit values in rs2 from rs1, and store the sign-extended result in rd

B.200.3. Access

M	S	U
Always	Always	Always

B.200.4. Decode Variables

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

B.200.5. Execution

```
Bits<32> t0 = X[rs1][31:0];
Bits<32> t1 = X[rs2][31:0];
X[rd] = sext(t0 - t1, 31);
```

B.200.6. Exceptions

This instruction does not generate synchronous exceptions.

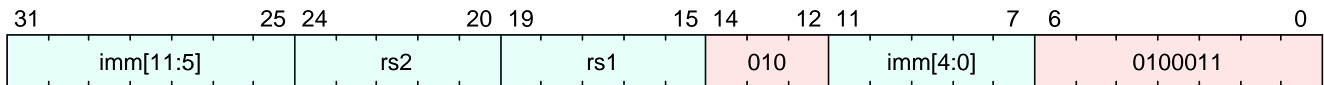
B.201. sw

Store word

This instruction is defined by:

- I, version >= 0

B.201.1. Encoding



B.201.2. Synopsis

Store 32 bits of data from register rs2 to an address formed by adding rs1 to a signed offset.

B.201.3. Access

M	S	U
Always	Always	Always

B.201.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.201.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.201.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

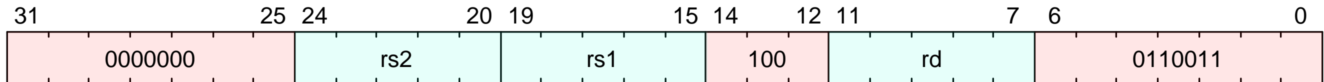
B.202. xor

Exclusive Or

This instruction is defined by:

- I, version >= 0

B.202.1. Encoding



B.202.2. Synopsis

Exclusive or rs1 with rs2, and store the result in rd

B.202.3. Access

M	S	U
Always	Always	Always

B.202.4. Decode Variables

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

B.202.5. Execution

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

B.202.6. Exceptions

This instruction does not generate synchronous exceptions.

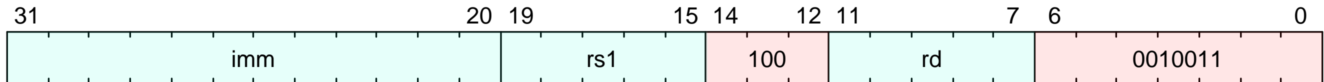
B.203. xori

Exclusive Or immediate

This instruction is defined by:

- I, version >= 0

B.203.1. Encoding



B.203.2. Synopsis

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

B.203.3. Access

M	S	U
Always	Always	Always

B.203.4. Decode Variables

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

B.203.5. Execution

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

B.203.6. Exceptions

This instruction does not generate synchronous exceptions.

Appendix C: CSR Details

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, version >= 0
Length	64-bit
Privilege Mode	U

C.1.2. Format

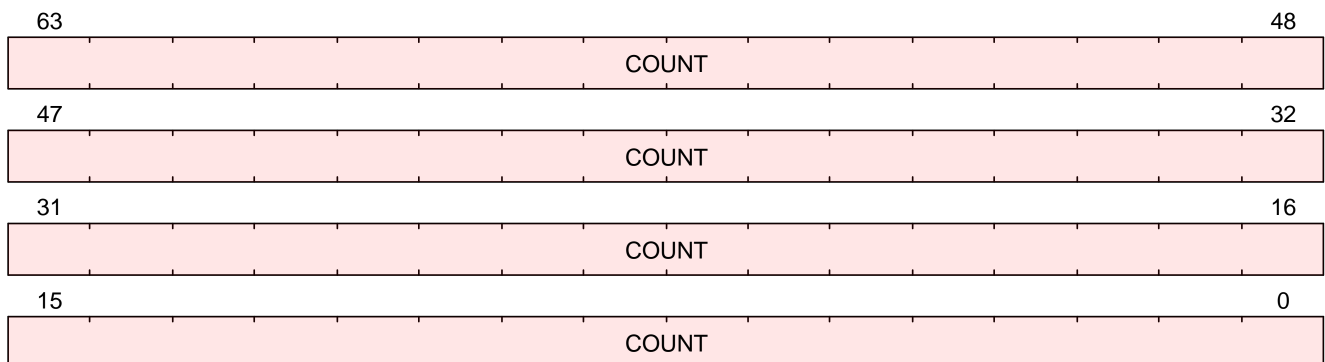


Figure 1. cycle format

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, version >= 0
Length	32-bit
Privilege Mode	U

C.2.2. Format

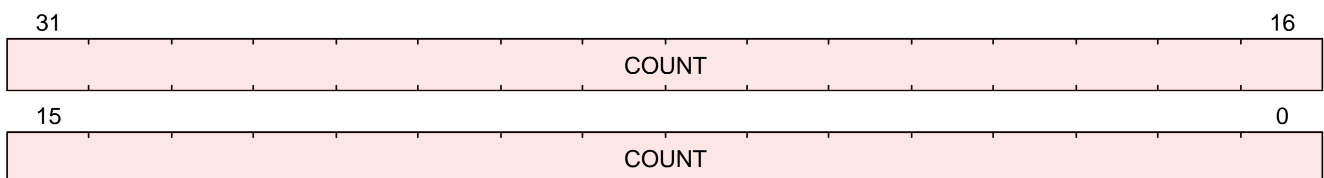


Figure 2. cycleh format

C.3. fcsr

Floating-point control and status register (`frm` + `fflags`)

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 RVA20.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 9. 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).

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 10. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 36. 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, version >= 0
Length	32-bit
Privilege Mode	U

C.3.2. Format

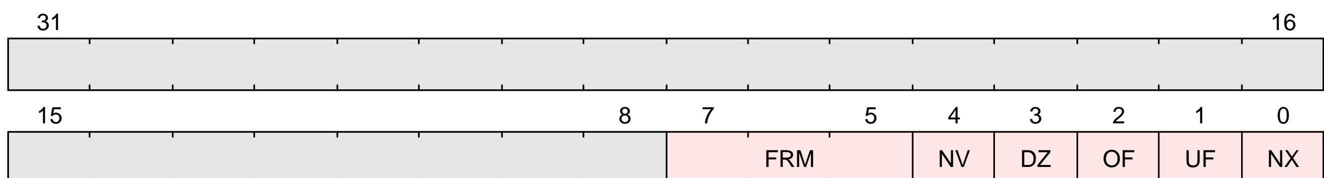


Figure 3. fcsr format

C.4. hpmcounter10

User-mode Hardware Performance Counter 7

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.4.1. Attributes

CSR Address	0xc0a

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.4.2. Format

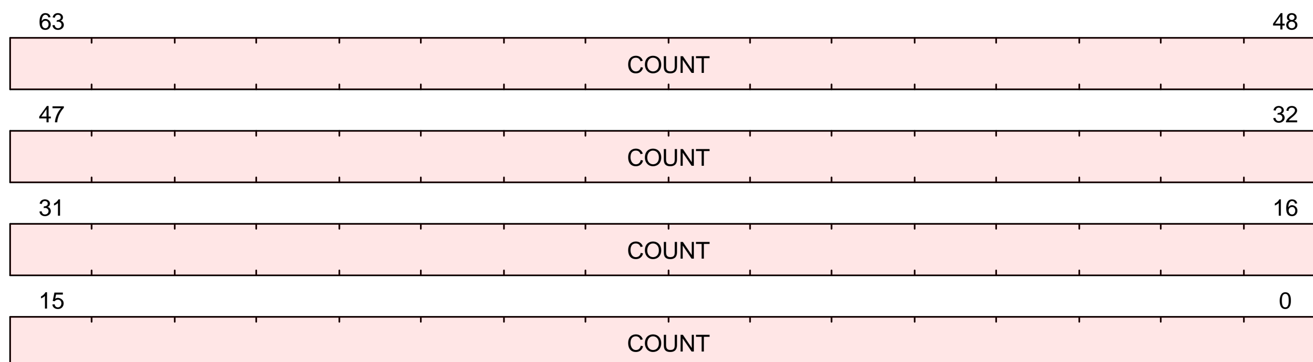


Figure 4. hpmcounter10 format

C.5. hpmcounter11

User-mode Hardware Performance Counter 8

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	-	-	`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.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.5.1. Attributes

CSR Address	0xc0b

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.5.2. Format

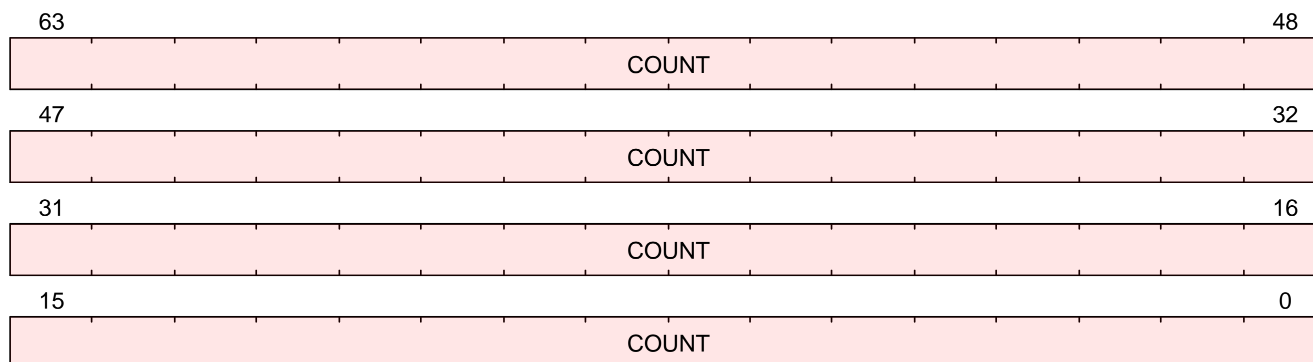


Figure 5. hpmcounter11 format

C.6. hpmcounter12

User-mode Hardware Performance Counter 9

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	-	-	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.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.6.1. Attributes

CSR Address	0xc0c

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.6.2. Format

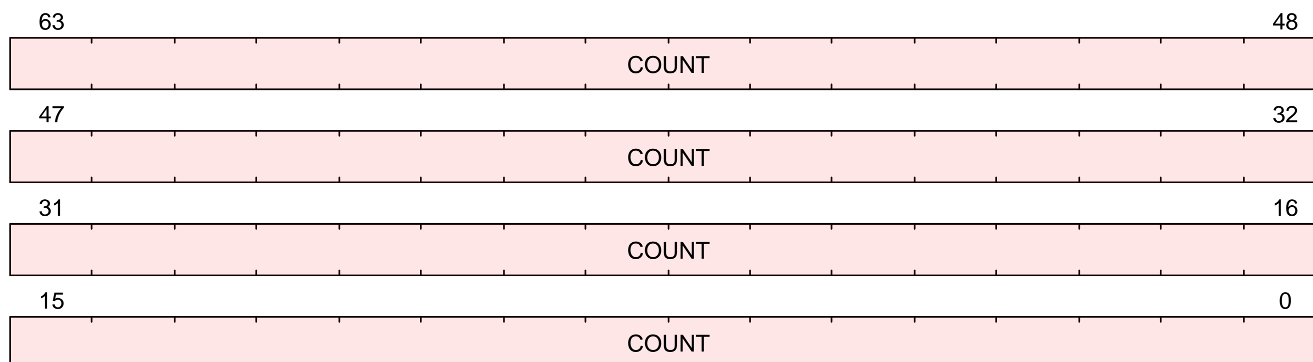


Figure 6. hpmcounter12 format

C.7. hpmcounter13

User-mode Hardware Performance Counter 10

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	-	-	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.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.7.1. Attributes

