

1 RISC-V Instruction Translation

1.1 In this question, translate the following RISC-V instructions into their binary and hexadecimal values.

a) `addi s1 x0 -24` = 0b1111 1110 1000 0000 0000 0100 1001 0011
1.1
= 0xFE800493
1.1

For this question, use the [CS 61C reference card](#) to obtain the information needed to convert each instruction to its binary representation. One thing that helps is splitting the parsing into sections.

For question 1, reading pages 1 and 2 of the reference card we can find:

```
addi s1 x0 -24:
Instruction format: I-type
immediate (12 bits): -24 = 0b1111 1110 1000
opcode (7 bits): 0b001 0011
funct3 (3 bits): 0b000
rs1 (5 bits): x0 = 0b00000
rd (5 bits): s1 = x9 = 0b01001
```

I-type instructions have the format:

```
imm[11:0] | rs1 | funct3 | rd | opcode].
```

Combining the values for our `addi` instruction into the I-type format gives us: `0b1111 1110 1000 0000 0000 0100 1001 0011 = 0xFE800493`

1.2 In this question, translate the following hexadecimal values into RISC-V instructions.

a) `0xFE05 0CE3` = `beq a0, x0, -8`
1.2

`0xFE05 0CE3` = `0b1111 1110 0000 0101 0000 1100 1110 0011`

For the reverse conversion, we first need to determine the instruction type. To do so, we examine the lower 7 bits `instruction[6:0]` as this will always be our opcode. Then, if necessary, we examine `funct3` / `funct7` as necessary to narrow down specific instruction.

`opcode` = `0b110 0011` which is for a B-type instruction.

A B-type instruction has the fields: [`imm[12|10:5]` | `rs2` | `rs1` | `funct3` | `imm[4:1|11]` | `opcode`] (note: an expression like [`imm[12|10:5]` | ...] is equivalent to [`imm[12]` | `imm[10:5]` | ...]).

We can pattern match as follows:

```
imm[12]: 0b1
imm[10:5]: 0b11 1111
rs2: 0b000000 = x0
rs1: 0b01010 = x10 = a0
funct3: 0b000
imm[4:1] = 0b1100
imm[11] = 0b1
```

With a B-type opcode and `funct3` = `0b000` we know that this is a `beq` instruction.

Assembling the full immediate, we get the 13-bit branch immediate to be [`imm[12]` | `imm[11]` | `imm[10:5]` | `imm[4:1]` | `0`] = `0b1111 1111 1100 0`. Notice that branch immediates have an implicit zero (see following question for explanation why). Converting from binary to decimal (recall immediates are in two's complement) we get `imm` = `-8`.

Thus, we can assemble our instruction as `0x2345 5487` = `beq a0, x0, -8`

- 1.3 Given the following RISC-V code and instruction addresses, translate the **jal** and **bne** instructions (you'll need your RISC-V reference sheet!) and determine the value of **R[ra]** during the execution of **loop**.

```

loop:
0x002CFF00:    add t1, t2, t0        0x00538333

0x002CFF04:    jal ra, foo            0x028000EF
                                   1.3

0x002CFF08:    bne t1, zero, loop     0xFE031CE3
                                   1.3
...

foo:

0x002CFF2C:    jr ra                R[ra] = 0x002CFF08
                                   1.3

```

For the first **jal** instruction, we can find that **rd = ra = 0b00001**. To determine the immediate, we need to move our PC to the first instruction starting at the label **foo**. $0x002CFF2C - 0x002CFF04 = 0x00000028 = 40$ in decimal. Thus, our 21-bit offset will be **0b0...00101000** which means our J-type immediate will be **imm[20:1] = 0b0...0010100** (recall the implicit 0). Reassembling the J-type instruction format, we get **0x028000EF**

For **bne**, we can decode the fields following the steps outlined in the previous questions. To find the B-type immediate, we need to move the PC from **0x002CFF08** to **0x002CFF00** (the start of label **loop**) which is **PC-8** bytes away. Thus, we have an offset of **-8 = 0b11...111000** which gives the immediate **imm[12:1] = 0b111111111100**. Assembling the instruction, we get **0xFE031CE3**.

R[ra] = 0x002CFF08 because the **jal** instruction sets the return address register to be **PC + 4** so that the callee can “return” to the caller by jumping back to the next instruction that should be executed.

2 RISC-V Addressing

We have several *addressing modes* to access memory (immediate not listed):

- a) Base displacement addressing adds an immediate to a register value to create a data memory address (used for `lw`, `lb`, `sw`, `sb`).
- b) PC-relative addressing uses the PC and adds the immediate value of the instruction to create an instruction address (used by branch and jump instructions).
- c) Register Addressing uses the value in a register as an instruction address. For instance, `jalr`, `jr`, and `ret`, where `jr` and `ret` are just pseudoinstructions that get converted to `jalr`.

