Chapter 7

RV64I Base Integer Instruction Set, Version 2.1

This chapter describes the RV64I base integer instruction set, which builds upon the RV32I variant described in Chapter 2. This chapter presents only the differences with RV32I, so should be read in conjunction with the earlier chapter.

7.1 Register State

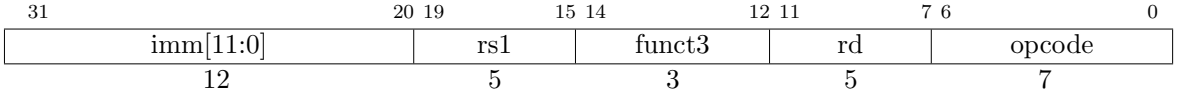
RV64I widens the integer registers and supported user address space to 64 bits (XLEN=64 in Figure 2.1).

7.2 Integer Computational Instructions

Most integer computational instructions operate on XLEN-bit values. Additional instruction variants are provided to manipulate 32-bit values in RV64I, indicated by a 'W' suffix to the opcode. These "*W" instructions ignore the upper 32 bits of their inputs and always produce 32-bit signed values, sign-extending them to 64 bits, i.e. bits XLEN-1 through 31 are equal.

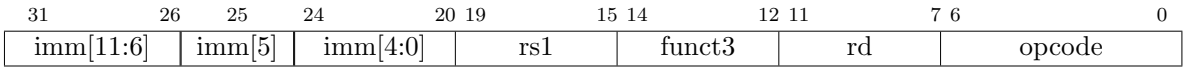
The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion from a signed 32-bit integer to a signed 64-bit integer. Existing 64-bit wide SLTU and unsigned branch compares still operate correctly on unsigned 32-bit integers under this invariant. Similarly, existing 64-bit wide logical operations on 32-bit sign-extended integers preserve the sign-extension property. A few new instructions (ADD[I]W/SUBW/SxxW) are required for addition and shifts to ensure reasonable performance for 32-bit values.

Integer Register-Immediate Instructions



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

ADDIW is an RV64I instruction that adds the sign-extended 12-bit immediate to register *rs1* and produces the proper sign-extension of a 32-bit result in *rd*. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW *rd, rs1, 0* writes the sign-extension of the lower 32 bits of register *rs1* into register *rd* (assembler pseudoinstruction SEXT.W).

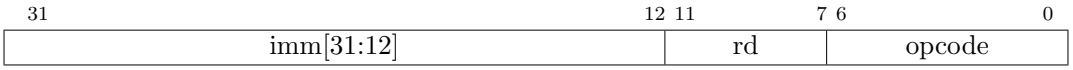


31 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

Shifts by a constant are encoded as a specialization of the I-type format using the same instruction opcode as RV32I. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the I-immediate field for RV64I. The right shift type is encoded in bit 30. 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).

SLLIW, SRLIW, and SRAIW are RV64I-only instructions that are analogously defined but operate on 32-bit values and sign-extend their 32-bit results to 64 bits. SLLIW, SRLIW, and SRAIW encodings with $imm[5] \neq 0$ are reserved.

Previously, SLLIW, SRLIW, and SRAIW with $imm[5] \neq 0$ were defined to cause illegal instruction exceptions, whereas now they are marked as reserved. This is a backwards-compatible change.



31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 32-bit U-immediate into register *rd*, filling in the lowest 12 bits with zeros. The 32-bit result is sign-extended to 64 bits.

AUIPC (add upper immediate to `pc`) uses the same opcode as RV32I. AUIPC is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, sign-extends the result to 64 bits, adds it to the address of the AUIPC instruction, then places the result in register *rd*.

Note that the set of address offsets that can be formed by pairing LUI with LD, AUIPC with JALR, etc. in RV64I is [$-2^{31}-2^{11}$, $2^{31}-2^{11}-1$].

Integer Register-Register Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

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 RV64I, only the low 6 bits of *rs2* are considered for the shift amount.

SLLW, SRLW, and SRAW are RV64I-only instructions that are analogously defined but operate on 32-bit values and sign-extend their 32-bit results to 64 bits. The shift amount is given by *rs2[4:0]*.

7.3 Load and Store Instructions

RV64I extends the address space to 64 bits. The execution environment will define what portions of the address space are legal to access.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

The LD instruction loads a 64-bit value from memory into register *rd* for RV64I.

The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register *rd* for RV64I. The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64I. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory respectively.

7.4 HINT Instructions

All instructions that are microarchitectural HINTs in RV32I (see Section 2.9) are also HINTs in RV64I. The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.

Table 7.1 lists all RV64I HINT code points. 91% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs: no standard HINTs will ever be defined in this subspace.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADDIW	*rd*=x0	2^{17}	
ADD	*rd*=x0, *rs1*≠x0	$2^{10}-32$	
ADD	*rd*=x0, *rs1*=x0, *rs2*≠x2–x5	28	
ADD	*rd*=x0, *rs1*=x0, *rs2*=x2–x5	4	(*rs2*=x2) NTL.P1 (*rs2*=x3) NTL.PALL (*rs2*=x4) NTL.S1 (*rs2*=x5) NTL.ALL
SUB	*rd*=x0	2^{10}	*Reserved for future standard use*
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
ADDW	*rd*=x0	2^{10}	
SUBW	*rd*=x0	2^{10}	
SLLW	*rd*=x0	2^{10}	
SRLW	*rd*=x0	2^{10}	
SRAW	*rd*=x0	2^{10}	
FENCE	*rd*=x0, *rs1*≠x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*≠x0, *rs1*=x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=0, *succ*≠0	15	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*≠W, *succ*=0	15	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=W, *succ*=0	1	PAUSE
SLTI	*rd*=x0	2^{17}	*Designated for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{11}	
SRLI	*rd*=x0	2^{11}	
SRAI	*rd*=x0	2^{11}	
SLLIW	*rd*=x0	2^{10}	
SRLIW	*rd*=x0	2^{10}	
SRAIW	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

Table 7.1: RV64I HINT instructions.