CSR Address	0xc0d

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.7.2. Format

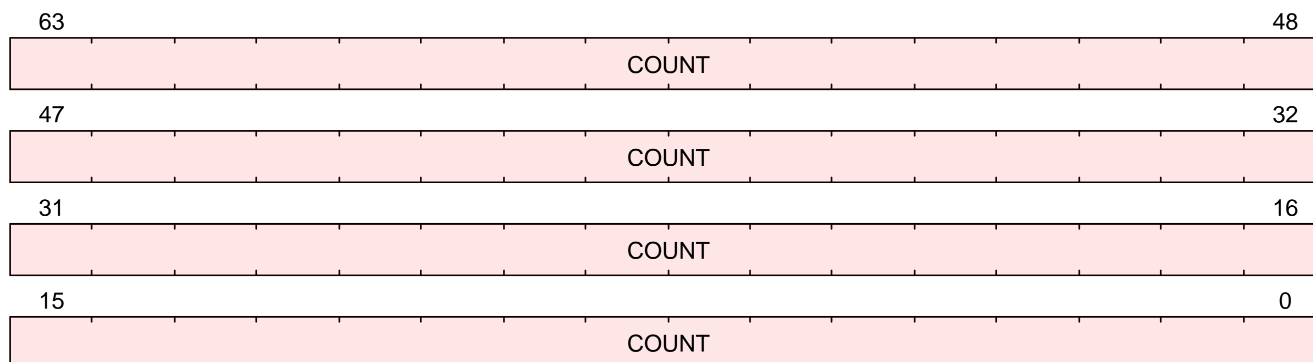


Figure 7. hpmcounter13 format

C.8. hpmcounter14

User-mode Hardware Performance Counter 11

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	-	-	`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.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.8.1. Attributes

CSR Address	0xc0e

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.8.2. Format

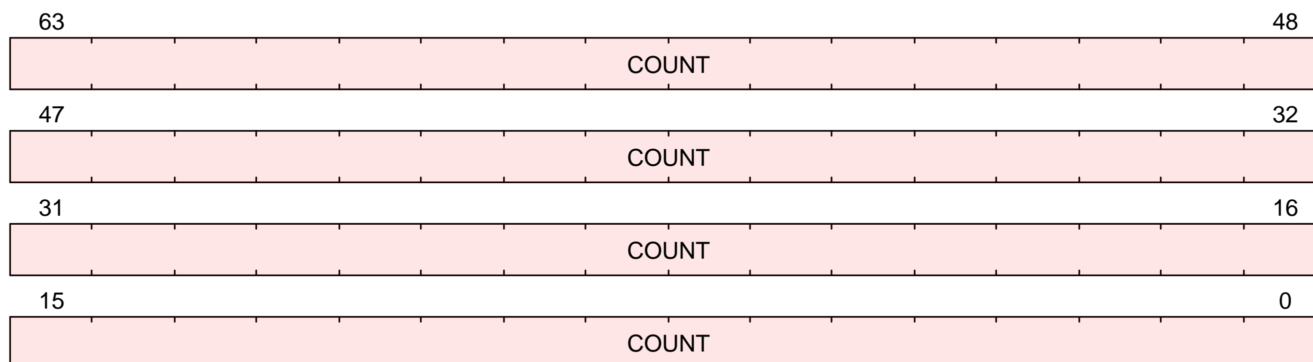


Figure 8. hpmcounter14 format

C.9. hpmcounter15

User-mode Hardware Performance Counter 12

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	-	-	`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.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.9.1. Attributes

CSR Address	0xc0f

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.9.2. Format

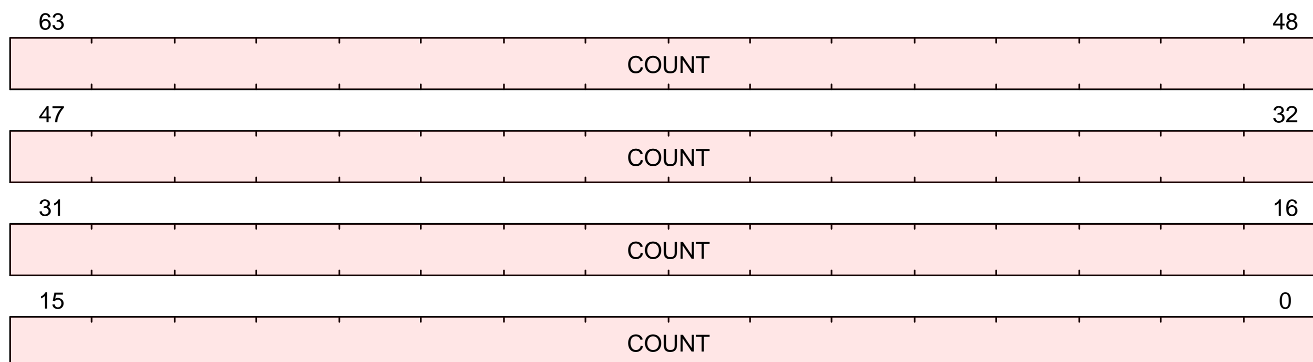


Figure 9. hpmcounter15 format

C.10. hpmcounter16

User-mode Hardware Performance Counter 13

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.10.1. Attributes

CSR Address	0xc10

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.10.2. Format

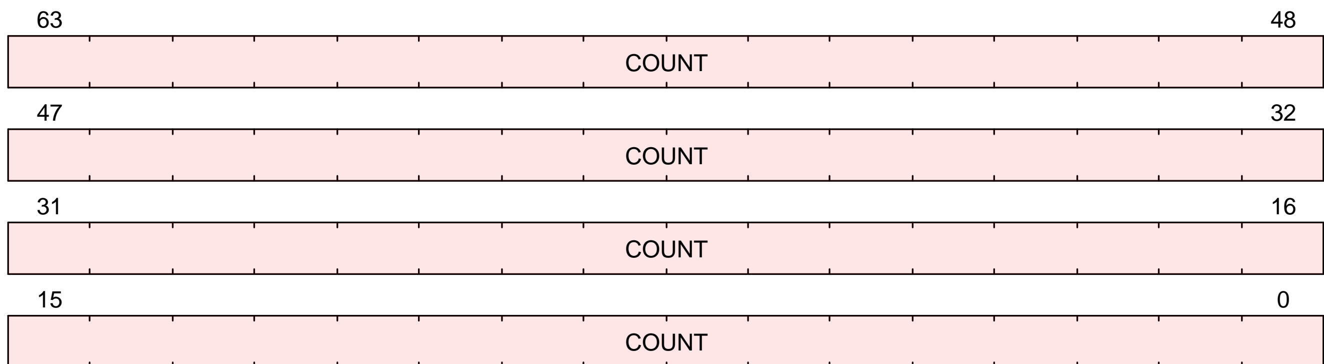


Figure 10. hpmcounter16 format

C.11. hpmcounter17

User-mode Hardware Performance Counter 14

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	-	-	`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.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.11.1. Attributes

CSR Address	0xc11

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.11.2. Format

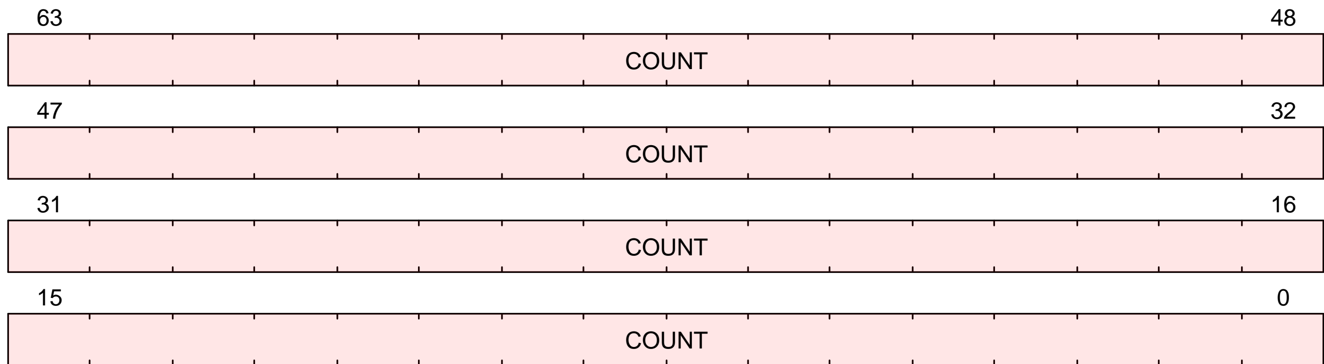


Figure 11. hpmcounter17 format

C.12. hpmcounter18

User-mode Hardware Performance Counter 15

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	-	-	`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.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.12.1. Attributes

CSR Address	0xc12

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.12.2. Format

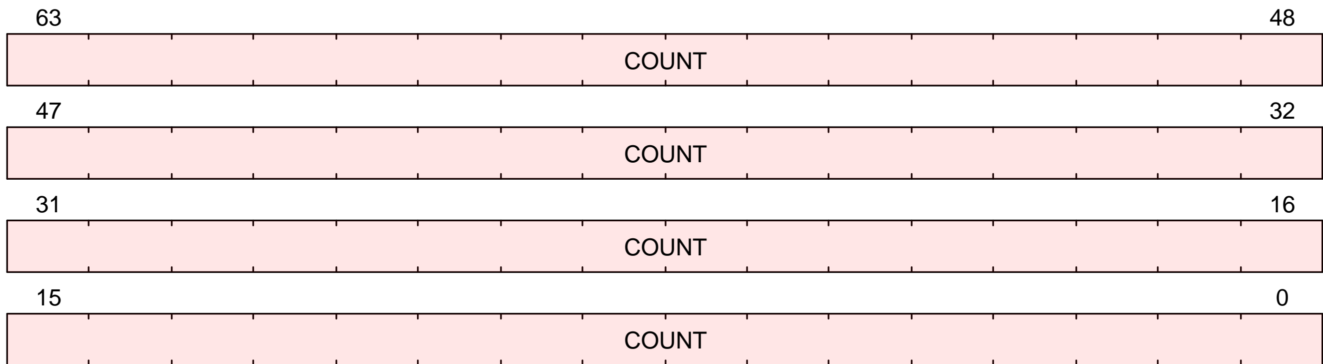


Figure 12. hpmcounter18 format

C.13. hpmcounter19

User-mode Hardware Performance Counter 16

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	-	-	`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.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.13.1. Attributes

