CSM CS61C Note #3: RISC-V Calling Conventions and Instruction Formats

Anthony Han

1 RISC-V Calling Conventions

In RISC-V, registers are divided into **caller-saved registers** and **callee-saved registers**. Programmers are expected to make sure that before and after a function call, the value stored in callee-saved registers stay the same. This is the RISC-V **calling convention**.

Register	Name	Description	Saver
x0	zero	Always 0	N/A
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	N/A
x4	tp	Thread pointer	N/A
x5 – 7	t0 – 2	Temporary	Caller
x8	s0 / fp	Saved register / Frame pointer	Callee
x9	s1	Saved register	Callee
x10 – 17	a0 – 7	Function arguments / Return values	Caller
x18 – 27	s2 – 11	Saved registers	Callee
x28 – 31	t3 – 6	Temporaries	Caller

You can also find this table on the RISC-V green sheet. Note that:

- Return address ra stores where to return to in a nested procedure.
- Stack pointer sp points to the bottom of call frame. Convention is grow stack down from high to low addresses: push decrements sp, pop increments sp.
- If there are more than 8 arguments, put on stack:

```
sw t0, 0(sp) # Argument #9
sw t1, 4(sp) # Argument #10
# You need to memorize where they are stored.
```

As a result of this calling convention, within each function, we tend to have a prologue and epilogue that looks like

```
func:
    # Prologue
    addi sp, sp, -8 # Adjust stack for 2 items
    sw ra, 4(sp)    # Save ra for use afterwards
    sw s0, 0(sp)    # Save s0 for use afterwards

    # Do stuff
```

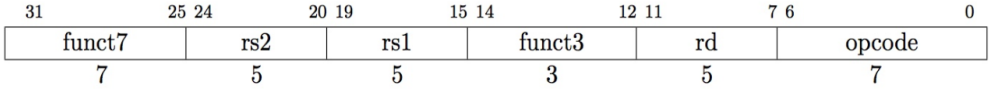
```
8
9       # Epilogue
10      lw s0, 0(sp)     # Restore register s0 for caller
11      lw ra, 4(sp)     # Restore register ra for return
12      addi sp, sp, 8   # Adjust stack to delete 2 items
13      jr ra            # Jump back to calling routine
```

2 RISC-V Instruction Formats

To let the computer execute the assembly we write, we need to turn the instructions into raw bits. Last time around, we classified RISC-V instructions into five categories by what these instructions do. This time, we are going to classify these instructions by how they are encoded in machine code. There are some similarities, yet there are differences.

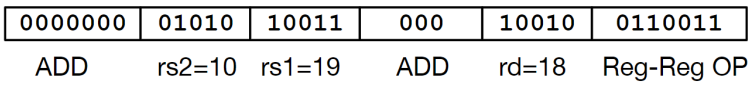
RISC-V Instruction Format - R Format

This corresponds to the register instructions we discussed last time.



- The **opcode** field of an instruction partially specifies which instruction it is. It is **equal to $(0110011)_2$ for all R-type instructions.**
- The **funct7** and **funct3** fields, when combined with the opcode, describe the operation to be performed.
- Each register field (**rs1**, **rs2**, **rd**) holds a 5-bit unsigned integer (0–31) corresponding to a register (x0–31).
  - **rs1** (Source register #1): specifies register containing first operand.
  - **rs2** (Source register #2): specifies register containing second operand.
  - **rd** (Destination register): specifies register which will receive result of computation.
- R format example:

```
1 add x18, x19, x10
```



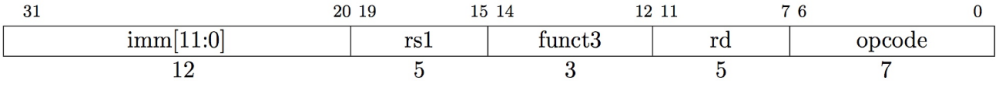
- All R-type instructions:

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

RISC-V Instruction Format - I Format

Ideally, RISC-V only uses one instruction format; this is not possible, however: 5b field only represents numbers up to the value 31.

Therefore, to represent the immediate instructions, we define a format most consistent with R-format which uses at most 2 registers (one source, one destination):



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

- Here, the rs2 and funct7 fields are replaced by a 12-bit signed immediate, **imm[11:0]**. The remaining fields stay the same.
- Immediate is **always sign-extended to 32b** before used in an arithmetic operation.
- Some I-type instructions:

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

  For logical shifts, we note that:

  – One of the higher-order immediate bits is used to distinguish SRLI from SRAI.

