CS61C Spring 2025

Instruction Translation, AMAT

Discussion 5

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`

b) `sh s1 4(t1)` = `0b0000 0000 1001 0011 0001 0010 0010 0011` 1.1

= `0x00931223` 1.1

For the second question, following a similar method using the CS 61C reference card:

```
sh s1 4(t1):
Instruction format: S-type
rs1: s1 = t1 = 0b00110
rs2: s1 = x9 = 0b01001
immediate: 4 = 0b0000 0000 0100
opcode: 0b010 0011
funct3: 0b001
```

Notice that S-type instructions are encoded as follows: `[imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode]`. `imm[11:5]` is bits 11-5 inclusive of the immediate, `imm[4:0]` is bits 4-0 of the immediate, and so on.

Thus, assembling the S-type instruction: `0b0000 0000 1001 0011 0001 0010 0010 0011 = 0x00931223`

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: 0b00000 = 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`

b) `0x2345 54B7 = lui s1 0x23455` (1.2)

`0x2345 54B7 = 0b0010 0011 0100 0101 0101 0100 1011 0111`

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

`opcode = 0b011 0111` which is for a U-type instruction (specifically for `lui`).

A U-type `lui` instruction has the fields: `imm[31:12] | rd | opcode`, so we can pattern match as follows:

```
lui rd immu
immu = Immediate[31:12] = 0b0010 0011 0100 0101 0101
     = 0x23455
rd: 0b01001 = s1
```

To get the answer `0x2345 54B7 = lui s1 0x23455`

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 Caches Intro: AMAT

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

3.1 Suppose your system takes 100ns to access main memory. We decide to add a cache with a measured hit time of 25ns and miss rate of 25%. What is the average memory access time of the system?

Answer: 50ns

We are looking for a solution to the AMAT equation. The hit time for the new L1$ is 25ns. The miss rate is 25% and the miss penalty will be the 100ns required to access main memory in the case of a cache miss. Thus, our solution is $\text{AMAT} = 25\text{ns} + 0.25 * 100\text{ns} = 50\text{ns}$. By adding a cache, we have effectively halved the time spent waiting for memory accesses.

3.2 In a new 2-level cache system, after 100 total accesses to the cache system, we find that the L2$ (L2 Cache) ended up missing 20 times. What is the global miss rate of L2$?

Answer: 20%

$$\text{Global Miss Rate} = \frac{\text{Local Missed Accesses}}{\text{Total System Accesses}} = \frac{20}{100} = 20\%$$

3.3 Given the system from the previous subpart (100 total accesses, 20 L2$ misses), if L1$ had a local miss rate of 50%, what is the local miss rate of L2$?

Answer: 40%

$$\text{Local Miss Rate} = \frac{\text{Local Missed Accesses}}{\text{Local Cache Accesses}} = \frac{20}{50\% * 100} = \frac{20}{50} = 40\%$$

We know that L2$ is accessed when L1$ misses, so if L1$ misses 50% of the time, that means we access L2$ 50 times, of which we ended up having 20 misses in L2$.

For the following subparts, suppose we have a new system that consists of:
1. An L1$ that has a hit time of 2 cycles and a local miss rate of 20%
2. An L2$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

EDIT 2/26: the original worksheet listed L2$ hit time as 16 cycles. The correct L2$ hit time is 15 cycles.

3.4 What is the local miss rate of L2$?

Answer: 25%

The number of accesses to the L2$ is the number of misses in L1$, so we divide the global miss rate of L2$ with the miss rate of L1$.

$$\text{L2\$ Local Miss Rate} = \frac{\text{Misses in L2\$}}{\text{Accesses in L2}} = \frac{\text{Misses in L2\$}}{\text{Total Accesses}} / \frac{\text{Misses in L1\$}}{\text{Total Accesses}} = \frac{\text{Global Miss Rate}}{\text{L1\$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%$$

3.5 What is the AMAT of the system?

Answer: AMAT $= 2 + 20\% \times (15 + 25\% \times 100) = 10$ cycles

The miss penalty of the L1$ is the "local" AMAT of the L2$.

3.6 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$. If the L3$ has a local miss rate of 30%, what is the largest hit time that L3$ can have?

Answer: 30 cycles

Let H = hit time of the cache. Extending the AMAT equation so that the Miss Penalty of the L2$ is the "local" AMAT of the L3$, we can write:

$$\text{AMAT} = 2 + 20\% * (15 + 25\% * (H + 30\% * 100)) \leq 8$$

Solving for H, we find that $H \leq 30$. So, the largest hit time is 30 cycles.