ICSI 404 – Assignment 3: CPU Design

The third programming assignment builds on the previous ones to emulate the architectural design of a computer. Consequently, it needs an implementation of the byte-array abstraction of the main memory. For this series of assignments, we do not consider the timing aspects of CPU instruction cycle and memory access. Consequently, we completely avoid all aspects of cache emulation and only implement the single-cycle execution abstraction of the CPU. The simple instruction set includes only the basic capabilities of arithmetic/logic operations (implemented in the ALU design assignment), input/output using software interrupt, unconditional jump and conditional branches. While we would not implement stack operations as part of this assignment, we leave the opcode space open for future enhancement.

Task 1: Memory

The `Memory` class implementation **must follow these specifications**:

1. The `Memory` class must be **a new class** that does not inherit from (i.e., extend) any other class. A memory object is meant to be instantiated by a computer object (see Task 2). However, you **must** implement the Memory class independently in the `Memory.java` file, and **not** as a nested class within the `Computer` class. The `Memory` class implementation needs to be tested thoroughly with a tester class created in `TestMemory.java`, before proceeding to develop the Computer class for the next task.
2. A Memory object must have an array of bytes as part of its data field. For our emulated computer, we do not plan on working with a large program or large amount of data. Thus, the memory size can be hard-coded to 256 bytes, which will limit the volume of printed output when we do a memory dump to standard output for testing or debugging purposes. You may like to define the memory size as a constant (setting it to 256) and make the size readable via a public accessor. This way you can change the size later if you need to.
3. You **must** create the public interface (not referring to Java interface) for your class by implementing the following methods:
   a) At creation, all bytes in the memory need to be initialized to zero.
   b) Implement the **accessor** for accessing up to a maximum 4 bytes of data (check must be enforced for illegal argument value) starting from a given memory address (a long-word whose integer value serves as the index of the internal byte-array). The return value of the accessor is a long word that can be stored in a CPU register. Assume that data is stored in memory in the big-endian convention. The accessor will be used for both instruction fetch (2 bytes) and data fetch (1, 2 or 4 bytes). For example, assuming the memory location 0 and 1 have the bytes A1 and B2 in hex, a memory read of 2 bytes from address 0 should return a long-word with hex value 0000A1B2. The method prototype may look like

      ```
      public LongWord read(LongWord address, int numBytes)
      ```

   c) Implement the **mutator** for writing to memory up to a maximum of 4 bytes starting from a given memory address. Again, we need to adhere to the big-endian convention for writing to memory, but count the number of bytes from low order of significance. Continuing with the last example,

writing 2 bytes from a long-word with hex value xxxxA1B2 at memory address 0 will place the byte A1 at address 0 and B2 at address 1. The method prototype may look like

```
public void write(LongWord address, LongWord word, int numBytes)
```

4. Fully test the memory read-write functionalities by creating a separate MemoryTest.java file and implementing an appropriate runTests method, before proceeding to the next task.

Task 2: Computer

With memory and ALU classes, it is time to start building a fully functional CPU. This will be done by creating a new class called Computer, which must follow the following specifications:

