ECE 552 RISC-V ISA Specification (WISC-F25)

Authors: ECE 552 Staff

Version v0.0.0, 2025-10-12: Draft

Chapter 1. Introduction

The WISC-F25 architecture you will implement for this course is based on the 32-bit RISC-V base instruction set (RV32I). RISC-V is an open source instruction set originally developed at UC Berkeley for research, and has since seen increasing industry adoption. While all core instructions specified in this document are compatible with the standard RISC-V ISA, a few instructions have been removed to simplify the implementation. Additionally, a couple of custom instructions have been added, which don't interfere with any existing instructions.

Through the course, you will build up an in-order, 5 stage pipelined processor that implements this instruction set architecture (ISA). However, microarchitectural details are not specified in this document - giving you flexibility in how you implement the processor, as long as it conforms to the instruction specifications listed here.

While this document should be all you need to implement a WISC-F25 compatible processor, it has been intentionally simplified from the official RISC-V specification. For those looking for more details or interested to learn more, we recommend reading the official specification at drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link. It's very long, but the most relevant parts are chapters 1, 2, and 35.

1.1. Datapath and High-Level Architecture

As the name implies, RV32I operates on 32-bit words stored in registers. Additionally, each instruction is 32 bits wide (this is also true in RV64I, a similar ISA that operates on 64-bit words). While many extensions exist for advanced features, the base ISA consists of only 37 instructions that operate on integers. While the size of the program counter is not important (and effectively depends on the size of memory), it can be assumed to be 32 bits wide for simplicity.

1.2. Register File

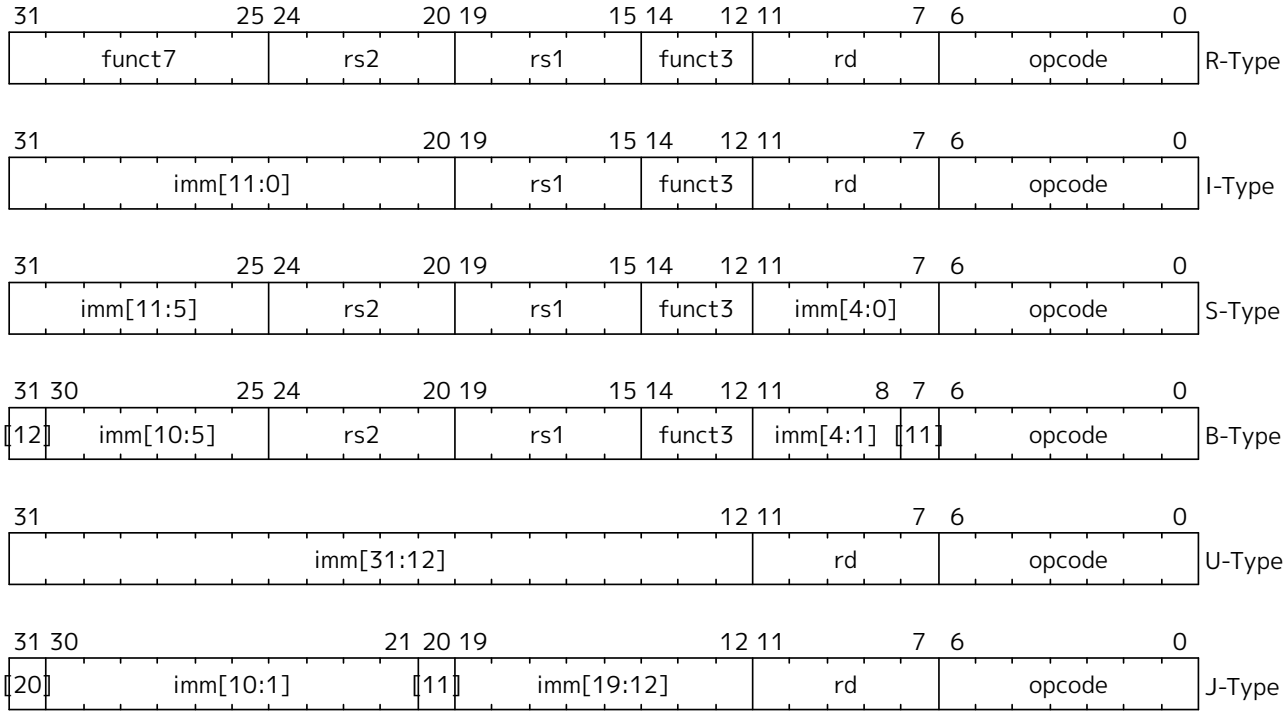
RV32I has 32 general-purpose registers labeled `x0` to `x31`, each 32 bits wide. `x0` is hardwired to zero (and is therefore also referred to as `zero`). While software convention assigns some registers to functions (`x1` is the link/return address register, `x2` is the stack pointer) these are not hardcoded in the ISA. The program counter is separate from the general purpose register file.

1.3. Memory Model

RV32I is a load/store architecture, meaning that all reads and writes to memory are performed through explicit load and store instructions. Memory is byte-addressable and can be accessed in 8-bit (byte), 16-bit (half-word), and 32-bit (word) chunks. RV32I is little endian, both in the way data is stored to memory and the way instructions are encoded.

Chapter 2. Instruction Formats

RV32I has 6 instruction formats used to encode different types of instructions. There are 4 primary formats (R-type, I-type, S-type, and U-type) which encode different operands and 2 additional formats (B-type and J-type) which differ only based on how they encode the immediate. Overall, RV32I has been designed to be very simple to decode, as we will see below.



