Note 9: RISC-V Instruction Formats

Kevin Moy

1 Stored Program Concept

1.1 ENIAC

The ENIAC computer was developed in Pennsylvania in 1946. It was incredibly fast. However, it was programmed with a ton of cords and switches, making it very hardwired and hard to reprogram. The entire computer would have to be tinkered with to edit/add programs.

1.2 Stored Programs

So people wanted to find a faster way to store programs. What if programs were, like data, just bit patterns? Then we could store programs into memory as well! Then, reprogramming just involves rewriting memory rather than the whole computer! This is the **stored program concept** in computers, and Such computers utilize it are called *Von Neumann* computers.

1.3 Memory Addresses are Universal

So now programs and data are in memory, and so everything now has an associated memory address! Remember C pointers are just containers for memory addresses. So our **program counter** (`PC`) is just a pointer to current instruction in code.

1.4 Code vs. Data

How do we distinguish data and instructions though? Again, just how you interpret the bits.

1.5 Binary Compatibility

Normally, programs are distributed in binary form- meaning that such programs are bound to a specific *instruction set*. However, with new machines (in same family as old machines) had new instruction sets. The solution is adding **backwards-compatible** instructions (support for the old machine instructions) in order to run the older programs ("**binaries**"). However, this makes things more complicated and possibly slower as overall instruction sets will grow over time.

The **assembler** translates assembly instructions into machine code.

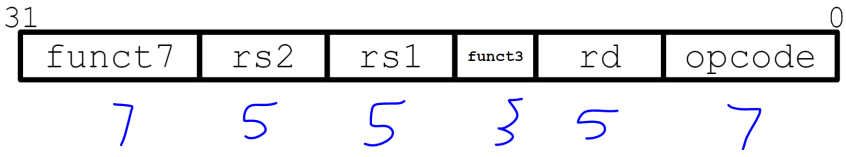
2 Instruction Fields

RISC-V instructions are each a word long- 32 bits. To process these instructions, the assembler splits up these 32 bits into multiple fields based on the **instruction format**. There are 6 formats we will cover in this note, all based on instruction functionality:

- R format: all registers (3)
- I format: immediate instructions / LOAD instructions
- S format: store instructions
- U format: upper-immediate instructions
- SB format: branch instructions
- UJ format: jump instruction `jal`

3 R-Type Instructions

Below is how R-Type instruction fields are split up:



Each field is treated as an **unsigned integer**. This is true in general for all instruction format fields, EXCEPT immediate fields.

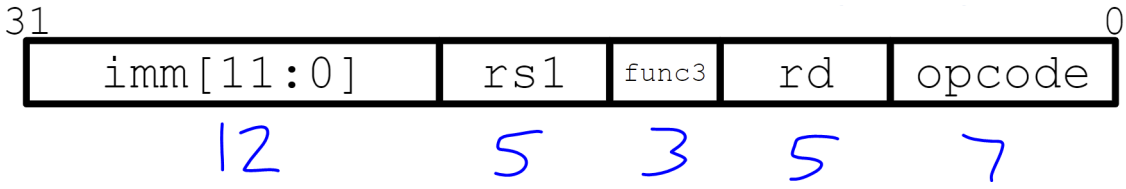
The `opcode` field is a PARTIAL specifier of operation. Together with `funct7` and `funct3` (and the opcode), the 3 fields fully indicate the specific operation. The opcode for R-types is fixed, so there are $2^7 * 2^3 = 2^{10}$ possible R-format instructions encodable.

R-type instructions utilize 2 operand registers and one destination register. `rs1` is the first operand register. `rs2` is the second operand register. `rd` is the destination register. All are 5 bit fields each for $2^5 = 32$ possible registers. The register field is based on the "official" register ID (`x0` to `x31`).

4 I-Type Instructions

Now we deal with immediates. Ideally, RISC-V would have a universal instruction format (same fields for all types) but we cannot- so we compromise and make I-format as close to R-format as possible.

