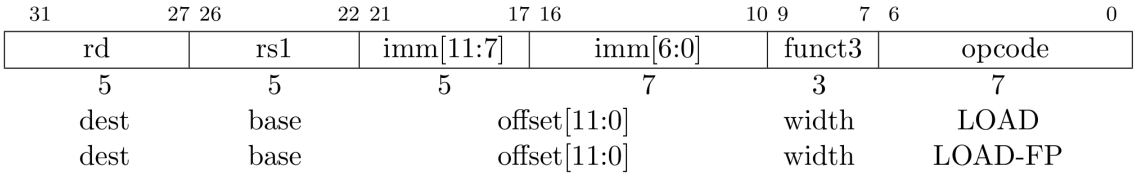
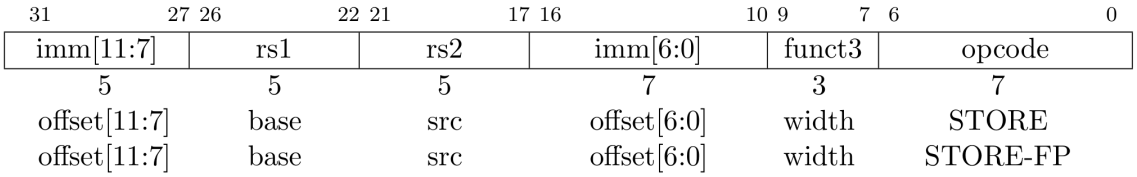
Summary of Subset of RISC-V ISA

1 Load/Store



31–27	26–22	21–17	16–10	9–7	6–0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	base	offset[11:0]		width	LOAD
dest	base	offset[11:0]		width	LOAD-FP



31–27	26–22	21–17	16–10	9–7	6–0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7
offset[11:7]	base	src	offset[6:0]	width	STORE
offset[11:7]	base	src	offset[6:0]	width	STORE-FP

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format, and stores are B-type. The effective byte address is obtained by adding register *rs1* to the sign-extended immediate. Loads write to register *rd* a value in memory. Stores write to memory the value in register *rs2*.

2 Shifts

31 27	26 22	21 16	15	14 10	9 7	6 0
rd	rs1	imm[11:6]	imm[5]	imm[4:0]	funct3	opcode
5	5	6	1	5	3	7
dest	src	SRA/SRL	shamt[5]	shamt[4:0]	SRxI	OP-IMM
dest	src	SRA/SRL	0	shamt[4:0]	SRxIW	OP-IMM-32
dest	src	0	shamt[5]	shamt[4:0]	SLLI	OP-IMM
dest	src	0	0	shamt[4:0]	SLLIW	OP-IMM-32

Shifts by a constant are also encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the immediate field for RV64, and in the lower 5 bits for RV32. The shift type is encoded in the upper bits of the immediate field.

SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). In RV32, SLLI, SRLI, and SRAI generate an illegal instruction trap if $imm[5] \neq 0$.

SLLIW, SRLIW, and SRAIW are RV64-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. SLLIW, SRLIW, and SRAIW generate an illegal instruction trap if $imm[5] \neq 0$.

3 ADDI, ANDI, ORI, XORI and (implicitly) NOT with Immediates

Integer Register-Immediate Instructions

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	src	immediate[11:0]		ADDI/SLTI[U]	OP-IMM
dest	src	immediate[11:0]		ANDI/ORI/XORI	OP-IMM
dest	src	immediate[11:0]		ADDIW	OP-IMM-32

ADDI and ADDIW add the sign-extended 12-bit immediate to register *rs1*. ADDIW is an RV64-only instruction that produces the proper sign-extension of a 32-bit result. Note, ADDIW *rd, rs1, 0* writes the sign-extension of the lower 32 bits of register *rs1* into register *rd*.

SLTI (set less than immediate) 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*. SLTIU is similar but compares the values as unsigned numbers.

ANDI, ORI, XORI are logical operations that perform bit-wise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a logical inversion (NOT) of register *rs1*.

4 ADD, AND, OR, XOR with Registers

RISC-V defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct* field selects the type of operation.