2.1. Opcodes

All instructions have a 7 bit opcode field in the 7 least significant bits of the instruction. This opcode is used to determine the major instruction function and select the appropriate instruction format (note that the location of the opcode does not depend on the format). While this means there are 128 possible opcodes, only a small fraction of them are used in WISC-F25.

You may notice that all the opcodes listed in Chapter 3 end in 11 *in the two least significant bits. This is because these bits are used to indicate an alternate 16-bit compressed encoding in the compressed extensions (C, Zc*). WISC-F25 does not implement any compression and therefore all instructions are 32 bits long and end in* 11.

2.2. Source Operands

Instructions read up to two source registers, called *rs1* and *rs2*, and optionally write to a destination register *rd*. As there are 32 general purpose registers, these indices are encoded in 5 bits each. When the source register is present in the format, it is always in the same place - making it very easy to fetch operands from the register file without decoding the instruction format.

2.3. Immediates

Instructions also contain immediate values. These are usually 12 bits, but are larger in a couple of formats (U-type and J-type) to allow loading larger constants or jumping to distant PC-relative addresses. All immediates are sign-extended to 32 bits before being used. Not all instruction formats

start at bit 0 - as you'll see in Chapter 3, the jump instructions store 2-byte aligned offsets and the U-type format only loads the upper 20 bits of the immediate, setting the lower 12 bits to zero.

Immediate types shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit produces each bit of the immediate value.

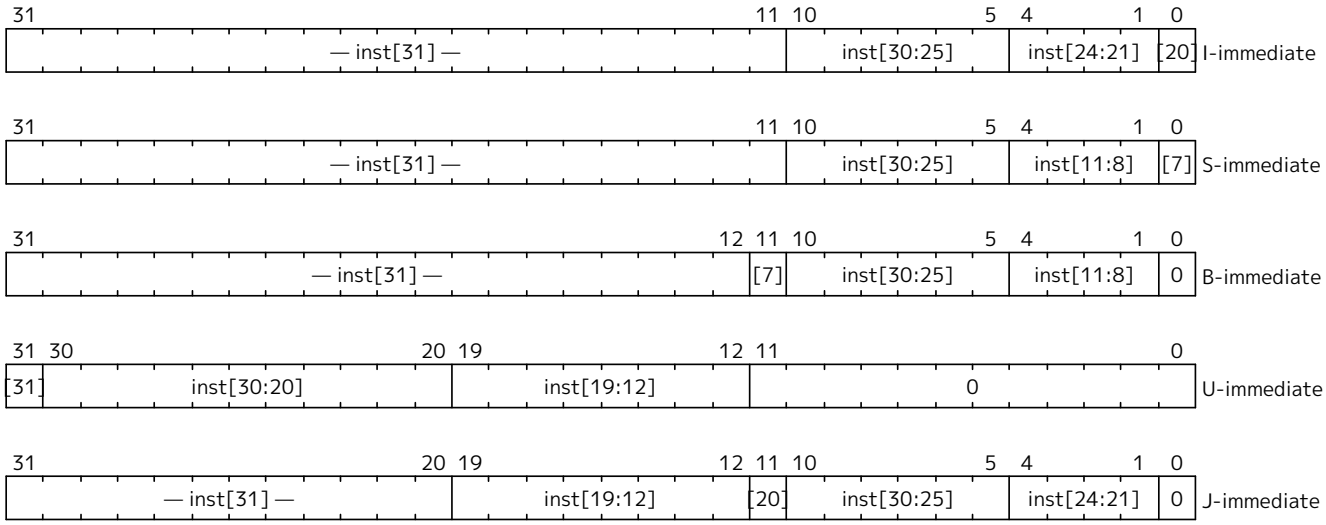


Figure 1. Types of immediate produced by RISC-V instructions.

You may find the immediate encodings confusing at first - why are there so many different formats, and why are the bits of the immediate shuffled around? This is for a couple of reasons:

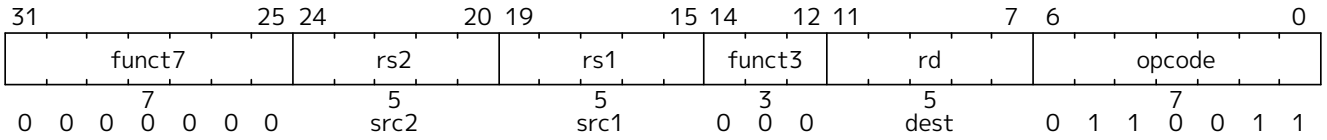
1. *Encodings that have lower bits set to zero (B-type and J-type) do not specify these bits to avoid wasting instruction bits, and instead encode more significant bits to increase the jump target range.*
2. *As you may notice in Immediate types, the immediate bits are shuffled and organized to minimize the number of different instruction bits in any particular bit of the immediate. This reduces the number of multiplexers needed in the immediate generator, reducing the area of the instruction decoder.*

Chapter 3. Instruction Reference

3.1. Register Arithmetic Instructions

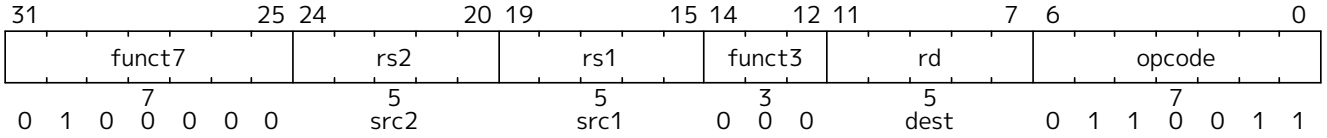
RV32I defines several arithmetic operations that operate on two source registers *rs1* and *rs2* and write the result into a destination register *rd*. They are encoded using the R-type format. The *funct3* and sometimes *funct7* field select the type of operation. For instructions that do not use the *funct7* field, it must be set to all zeros as shown. It is invalid to set the field to anything else (e.g. junk or don't-cares).

