

2.2 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

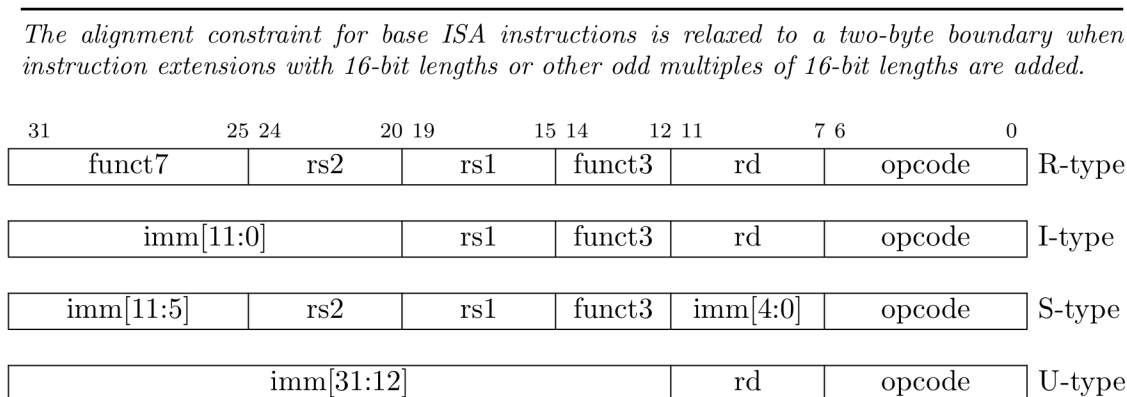


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction’s immediate field as is usually done.

The RISC-V ISA keeps the source ($rs1$ and $rs2$) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Section 2.8), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR [18]).

In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load upper immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

Immediates are sign-extended because we did not observe a benefit to using zero-extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.

2.3 Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 2.3.

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits ($\text{imm}[10:1]$) and sign bit stay in fixed positions, while the lowest bit in S format ($\text{inst}[7]$) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

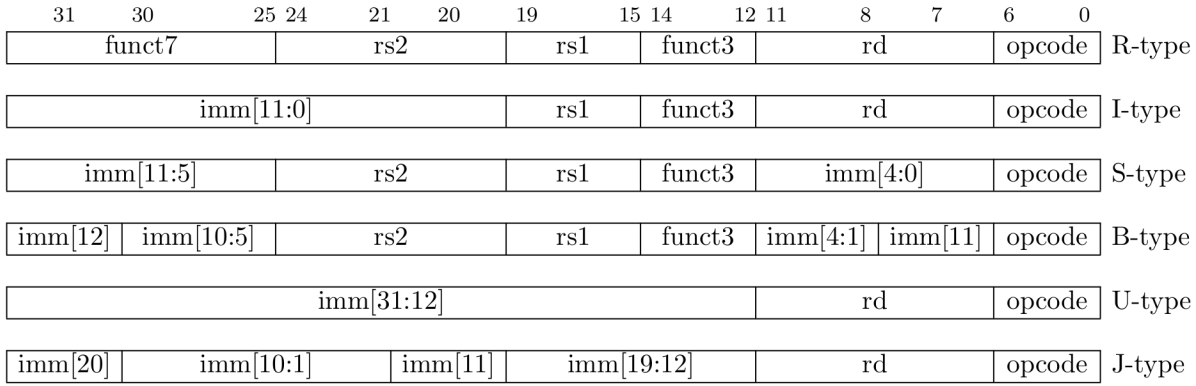


Figure 2.3: RISC-V base instruction formats showing immediate variants.

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit ($\text{inst}[y]$) produces each bit of the immediate value.