CSR Address	0xc13

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.13.2. Format

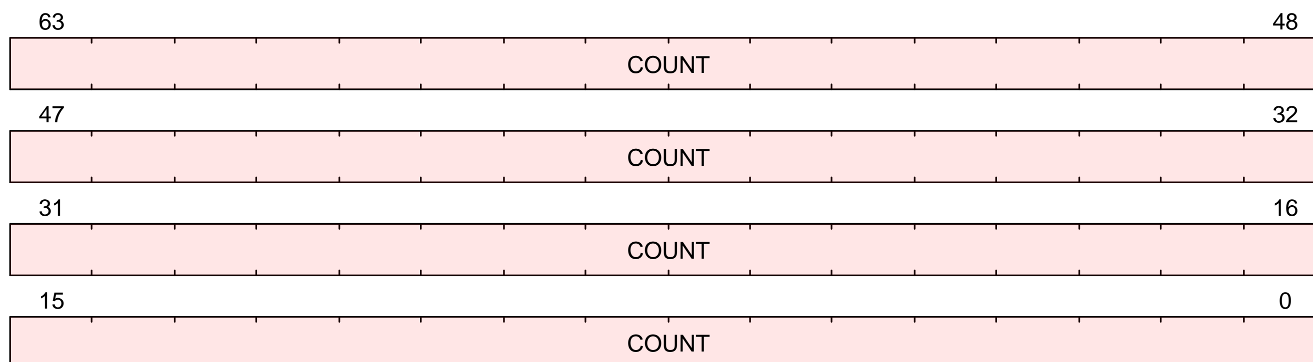


Figure 13. hpmcounter19 format

C.14. hpmcounter20

User-mode Hardware Performance Counter 17

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	-	-	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.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.14.1. Attributes

CSR Address	0xc14

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.14.2. Format

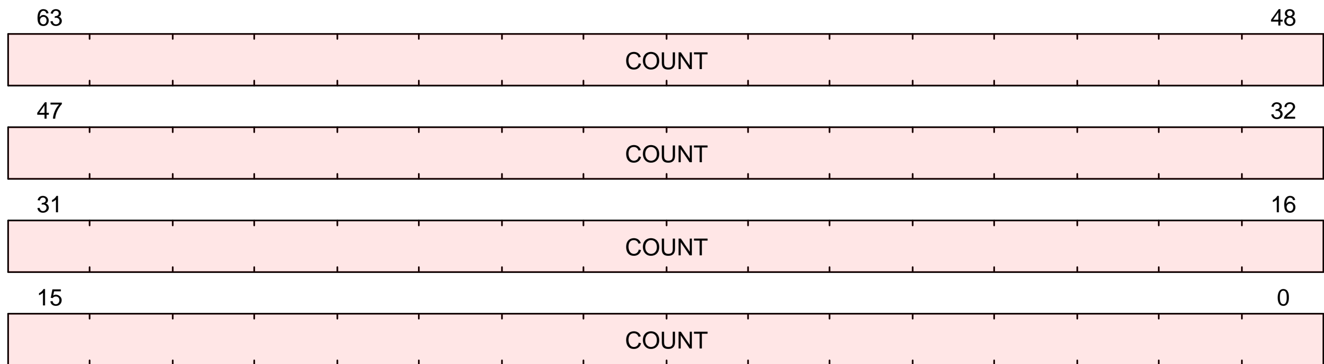


Figure 14. hpmcounter20 format

C.15. hpmcounter21

User-mode Hardware Performance Counter 18

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	-	-	`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.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.15.1. Attributes

CSR Address	0xc15

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.15.2. Format

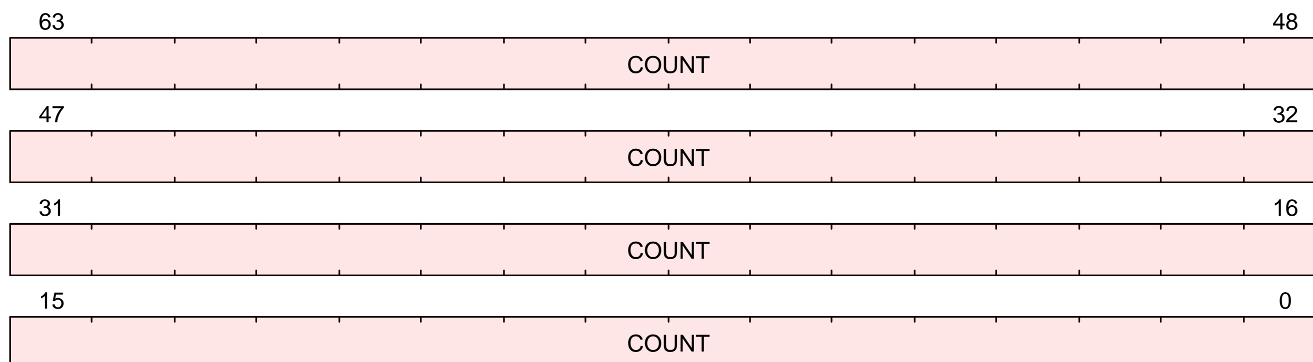


Figure 15. hpmcounter21 format

C.16. hpmcounter22

User-mode Hardware Performance Counter 19

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	-	-	`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.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.16.1. Attributes

CSR Address	0xc16

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.16.2. Format

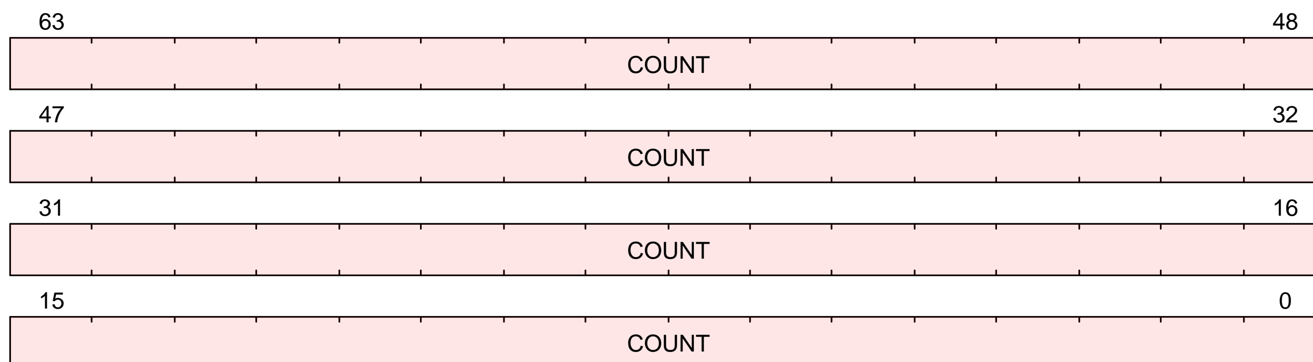


Figure 16. hpmcounter22 format

C.17. hpmcounter23

User-mode Hardware Performance Counter 20

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	-	-	`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.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.17.1. Attributes

CSR Address	0xc17

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.17.2. Format

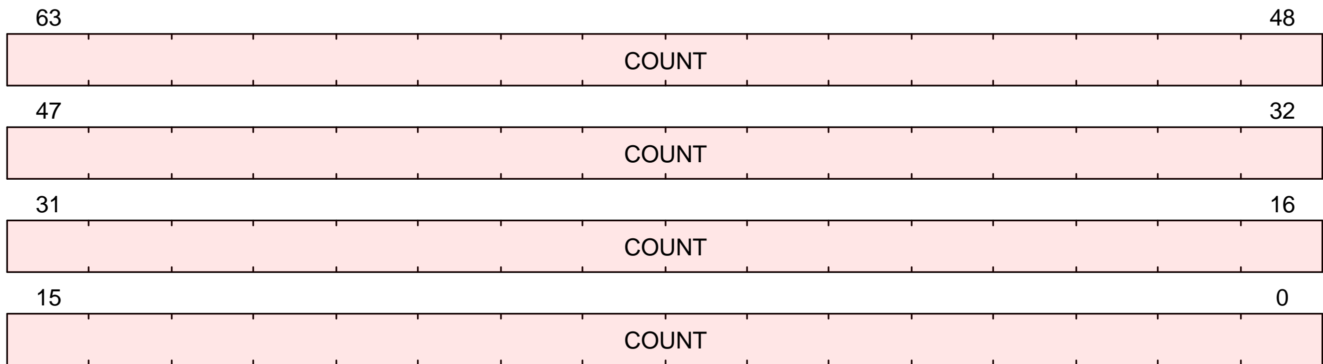


Figure 17. hpmcounter23 format

C.18. hpmcounter24

User-mode Hardware Performance Counter 21

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.18.1. Attributes

CSR Address	0xc18

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.18.2. Format

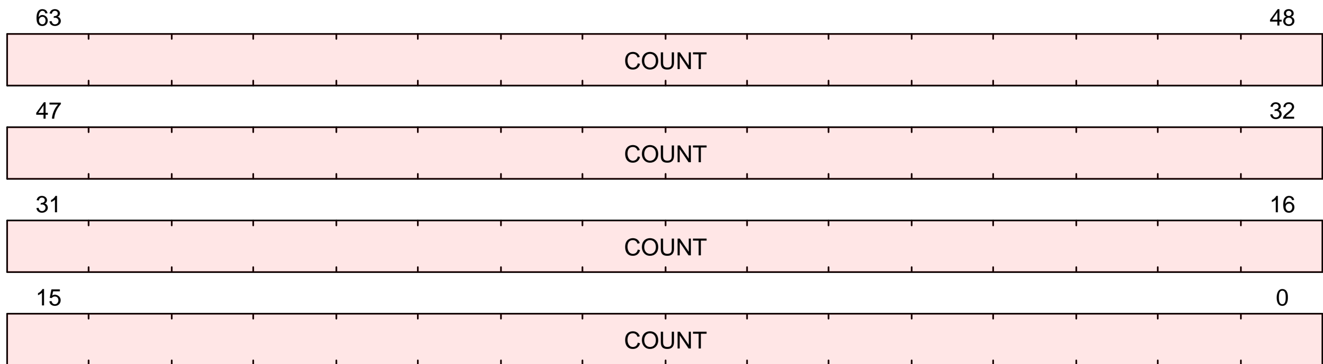


Figure 18. hpmcounter24 format

C.19. hpmcounter25

User-mode Hardware Performance Counter 22

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	-	-	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.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.19.1. Attributes