3.1.1. ADD: Add



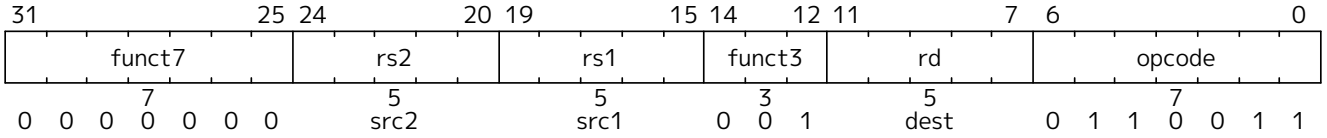
ADD performs the addition of two source registers *rs1* and *rs2* and places the result in the destination register *rd*. Overflow is ignored, and only the lower 32 bits of the result are written. The *funct7* field is set to all zeros to indicate addition instead of subtraction (see the next instruction).

3.1.2. SUB: Subtract



SUB performs the subtraction of *rs2* from *rs1* (*rs1 - rs2*) and places the result in the destination register *rd*. Overflow is ignored, and only the lower 32 bits of the result are written. Bit 5 of the *funct7* field (bit 30 of the instruction) is set to 1 to indicate subtraction.

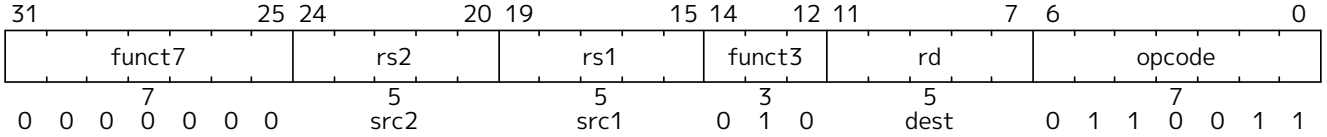
3.1.3. SLL: Shift Left Logical



SLL performs a logical left shift on the value in register *rs1* by the shift amount held in the least significant 5 bits of register *rs2* (effectively the value in *rs2* modulo 32) and places the result in the destination register *rd*. The most significant 27 bits of *rs2* are ignored.

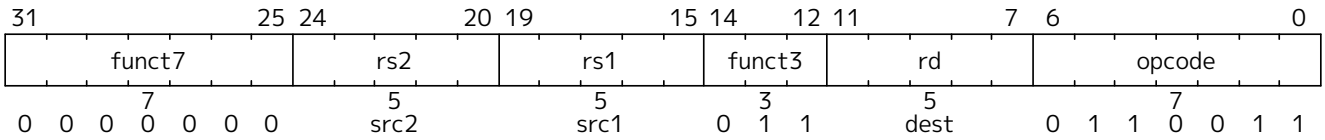
There is no SLA (shift left arithmetic) instruction because logical and arithmetic left shifts are equivalent. Both shift operations fill the least significant bits with zeros.

3.1.4. SLT: Set Less Than



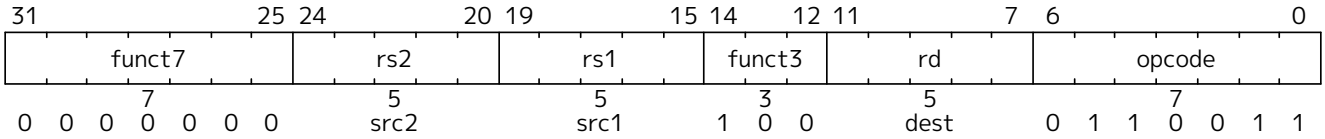
SLT performs a **signed** comparison of the values in *rs1* and *rs2*. If *rs1* is less than *rs2*, the destination register *rd* is set to `1`, otherwise it is set to `0`.

3.1.5. SLTU: Set Less Than Unsigned



SLT performs an **unsigned** comparison of the values in *rs1* and *rs2*. If *rs1* is less than *rs2*, the destination register *rd* is set to `1`, otherwise it is set to `0`.

3.1.6. XOR: Bitwise Exclusive Or



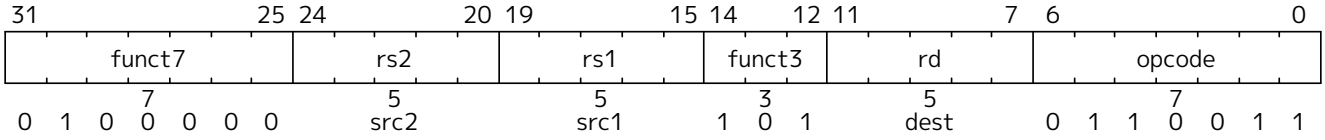
XOR is a logical operation that performs bitwise XOR on registers *rs1* and *rs2* and places the result in *rd*.

3.1.7. SRL: Shift Right Logical

SRL performs a logical right shift on the value in register *rs1* by the shift amount held in the least significant 5 bits of register *rs2* (effectively the value in *rs2* modulo 32) and places the result in the destination register *rd*. The most significant 27 bits of *rs2* are ignored. The *funct7* field is set to all zeros to specify that the right shift is logical and not arithmetic (see the next instruction).

