

Instruction Set Architecture

George Herbert

February 13, 2023

Abstract

This document outlines the instruction set architecture (ISA) of the processor simulator. The ISA is a subset of the RISC-V Unprivileged ISA, with several changes. Many of the sections in this document very closely correspond to the sections in ‘RISC-V Instruction Set Manual: Volume I: Unprivileged ISA’. While this is deliberate for ease of reference, it is not intended to imply that the ISA is identical.

1 RV32I Version 2.1

1.1 Registers

For RV32I, the 32 x registers are each 32 bits wide (i.e. $XLEN=32$). Register $x0$ is hardwired with all bits equal to 0. General purpose registers $x1$ - $x31$ hold values that various instructions interpret as a collection of Boolean values, or as two’s complement signed binary integers or unsigned binary integers.

There is one additional unprivileged register: the program counter pc holds the address of the current instruction.

1.2 Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U). All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. Immediates are always sign-extended.

RISC-V base instruction formats, with corresponding numbers of bits for each field in brackets:

- R-type: opcode (7), rd (5), funct3 (3), rs1 (5), rs2 (5), funct7 (7)
- I-type: opcode (7), rd (5), funct3 (3), rs1 (5), imm (12)
- S-type: opcode (7), funct3 (3), rs1 (5), rs2 (5), imm (12)
- U-type: opcode(7), rd(5), imm(20)

1.3 Integer Computational Instructions

Most integer computational instructions operate on $XLEN$ bits of values held in the integer register file. Integer computational instructions are

either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format.

1.3.1 Integer Register-Immediate Instructions

Table 1: Integer Register-Immediate Instructions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
I-immediate												src			ADDI [000]			dest			OP-IMM [0010011]			I-type								
I-immediate												src			SLTI [010]			dest			OP-IMM [0010011]			I-type								
I-immediate												src			SLTIU [011]			dest			OP-IMM [0010011]			I-type								
I-immediate												src			ANDI [111]			dest			OP-IMM [0010011]			I-type								
I-immediate												src			ORI [110]			dest			OP-IMM [0010011]			I-type								
I-immediate												src			XORI [100]			dest			OP-IMM [0010011]			I-type								
0000000					shamt					src			SLLI [001]			dest			OP-IMM [0010011]			I-type										
0000000					shamt					src			SRLI [101]			dest			OP-IMM [0010011]			I-type										
0100000					shamt					src			SRAI [101]			dest			OP-IMM [0010011]			I-type										
U-immediate																			dest			LUI [0110111]			U-type							
U-immediate																			dest			AUIPC [0010111]			U-type							

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

SLTI (set less than immediate) places the value 1 in the 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 (i.e. the immediate is first sign-extended to XLEN bits then treated as an unsigned number).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR and XOR on registers *rs1* and the sign-extended 12-bit immediate and place the result in *rd*.

Shifts by a constant are encoded as a specialisation of the I-type format. The operand to be shifted is in *rs1* and the shift amount is encoded in the lower 5 bits of the I-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).

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20th-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

1.3.2 Integer Register-Register Operations

RV32I 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 *funct7* and *funct3* fields select the type of operation.

Table 2: Integer Register-Register Operations

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0000000		src2		src1		ADD		[000]		dest		OP		[0110011]		R-type																
0000000		src2		src1		SLT		[010]		dest		OP		[0110011]		R-type																
0000000		src2		src1		SLTU		[011]		dest		OP		[0110011]		R-type																
0000000		src2		src1		AND		[111]		dest		OP		[0110011]		R-type																
0000000		src2		src1		OR		[110]		dest		OP		[0110011]		R-type																
0000000		src2		src1		XOR		[100]		dest		OP		[0110011]		R-type																
0000000		src2		src1		SLL		[001]		dest		OP		[0110011]		R-type																
0000000		src2		src1		SRL		[101]		dest		OP		[0110011]		R-type																
0100000		src2		src1		SUB		[000]		dest		OP		[0110011]		R-type																
0100000		src2		src1		SRA		[101]		dest		OP		[0110011]		R-type																

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of the results are written to the destination *rd*. 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.

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 the lower 5 bits of register *rs2*.

1.4 Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches.

1.4.1 Unconditional Jumps

Table 3: Unconditional Jumps

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																		offset			dest		JAL	[1101111]	U-type						
										offset			base			000		dest		JALR	[1100111]	I-type									

The jump and link (JAL) instruction uses the U-type format, where the U-immediate encodes a signed offset in multiples of 2 bytes. The offset is

sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump ($pc + 4$) into register rd . The standard software calling convention uses $x1$ as the return address register and $x5$ as an alternate link register.

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register $rs1$, then setting the least-significant bit of the result to zero. The address of the instruction following the jump ($pc + 4$) is written to register rd .

1.4.2 Conditional Branches

Table 4: Conditional Branches

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
offset												src2				src1				BEQ [000]			BRANCH [1100011]				S-type					
offset												src2				src1				BNE [001]			BRANCH [1100011]				S-type					
offset												src2				src1				BLT [100]			BRANCH [1100011]				S-type					
offset												src2				src1				BLTU [110]			BRANCH [1100011]				S-type					
offset												src2				src1				BGE [101]			BRANCH [1100011]				S-type					
offset												src2				src1				BGEU [111]			BRANCH [1100011]				S-type					

All branch instructions use the S-type instruction format. The 12-bit S-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ± 4 KiB.

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 synthesised by reversing the operands to BLT, BLTU, BGE and BGEU, respectively.

2 Load and Store Instructions

Table 5: Load and Store Instructions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
offset												base				LW [010]		dest				LOAD		0000011								I-type
offset												base				LH [001]		dest				LOAD		0000011								I-type
offset												base				LHU [101]		dest				LOAD		0000011								I-type
offset												base				LB [000]		dest				LOAD		0000011								I-type
offset												base				LBU [100]		dest				LOAD		0000011								I-type
offset												src				base				SW [010]		STORE		0100011								S-type
offset												src				base				SH [001]		STORE		0100011								S-type
offset												src				base				SB [000]		STORE		0100011								S-type

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed.

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding registers *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends it to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends it to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH and SB instructions store 32-bit, 16-bit and 8-bit values from the low bits of register *rs2* to memory.