2.1 What is the range of 32-bit instructions that can be reached from the current PC using a single branch instruction? Note that RISC-V branch instructions must support branching to 16-bit “compressed” instructions (enabled via an optional RISC-V extension).

Let’s first figure out how many bytes we can move the PC.

The B-format instruction encoding has support for a 12-bit immediate field. Because RISC-V must support 16-bit instructions, and our instructions will always be word-aligned (half-word for 16b instructions), the byte offset needed to branch to any instructions will always be divisible by 2. Thus, we assume an implicit 0 at bit 0 of our immediate field, allowing a 13-bit offset (that’s why the reference sheet describes the immediate as `imm[12:1]`!). It is signed, the branch immediate can move the PC a total range of $[-2^{12}, 2^{12} - 2]$ bytes (recall last bit is always 0).

To find the range of 32-bit instructions we can reach from the current PC, we look for all byte offsets we can reach that are divisible by 4. Thus, we have a range of $[-2^{10}, 2^{10} - 1]$ 32-bit instructions to branch to.

For those curious: the RISC-V “RVC” extension can be enabled to compress many common instructions to 16-bits.

2.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

The immediate field of the `jal` instruction is 20 bits, while that of the `jalr` instruction is only 12 bits, so `jal` can reach a wider range of instructions. As with above, let’s first find the number of bytes we can move the PC.

With a 20-bits of immediate (21 bits with implicit zero) for the `jal` instruction, we have a signed range of $[-2^{20}, 2^{20} - 2]$ bytes. The number of 4-byte instructions will be the range of addresses divisible by 4, so we can jump to reference within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

3 Two-Pass Assembly

Consider the following assembly code. Assume that `printf` exists in the C standard library and that `msg` exists at an unknown address in the `.data` section.

Address	Assembly
.data	msg: .string "Hello World"
.text	
0x0C	add t0, x0, x0
0x10	addi t1, x0, 4
0x14	loop: beq t0, t1, end
0x18	addi a0, a0, 1
0x1C	la a0, msg # load address of `msg`
0x20	jal ra, printf
0x24	n: addi t0, t0, 1
0x28	j loop
0x2C	end: ret

3.1 This code is output from the [Compiler](#) and [may](#) contain pseudoinstructions.

The compiler is responsible for translating high-level language code (e.g. C) to assembly. The compiler's output may contain pseudoinstructions which gets translated by the assembler.

3.2 Assume we are using a two-pass assembler. Fill out the symbol table after the first pass (top-to-bottom) of the assembler. Not all lines may be used. The order of entries in the table do not matter.

Symbol Table	
Label	Address
msg	?
loop	0x14
n	0x24
end	0x2C

The assembler performs two passes over the program to compute all the offsets. Branches and PC-relative jumps that target labels with *positive* offsets are unknown in the first pass (ex: during the assembler's first pass, it encounters the label `end` in `beq t0, t1, end` before it has seen the address of `end`).

The solution is to take two passes over the program. Pass 1 remembers the positions of labels which are stored in the symbol table, and pass 2 uses label positions to generate the machine code. References to static data and external functions cannot be determined at this stage, however, because full 32-bit addresses are unknown until the linker creates the executable.

For the first pass of the assembler:

- a) `msg` is defined in the data section and will be static data with an absolute address. Thus, it's address will be unknown until the linker stage. `msg` will be defined in the symbol table and passed to the relocation table.
- b) `loop` is defined at address `0x14` and is defined in the symbol table.
- c) `n` is defined at address `0x24` and is defined in the symbol table.
- d) `end` is defined at address `0x2C` and is defined in the symbol table.

Note that the symbol table contains all labels defined within the current file that can be used externally. Since `printf` is not defined in the current file, it isn't in the symbol table.

- 3.3 After the first pass of the assembler, which of the instructions do not have their addresses fully resolved?

Answer:

- a) `beq t0, t1, end`
- b) `la a0, msg`
- c) `jal ra, printf`

At the end of the first-pass, all instructions with forward references (positive PC-relative offsets) are still unknown. This is the case for `beq t0, t1, end`. Additionally, any absolute addresses or links to external library functions are unknown and cannot be translated until the linker stage (`jal ra, printf` and `la a0, msg`).

- 3.4 After the second pass of the assembler but before the linker, which of the instructions do not have their addresses fully resolved?

Answer: `jal ra, printf` and `la a0, msg`

The address of the label `end` is known in the symbol table at the end of the assembler's first pass. On the second pass, the address is resolved and the instruction can be translated to machine code.

`printf` is an external library function whose address is unknown and is determined by the linker. Additionally, `msg` will be stored as absolute address (not PC-relative) at an address determined by the linker, and thus `la a0, msg` cannot be translated at this stage.