Programming Assignment #1
Instruction Processing

CS 3220 / 5220 Spring 2024
20 points
due Saturday, Feb. 3rd, 11:59 pm

1 Instruction Processing

You'll write a problem to simulate the processing of machine instructions in a CPU. These instructions are similar in form to RISC-V instructions, which we'll study in detail.

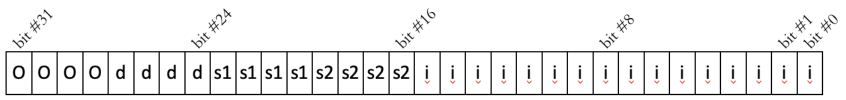
You may work individually or with a partner.

2 Instructions

An instruction is a four-byte (32-bit) integer.

2.1 Instruction format

The fields of an instruction are located in fixed positions in these 32 bits, as shown in the following diagram:


bit #31
bit #24
bit #16
bit #8
bit #1
bit #0
O O O O d d d d s1 s1 s1 s1 s2 s2 s2 s2 i i i i i i i i i i i i i i i i

O	opcode	4 bits
d	Rd (destination register)	4 bits
s1	Rs1 (source register #1)	4 bits
s2	Rs2 (source register #2)	4 bits
i	immediate value	16 bits

Undergraduates: assume that an immediate value is non-negative. Graduate students: treat signed immediate values correctly, using two's complement representation (see below).

In your program, the instructions will be stored in an integer array (or list) that represents the memory of the system.

2.2 Registers

A register is a storage location for a single four-byte integer. There are a small number of registers. They are named R0, R1, ..., Rn-1.

The register R0 is a read-only register that always has the value zero.

2.3 Instructions to implement

In the following, Rd denotes the destination register; Rs1 is the first source register, and Rs2 is the second source register.

Implement the following instructions:

NOOP

opcode = 0 (0000). Do nothing.

ADD Rd, Rs1, Rs2

opcode = 1 (0001). Add the contents of `Rs1` and `Rs2` and put the result in `Rd`.

ADDI Rd, Rs1, immed

opcode = 2 (0010). Add the contents of `Rs1` to the immediate value and put the result in `Rd`.

BEQ Rs1, Rs2, immed

opcode = 3 (0011). If the contents of `Rs1` and `Rs2` are the same, then set `next_pc` to `pc + immed`; otherwise do nothing.

JAL Rd, immed

opcode = 4 (0100). Save `pc + 1` to `Rd` and set `next_pc` to the contents of `pc + immed`.

LW Rd, immed(Rs1)

opcode = 5 (0101). Load the value from memory location `immed + Rs1` and put that value into `Rd`.

SW Rs1, immed(Rs2)

opcode = 6 (0110). Store the value from `Rs1` into memory location `immed + Rs2`.

RETURN

opcode = 7 (0111). Terminate the processing of instructions.

3 The Program

The basic function of a microprocessor can be described in pseudocode:

```
while true:
  fetch the next instruction, which is the instruction at location pc
  set next_pc to pc + 1
  decode the instruction
  execute the instruction // JAL will change next_pc; BEQ could change next_pc
  set pc to next_pc
```

Write a program to implement this. Create a `CPU` class and an `Instruction` class. `CPU` should have these member variables:

- `pc`: an integer
- `next_pc`: an integer
- memory: an array (list) of `MEM_SIZE` integers
- regs: an array (list) of `NUM_REGISTERS` integers

Set `NUM_REGISTERS` to 16 and `MEM_SIZE` to 65536. (In Python, memory and registers will be integer lists.)

Put the value zero into each register and into each memory location initially.

The `Instruction` class should have these five member variables: `opcode`, `Rd`, `Rs1`, `Rs2`, and `immed`. All are integers.

The instructions themselves will be in the memory array/list. The variable `pc` is an index into this array. So for example, if the value of `pc` is 100, then then next instruction fetched would be the integer at position 100 in the array/list.

3.1 Building instructions

Write a function to take the components that make up an instruction (the opcode, the destination register, source register #1, source register #2, the immediate value) and build a four-byte instruction. Not every instruction will have each of these components.

I did the following:

```
def build_instruction(opcode, Rd, Rs1, Rs2, immed):
  instr = opcode << 28
  if Rd is not None:
    instr = instr + (Rd << 24)
  if Rs1 is not None:
    instr = instr + (Rs1 << 20)
  if Rs2 is not None:
    instr = instr + (Rs2 << 16)
  if immed is not None:
    instr = instr + immed
  return instr
```

4 Processing

Write the following functions. I made each of these functions a member function of `CPU`.

4.1 IF: Instruction fetch

Read the instruction at location `mem[pc]`. Set `next_pc` to `pc+1`. Each instruction will be a four-byte integer.

4.2 ID: Instruction decode

Pull out the opcode, the destination register (Rd), source register #1 (Rs1), source register #2 (Rs2), and the immediate value (immed). Not every instruction will have each of these values, but ID treats each instruction as if it does have each of these values. (This is called *fixed-field decoding* and is a means for improving the efficiency of instruction processing.)

The opcode is in bits 28-31 of the instruction. Rd is in bits 24-27. Rs1 is in bits 20-23. Rs2 is in bits 16-19. The immediate value is in bits 0-15. Use bitwise operations to extract these values. After extracting these values, create an instance of `Instruction` having its values set to the various extracted values.

Note: this doesn't represent the actual encoding of RISC-V instructions, but it's conceptually similar.

To extract specific bits from an integer, use a combination of bit shifting and bitwise operations.