CSR Address	0xc19

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.19.2. Format

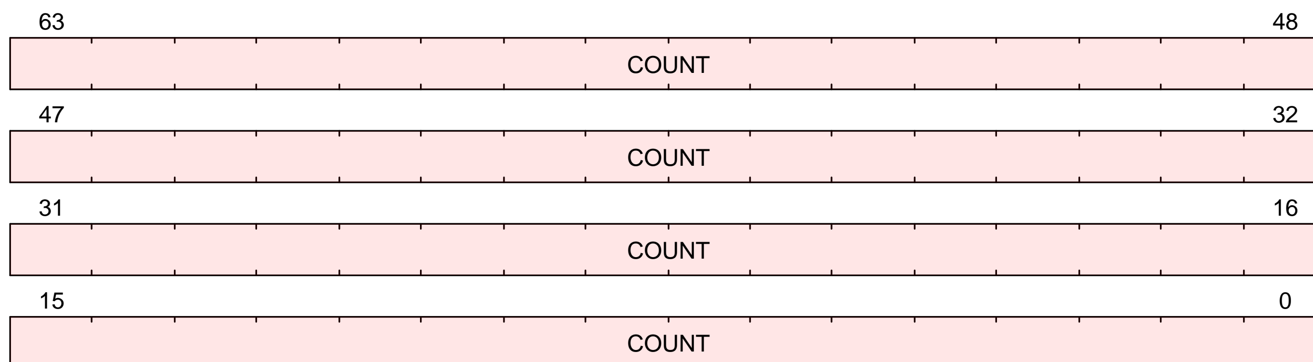


Figure 19. hpmcounter25 format

C.20. hpmcounter26

User-mode Hardware Performance Counter 23

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	-	-	`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.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.20.1. Attributes

CSR Address	0xc1a

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.20.2. Format

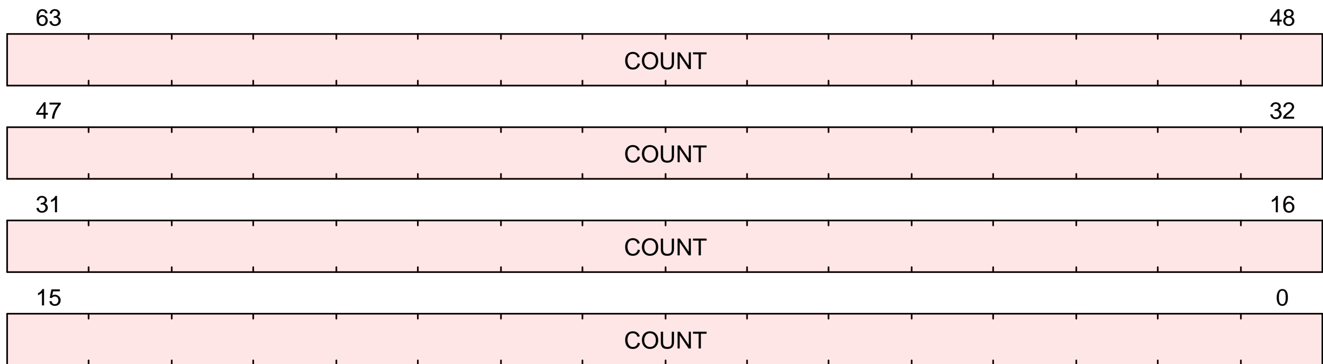


Figure 20. hpmcounter26 format

C.21. hpmcounter27

User-mode Hardware Performance Counter 24

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	-	-	`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.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.21.1. Attributes

CSR Address	0xc1b

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.21.2. Format

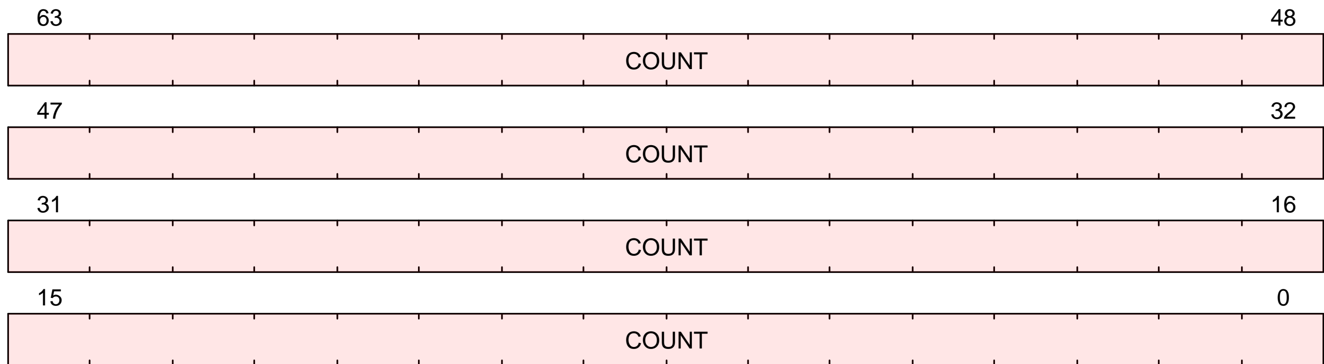


Figure 21. hpmcounter27 format

C.22. hpmcounter28

User-mode Hardware Performance Counter 25

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	-	-	`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.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.22.1. Attributes

CSR Address	0xc1c

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.22.2. Format

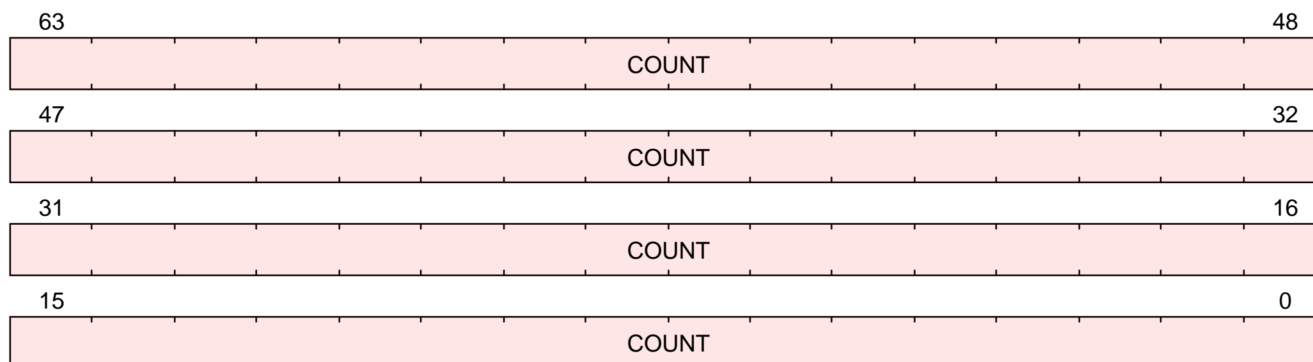


Figure 22. hpmcounter28 format

C.23. hpmcounter29

User-mode Hardware Performance Counter 26

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	-	-	`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.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.23.1. Attributes

CSR Address	0xc1d

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.23.2. Format

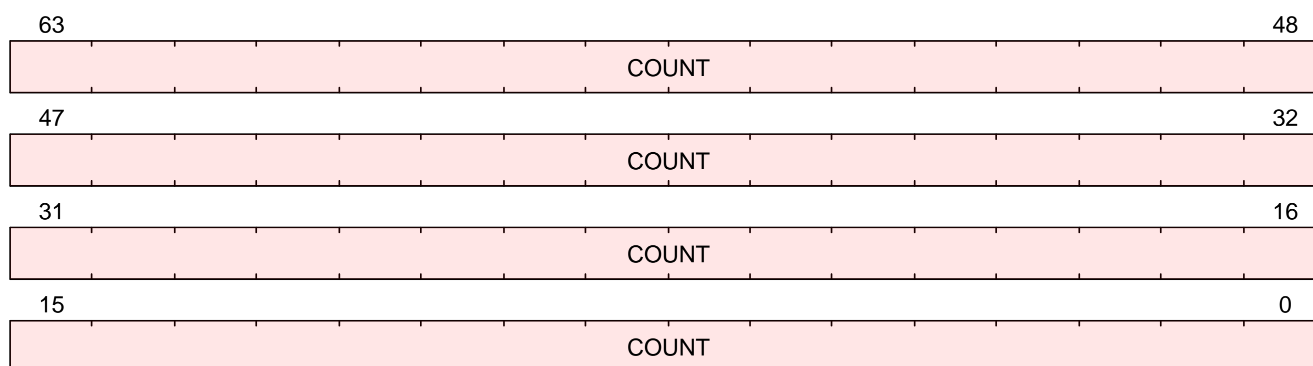


Figure 23. hpmcounter29 format

C.24. hpmcounter3

User-mode Hardware Performance Counter 0

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	-	-	`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.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.24.1. Attributes

CSR Address	0xc03

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.24.2. Format

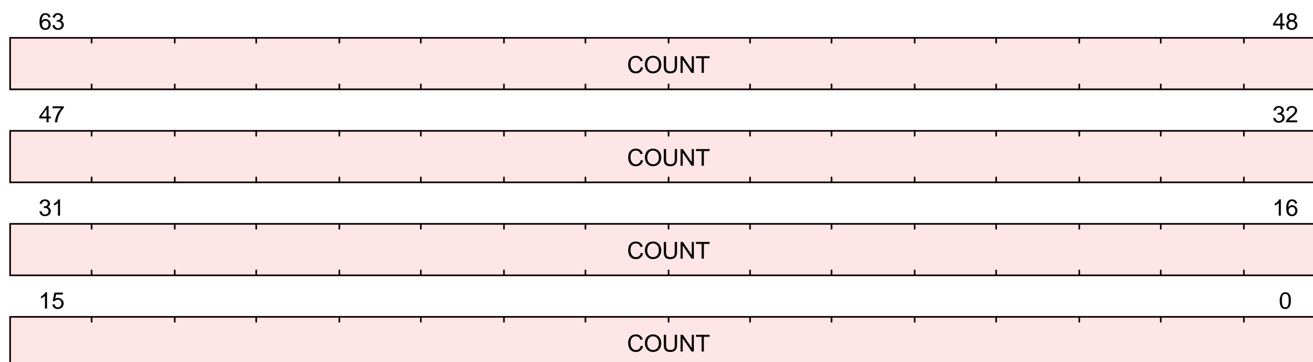


Figure 24. hpmcounter3 format

C.25. hpmcounter30

User-mode Hardware Performance Counter 27

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.25.1. Attributes

CSR Address	0xc1e

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.25.2. Format

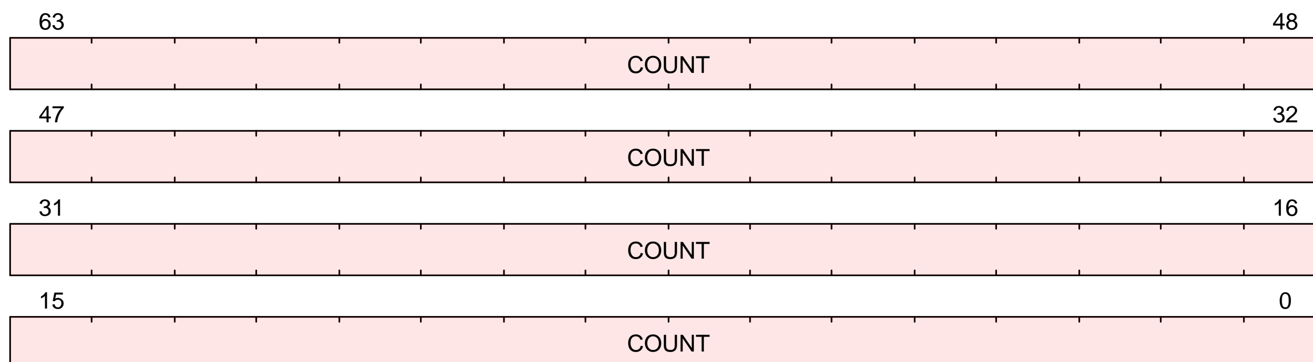


Figure 25. hpmcounter30 format

C.26. hpmcounter31

User-mode Hardware Performance Counter 28

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	-	-	`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.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.26.1. Attributes

