CSM 61C Spring 2020

RISC-V Instruction Formats

Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

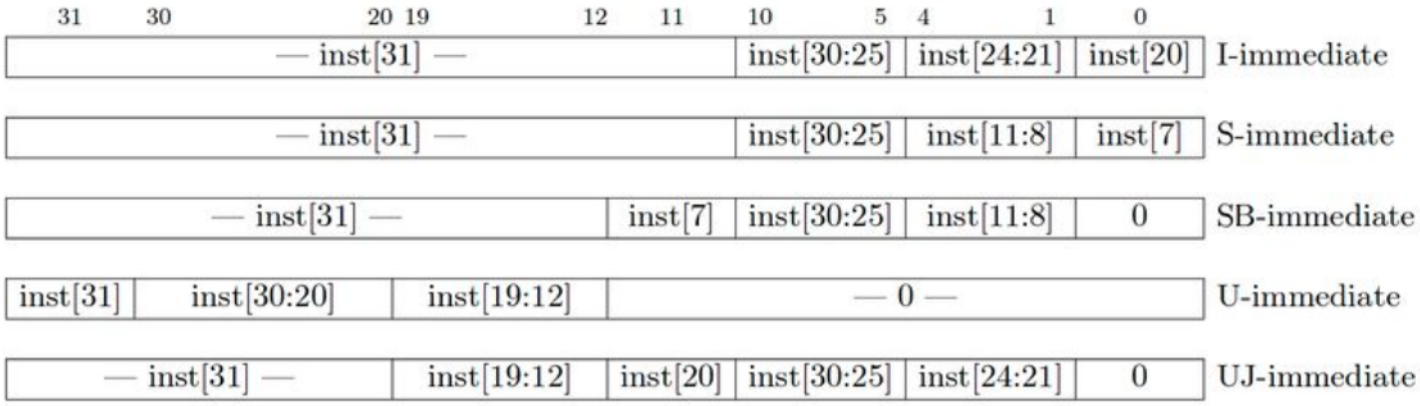
- Summer 2019 Midterm 2 Q2
- Fall 2019 Final Q2c
- Spring 2018 Midterm 1 Q5
- Fall 2018 Midterm Q4
- Fall 2019 Final Q10h
- Summer 2018 Midterm 1 Q5
- Spring 2018 Midterm 2 Q10f-h
- Fall 2019 Final Q4

Question 2: ReCALL This Information (or have it written down I guess) - 16 pts

Consider the following assembly code in a file foo.s:

```
      .text
1.              mv s1 a0
2.              addi s2 s2 4
3.    Start:    beq s1 x0 End
4.              lw a0 0(s1)
5.              jal ra printf
6.              add s1 s2 s1
7.              lw s1 0(s1)
8.              jal x0 Start
9.    End:      jalr x0, ra, 0
```

Recall that immediate values are generated from instructions with the following table:



We will refer to the number produced after this process is completed as the "immediate value."

1. Fill in all fields (or write Does Not Apply) for the machine code generated for **beq s1 x0 End** (line 3).

Immediate value: 24

funct3: 0x0

opcode: 0x63

funct7: N/A

rs1: 9

rs2: 0

rd: N/A

Given the hex representation, which line number in the above program does it correspond to?

2. 0x0004A483

Line: 7

3. 0xFEDFF06F

Line: 8

Q2) Open to Interpretation **(11 pts = 2 + 3 + 4 + 2)**

Let's consider the hexadecimal value **`0xFF000003`**. How is this data interpreted, if we treat this number as…

c) a RISC-V instruction? If there's an immediate, write it in decimal.