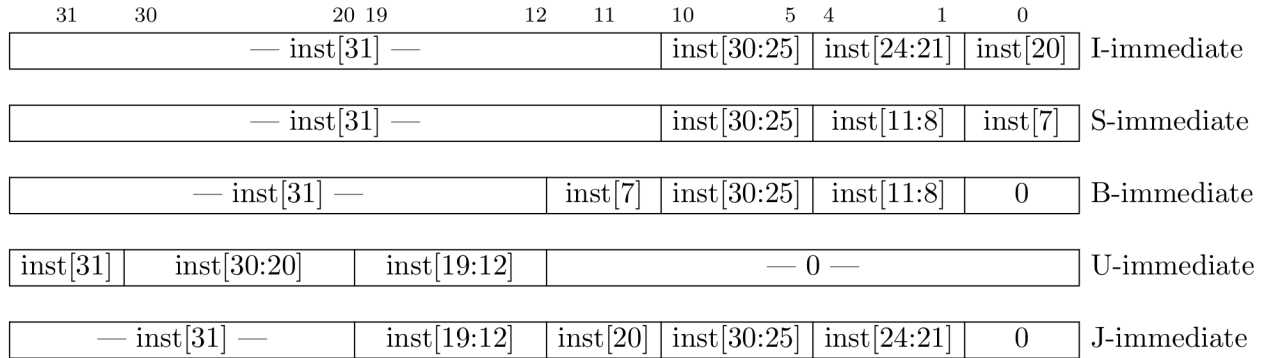


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses $\text{inst}[31]$.

Sign-extension is one of the most critical operations on immediates (particularly in RV64I), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across

types of instruction, we wanted to reduce the hardware cost of the simplest implementations. By rotating bits in the instruction encoding of *B* and *J* immediates instead of using dynamic hardware muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

2.4 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. The destination is register *rd* for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

We did not include special instruction set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.

For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. This covers the common case of addition with an immediate operand.

For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

In RV64, checks of 32-bit signed additions can be optimized further by comparing the results of `ADD` and `ADDW` on the operands.

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	ADDI/SLTI[U]	dest	OP-IMM	
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

`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. `ADDI rd, rs1, 0` is used to implement the `MV rd, rs1` assembler pseudo-instruction.

`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 (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1, 1* sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudo-op SEQZ *rd, rs*).

ANDI, ORI, XORI are logical operations that perform bitwise 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 bitwise logical inversion of register *rs1* (assembler pseudo-instruction NOT *rd, rs*).

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM	
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM	
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM	

Shifts by a constant are 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 5 bits of the I-immediate field. The right shift type is encoded in a high bit of the I-immediate. 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).

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) 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 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register *rd*.

The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.

The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the PC, it might cause pipeline breaks in simpler microarchitectures or pollute the BTB structures in more complex microarchitectures.

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.

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	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudo-op SNEZ *rd*, *rs*). 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*.

NOP Instruction

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
0	0	ADDI	0	OP-IMM	

The NOP instruction does not change any user-visible state, except for advancing the pc. NOP is encoded as ADDI *x0*, *x0*, 0.

NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as for more readable disassembly output.

2.5 Control Transfer Instructions

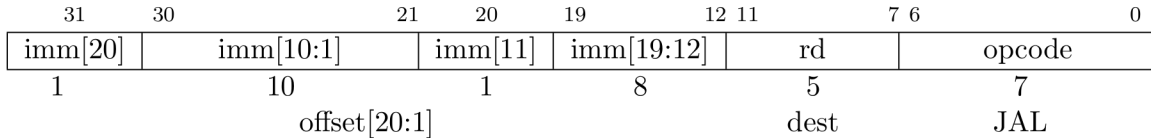
RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do *not* have architecturally visible delay slots.

Unconditional Jumps

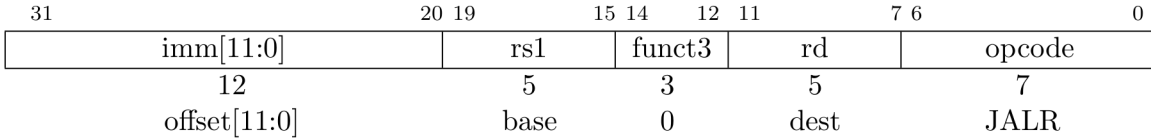
The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following 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 alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register `x5` was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.

Plain unconditional jumps (assembler pseudo-op `J`) are encoded as a `JAL` with `rd=x0`.



The indirect jump instruction `JALR` (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed 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`. Register `x0` can be used as the destination if the result is not required.



The unconditional jump instructions all use PC-relative addressing to help support position-independent code. The `JALR` instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit absolute address range. A `LUI` instruction can first load `rs1` with the upper 20 bits of a target address, then `JALR` can add in the lower bits. Similarly, `AUIPC` then `JALR` can jump anywhere in a 32-bit `pc`-relative address range.

Note that the `JALR` instruction does not treat the 12-bit immediate as multiples of 2 bytes, unlike the conditional branch instructions. This avoids one more immediate format in hardware. In practice, most uses of `JALR` will have either a zero immediate or be paired with a `LUI` or `AUIPC`, so the slight reduction in range is not significant.