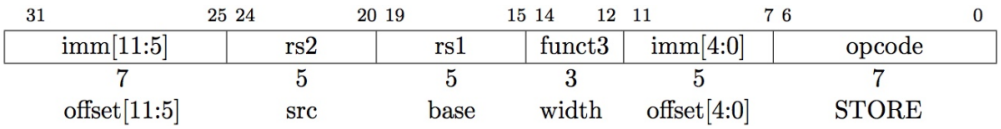
  – We only use the lower 5 bits of the immediate value for shift amount, since we can only shift by 0–31 bit positions.

- **Load instructions are also I-type.** We use the funct3 field to encodes the size and signedness of data we want to load.

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

RISC-V Instruction Format - S Format

This corresponds to the store instructions we discussed last time.



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

- **We use rs1 for the base memory address, rs2 for the data to be stored.**
- All S-type instructions:

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

RISC-V Instruction Format - SB Format

To encode the branch instructions, we need a way to represent the labels. In RISC-V, this is done through **PC-relative addressing**.

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

- We use the **immediate** field as a **two's complement offset relative to the PC**. The remaining fields are identical to those of S-type instructions.
- Since RISC-V instructions are at least 16b, the byte offset between instructions must be even, and **the least significant bit can be omitted**; and we can encode imm[12:1] instead of imm[11:0].
- All SB-type instructions:

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

- If the offset is more than 2^{12} bytes away from the branch instruction, we can use a jump instruction instead:

```
beq x10, x0, far
# next instr

# CHANGE TO
bne x10, x0, next
j   far
next: # next instr
```

RISC-V Instruction Format - U Format

As we can see, the immediate in I-type instruction is fairly small. What if we want to express large immediates? Hence the upper immediate instruction. In the case of U-type instructions, the immediate is the **upper 20 bits** of a 32b instruction word.

- **LUI - Load Upper Immediate**
  - **Writes the upper 20 bits** of the destination with the immediate value, and **clears the lower 12 bits.**
  - Together with an **addi to set low 12 bits**, we can create any 32b value in a register using two instructions.

```
lui  x10, 0x87654     # x10 = 0x87654000
addi x10, x10, 0x321 # x10 = 0x87654321
```

– A corner case: if we want to set 0xDEADBEEF...

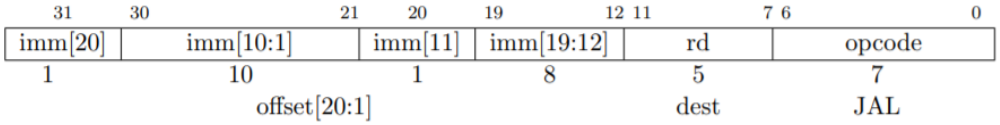
```
lui  x10, 0xDEADB    # x10 = 0xDEADB000
addi x10, x10, 0xEEF # x10 = 0xDEADAEEF
# Wrong result because immediate in addi is sign-extended
# Instead, pre-increment the value in upper 20 bits:
lui  x10, 0xDEADC    # x10 = 0xDEADC000
addi x10, x10, 0xEEF # x10 = 0xDEADBEEF

# In fact, the assembler pseudo-instruction handles it all:
li x10, 0xDEADBEEF # Creates two instructions
```

- **AUIPC - Add Upper Immediate to PC**

RISC-V Instruction Format - UJ Format

JAL's syntax is fairly different, and so we put it in its own category. Since JAL takes in a label, we are doing a **PC-relative jump, just like in SB-type instructions**.



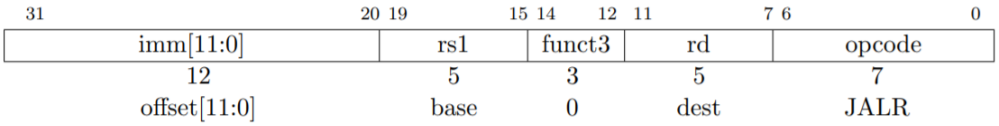
- **Saves PC + 4 in register rd** as return address;
- **Sets PC = PC + offset**.

```
# j - pseudo-instruction, sets rd = x0
j label # == jal x0, label # Discard return address

# Call function within 2**18 instructions of PC
jal ra, FuncName
```

JALR Instruction - I Format

Note that in RISC-V, jalr is an I-format instruction:



- **Writes PC + 4 to rd** as return address;
- **Sets PC = rs + immediate.**
- Uses same immediates as arithmetic and loads - **no multiplication by 2 bytes.**

```
# jr - pseudo-instruction, sets rd = x0, imm = 0
jr ra # == jalr x0, ra, 0
# ret - pseudo-instruction
ret   # == jr ra

# Call function at any 32-bit absolute address
lui  x1, <hi20bits>
jalr ra, x1, <lo12bits>
```

```
9
10 # Jump PC-relative with 32-bit offset
11 auipc x1, <hi20bits>
12     # Adds upper immediate value to and places result in x1
13 jalr x0, x1, <lo12bits>
14     # Same sign extension trick needed as LUI
```