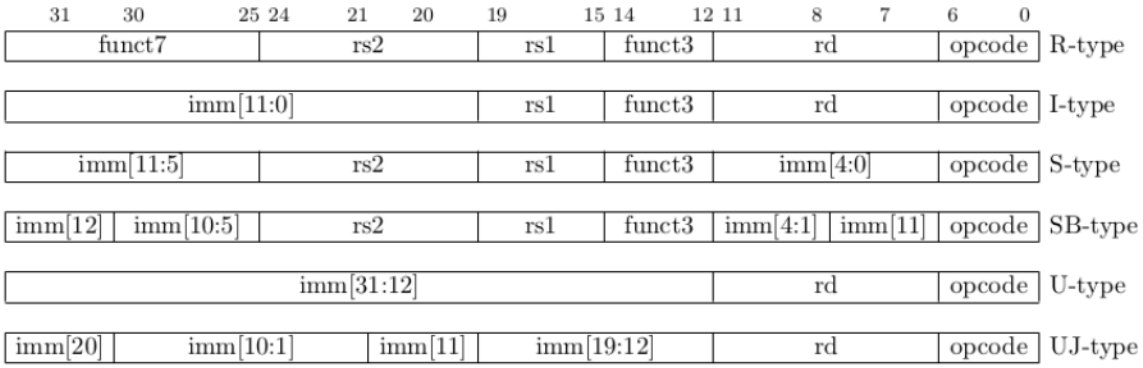
`lb x0 -16(x0)`

SHOW YOUR WORK

The opcode is 0b0000011 and the func3 is 0b000, which corresponds to the "lb" instruction. "lb" is an I-type instruction, so we extract rd = 0b00000, rs1 = 0b00000, Imm = 0b111111110000. The register 0b00000 is the x0 register. Finally, we calculate the immediate, taking care to note that immediates are stored in two's complement signed form. Negating the immediate yields 0b000000001111+1 = 0b000000010000 = 16, so the immediate is -16. We then write the instruction in lb format.

Problem 5 ***RISC-U ISA*** **(15 points)**

Here are the standard 32-bit RISC-V instruction formats taught in lecture for your reference:



Considering the standard 32-bit RISC-V instruction formats, convert `lw t5, 17(t6)` to machine code:

(a) **Solution:** 0x011FAF03

Prof. Wawrzynek decides to design a new ISA for his ternary neural network accelerator. He only needs to perform 7 different operations with his ISA: XOR, ADD, LD, SW, LUI, ADDI, and BLT. He decides that each instruction should be 17 bits wide, as he likes the number 17. There are no funct7 or funct3 fields in this new ISA.

(b) What is the minimum number of bits required for the opcode field?

Solution: $\lceil log_2 7 \rceil = 3$

(c) Suppose Prof. Wawrzynek decides to make the opcode field 6 bits. If we would like to support instructions with 3 register fields, what is the maximum number of registers we could address?

Solution: $\lfloor (17-6)/3 \rfloor = 3$ bits per register field which means 8 registers we could address

(d) Given that the opcode field is 6 bits wide and each register field is 2 bits wide in the 17 bit instruction, answer the following questions:

(i) Using the assumptions stated in the description of part (d), how many bits are left for the immediate field for the instruction BLT (Assume it takes opcode, rs1, rs2, and imm as inputs)?

Solution: 17-6-2-2 = 7

(ii) Let `n` be your answer in part (i). Suppose that BLT's branch immediate is in units of instructions (i.e. an immediate of value 1 means branching 1 instruction away). What is the maximum number of **bits** a BLT instruction can jump **forward** from the current PC using these assumptions? Write your answer in terms of `n`.

Solution: $(2^{n-1} - 1) * 17$

(iii) Using the assumptions stated in the description of part (d), what is the most negative immediate that could be used in the ADDI instruction (Assume it takes opcode, rs1, rd, and imm as inputs)?

Solution: -64

(iv) For LUI, we need opcode, rd, and imm as inputs. Using the assumptions stated in the description of part (d), how many bits can we use for the immediate value?