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	ADD/SUB/SLT/SLTU	OP
dest	src1	src2	AND/OR/XOR	OP
dest	src1	src2	SLL/SRL/SRA	OP
dest	src1	src2	ADDW/SUBW	OP-32
dest	src1	src2	SLLW/SRLW/SRAW	OP-32

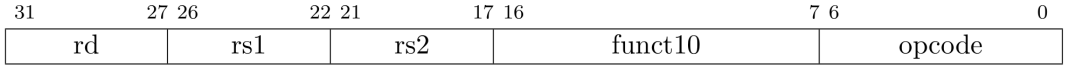
ADD and SUB perform addition and subtraction respectively. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* $<$ *rs2*, 0 otherwise. AND, OR, and XOR perform bitwise logical operations.

ADDW and SUBW are RV64-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in register *rs2*. In RV64, only the low 6 bits of *rs2* are considered for the shift amount. Similarly for RV32, only the low 5 bits of *rs2* are considered.

SLLW, SRLW, and SRAW are RV64-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. The shift amount is given by *rs2[4:0]*.

5 Multiply Divide (May want to incorporate later)



31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	MUL/MULH[[S]U]	OP
dest	dividend	divisor	DIV[U]/REM[U]	OP
dest	src1	src2	MUL[U]W	OP-32
dest	dividend	divisor	DIV[U]W/REM[U]W	OP-32

MUL performs an XPRLEN-bit×XPRLEN-bit multiplication and places the lower XPRLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XPRLEN bits of the full 2×XPRLEN-bit product, for signed×signed, unsigned×unsigned, and signed×unsigned multiplication respectively. 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.

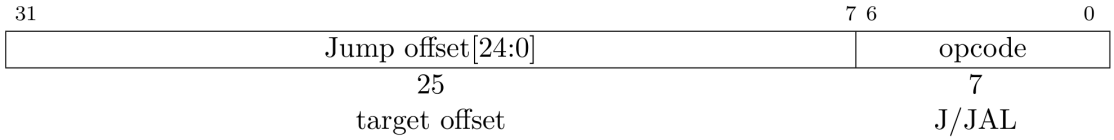
MULW is an RV64-only instruction that multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register. MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear.

DIV and DIVU perform signed and unsigned integer division of XPRLEN bits by XPRLEN bits. REM and REMU provide the remainder of the corresponding division operation. If both the quotient and remainder are required from the same division, the recommended code sequence is: DIV[U] *rdq, rs1, rs2*; REM[U] *rdr, rs1, rs2* (*rdq* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

DIVW and DIVUW are RV64-only instructions that divide the lower 32 bits *rs1* by the lower 32 bits of *rs2*, treating them as signed and unsigned integers respectively, placing the 32-bit quotient in *rd*. REMW and REMUW are RV64-only instructions that provide the corresponding signed and unsigned remainder operations respectively.

6 Unconditional Jump/Branch

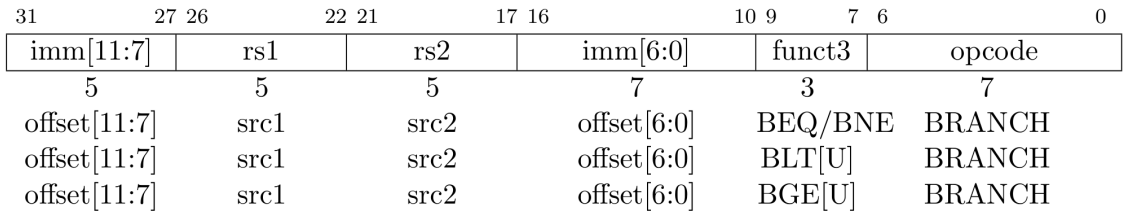
Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the `pc` to form the jump target address. Jumps can therefore target a ±32 MB range. JAL stores the address of the instruction following the jump (`pc`+4) into register `x1`.



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. It has three variants that are functionally identical but provide hints to the implementation: JALR.C is used to call subroutines; JALR.R is used to return from subroutines; and JALR.J is used for indirect jumps. The target address is obtained by sign-extending the 12-bit immediate then adding it to the address contained in register *rs1*. The address of the instruction following the jump (`pc`+4) is written to register *rd*. Register `x0` can be used as the destination if the result is not required.

7 Conditional Jump/Branch

All branch instructions use the B-type encoding. The 12-bit immediate is sign-extended, shifted left one bit, then added to the current `pc` to give the target address.



Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

31 27	26 22	21 17	16 15	14 12	11 10	9 8 7	6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Unimplemented Instruction

Control Transfer Instructions

imm25					1100111	J imm25
imm25					1101111	JAL imm25
imm12hi	rs1	rs2	imm12lo	000	1100011	BEQ rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	001	1100011	BNE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	100	1100011	BLT rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	101	1100011	BGE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	110	1100011	BLTU rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	111	1100011	BGEU rs1,rs2,imm12
rd	rs1	imm12		000	1101011	JALR.C rd,rs1,imm12
rd	rs1	imm12		001	1101011	JALR.R rd,rs1,imm12
rd	rs1	imm12		010	1101011	JALR.J rd,rs1,imm12
rd	00000	000000000000		100	1101011	RDNPC rd

Memory Instructions

rd	rs1	imm12		000	0000011	LB rd,rs1,imm12
rd	rs1	imm12		001	0000011	LH rd,rs1,imm12
rd	rs1	imm12		010	0000011	LW rd,rs1,imm12
rd	rs1	imm12		011	0000011	LD rd,rs1,imm12
rd	rs1	imm12		100	0000011	LBU rd,rs1,imm12
rd	rs1	imm12		101	0000011	LHU rd,rs1,imm12
rd	rs1	imm12		110	0000011	LWU rd,rs1,imm12
imm12hi	rs1	rs2	imm12lo	000	0100011	SB rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	001	0100011	SH rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	010	0100011	SW rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	011	0100011	SD rs1,rs2,imm12

Atomic Memory Instructions

rd	rs1	rs2	0000000	010	0101011	AMOADD.W rd,rs1,rs2
rd	rs1	rs2	0000001	010	0101011	AMOSWAP.W rd,rs1,rs2
rd	rs1	rs2	0000010	010	0101011	AMOAND.W rd,rs1,rs2
rd	rs1	rs2	0000011	010	0101011	AMOOR.W rd,rs1,rs2
rd	rs1	rs2	0000100	010	0101011	AMOMIN.W rd,rs1,rs2
rd	rs1	rs2	0000101	010	0101011	AMOMAX.W rd,rs1,rs2
rd	rs1	rs2	0000110	010	0101011	AMOMINU.W rd,rs1,rs2
rd	rs1	rs2	0000111	010	0101011	AMOMAXU.W rd,rs1,rs2
rd	rs1	rs2	0000000	011	0101011	AMOADD.D rd,rs1,rs2
rd	rs1	rs2	0000001	011	0101011	AMOSWAP.D rd,rs1,rs2
rd	rs1	rs2	0000010	011	0101011	AMOAND.D rd,rs1,rs2
rd	rs1	rs2	0000011	011	0101011	AMOOR.D rd,rs1,rs2
rd	rs1	rs2	0000100	011	0101011	AMOMIN.D rd,rs1,rs2
rd	rs1	rs2	0000101	011	0101011	AMOMAX.D rd,rs1,rs2
rd	rs1	rs2	0000110	011	0101011	AMOMINU.D rd,rs1,rs2
rd	rs1	rs2	0000111	011	0101011	AMOMAXU.D rd,rs1,rs2

31 27	26 22	21 17	16 15	14 12	11 10	9 8 7	6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Integer Compute Instructions

rd	rs1	imm12			000	0010011	ADDI rd,rs1,imm12
rd	rs1	000000	shamt		001	0010011	SLLI rd,rs1,shamt
rd	rs1	imm12			010	0010011	SLTI rd,rs1,imm12
rd	rs1	imm12			011	0010011	SLTIU rd,rs1,imm12
rd	rs1	imm12			100	0010011	XORI rd,rs1,imm12
rd	rs1	000000	shamt		101	0010011	SRLI rd,rs1,shamt
rd	rs1	000001	shamt		101	0010011	SRAI rd,rs1,shamt
rd	rs1	imm12			110	0010011	ORI rd,rs1,imm12
rd	rs1	imm12			111	0010011	ANDI rd,rs1,imm12
rd	rs1	rs2	0000000		000	0110011	ADD rd,rs1,rs2
rd	rs1	rs2	1000000		000	0110011	SUB rd,rs1,rs2
rd	rs1	rs2	0000000		001	0110011	SLL rd,rs1,rs2
rd	rs1	rs2	0000000		010	0110011	SLT rd,rs1,rs2
rd	rs1	rs2	0000000		011	0110011	SLTU rd,rs1,rs2
rd	rs1	rs2	0000000		100	0110011	XOR rd,rs1,rs2
rd	rs1	rs2	0000000		101	0110011	SRL rd,rs1,rs2
rd	rs1	rs2	1000000		101	0110011	SRA rd,rs1,rs2
rd	rs1	rs2	0000000		110	0110011	OR rd,rs1,rs2
rd	rs1	rs2	0000000		111	0110011	AND rd,rs1,rs2
rd	rs1	rs2	0000001		000	0110011	MUL rd,rs1,rs2
rd	rs1	rs2	0000001		001	0110011	MULH rd,rs1,rs2
rd	rs1	rs2	0000001		010	0110011	MULHSU rd,rs1,rs2
rd	rs1	rs2	0000001		011	0110011	MULHU rd,rs1,rs2
rd	rs1	rs2	0000001		100	0110011	DIV rd,rs1,rs2
rd	rs1	rs2	0000001		101	0110011	DIVU rd,rs1,rs2
rd	rs1	rs2	0000001		110	0110011	REM rd,rs1,rs2
rd	rs1	rs2	0000001		111	0110011	REMU rd,rs1,rs2
rd	imm20					0110111	LUI rd,imm20