1) Like all other classes built before, the `Computer` class must be **a new class** that does not extend any other class. However, it must be composed of objects of the classes that we built so far. This includes the LongWord (for registers), Memory and ALU classes.
2) A Computer object must have the following fields (additional ones may be added, if needed):
   a) A Boolean flag to indicate whether the computer is halted or not. This flag should be initialized as "not halted" and will only be flipped by the halt instruction (see later)
   b) An instance of ALU object (from last assignment)
   c) An instance of Memory object (from Task 1)
   d) An array of long-words to emulate the bank of 16 addressable registers. As discussed in class, a machine-code program would make every attempt to store "local variables" in these registers to avoid expensive memory access as much as possible. A register from the bank will be addressed by 4-bit address representing integers in the range 0-15. We can use the register address as the index of the array, and refer to the registers as R0 – R15 in the assembly code.
   e) A long-word implementing the program counter (PC), that tracks the sequence of machine instructions executed from a program. The PC value needs to be initialized to 0, which will be the starting address of a program in our computer. Under the single-instruction execution model of our computer, the PC should be incremented by 2 bytes (the size of every instruction in our instruction set), unless an explicit jump or conditional branch instruction loads a new instruction address to the PC.
   f) A long-word implementing the instruction register (IR) that holds an instruction fetched from memory until its execution is complete and it is time for fetching the next instruction as indicated by the PC.
   g) Long-words implementing the OP1 and OP2 registers to store the operands (actual values, not the referral address to a register) for the ALU, and the register to temporarily hold the output from the ALU before storing back to the destination register.
   h) Additional long-words and integers as needed to hold mask values, extracted fields from the machine instructions (e.g., opcode, register indexes, literal values, etc.)
3) Apart from the constructor, you **must** restrict the public interface to your Computer class to a single `run` method with no return value (void). Once called, this method will continue looping to execute one machine instruction from a program per iteration, starting at address 0 (that is how we initialize the PC value). The loop should exit when the Boolean "halted" flag as set (which can happen only after executing a halt instruction in the course of running the program). An important reminder here is that the potential that run method falls in an infinite loop if it never encounters a halt instruction.

4) The complete execution of every machine instruction should be clearly divided into the four stages to mimic the operations in an actual CPU. You must create the four void methods fetch(), decode(), execute() and store() and clearly divide the actions performed among the stages, as they happen in a real CPU. These methods are to be called in sequence once per loop iteration in the run() method. Since run() is the only method that invokes these four methods, they should be made private. What follows is a brief outline of how the activities are divided among methods representing four stages:
   a) Fetch: Read the next 2 bytes (all instructions are 16-bits) from memory using the PC as the address and store the returned long-word in the one implementing the IR. Also, increment the PC by 2, but refrain from using the ALU as that would mess up the flags (in status register). In real CPU, this is done using a special hardware (a simple counter or AGU). You can do this by extracting the integer value of the PC, adding 2 to it and setting PC back to the incremented value. This will be the potential address of the next instruction, unless a jump or conditional branch needs to overwrite the PC.
   b) Decode: Examine IR to determine which register(s) have our operands. This will require shifting and masking. You can do this either using the methods of LongWord first followed by convertion to integer index, or converting the long-word address to an integer first followed by Java bit operators (~, &, |, <<. >>, >>>) on the integers. Considering arithmetic and logic operations for example, the data from the two operands' source registers are to be copied into OP1 and OP2.
   c) Execute: As for the examples of arithmetic and logic instructions, pass the ALU control bits (the code rightmost 3-bits of the 4-bit opcode – see chart from last assignment) to the ALU operate() method along with OP1 and OP2. Hold the long-word returned in the "result" long-word.
   d) Store: Copy the value from the result long-word into the destination register as indicated by the instruction. To find the integer index of the destination register, use shifting and masking in a way similar to the decode stage.

   An obvious question is why we get the values from the registers and prepare OP1 and OP2 for the ALU, and make a call to the ALU method in the execute phase and postpone storing the result to the destination register for the store phase. As programmers, we can do all these in a single shot within a single method and this may even reduce the need for some unnecessary data fields in the computer class. The first reason is that we are emulating the hardware mechanism, and it takes non-overlapping time for each stage of fetch-decode-execute-store to do their work. The second reason is that a register can only be read or written at a time, and we update the register only in the store phase while having read. An instruction that performs "R1 ← R1 + R2" (i.e., R1 += R2) needs to have clear write-access to R1 when it is guaranteed to be not being read.

Instruction Format (for ALU-based and other new instructions)

All machine instructions are 2-bytes long (16-bits) and are compactly stored in memory. The machine code program starts at address 0 (the address in 32-bit long and is stored as a long-word). Thus, the next instruction is at address 2 with the following one is at address 4, and so on.