Recall that a logical right shift fills the most significant bits with zeros.

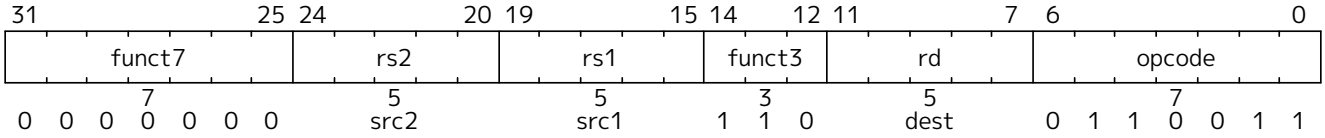
3.1.8. SRA: Shift Right Arithmetic



SRA performs an arithmetic right shift on the value in register *rs1* by the shift amount held in the least significant 5 bits of register *rs2* (effectively the value in *rs2* modulo 32) and places the result in the destination register *rd*. The most significant 27 bits of *rs2* are ignored. Bit 5 of the *funct7* field (bit 30 of the instruction) is set to 1 to indicate an arithmetic shift.

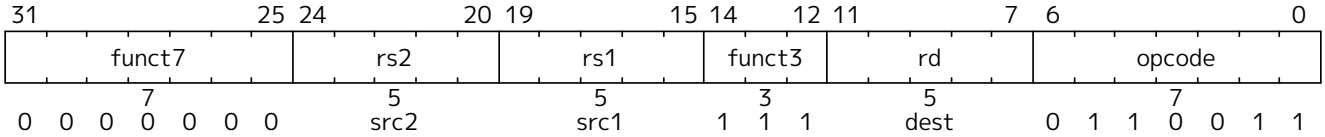
Recall that an arithmetic right shift fills the most significant bits with the MSB of the original value in rs1, preserving the sign of the input.

3.1.9. OR: Bitwise Or



OR is a logical operation that performs bitwise OR on registers *rs1* and *rs2* and places the result in *rd*.

3.1.10. AND: Bitwise And

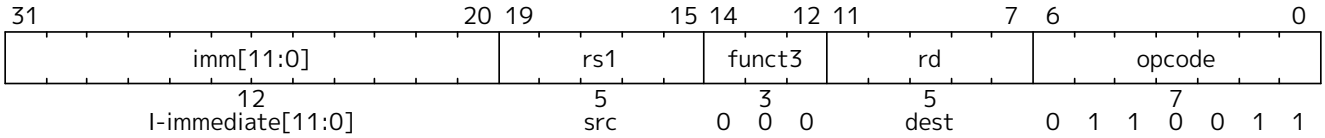


AND is a logical operation that performs bitwise AND on registers *rs1* and *rs2* and places the result in *rd*.

3.2. Register-Immediate Computational Instructions

RV32I defines several arithmetic immediate instructions that operate on a single source register *rs1* and a 12-bit immediate value (sign-extended to 32 bits) and write the result into a destination register *rd*. They are encoded using the I-type format. The *funct3* field selects the type of arithmetic operation.

3.2.1. ADDI: Add Immediate

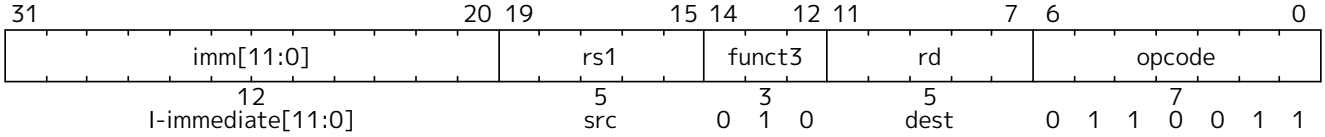


ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the lowest 32 bits of the result.

There is no SUBI (subtract immediate) instruction because the 12-bit immediate is sign-extended to 32 bits, and can therefore encode both positive and negative biases (constants).

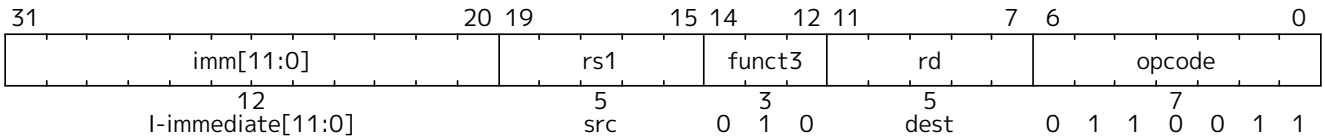
While there is no explicit NOP instruction in RV32I, recall that there are multiple ways to encode a NOP instruction in RISC architectures using immediate integer operations and the `zero` *register. RV32I defines ADDI, x0, x0, 0 as the canonical NOP encoding to simplify compilers and help with disassembly. It is also used by advanced microarchitectures to perform optimizations, but you* **do not** *need to treat NOP specially in WISC-F25.*

3.2.2. SLTI: Set Less Than Immediate



SLTI compares the **signed** value in register *rs1* with the sign-extended immediate. If *rs1* is less than the immediate, 1 is written to *rd*, otherwise 0 is written to *rd*.