For example: suppose I want the value that bits 3-7 of the integer `val` represent. I first shift `val` 3 bits to the right and then perform a bitwise AND with 15 (which is 1111 in binary): `result = (val >> 3) & 15`

4.3 EX: Execute

Perform the actual arithmetic operation specified by the opcode, using register value(s) or immediate value, as appropriate. After the specific processing for each opcode, set `pc` to `next_pc`.

In each case, `Rd` is an entry in the `cpu.regs[]` array/list. For example, `R4` corresponds to `cpu.regs[4]`. Similar for `Rs1` and `Rs2`.

4.3.1 ADD and ADDI

Perform the addition, using either the two source registers (for `ADD`) or `Rs1` and the immediate value (for ADDI). Put the result in `alu_result`, and then put `alu_result` into `Rd`.

4.3.2 BEQ

Compare the values in the two source registers. If the values are equal, then set `next_pc` to the value of `pc` plus the immediate value; otherwise, do nothing.

4.3.3 LW

Calculate the effective address: `Rs1` plus the immediate value. Put this in `eff_address`. Then, put the contents of `mem[eff_address]` into `Rd`.

4.3.4 SW

Calculate the effective address: `Rs2` plus the immediate value. Put this in `eff_address`. Then, put the contents of `Rs1` into `mem[eff_address]`.

4.3.5 JAL

Set `alu_result` to `pc+1`, put `alu_result` into `Rd`, and set `next_pc` to `pc` plus the immediate value.

5 Testing your code

5.1 Testcase #1

Here's a sequence of twelve instructions:

```
noop
addi r1, r0, 8     # r1 now has 8
addi r2, r0, 7     # r2 now has 7
add  r3, r1, r1    # r3 now has 16
add  r4, r2, r2    # r4 now has 14
beq  r3, r4, 3     # branch if r3 == r4; branch is not taken
addi r8, r0, 10    # r8 now has 10
jal  r0, 2         # save pc+1 to r0 and jump to pc+2
addi r8, r0, 1000  # this instruction is skipped
sw   r2, 16(r8)    # save the value 7 to mem[16+10]
lw   r5, 16(r8)    # load mem[16+10] into r5; r5 now has the value 7
return
```

After executing these instructions, you should have these values in the registers, assuming that your first instruction was at location 100:

```
R0=0 R1=8 R2=7 R3=16 R4=14 R5=7 R6=0 R7=0
R8=10 R9=0  R10=0 R11=0 R12=0 R13=0 R14=0 R15=0
```

and `memory[26]` should be 7.

5.1.1 Creating this testcase

Create constant values to represent the opcodes:

```
NOOP = 0
ADD  = 1
ADDI = 2
BEQ  = 3
JAL  = 4
LW   = 5
SW   = 6
RETURN = 7
```

Then, here's how to load instructions into the integer array/list that represents memory:

```
i0 = build_instruction(NOOP, None, None, None, None)
i1 = build_instruction(ADDI, R1, R0, None, 8)
i2 = build_instruction(ADDI, R2, R0, None, 7)
i3 = build_instruction(ADD, R3, R1, R1, None)
# etc.

cpu.mem[100] = i0
cpu.mem[101] = i1
cpu.mem[102] = i2
cpu.mem[103] = i3
cpu.mem[104] = i4
cpu.mem[105] = i5
cpu.mem[106] = i6
cpu.mem[107] = i7
cpu.mem[108] = i8
cpu.mem[109] = i9
cpu.mem[110] = i10
cpu.mem[111] = i11
```

Then set `cpu.pc` to 100 and start the while loop.

5.2 Testcase #2

To make sure that your BEQ instruction is correct:

```
addi r1, r0, 5    # r1 now has 5
addi r2, r0, 6    # r2 now has 6
add  r3, r2, r1   # r3 now has 11
add  r4, r1, r2   # r4 now has 11
beq  r3, r4, 3    # branch if r3 == r4; branch is taken
addi r8, r0, 10   # skip
jal  r0, 2        # skip
addi r8, r0, 30 # r8 now has 30
sw   r3, 10(r8)   # store 11 in mem[10+30]
lw   r5, 10(r8)   # load mem[10+30] into r5; r5 now has the value 11
return
```

After executing these instructions, you should have these values in the registers:

```
R0=0 R1=5 R2=6 R3=11 R4=11 R5=11 R6=0 R7=0
R8=30 R9=0 R10=0 R11=0 R12=0 R13=0 R14=0 R15=0
```

and `memory[40]` should be 11.

6 What to Submit

Submit your code.

7 Graduate students

Graduate students, and undergraduates for extra credit, do the following.

7.1 Reading from a file

Implement the SUB, SUBI, and JALR opcodes. Also, provide the capability to read instructions from a text file. The text file should contain a sequence of instructions, one per line. You can ignore case.

7.2 Signed integers

Also, correctly treat signed immediate values. With 16 bits, a signed value can range from -32768 to 32767. Assume a two's-complement representation of signed integers.

Two's complement: if the most significant bit of an N-bit binary number b is set, then, interpret the signed number as $-(2^N - b)$, with b as an unsigned number in this difference. Example using four bits: 1010 should be interpreted as $-(16 - 10) = -6$; similarly, 1000 as a signed integer is -8.

7.3 Additional opcodes

SUB Rd, Rs1, Rs2

opcode = 9 (1001). Subtract the contents of `Rs2` from `Rs1` and put the result in `Rd`.

SUBI Rd, Rs1, immed

opcode = 10 (1010). Subtract the immediate value from `Rs1` and put the result in `Rd`.

JALR Rd, Rs1, immed

opcode = 12 (1100). Save `pc + 1` to `Rd` and set `next_pc` to the contents of `Rs1 + immed`.