Solution: 17-6-2 = 9

Q4) *RISC-V business: I'm in a CS61C midterm & I'm being chased by Guido the killer pimp...* (14 points)

a) Write a function in RISC-V code to return *0 if the input 32-bit float = ∞*, else a non-zero value. The input and output will be stored in **a0**, as usual.
(If you use 2 lines=3pts. 3 lines=2 pts)

```
isNotInfinity: lui a1, 0x7F800
               xor a0, a0, a1
               ret
```

(the rest of Q3 deals with the code on the right)
Consider the following RISC-V code run on a 32-bit machine:

```
done: li a0, 1
      ret
fun:  beq a0, x0, done
      addi sp, sp, -12
      addi a0, a0, -1
      sw ra, 8(sp)
      sw a0, 4(sp)
      sw s0, 0(sp)
      jal fun
      mv s0, a0
      lw a0, 4(sp)
      jal fun
      add a0, a0, s0
      lw s0, 0(sp)
      lw ra, 8(sp)
      addi sp, sp, 12
      ret
```

```
beq a0, x0, done
   rs1 rs2 label
```

c) What is the hex value of the machine code for the underlined line? (choose ONE)

○0xFE050EEA ○0xFE050EE3 ●0xFE050CE3 ○0xFE050FE3 ○0xFE050EFA ○0xFE050FEA

```
beq --> look that up it's opcode 0b1100011 with funct3 of 0b000
The branch moves 2 spots up, so that's -2 instructions or -8 bytes
(8_10)=0...001000 --> (flip-bits of 8_10)=1...110111 (then add 1 for 2s comp)+ 1 --> 1...111000
...and in terms of bits it's                                                      543210
...so bits 12|10:5 are all 1s and bits 4:1|11 are 0b1100|0b1 --> 0b11001
rs1 (lookup in green sheet) for a0 is x10, which is 0b01010, rs2 is 0s
So putting all the bits in the right place above and clustering by nibbles we get 0xFE050CE3
    imm[12|10:5]    |     rs2      |     rs1      | funct3 |  imm[4:1|11] |       opcode
 1  1  1  1  1  1  1| 0  0  0  0  0| 0  1  0  1  0| 0  0  0| 1  1  0  0  1| 1  1  0  0  0  1  1
31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
      F          E          0          5          0          C          E          3
```

d) What is the one-line C disassembly of **fun** *with* recursion, and generates the same # of function calls:

```
uint32_t fun(uint32_t a0) { return !a0 ? 1 : fun(a0-1) + fun(a0-1) }
```

e) What is the one-line C disassembly of **fun** that has *no* recursion (i.e., see if you can optimize it):

```
uint32_t fun(uint32_t a0) { return 1 << a0 }
```

f) Show the call and the return value for the *largest possible value* returned by (e) above:

```
fun( 31 ) ⇒ 2^31
```

SOLUTIONS Fall 2019 Final
Question 10h

h) You are designing a 64-bit ISA for a simplified CPU with 3 bit-fields: `immediate | register | opcode`. You reserve enough of the rightmost bits to handle 1,500 opcodes, and enough of the leftmost bits to encode unsigned numbers up to 500 trillion. What's the greatest number of registers can you have?

16

SHOW YOUR WORK

1500 opcodes requires 11 bits (2048)
500 trillion requires 49 bits (512 Tebi)
64 - (11 + 49) = 4 bits for registers
2^4=16

Question 5: RISC-V Instruction Formats (12 pts)

You are given the following RISC-V code:

```
Loop:   andi  t2 t1 1
        srli  t3 t1 1
        bltu  t1 a0 Loop
        jalr s0 s1
        MAX_POS_IMM ...
```

1) What is the value of the **byte offset** that would be stored in the immediate field of the `bltu` instruction?

Two instructions away = -8 bytes

2) What is the binary encoding of the `bltu` instruction? Feel free to use the following space for scratch work—it will not be graded. Put your final answer in hexadecimal.

31 0

1111 1110 1010 0011 0110 1100 1110 0011

0xFEA36CE3

As a curious 61C student, you question why there are so many possible opcode, but only 47 instructions. Thus, you propose a revision to the standard 32-bit RISC-V instruction formats where **each instruction has a unique opcode (which still is 7 bits)**. You believe this justifies taking out the funct3 field from the R, I, S, and SB instructions, allowing you to allocate bits to other instruction fields **except the opcode field**.

1) What is the largest number of registers that can now be supported in hardware?

Now that we eliminate the funct3 field, we have 3 bits at our disposal for the R, I, S, and SB instructions. Since the R type instruction has three registers fields, we could add a bit to each of those fields and thus increase the number of registers to 64.

2) With the new register sizes, how far can a `jal` instruction jump to **(in halfwords)**?