3.2.3. SLTIU: Set Less Than Immediate Unsigned

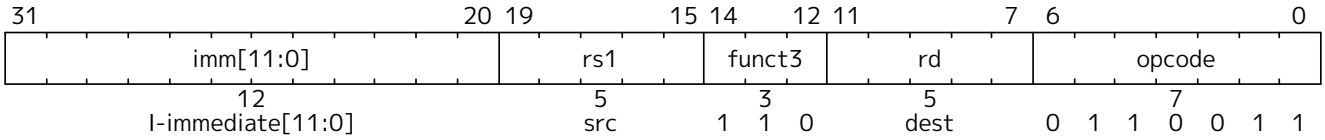


SLTIU is similar to SLTI, but compares the values as **unsigned** numbers. That is, the unsigned value in *rs1* is compared with the immediate, which is first **sign-extended to 32 bits** and then treated as an unsigned number. If *rs1* is less than the immediate, 1 is written to *rd*, otherwise 0 is written to *rd*.

3.2.4. ANDI: Bitwise And Immediate

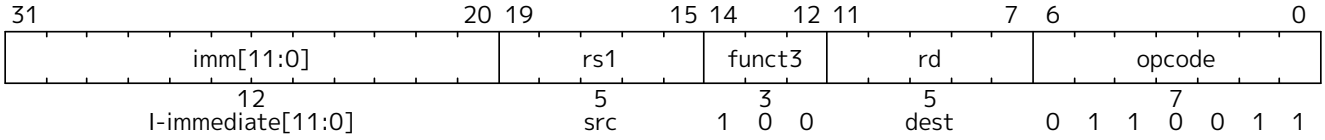
ANDI is a logical operation that performs bitwise AND on register *rs1* and the sign-extended 12-bit immediate and places the result in *rd*.

3.2.5. ORI: Bitwise Or Immediate



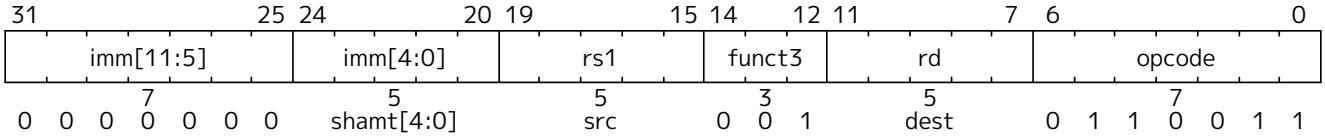
ORI is a logical operation that performs bitwise OR on register *rs1* and the sign-extended 12-bit immediate and places the result in *rd*.

3.2.6. XORI: Bitwise Exclusive Or Immediate



XORI is a logical operation that performs bitwise XOR on register *rs1* and the sign-extended 12-bit immediate and places the result in *rd*.

3.2.7. SLLI: Shift Left Logical Immediate



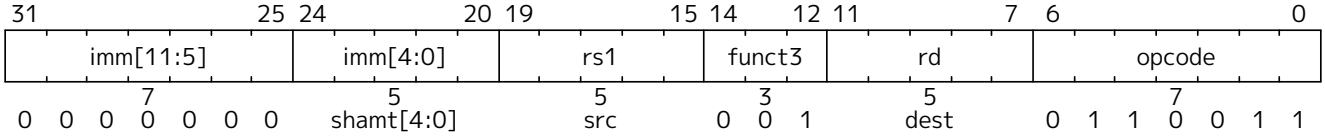
Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in

rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field.

SLLI encodes the upper 7 bits of the I-immediate as zeros. It is invalid to set these bits to anything else (e.g. junk or dont-cares).

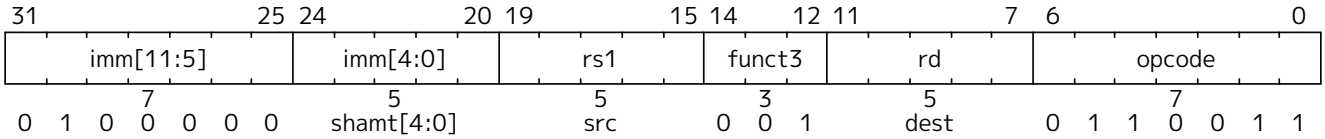
There is no SLAI (shift left arithmetic immediate) instruction because logical and arithmetic left shifts are equivalent. The least significant bits are filled in with zeros.

3.2.8. SRLI: Shift Right Logical Immediate



SRLI encodes the upper 7 bits of the I-immediate as zeros. It is invalid to set these bits to anything else (e.g. junk or dont-cares). Bit 10 of the immediate (bit 30 of the instruction) is used to encode the right shift type, and is set to 0 for logical shifts.

3.2.9. SRAI: Shift Right Arithmetic Immediate

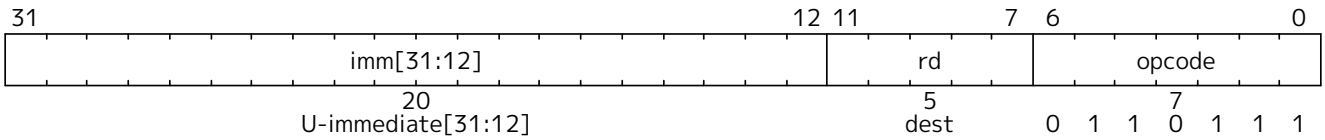


SRAI sets bit 10 of the immediate (bit 30 of the instruction) to 1 to indicate an arithmetic shift, and sets the other upper 6 bits of the immediate to zeros. It is invalid to set these bits to anything else (e.g. junk or dont-cares).

3.3. Integer Immediate Instructions

The integer immediate instructions are used to load large immediates and to calculate *far* `pc`-relative addresses. For this reason, they use the U-type format, which includes a 20-bit immediate field, and only operate on a destination register *rd*.