The `JALR` instruction ignores the lowest bit of the calculated target address. This both simplifies the hardware slightly and allows the low bit of function pointers to be used to store auxiliary information. Although there is potentially a slight loss of error checking in this case, in practice jumps to an incorrect instruction address will usually quickly raise an exception.

When used with a base `rs1=x0`, `JALR` can be used to implement a single instruction subroutine call to the lowest 2 KiB or highest 2 KiB address region from anywhere in the address space, which could be used to implement fast calls to a small runtime library.

The `JAL` and `JALR` instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.

Instruction fetch misaligned exceptions are not possible on machines that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension, C.

Return-address prediction stacks are a common feature of high-performance instruction-fetch units, but require accurate detection of instructions used for procedure calls and returns to be effective. For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. A `JAL` instruction should push the return address onto a return-address stack (RAS) only when `rd=x1/x5`. `JALR` instructions should push/pop a RAS as shown in the Table 2.1.

<i>rd</i>	<i>rs1</i>	<i>rs1=rd</i>	RAS action
<i>!link</i>	<i>!link</i>	-	none
<i>!link</i>	<i>link</i>	-	pop
<i>link</i>	<i>!link</i>	-	push
<i>link</i>	<i>link</i>	0	push and pop
<i>link</i>	<i>link</i>	1	push

Table 2.1: Return-address stack prediction hints encoded in register specifiers used in the instruction. In the above, *link* is true when the register is either **x1** or **x5**.

Some other ISAs added explicit hint bits to their indirect-jump instructions to guide return-address stack manipulation. We use implicit hinting tied to register numbers and the calling convention to reduce the encoding space used for these hints.

*When two different link registers (**x1** and **x5**) are given as *rs1* and *rd*, then the RAS is both pushed and popped to support coroutines. If *rs1* and *rd* are the same link register (either **x1** or **x5**), the RAS is only pushed to enable macro-op fusion of the sequences: `lui ra, imm20; jalr ra, ra, imm12` and `auipc ra, imm20; jalr ra, ra, imm12`*

Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current **pc** to give the target address. The conditional branch range is ± 4 KiB.

31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode		
1	6	5	5	3	4	1	7		
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH		
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH		
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH		

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.

Signed array bounds may be checked with a single BLTU instruction, since any negative index will compare greater than any nonnegative bound.

Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

Unlike some other architectures, the RISC-V jump (JAL with $rd=x0$) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always-true condition. RISC-V jumps are also PC-relative and support a much wider offset range than branches, and will not pressure conditional branch prediction tables.

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC and Xtensa ISA), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Another point is that comparisons against zero require non-trivial circuit delay (especially after the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier. There is perhaps an advantage to a design with condition codes in the case where multiple branches can be taken based on the same condition codes, but we believe this case to be relatively rare.

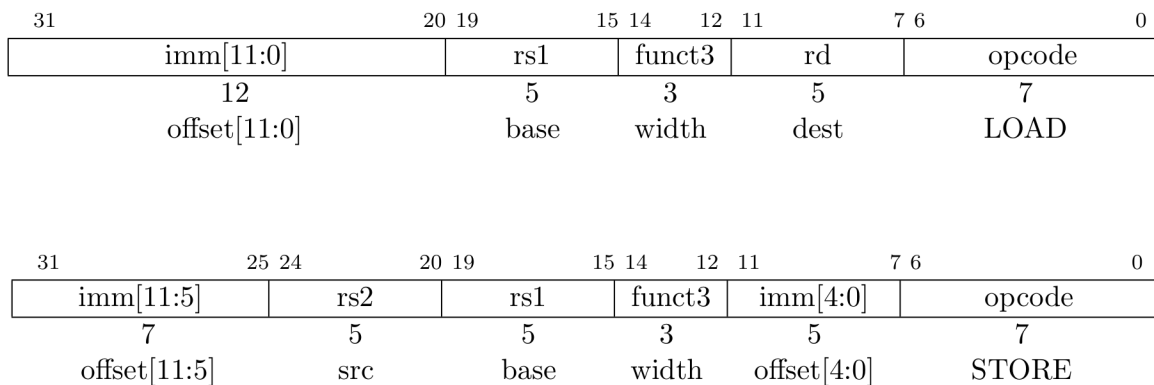
We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs.