CSR Address	0xc1f

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.26.2. Format

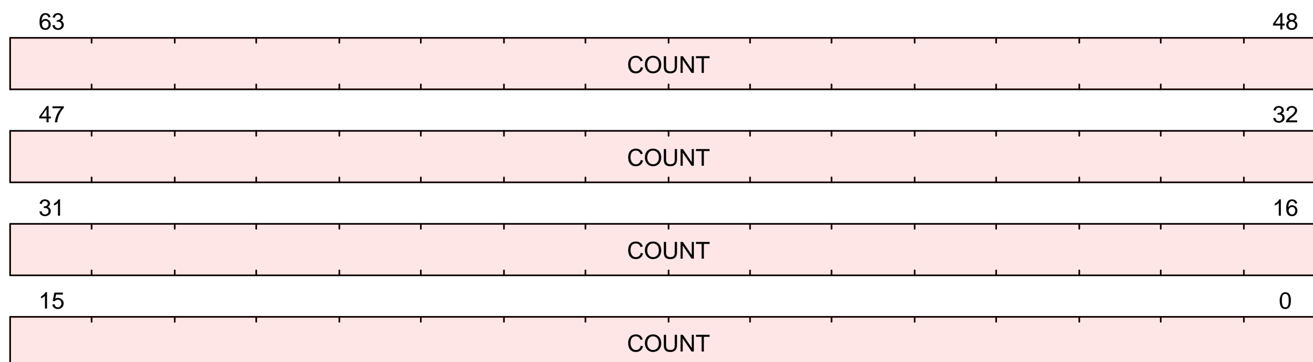


Figure 26. hpmcounter31 format

C.27. hpmcounter4

User-mode Hardware Performance Counter 1

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	-	-	`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.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.27.1. Attributes

CSR Address	0xc04

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.27.2. Format

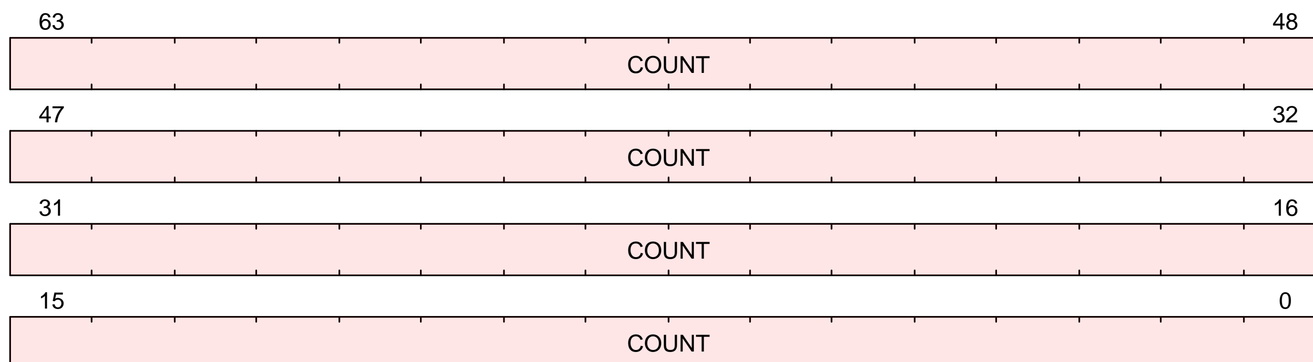


Figure 27. hpmcounter4 format

C.28. hpmcounter5

User-mode Hardware Performance Counter 2

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	-	-	`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.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.28.1. Attributes

CSR Address	0xc05

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.28.2. Format

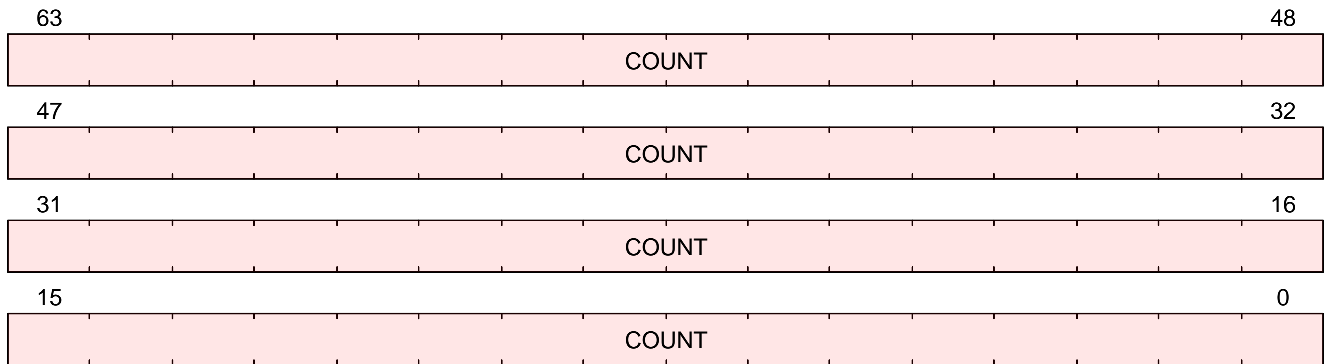


Figure 28. hpmcounter5 format

C.29. hpmcounter6

User-mode Hardware Performance Counter 3

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	-	-	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.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.29.1. Attributes

CSR Address	0xc06

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.29.2. Format

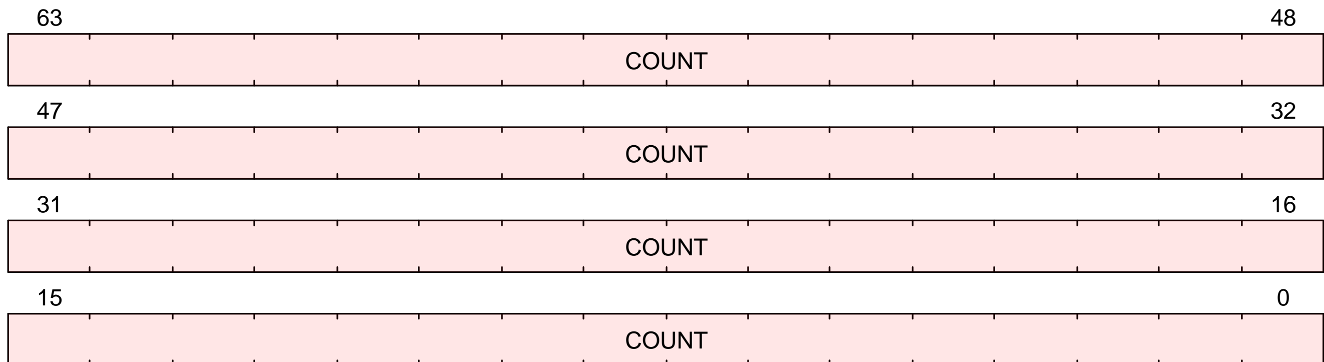


Figure 29. hpmcounter6 format

C.30. hpmcounter7

User-mode Hardware Performance Counter 4

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.30.1. Attributes

CSR Address	0xc07

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.30.2. Format

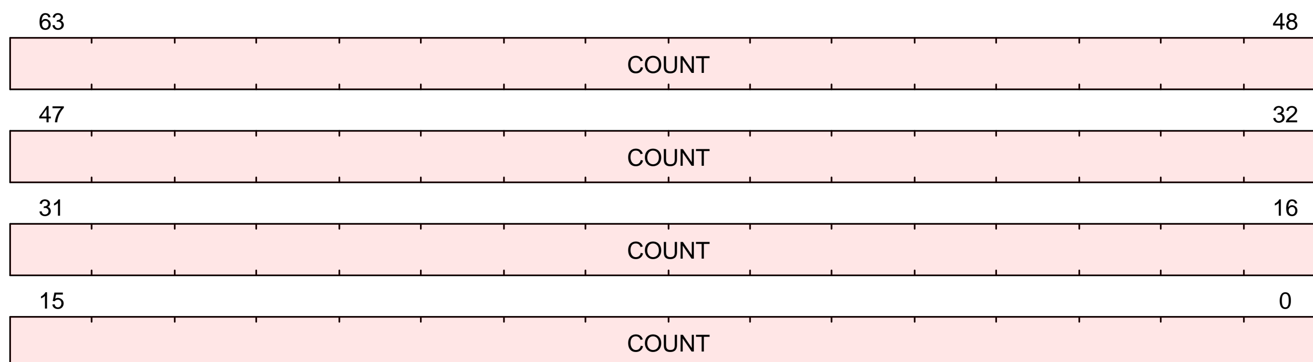


Figure 30. hpmcounter7 format

C.31. hpmcounter8

User-mode Hardware Performance Counter 5

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	-	-	`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.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.31.1. Attributes