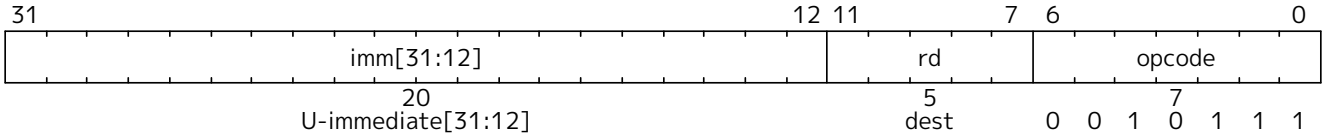
3.3.1. LUI: Load Upper Immediate



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

Because it is impossible to encode a 32-bit constant in a single 32-bit instruction (and leave space for opcodes, registers, and other fields), RV32I takes advantage of the fact that a majority of integer immediates are small, and can be represented in 12 bits (with sign-extension) and loaded through ADDI. For larger constants, the upper 20 bits can be loaded first using LUI, and then lower 12 bits can be loaded using ADDI.

3.3.2. AUIPC: Add Upper Immediate to PC



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 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*.

Similar to LUI, AUIPC is used to build 32 bit integers, but `pc`-relative rather than absolute. These are especially useful for long indirect jumps and function calls, where the target address is more than 12 bits (sign-extended) away from the current PC. AUIPC is also a convenient way to obtain the current PC when the immediate is set to 0.

3.4. Conditional Branch Instructions

All branch instructionis use the B-type instruction format. They compare two source registers *rs1* and *rs2* and, depending on the result, add a sign-extended immediate offset to the current PC, or do nothing and continue execution at the next instruction (`pc` + 4).

The 12-bit 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. This allows both jumping forward (generally for if statements) and backwards (generally for loops).

The conditional branch instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a 4-byte boundary and the branch condition evaluates to true. If the branch condition evaluates to false, the instruction-address-misaligned exception will not be raised.

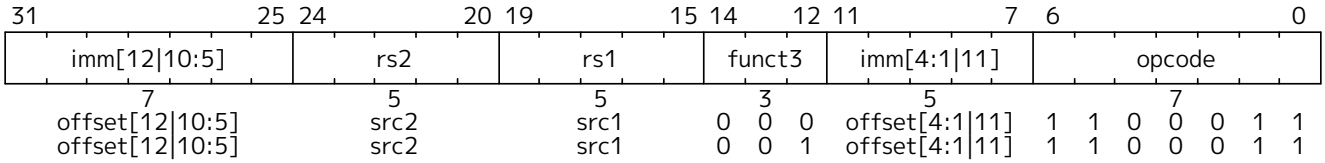
Because RV32I instructions are always 4 bytes long, instruction addresses are 4-byte aligned and the 2 LSBs of the immediate are always zero. You might have noticed however that only one LSB of the immediate is ignored in the encoding. This is because there are extensions to RISC-V (C and Zc) that add 16 bit* **compressed** *instructions as an optimization. We do not implement these in WISC-F25, but keep the official branch encoding for compatibility.*

You may recall that some ISAs including x86 have a "flags register" which stores the result of an arithmetic operation; branches are then performed by checking a flag bit. RISC-V instead takes the approach of combining the arithmetic comparison and branch into a single instruction. There are benefits to both approaches, but the combined approach simplifies advanced microarchitectures significantly (as there is no implicit flags register to track) and performs similarly if not better. Here's an excerpt on this decision from the official specification:

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC, Xtensa, and MIPS R6), 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.

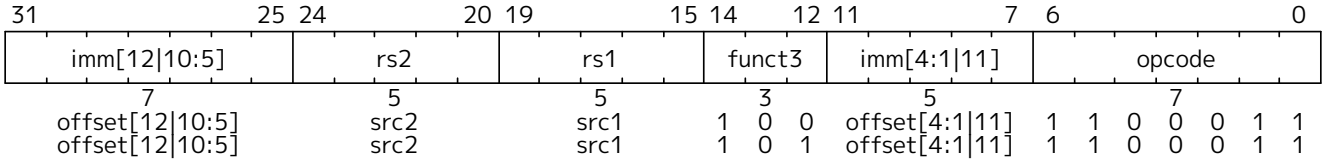
3.4.1. BEQ/BNE: Branch If Equal/Not Equal



BEQ compares the source registers *rs1* and *rs2* for equality and jumps to the target offset if they are equal. BNE performs the same comparison but inverts the condition, jumping to the target if the registers are not equal.

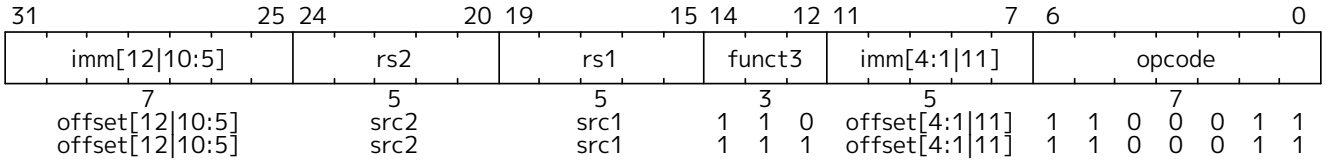
When programming in assembly, labels are used instead of manually calculating immediate offsets, which simplifies things considerably and maintains correctness/reduces the risk of bugs when program structure is changed.

3.4.2. BLT/BGE: Branch If Less Than/Greater Than Or Equal



BLT performs **signed** comparison between the source registers *rs1* and *r2*. If *rs1* is less than *rs2*, it jumps to the target offset. BGE performs the same comparison but inverts the condition, jumping to the target if *rs1* is greater than or equal to *rs2*.