We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point operations). Predication adds additional flag state to a system, additional instructions to set and clear flags, and additional encoding overhead on every instruction. Both conditional move and predicated instructions add complexity to out-of-order microarchitectures, adding an implicit third source operand due to the need to copy the original value of the destination architectural register into the renamed destination physical register if the predicate is false. Also, static compile-time decisions to use predication instead of branches can result in lower performance on inputs not included in the compiler training set, especially given that unpredictable branches are rare, and becoming rarer as branch prediction techniques improve.

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict [13, 17, 16] and have been implemented in commercial processors [27]. The simplest techniques just reduce the penalty of recovering from a mispredicted short forward branch by only flushing instructions in the branch shadow instead of the entire fetch pipeline, or by fetching instructions from both sides using wide instruction fetch or idle instruction fetch slots. More complex techniques for out-of-order cores add internal predicates on instructions in the branch shadow, with the internal predicate value written by the branch instruction, allowing the branch and following instructions to be executed speculatively and out-of-order with respect to other code [27].

2.6 Load and Store Instructions

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 user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access. Loads with a destination of $x0$ must still raise any exceptions and action any other side effects even though the load value is discarded.



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 byte address is obtained by adding register *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 to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends 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.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

Misaligned accesses are occasionally required when porting legacy code, and are essential for good performance on many applications when using any form of packed-SIMD extension. Our rationale for supporting misaligned accesses via the regular load and store instructions is to simplify the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISA and then provide some separate ISA support for misaligned accesses, either special instructions to help software handle misaligned accesses or a new hardware addressing mode for misaligned accesses. Special instructions are difficult to use, complicate the ISA, and often add new processor state (e.g., SPARC VIS align address offset register) or complicate access to existing processor state (e.g., MIPS LWL/LWR partial register writes). In addition, for loop-oriented packed-SIMD code, the extra overhead when operands are misaligned motivates software to provide multiple forms of loop depending on operand alignment, which complicates code generation and adds to loop startup overhead. New misaligned hardware addressing modes take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).

We do not mandate atomicity for misaligned accesses so simple implementations can just use a machine trap and software handler to handle some or all misaligned accesses. If hardware misaligned support is provided, software can exploit this by simply using regular load and store instructions. Hardware can then automatically optimize accesses depending on whether runtime addresses are aligned.

2.7 Memory Model

This section is out of date as the RISC-V memory model is currently under revision to ensure it can efficiently support current programming language memory models. The revised base memory model will contain further ordering constraints, including at least that loads to the same address from the same hart cannot be reordered, and that syntactic data dependencies between instructions are respected.

The base RISC-V ISA supports multiple concurrent threads of execution within a single user address space. Each RISC-V hardware thread, or *hart*, has its own user register state and program counter, and executes an independent sequential instruction stream. The execution environment will define how RISC-V harts are created and managed. RISC-V harts can communicate and synchronize with other harts either via calls to the execution environment, which are documented separately in the specification for each execution environment, or directly via the shared memory system. RISC-V harts can also interact with I/O devices, and indirectly with each other, via loads and stores to portions of the address space assigned to I/O.

We use the term hart to unambiguously and concisely describe a hardware thread as opposed to software-managed thread contexts.

In the base RISC-V ISA, each RISC-V hart observes its own memory operations as if they executed sequentially in program order. RISC-V has a relaxed memory model between harts, requiring an explicit FENCE instruction to guarantee ordering between memory operations from different RISC-V harts. Chapter 7 describes the optional atomic memory instruction extensions “A”, which provide additional synchronization operations.

31	28	27	26	25	24	23	22	21	20	19	15	14	12	11	7	6	0
0	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode					
4	1	1	1	1	1	1	1	1	5	3	5	7					
0	predecessor					successor				0	FENCE	0	MISC-MEM				

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 execution environment will define what I/O operations are possible, and in particular, which load and store instructions might be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new coprocessor I/O instructions that will also be ordered using the I and O bits in a FENCE.

The unused fields in the FENCE instruction, *imm[11:8]*, *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.

We chose a relaxed memory model to allow high performance from simple machine implementations, however a completely relaxed memory model is too weak to support programming language memory models and so the memory model is being tightened.

A relaxed memory model is also most compatible with likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative non-memory paths to control added coprocessors or I/O devices. Simple implementations may additionally ignore the predecessor and successor fields and always execute a conservative fence on all operations.

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
0	0	FENCE.I	0	MISC-MEM	

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 the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only 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 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.

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, then only the pipeline needs to be flushed at a FENCE.I.

We considered but did not include a “store instruction word” instruction (as in MAJC [30]). 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.