Since register fields now have 6 bits, then in the UJ type, the jump immediate field is now reduced to 19 bits instead of 20. Thus, we can only jump to $\pm(2^{18})$ halfwords

`jal` jump range: [-2^{18}, $2^{18} - 1$]

3) Assume register `s0` = 0x1000 0000, `s1` = 0x4000 0000, `PC` = 0xA000 0000. Let's analyze the instruction:

```
jalr s0, s1, MAX_POS_IMM
```

where `MAX_POS_IMM` is the maximum possible positive immediate for `jalr`.

Once again, use the new register sizes from part 1. After the instruction executes, what are the values in the following registers?

Once again, we know that rd and rs1 fields are now 6 bits. `jalr` is an I-type instruction, so we take out the funct3 bits but we give each of rd and rs1 fields 1 bit, meaning we have 1 bit leftover to give to the immediate field. Thus, we now have a 13-bit immediate. Thus, the maximum possible immediate a jalr instruction can hold is $+2^{12} - 1$ halfwords away, which is represented as 0b0 1111 1111 1111, which is 0x0FFF.

s0 is the linking register—it's value is PC + 4
s1 does not get written into so it stays the same
PC = R[s1] + 0x0FFF

s0 = 0xA000 0004 s1 = 0x4000 0000 PC = 0x4000 0FFF

(f) Complete the following RISC-V procedure `jal_address_fixing` that handles address relocation for all `jal` instructions. It first calls `find_next_jal` to find a `jal` instruction that does not yet have its offset filled in (the immediate bits are all zeroes), calculates the jump offset, and fills the immediate field of the `jal` instruction.

- Fill in **one** instruction for each of the 5 blanks.
- You can assume `jal_address_fixing` has the ability to modify text segment instruction memory.
- The function `find_next_jal` returns two values: the first is the address of a `jal` instruction stored in `a0`; the second is the address of the target instruction this `jal` instruction is jumping to stored in `a1`. If there are no more `jal` instructions to fill in offsets for, it returns 0 and 0.

```
jal_address_fixing:
     jal ra, find_next_jal
     beq a0, x0, DONE
     sub a1, a1, a0          # set a1 as the jump offset
IMM_20:                        # sets imm[20]
     srai a5, a1, 20          # now a5 has imm[20]
     slli a5, a5, 31      # a5 has imm[20]
IMM_19_12:     # sets imm[19:12]
     li a3, 0xFF000
     and a3, a1, a3
     or a5, a5, a3      # now a5 has imm[20] and imm[19:12]
IMM_10_1:     # sets imm[10:1]
     li a3, 0x7FE
     and a3, a1, a3
     slli a3, a3, 20
     or a5, a5, a3      # now a5 has imm[20], imm[10:1], and imm[19:12]
IMM_11:     # sets imm[11]
```

```
    li a3, 0x800
    and a3, a1, a3
    slli a3, a3, 9
    or a5, a5, a3      # now a5 has imm[20], imm[10:1], imm[11], and imm[19:12])
UPDATE:                # inserts immediate into jal instruction
    lw t0, 0(a0)      # load the current jal instruction (from mem[a0])
    or t0, t0, a5      # update the instruction (add also works)
    sw t0, 0(a0)      # save the updated instruction (to mem[a0])
    j jal_address_fixing          # jump back to fix the next one
DONE:
    ...
```

Solution: The above procedure in C:

```
void jal_address_fixing(int address, int target) {
    int offset = target - address;
    int imm_10_1 = (offset & 0x7FE) >> 1;
    int imm_11 = (offset & 0x800) >> 11;
    int imm_19_12 = (offset & 0xFF000) >> 12;
    int imm_20 = (offset & 0x100000) >> 20;
    offset = (imm_20 << 31) | (imm_10_1 << 21) |
                (imm_11 << 20) | (imm_19_12 << 12);
    int* addr = (int*) address;
    *addr &= offset;
}
```

(g) The above code works for a `jal` target address that is 2^{16} bytes smaller than the `jal` instruction address.

● True ○ False

Solution: This code correctly updates the offset for `jal` instruction if the address is 2^{16} bytes smaller.