3.4.3. BLTU/BGEU: Branch If Less Than/Greater Than Or Equal Unsigned



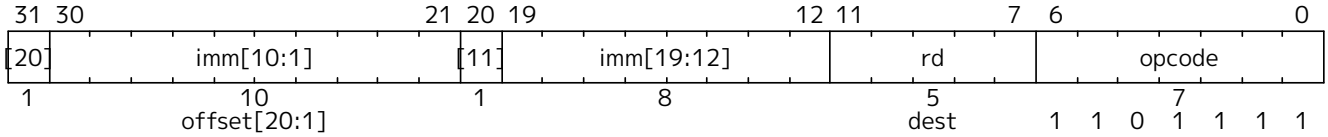
BLTU performs **unsigned** comparison between the source registers *rs1* and *r2*. If *rs1* is less than *rs2*, it jumps to the target offset. BGEU performs the same comparison but inverts the condition, jumping to the target if *rs1* is greater than or equal to *rs2*.

3.5. Unconditional Jumps

The unconditional jump instructions jump to a target address without comparing any values, and hence have more space in their instruction encoding for larger signed immediate offsets (leading to larger jump ranges). Before jumping, the next sequential value of the PC is saved in a specified destination register, called "linking".

Like with all other instructions that write to a destination register, the link register can be set to `zero` to discard the value if not needed by the program.

3.5.1. JAL: Jump and Link

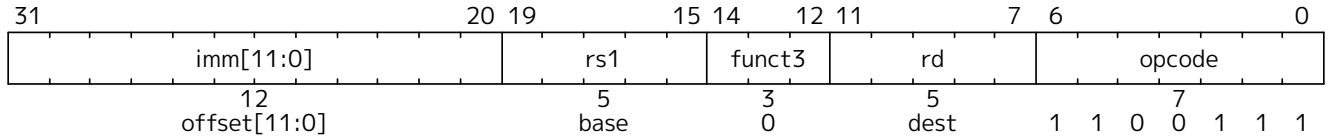


JAL uses the J-type format. The destination register *rd* is set to `pc` + 4, and a sign-extended and aligned immediate offset is added to the current PC.

When used with a destination link register, JAL implements a function call (which is an explicit CALL instruction in some ISAs). When the link register is discarded, it is simply an unconditional jump to a PC-relative address.

To support nested or recursive function calls, the program must either use multiple link registers or push the previous link register onto the stack using the SW instruction to avoid cloberring the old return address.

3.5.2. JALR: Jump and Link Register



JALR is very similar to JAL, but jumps to an address specified in a source register *rs1* plus a small sign-extended immediate offset (this is often called an *indirect jump*). It uses the I format. The destination register is set to `pc` + 4 and the PC is set to the aligned upper 31 bits of the sum of the *rs1* and the sign-extended immediate. The LSB of the PC is always set to 0.

Read the target calculation above carefully. You may fail the JALR test cases if your implementation does not correctly **clear** *the LSB of the target address to align the PC to a 2-byte boundary.*

When used with a destination link register, JALR can implement indirect function calls, which are useful for implementing interfaces and virtual methods/inheritance (vtables) in many programming languages. When the link register is discarded and rs1 is set to a previous link register, JALR implements a function return (which is an explicit RET instruction in some ISAs).

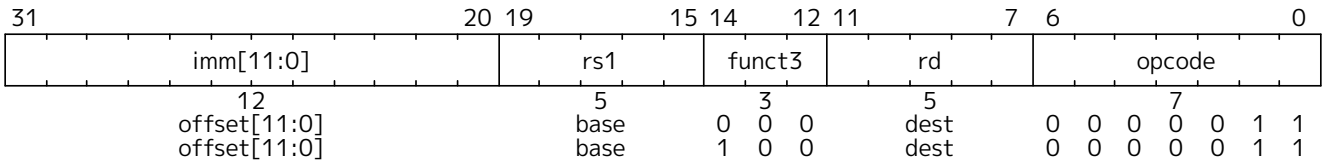
3.6. Memory Instructions

RV32I is a load/store architecture, meaning that all operations on memory are performed through explicit load and store instructions. RV32I allows memory to be accessed at 8-bit (byte), 16-bit (half-word), and 32-bit (word) granularity. Load instructions use the I-type format and store instructions use the S-type format.

The official RISC-V specification allows implementations to either require aligned memory accesses (i.e. half-word access are 2-byte aligned, and word accesses are 4-byte aligned) or allow unaligned

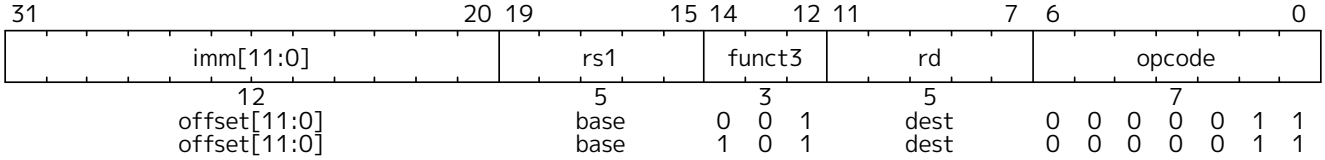
accesses. To simplify the implementation, WISC-F25 requires aligned accesses (and therefore you can ignore the somewhat complex logic required to assemble unaligned memory fragments).