32-bit Integer Compute Instructions

rd	rs1	imm12			000	0011011	ADDIW rd,rs1,imm12
rd	rs1	0000000	shamtw		001	0011011	SLLIW rd,rs1,shamtw
rd	rs1	0000000	shamtw		101	0011011	SRLIW rd,rs1,shamtw
rd	rs1	0000010	shamtw		101	0011011	SRAIW rd,rs1,shamtw
rd	rs1	rs2	0000000		000	0111011	ADDW rd,rs1,rs2
rd	rs1	rs2	1000000		000	0111011	SUBW rd,rs1,rs2
rd	rs1	rs2	0000000		001	0111011	SLLW rd,rs1,rs2
rd	rs1	rs2	0000000		101	0111011	SRLW rd,rs1,rs2
rd	rs1	rs2	1000000		101	0111011	SRAW rd,rs1,rs2
rd	rs1	rs2	0000001		000	0111011	MULW rd,rs1,rs2
rd	rs1	rs2	0000001		100	0111011	DIVW rd,rs1,rs2
rd	rs1	rs2	0000001		101	0111011	DIVUW rd,rs1,rs2
rd	rs1	rs2	0000001		110	0111011	REMW rd,rs1,rs2
rd	rs1	rs2	0000001		111	0111011	REMUW rd,rs1,rs2

31 27	26 22	21 17	16 15	14 12	11 10	9 8	7 6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Floating-Point Memory Instructions

rd	rs1	imm12		010	0000111	FLW rd,rs1,imm12
rd	rs1	imm12		011	0000111	FLD rd,rs1,imm12
imm12hi	rs1	rs2	imm12lo	010	0100111	FSW rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	011	0100111	FSD rs1,rs2,imm12

Floating-Point Compute Instructions

rd	rs1	rs2	00000	rm	00	1010011	FADD.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00001	rm	00	1010011	FSUB.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00010	rm	00	1010011	FMUL.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00011	rm	00	1010011	FDIV.S rd,rs1,rs2[,rm]
rd	rs1	00000	00100	rm	00	1010011	FSQRT.S rd,rs1[,rm]
rd	rs1	rs2	11000	000	00	1010011	FMIN.S rd,rs1,rs2
rd	rs1	rs2	11001	000	00	1010011	FMAX.S rd,rs1,rs2
rd	rs1	rs2	00000	rm	01	1010011	FADD.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00001	rm	01	1010011	FSUB.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00010	rm	01	1010011	FMUL.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00011	rm	01	1010011	FDIV.D rd,rs1,rs2[,rm]
rd	rs1	00000	00100	rm	01	1010011	FSQRT.D rd,rs1[,rm]
rd	rs1	rs2	11000	000	01	1010011	FMIN.D rd,rs1,rs2
rd	rs1	rs2	11001	000	01	1010011	FMAX.D rd,rs1,rs2
rd	rs1	rs2	rs3	rm	00	1000011	FMADD.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1000111	FMSUB.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1001011	FNMSUB.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1001111	FNMADD.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1000011	FMADD.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1000111	FMSUB.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1001011	FNMSUB.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1001111	FNMADD.D rd,rs1,rs2,rs3[,rm]