2.8 Control and Status Register Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in this section, with the two other user-level SYSTEM instructions described in the following section.

The SYSTEM instructions are defined to allow simpler implementations to always trap to a single software trap handler. More sophisticated implementations might execute more of each system instruction in hardware.

CSR Instructions

We define the full set of CSR instructions here, although in the standard user-level base ISA, only a handful of read-only counter CSRs are accessible.

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

For both CSRRS and CSRRC, if *rs1*=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if *rs1* specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the *rs1* field instead of a value from an integer

register. For CSRRSI and CSRRCI, if the `uimm[4:0]` field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if `rd=x0`, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

Some CSRs, such as the instructions retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes a CSR, the update occurs after the execution of the instruction. In particular, a value written to `instret` by one instruction will be the value read by the following instruction (i.e., the increment of `instret` caused by the first instruction retiring happens before the write of the new value).

The assembler pseudo-instruction to read a CSR, `CSRR rd, csr`, is encoded as `CSRRS rd, csr, x0`. The assembler pseudo-instruction to write a CSR, `CSRW csr, rs1`, is encoded as `CSRRW x0, csr, rs1`, while `CSRWI csr, uimm`, is encoded as `CSRRWI x0, csr, uimm`.

Further assembler pseudo-instructions are defined to set and clear bits in the CSR when the old value is not required: `CSRS/CSRC csr, rs1`; `CSRSI/CSRCI csr, uimm`.

Timers and Counters

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
RDCYCLE[H]	0	CSRRS	dest	SYSTEM	
RDTIME[H]	0	CSRRS	dest	SYSTEM	
RDINSTRET[H]	0	CSRRS	dest	SYSTEM	

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12-bit CSR address space and accessed in 32-bit pieces using CSRRS instructions.

The RDCYCLE pseudo-instruction reads the low XLEN bits of the `cycle` CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. RDCYCLEH is an RV32I-only instruction that reads bits 63–32 of the same cycle counter. The underlying 64-bit counter should never overflow in practice. The rate at which the cycle counter advances will depend on the implementation and operating environment. The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

The RDTIME pseudo-instruction reads the low XLEN bits of the `time` CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is an RV32I-only instruction that reads bits 63–32 of the same real-time counter. The underlying 64-bit counter should never overflow in practice. The execution environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant. The real-time clocks of all harts in a single user application should be synchronized to within one tick of the real-time clock. The environment should provide a means to determine the accuracy of the clock.

The RDINSTRET pseudo-instruction reads the low XLEN bits of the `instret` CSR, which counts

the number of instructions retired by this hart from some arbitrary start point in the past. RDINSTRETH is an RV32I-only instruction that reads bits 63–32 of the same instruction counter. The underlying 64-bit counter that should never overflow in practice.

The following code sequence will read a valid 64-bit cycle counter value into `x3:x2`, even if the counter overflows between reading its upper and lower halves.

```
again:
    rdcycleh    x3
    rdcycle     x2
    rdcycleh    x4
    bne         x3, x4, again
```

Figure 2.5: Sample code for reading the 64-bit cycle counter in RV32.

We mandate these basic counters be provided in all implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

We required the counters be 64 bits wide, even on RV32, as otherwise it is very difficult for software to determine if values have overflowed. For a low-end implementation, the upper 32 bits of each counter can be implemented using software counters incremented by a trap handler triggered by overflow of the lower 32 bits. The sample code described above shows how the full 64-bit width value can be safely read using the individual 32-bit instructions.

In some applications, it is important to be able to read multiple counters at the same instant in time. When run under a multitasking environment, a user thread can suffer a context switch while attempting to read the counters. One solution is for the user thread to read the real-time counter before and after reading the other counters to determine if a context switch occurred in the middle of the sequence, in which case the reads can be retried. We considered adding output latches to allow a user thread to snapshot the counter values atomically, but this would increase the size of the user context, especially for implementations with a richer set of counters.

2.9 Environment Call and Breakpoints

31	20 19	15 14	12 11	7 6	0
funct12	rs1	funct3	rd	opcode	
12	5	3	5	7	
ECALL	0	PRIV	0	SYSTEM	
EBREAK	0	PRIV	0	SYSTEM	

The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment.

ECALL and EBREAK were previously named SCALL and SBREAK. The instructions have

the same functionality and encoding, but were renamed to reflect that they can be used more generally than to call a supervisor-level operating system or debugger.