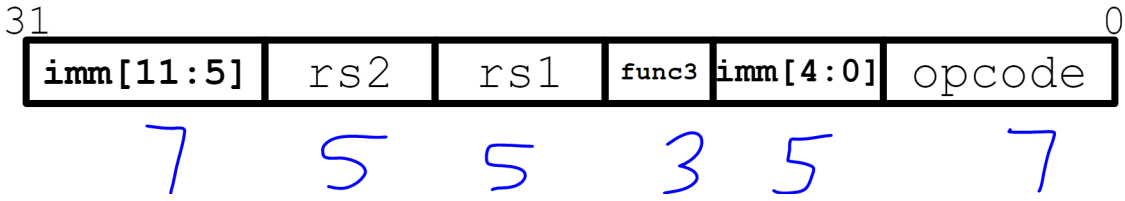
Immediate instructions only use 1 `src` register and 1 `dst` register. Here is the format for I type:



The key is that *only* the `imm` field is different from R-format: instead of `rs2` and `funct7`, we merge them to create a 12-bit field `imm`. This 12-bit number must be sign-extended to 32 bits before arithmetic computation (since its done with words).

5 S-Type Instructions

S-Type instructions deal purely with store instructions. `rs1` represents the base memory address and `rs2` represents the register containing the data to be stored. Here is the S-format:



Since we don't utilize a destination register `rd`, we'll just use the 5 bits instead for the lower 5 bits of our 12-bit immediate. So this way, `rs1` and `rs2` stay in the same place.

6 SB-Type Instructions

SB-format instructions are just branching instructions. A branching instruction takes a jump address as well as two registers to compare values of in its "conditional". Similar to store instructions, branch instructions do not actually write any registers. We typically use branch statements for loop constructs.

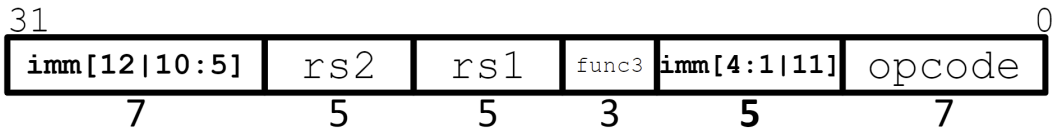
Remember also that instructions are stored in a localized area of memory- the code/text segment.

The PC stores the address of the current instruction- and jumps are made *relative* to the PC. This is called **PC-Relative Addressing**: the `imm` field is a (2's complement) *offset* to PC. Usually, these `imm`s aren't very big: basically it can add $\pm 2^{11}$ **word addresses**, or 2^{13} byte addresses, to the PC. Remember our 32-bit instructions are **word-aligned**: each address is a multiple of 4 (bytes). This way, by only adding word multiples to PC, we can ensure it always points to an instruction.

Side note: For systems supporting RISC-V *compressed instructions*, it uses half-word alignment (all addresses multiples of 2). So the branch offset would be.

So branching would update `PC = PC + (immediate*4)`.

Here is the format for SB:



This now accounts for half-word-aligned instructions. Our 12-bit `imm` represents $[-2^{12}, 2^{12} - 2]$ in 2-byte increments. Note the lowest bit of our immediate is always 0 (immediate must be even number) so we can actually encode 13-bit (signed) offsets.

To fully encode a SB-type instruction we need to calculate the **branch offset**, or basically the number of instructions after the branch to the label. Then multiply by 4 to get the branch offset (in bytes).

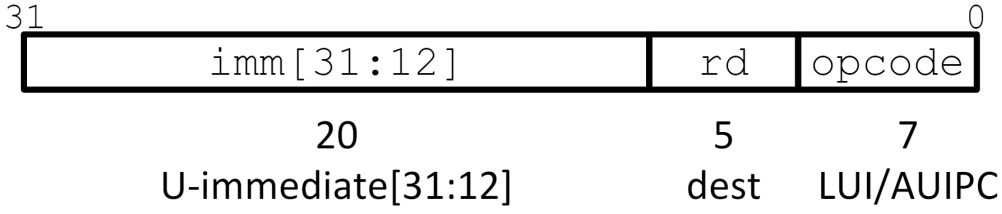
Why is this immediate field format so fucking weird? I don't know. I didn't write this shit.