31 27	26 22	21 17	16 15	14 12	11 10	9 8 7	6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Floating-Point Move & Conversion Instructions

rd	rs1	rs2	00101	000	00	1010011	FSGNJ.S rd,rs1,rs2
rd	rs1	rs2	00110	000	00	1010011	FSGNJN.S rd,rs1,rs2
rd	rs1	rs2	00111	000	00	1010011	FSGNJX.S rd,rs1,rs2
rd	rs1	rs2	00101	000	01	1010011	FSGNJ.D rd,rs1,rs2
rd	rs1	rs2	00110	000	01	1010011	FSGNJN.D rd,rs1,rs2
rd	rs1	rs2	00111	000	01	1010011	FSGNJX.D rd,rs1,rs2
rd	rs1	00000	10001	rm	00	1010011	FCVT.S.D rd,rs1[,rm]
rd	rs1	00000	10000	rm	01	1010011	FCVT.D.S rd,rs1[,rm]

Integer to Floating-Point Move & Conversion Instructions

rd	rs1	00000	01100	rm	00	1010011	FCVT.S.L rd,rs1[,rm]
rd	rs1	00000	01101	rm	00	1010011	FCVT.S.LU rd,rs1[,rm]
rd	rs1	00000	01110	rm	00	1010011	FCVT.S.W rd,rs1[,rm]
rd	rs1	00000	01111	rm	00	1010011	FCVT.S.WU rd,rs1[,rm]
rd	rs1	00000	01100	rm	01	1010011	FCVT.D.L rd,rs1[,rm]
rd	rs1	00000	01101	rm	01	1010011	FCVT.D.LU rd,rs1[,rm]
rd	rs1	00000	01110	rm	01	1010011	FCVT.D.W rd,rs1[,rm]
rd	rs1	00000	01111	rm	01	1010011	FCVT.D.WU rd,rs1[,rm]
rd	rs1	00000	11110	000	00	1010011	MXTF.S rd,rs1
rd	rs1	00000	11110	000	01	1010011	MXTF.D rd,rs1
rd	rs1	00000	11111	000	00	1010011	MTFSR rd,rs1

Floating-Point to Integer Move & Conversion Instructions

rd	rs1	00000	01000	rm	00	1010011	FCVT.L.S rd,rs1[,rm]
rd	rs1	00000	01001	rm	00	1010011	FCVT.LU.S rd,rs1[,rm]
rd	rs1	00000	01010	rm	00	1010011	FCVT.W.S rd,rs1[,rm]
rd	rs1	00000	01011	rm	00	1010011	FCVT.WU.S rd,rs1[,rm]
rd	rs1	00000	01000	rm	01	1010011	FCVT.L.D rd,rs1[,rm]
rd	rs1	00000	01001	rm	01	1010011	FCVT.LU.D rd,rs1[,rm]
rd	rs1	00000	01010	rm	01	1010011	FCVT.W.D rd,rs1[,rm]
rd	rs1	00000	01011	rm	01	1010011	FCVT.WU.D rd,rs1[,rm]
rd	00000	rs2	11100	000	00	1010011	MFTX.S rd,rs2
rd	00000	rs2	11100	000	01	1010011	MFTX.D rd,rs2
rd	00000	00000	11101	000	00	1010011	MFFSR rd

31 27	26 22	21 17	16 15	14 12	11 10	9 8	7 6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Floating-Point Compare Instructions

rd	rs1	rs2	10101	000	00	1010011	FEQ.S rd,rs1,rs2
rd	rs1	rs2	10110	000	00	1010011	FLT.S rd,rs1,rs2
rd	rs1	rs2	10111	000	00	1010011	FLE.S rd,rs1,rs2
rd	rs1	rs2	10101	000	01	1010011	FEQ.D rd,rs1,rs2
rd	rs1	rs2	10110	000	01	1010011	FLT.D rd,rs1,rs2
rd	rs1	rs2	10111	000	01	1010011	FLE.D rd,rs1,rs2

Miscellaneous Memory Instructions

rd	rs1	imm12	001	0101111	FENCE.I rd,rs1,imm12
rd	rs1	imm12	010	0101111	FENCE rd,rs1,imm12

System Instructions

00000	00000	00000	0000000	000	1110111	SYSCALL
00000	00000	00000	0000000	001	1110111	BREAK
rd	00000	00000	0000000	100	1110111	RDCYCLE rd
rd	00000	00000	0000001	100	1110111	RDTIME rd
rd	00000	00000	0000010	100	1110111	RDINSTRET rd

Table 6: Instruction listing for RISC-V