3.6.1. LB/LBU: Load Byte/Load Byte Unsigned



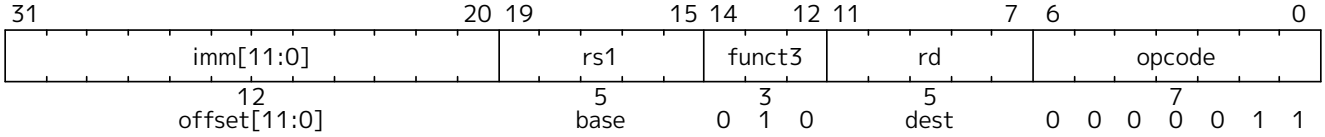
The LB instruction loads a byte at the memory address provided by the sum of the source register *rs1* and a sign-extended immediate. The byte is loaded into the least significant 8 bits of the destination register *rd* and the result is sign-extended to 32 bits. The LBU instruction is similar but zero-extends the 8 bit value to 32 bits.

3.6.2. LH/LHU: Load Half Word/Load Half Word Unsigned



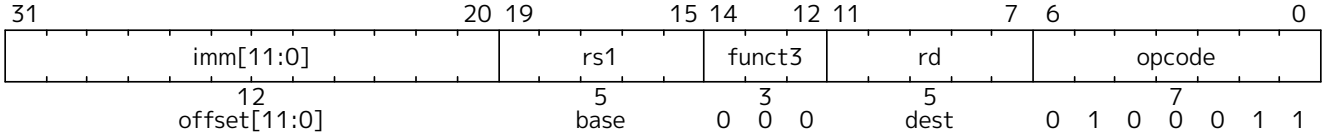
The LH instruction loads a half-word at the memory address provided by the sum of the source register *rs1* and a sign-extended immediate. The target address is guaranteed to be 2-byte aligned. The half-word is loaded into the least significant 16 bits of the destination register *rd* and the result is sign-extended to 32 bits. The LHU instruction is similar but zero-extends the 16 bit value to 32 bits.

3.6.3. LW: Load Word



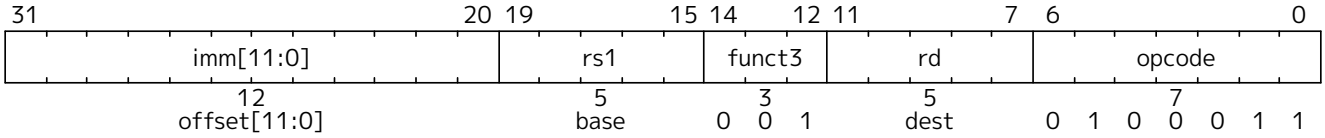
The LW instruction loads a word at the memory address provided by the sum of the source register *rs1* and a sign-extended immediate. The target address is guaranteeded to be 4-byte aligned. Because the 32-bit word fills the entire 32 bit destination register, there is no need for sign- or zero-extension.

3.6.4. SB: Store Byte



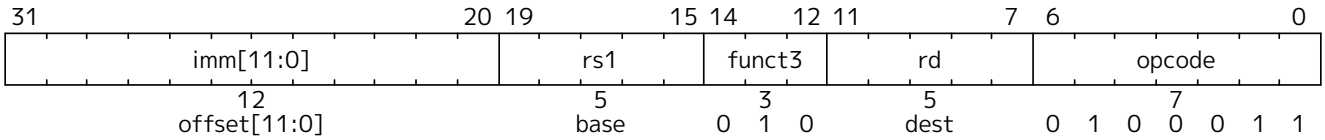
The SB instruction stores the least significant byte of the source register *rs2* to the memory address provided by the sum of *rs1* and a sign-extended immediate.

3.6.5. SH: Store Half Word



The SH instruction stores the least significant half-word of the source register *rs2* to the memory address provided by the sum of *rs1* and a sign-extended immediate. The target address is guaranteed to be 2-byte aligned.

3.6.6. SW: Store Word

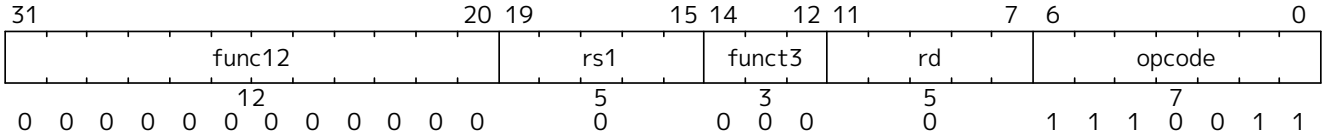


The SH instruction stores the value of the source register *rs2* to the memory address provided by the sum of *rs1* and a sign-extended immediate. The target address is guaranteed to be 4-byte aligned. === System Instructions

RV32I is a load/store architecture, meaning that all operations on memory are performed through explicit load and store instructions. RV32I allows memory to be accessed at 8-bit (byte), 16-bit (half-word), and 32-bit (word) granularity. Load instructions use the I-type format and store instructions use the S-type format.

The official RISC-V specification allows implementations to either require aligned memory accesses (i.e. half-word access are 2-byte aligned, and word accesses are 4-byte aligned) or allow unaligned accesses. To simplify the implementation, WISC-F25 requires aligned accesses (and therefore you can ignore the somewhat complex logic required to assemble unaligned memory fragments).

3.6.7. EBREAK: Break to environment



The EBREAK instruction is used to cause a breakpoint that traps the environment. In more advanced implementations, this is usually used to implement debugging features that trap to the operating system.

WISC-F25 uses the EBREAK instruction in a slightly different way - to halt execution. In other words, when the CPU executes an EBREAK instruction, it should stop fetching new instructions, and after retiring all existing instructions in the pipeline, it should assert the halt signal.