What if our destination is $> 2^{10}$ instructions away from our branch instruction? Then we have a neat little fix: we set our branch label to be 2 instructions away (so branch offset becomes 2) and then set the next instruction instead to be `j Dest_Label`. Jump instructions have a much larger range than branch instructions.

7 U-Type Instructions

For now, we only get 12 bits for our immediate. What if we need all 32 bits? We actually just need a way to load the UPPER 20 bits of our immediate. This is where U-type instructions, specifically ONLY `lui` and `auipc`, come in.

So U-type instructions designate a `rd`, a 20-bit `imm`, and `opcode`. That is all! Here's our format:



7.1 lui

`lui`, or load upper immediate, writes the upper 20 bits of `rd` with `imm` then clears the lower 12 bits. We would (ideally) then fill in the lower 12 with `addi`. Pseudoinstruction `li` is a combination of `lui` and `addi` to make a full 32-bit load.

However, be careful: `addi` *sign-extends*, so combining `lui` and `addi` might not always get what you want. The solution is to add 1 to the upper 20 bits (result of `lui`) BEFORE `addi`.

7.2 auipc

`auipc` adds an upper-immediate to `PC` and places the result in `rd`. We use this instruction for **PC-relative addressing**- since the value `rd` holds depends on `PC` (the address of the current `auipc` instruction!)

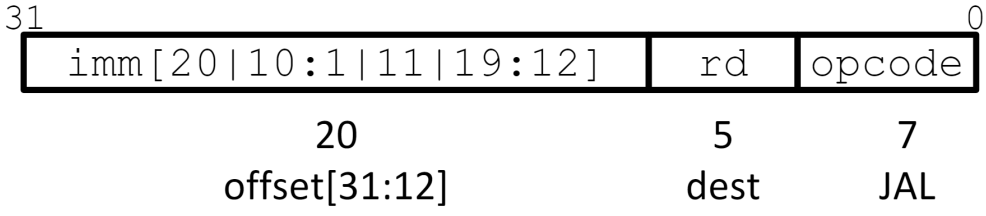
8 UJ-Type Instructions

Finally, we cover jump instructions: `jal, jalr, j`, etc. Branch instructions assume PC-relative addressing, i.e. the label to branch to isn't that far from `PC`. However, in the case that it *is* far, we need jumps.

8.1 jal

Remember that `jal rd Label` jumps to `Label` then places `PC+4` into `rd`. For such jumps, we can jump to *anywhere* in code memory (remember, this is where all the instructions and their addresses are!) We can't fit a 32-bit memory address into our instruction.

Instead, for `jal`, we take a 20-bit immediate, along with `rd` and of course `opcode`, as our format:



We're still doing PC-relative jumps with `offset`, and we set `PC = PC+offset` when jumping. Our instruction addresses are still half-word-aligned, and we have $\pm 2^{19}$ possible addresses from `PC` to jump to.

Remember that `j Label` is actually a pseudoinstruction for `jal x0 Label`.

The reason `imm` is encoded so weird is the same as all the other weirdly-encoded `imm` fields in other formats: we want to keep it as close to other formats (specifically in this case, branch instructions) as possible. This reduces hardware cost.

8.2 jalr

Remember that `jalr rd rs1 offset` writes `rd = PC+4` THEN sets `PC = rs1 + offset`. Notice this is the one jump instruction that DOESN'T use labels (Pseudocode `jr` does too). Notice our immediate is the exact same as arithmetic and load instructions-so we are now back to byte-addressing (no mul).

Note that `jalr x1` is the same as `jalr ra x1 0` - it is shorthand for jumping to the instruction address in `x1` then restting the return address to PC+4 (next instruction).

What are cases where we'd want to use `offset`? We'd want to use it for jumping to **absolute addreseses**. We'd set `x1` as our upper 20 bits of this address (minus 1) then set `offset` as our lower 12 bits.

There is another case with PC-relative jumps **with 32-bit offsets**. We can then use `auipc` to add the high 20 bits to `PC`, then use `jalr` and set the `offset` to the low 12 bits.