CSR Address	0xc08

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.31.2. Format

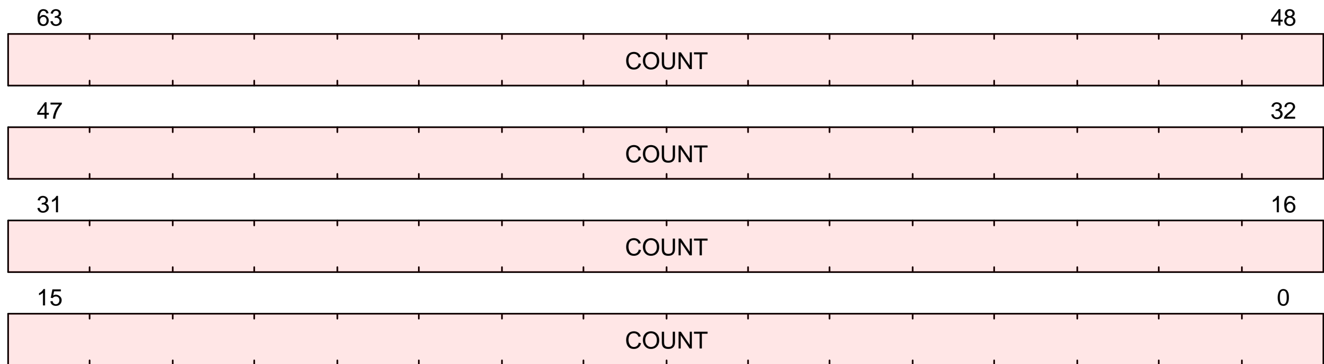


Figure 31. hpmcounter8 format

C.32. hpmcounter9

User-mode Hardware Performance Counter 6

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.32.1. Attributes

CSR Address	0xc09

Defining extension	• Zihpm, version >= 0
Length	64-bit
Privilege Mode	U

C.32.2. Format

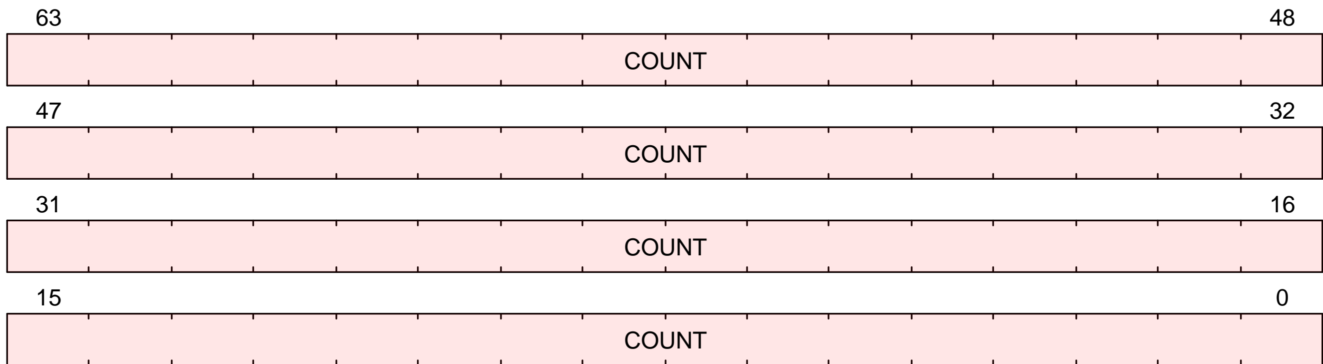


Figure 32. hpmcounter9 format

C.33. 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.33.1. Attributes

CSR Address	0xc02
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	U

C.33.2. Format

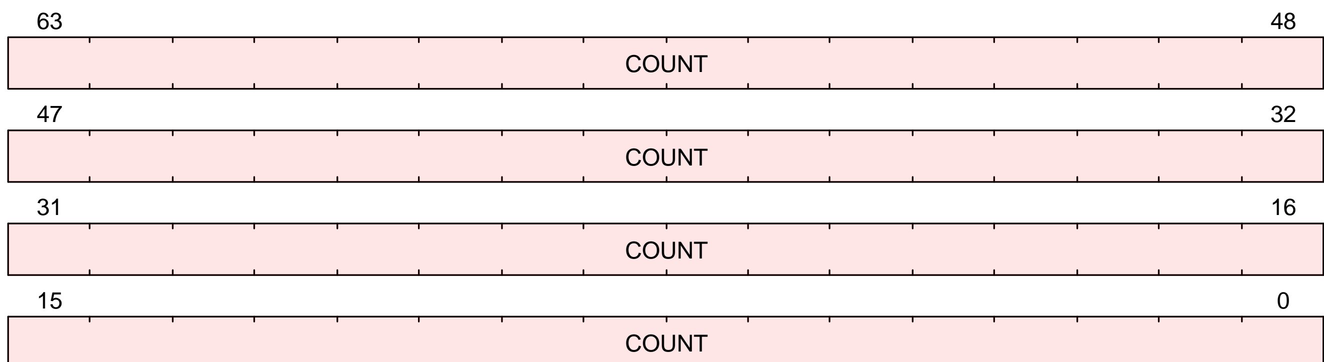


Figure 33. instret format

C.34. instreth

Instructions retired counter, high bits

instreth is only defined in RV32.

Alias for high bits of M-mode CSR minstret[63:32].

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.34.1. Attributes

CSR Address	0xc82
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	U

C.34.2. Format

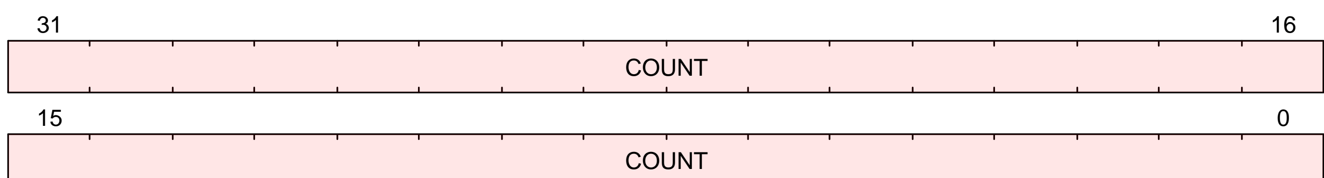


Figure 34. instreth format

C.35. 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.35.1. Attributes

CSR Address	0x306
Defining extension	• U, version >= 0
Length	32-bit
Privilege Mode	M

C.35.2. Format

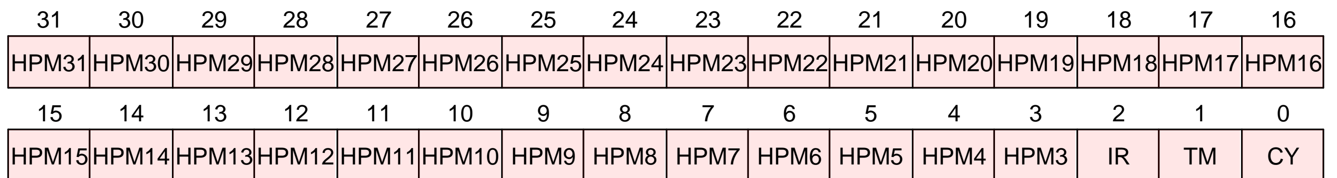


Figure 35. mcounteren format

C.36. 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.36.1. Attributes

CSR Address	0xb00
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	M

C.36.2. Format

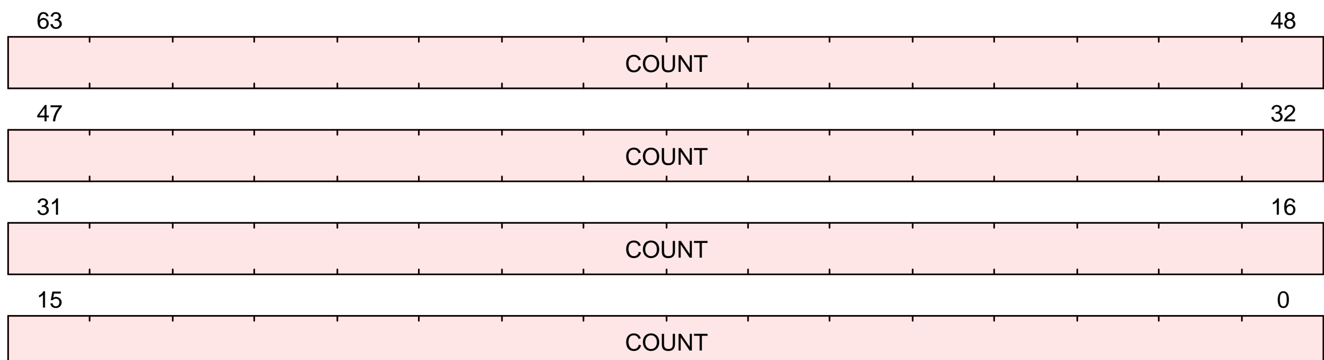


Figure 36. mcycle format

C.37. mcycleh

High-half machine Cycle Counter

mcycleh is only defined in RV32.

High-half alias of mcycle.