(h) The above code works for a `jal` target address that is 2^{24} bytes larger than the `jal` instruction address.

○ True ● False

Solution: To reach an address that is 2^{24} bytes larger, we need two instructions: `auipc` and `jalr`. This code does not work for far jump targets.

SOLUTIONS Fall 2019 Final

Q4) Felix Unger must have written this RISC-V code! **(30 pts = 3*10)**

```
1. mystery:
2.    la t6, loop
3. loop: addi x0, x0, 0        ### nop
4.    lw   t5, 0(t6)
5.    addi t5, t5, 0x80
6.    sw   t5, 0(t6)
7.    addi a0, a0, -1
8.    bnez a0, loop
9.    ret
```

What this function does (courtesy of Albert Zhan on piazza with minor edits):

1. mystery(x) entry point => where a0 = x, this is the label to jump to mystery.
2. la t6, loop => t6 register now stores the address of the "loop" (in this case, it points to an instruction in the text section of memory)
3. addi x0 x0 0 => instruction is executed. Note that addi instruction looks like
   _ _ _ _ _ _ _ _ _ _ _ _ + _ _ _ _ _ + 000 + _ _ _ _ _ + 0010011
   which are imm[11:0], rs1, and rd
4. lw t5, 0(t6) => loads the 32 bits of instruction into t5 register
5. addi t5, t5, 0x80 => adds
   0000 0000 0000 0000 0000 0000 0100 0000
   to the instruction at "loop" -> counting the bits, it adds 1 to rd (so the instruction on the first iteration becomes "addi x1, x0, 0")
6. sw t5, 0(t6) => stores the updated instruction into memory, so it modifies the instruction at "loop" -- self modifying the code.
7. addi a0, a0, -1 => decreases x by 1
8. bnez a0, loop => if a0 == 0, continue. Otherwise, go back to the instruction at "loop"
9. ret => jump to ra...

So it modifies the registers by modifying the code at "loop", which in turn modifies the registers x0, x1, etc.

You are given the code above, and told that you can read and write to any word of memory without error. The function **`mystery`** lives somewhere in memory, but *not* at address **`0x0`**. Your system has no caches.

a) At a functional level, in seven words or fewer, what does **`mystery(x)`** do when **`x < 10`**?

Resets the first x registers.
Resets register number 0 through x-1.

b) One by one, what are the values of **a0** that **bnez** sees with **mystery(13)** at every iteration? We've done the first few for you. List no more than 13; if it sees fewer than 13, write N/A for the rest.

12, 11, 10, 9, 8, 7, 6, 5, 4, 3, -1, -2, -3

We're merrily rolling along, resetting all the registers, when we reset x10 = a0! But then "addi a0,a0,-1" makes it -1 so it actually never hits the stopping "branch equal to zero" case then! So the bnez sees -1, then -2, then -3 as the resetter continues along its merry way.

Reset register # on "nop" line	a0 before addi line	bnez sees a0 value
0	13	12
1	12	11
2	11	10
3	10	9
4	9	8
5	8	7
6	7	6
7	6	5
8	5	4
9	4	3
10	0	-1 (or 2^{32} - 1)
11	-1	-2 (or 2^{32} - 2)
12	-2	-3 (or 2^{32} - 3)

c) How many times is the **bnez** instruction seen when **mystery(33)** is called before it reaches **ret** (if it ever does)? If it's infinity, write ∞. **2^{32} + 10**

d) Briefly (two sentences max) explain your answer for part (c) above.

After we get through a0 resetting (and then skipping 0, the stopping condition), we continue resetting all the registers until we get to t5 (x30). Resetting it doesn't do anything since we clobber it anyway with the lw command. The next iteration, the "nop" line will reset t6. So when we lw t5 0(t6=0) we are loading the first word of memory. We are told this does not cause an error. Then we change it and write it back. We're no longer modifying our own program! So we continue to do this merrily until a0 runs down, which is 2^{32} total iterations (seems like forever, I know). So the total iterations is 2^{32} (after it was -1) and 10 more before that for 2^{32} + 10 iterations.