C.37.1. Attributes

CSR Address	0xb80
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	M

C.37.2. Format

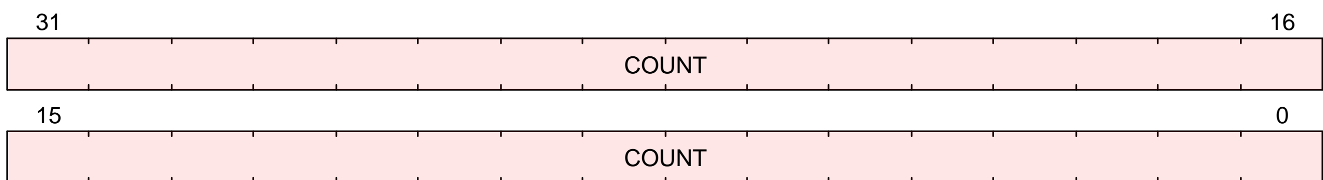


Figure 37. mcycleh format

C.38. 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.38.1. Attributes

CSR Address	0x302
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	M

C.38.2. Format

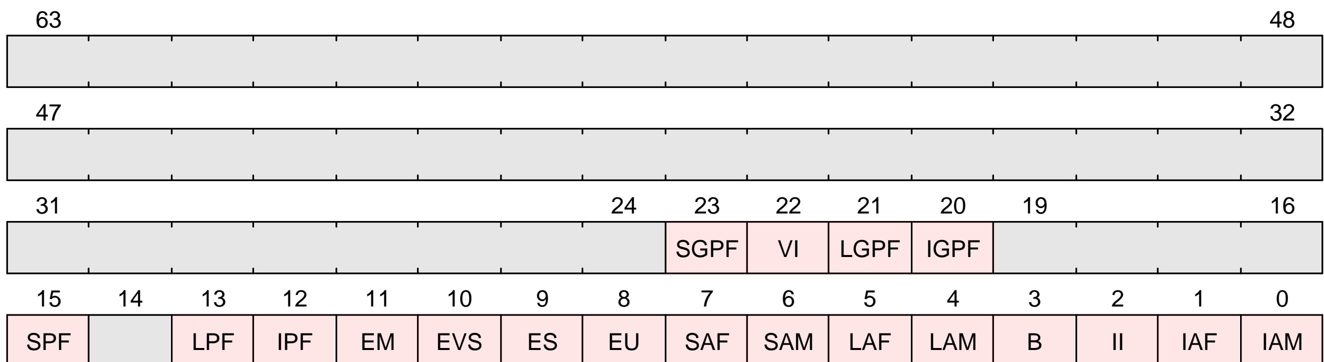


Figure 38. medeleg format

C.39. 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.39.1. Attributes

CSR Address	0xb02
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	M

C.39.2. Format

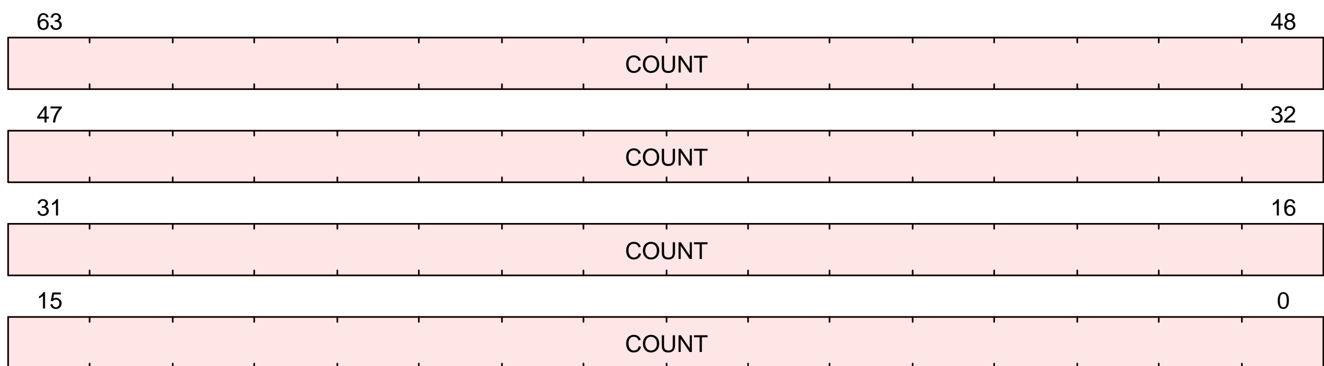


Figure 39. minstret format

C.40. minstreth

Machine Instructions Retired Counter

Upper half of 64-bit instructions retired counters.

See minstret for details.

C.40.1. Attributes

CSR Address	0xb02
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	M

C.40.2. Format

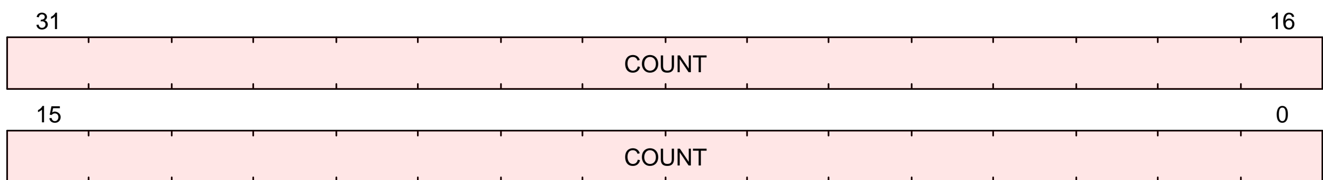


Figure 40. minstreth format

C.41. 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.41.1. Attributes

CSR Address	0x180
Defining extension	• S, version >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.41.2. Format

This CSR format changes dynamically with XLEN.

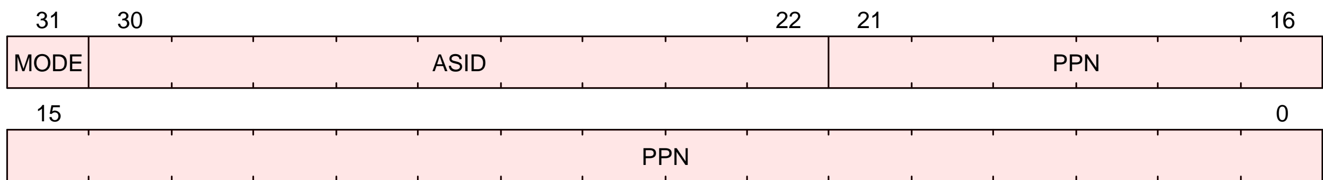


Figure 41. satp Format when CSR[mstatus].SXL == 0

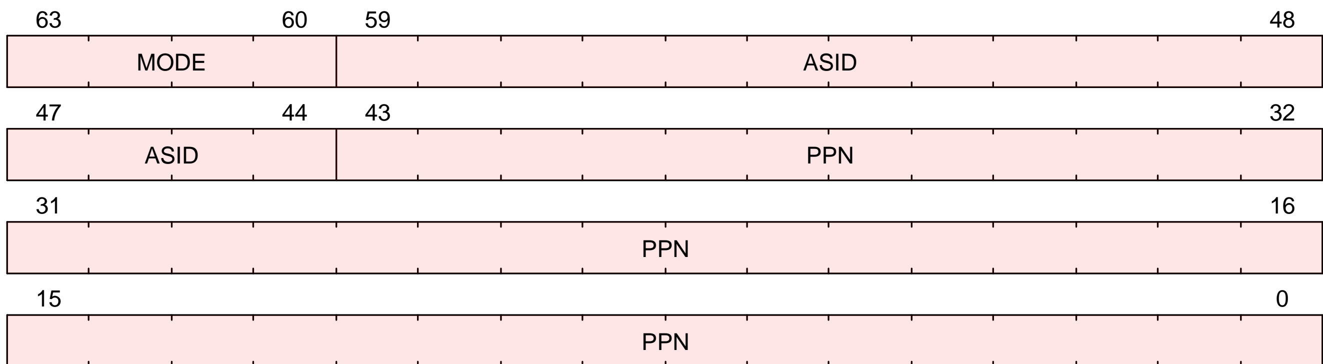


Figure 42. satp Format when CSR[mstatus].SXL == 1

C.42. scause

Supervisor Cause

Reports the cause of the latest exception.

C.42.1. Attributes

CSR Address	0x142
Defining extension	• S, version >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.42.2. Format

This CSR format changes dynamically with XLEN.

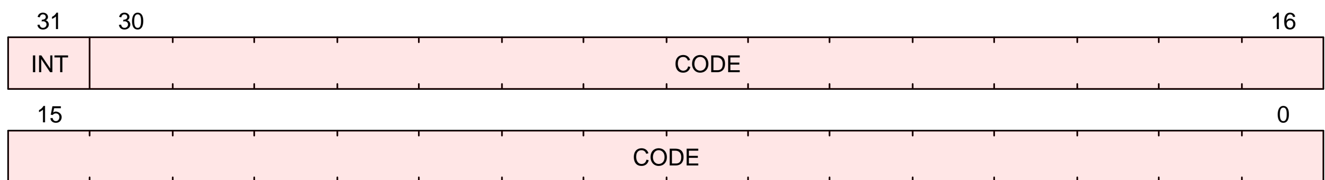


Figure 43. scause Format when CSR[mstatus].SXL == 0

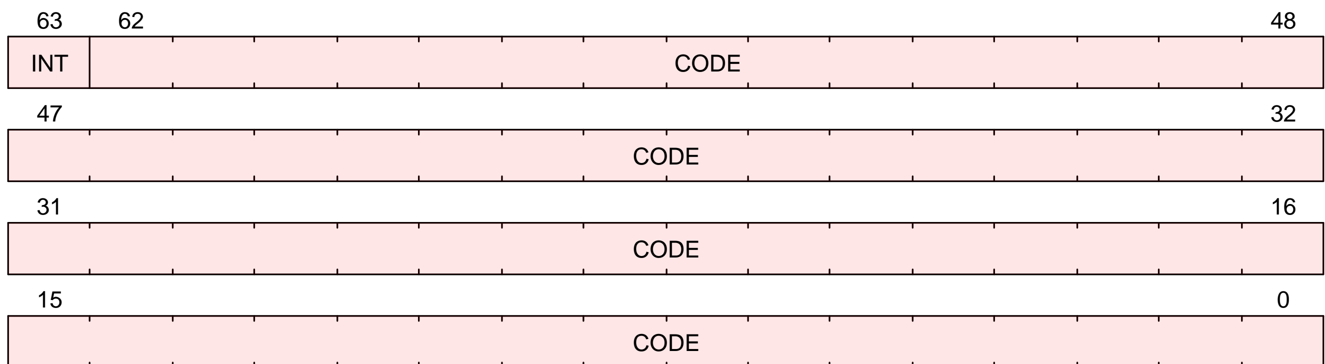


Figure 44. scause Format when CSR[mstatus].SXL == 1

C.43. scounteren

Supervisor Counter Enable

Delegates control of the hardware performance-monitoring counters to U-mode

C.43.1. Attributes

CSR Address	0x106
Defining extension	• S, version >= 0
Length	32-bit
Privilege Mode	S

C.43.2. Format

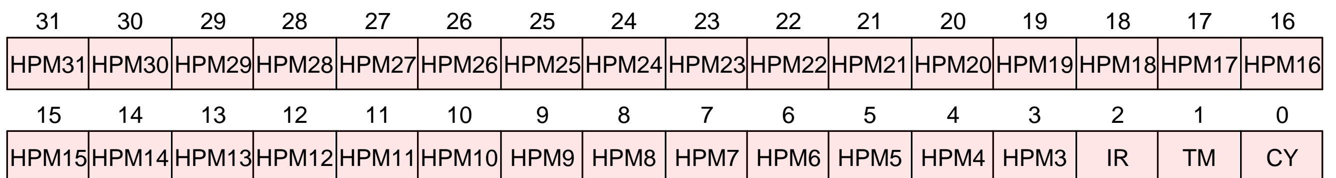


Figure 45. scounteren format

C.44. 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.44.1. Attributes

CSR Address	0x141
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	S

C.44.2. Format

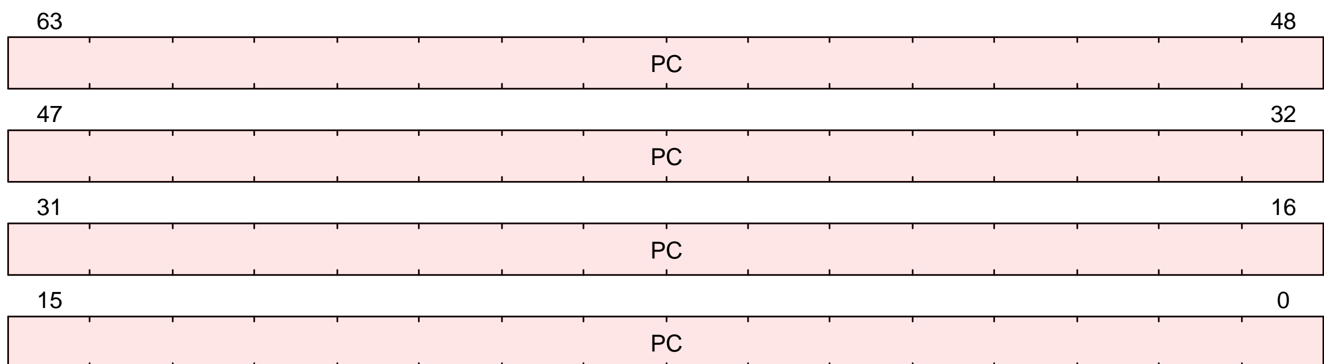


Figure 46. sepc format

C.45. 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.45.1. Attributes

CSR Address	0x144
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	S

C.45.2. Format

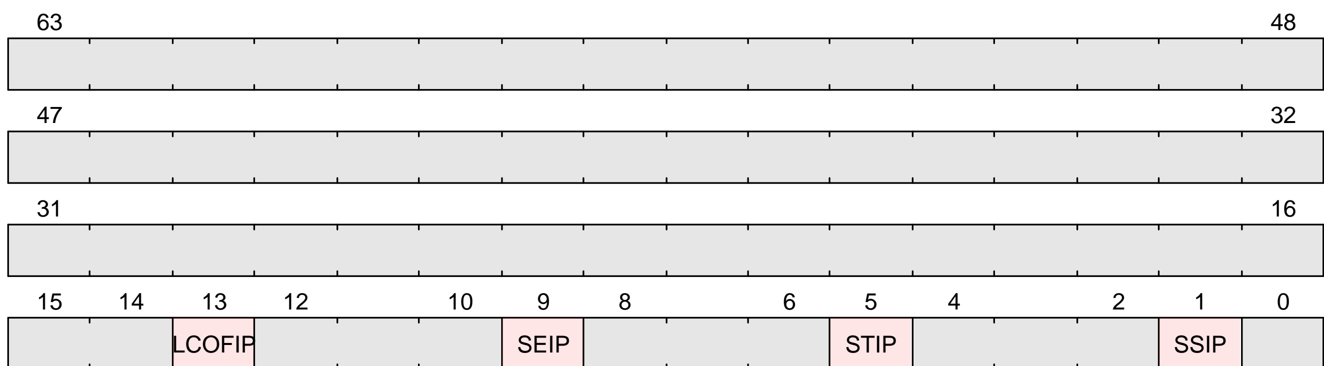


Figure 47. sip format

C.46. sscratch

Supervisor Scratch Register

Scratch register for software use. Bits are not interpreted by hardware.

C.46.1. Attributes

CSR Address	0x140
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	S

C.46.2. Format

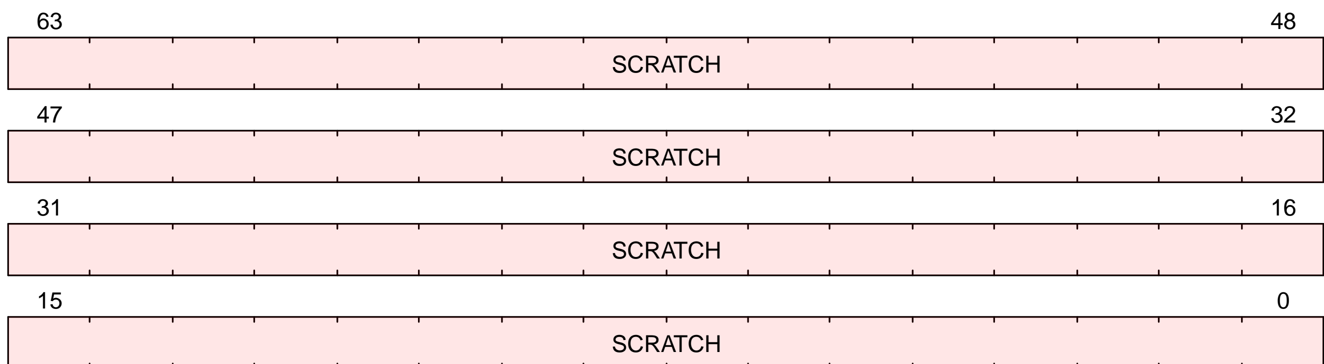


Figure 48. sscratch format

C.47. 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.47.1. Attributes

CSR Address	0x100
Defining extension	• S, version >= 0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.47.2. Format

This CSR format changes dynamically with XLEN.

31	30–20	19	18	17	16
SD		MXR	SUM		XS

15	14–13	12–11	10–9	8	7	6	5	4–2	1	0
XS	FS		VS	SPP		UBE	SPIE		SIE	

Figure 49. sstatus Format when CSR[mstatus].SXL == 0

63	62–48
SD	

47–34	33–32
	UXL

31–20	19	18	17	16
	MXR	SUM		XS

15	14–13	12–11	10–9	8	7	6	5	4–2	1	0
XS	FS		VS	SPP		UBE	SPIE		SIE	

Figure 50. sstatus Format when CSR[mstatus].SXL == 1

C.48. stval

Supervisor Trap Value

Holds trap-specific information

C.48.1. Attributes

CSR Address	0x143
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	S

C.48.2. Format

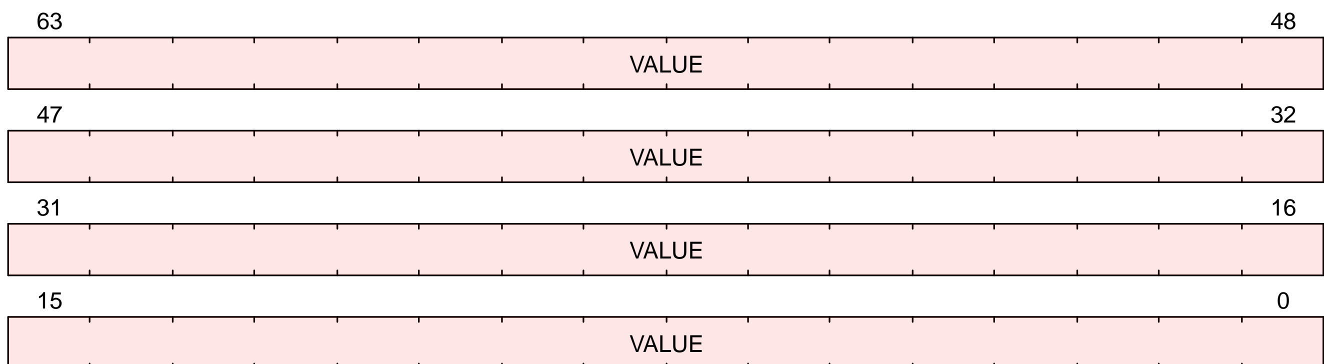


Figure 51. stval format

C.49. stvec

Supervisor Trap Vector

Controls where traps jump.

C.49.1. Attributes

CSR Address	0x105
Defining extension	• S, version >= 0
Length	64-bit
Privilege Mode	S

C.49.2. Format

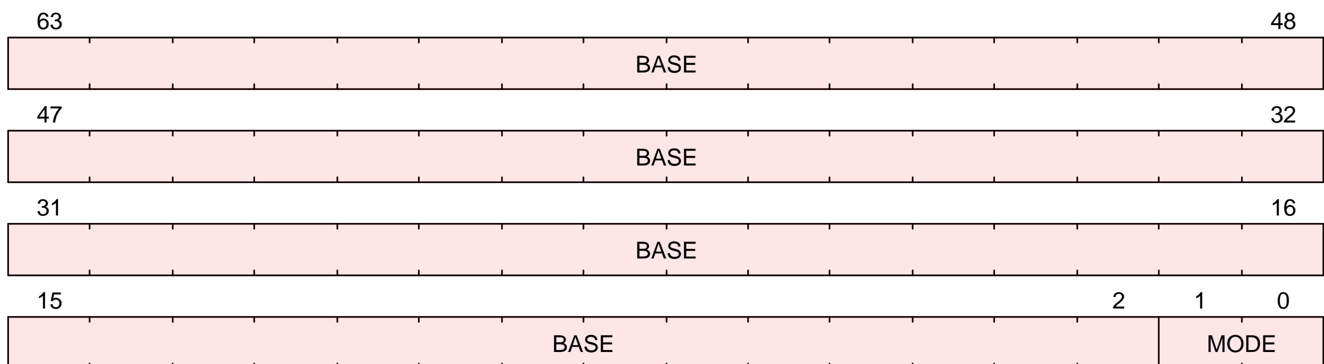


Figure 52. stvec format

C.50. time

Timer for RDTIME Instruction

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of 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.50.1. Attributes

CSR Address	0xc01
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	U

C.50.2. Format

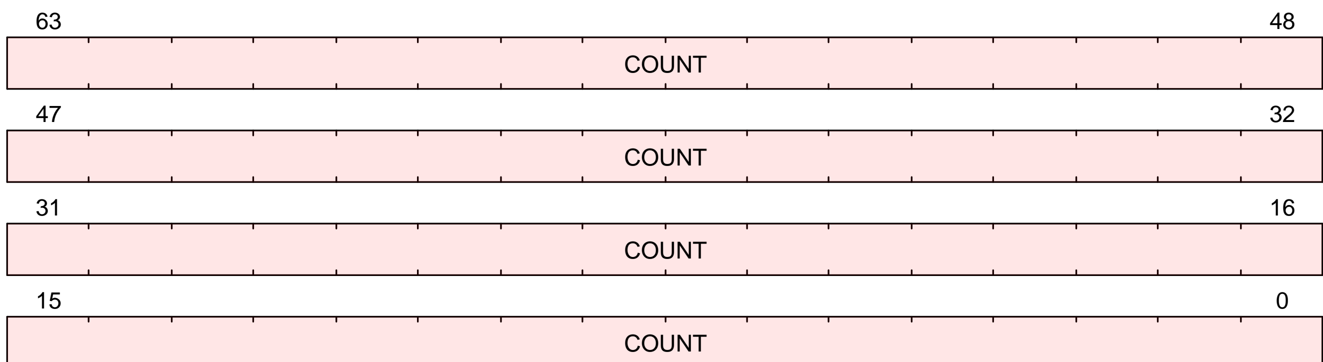


Figure 53. time format

C.51. timeh

High-half timer for RDTIME Instruction

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtimeh`.

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.51.1. Attributes

CSR Address	0xc81
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	U

C.51.2. Format

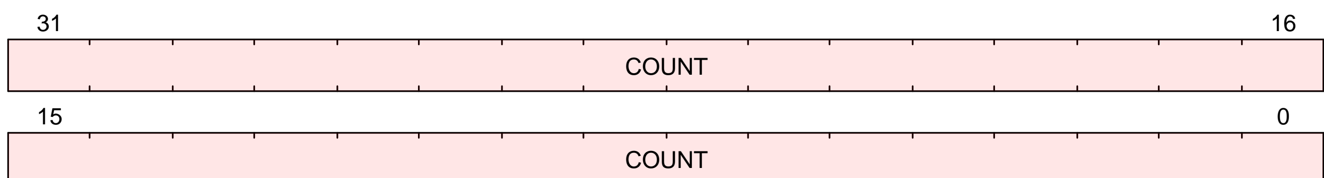


Figure 54. timeh format