Instructions are fetched by reading 2 bytes from memory (as specified by the address) into the 32-bit instruction register (IR). Even though there is space for fetching and holding two instructions in the IR, that is not how our machine operates. This also means that the upper (most significant) half of the IR is zeroed out at all times and never used.

The opcode occupies the most significant 4 bits (i.e., bits 15 – 12 within the 16-bit format). Remaining 12 bits are interpreted based on the opcodes as described below. The decode stage must prepare input data accordingly. The actions needed to execute the instructions, as well as any necessary store-back are also explained. Some instructions (such as ALU operations) are grouped when they share common decoding and execution rules.

1) Arithmetic and Logic instructions (opcodes in the range 1000 – 1111):

   As described in the last assignment, all opcodes with the leftmost bit or MSB (most significant bit) 1 make use of the ALU. Moreover, the remaining 3 bits from right serve as the ALU control code. When represented as integers, these opcodes fall in the range 8 – 15, where the ALU control code can be easily extracted as the remained when you divide the opcode value by 8. As all our ALU operations binary (you must have noted how we could achieve the unary NOT operation by XORing with -1), we need 3 register indexes for two operands and the destination. The format we choose for the these 3-Regeister (3R) instructions will have from left to right

   4-bit opcode (bits 12 – 15)
   4-bit destination/target register (bits 8 – 11) where the result goes
   4-bit opearnd1 (OP1)
   4-bit operand 2 (OP2).

   The assembly language instruction for "R3 ← R1 + R2" will thus be written as

   ADD R3 R1 R2

   and will assemble to machine code as (1011 is opcode for ADD)

   1011 0011 0001 0010.

2) The Halt (HLT) instruction (opcode 0000):

   Recall that we create a Boolean flag that indicates whether the computer is halted or should keep running. When we encounter a halt instruction, that bit should be set so as to indicate that the computer should not continue looping in the single-cycle instruction mode and the emulation program eventually stops. More specifically, this flag should be set in the execute stage and the halt instruction has no activity for the store stage. Also, this instruction takes no parameters, so decode and store should not alter the registers. The bits of halt after the 0000 are ignored, so 0x0000, 0x0123, 0x0FFF, etc. will perform the same halt operation. However, while writing machine code, we will write the unique code 0x0000 whenever we need a halt instruction.

3) The Interrupt/Trap (INT) instruction (opcode 0001):

   An interrupt is a special kind of instruction. There are two uses for interrupts – one is for hardware to interrupt the CPU (called hard interrupts), to make the devices can work asynchronously with the CPU. The second kind, called soft interrupt or trap, makes use of this explicit machine instruction to allow the program to change its state which would not have happened in normal course. These are used, for example, to call the kernel in an operating system. Here, we are going to simulate interrupt mechanism to do some output.

The interrupt instruction has one parameter – which interrupt to perform. Think of this as which hardware device needs attention or which kernel-call we are executing. For decoding purpose, we will use the lower byte for the interrupt code (allowing 256 possible unsigned code values, 0 – 255) and ignore bits 8 – 11. However, for execution, you only need to implement of two interrupts (namely 0 and 1), which will be used as the only output mechanism for our emulator.

0001 0000 0000 0000 – print all of the registers to the screen (using long-word's toString)
0001 0000 0000 0001 – print all 256 bytes of memory to the screen (4 bytes per line)

4) The Move (MOV) instruction (opcode 0010):

Having dealt with the output, we need the ability to for data input. The simplest way to do this is by assigning the input value to a register, but our computer does not yet have the ability to set a register to a given value. The Move instructions add this capability. Note that it needs a destination register just like the ALU instructions, but only one operand. However, there is need for more bits for this operand, as the operand value directly resides as a literal within the instruction, such as

MOV R1 10 – moves the literal value 10 into R2.

The assembled machine code for this instruction would look like

0010 0001 0000 1010 (4 for the opcode, 4 to indicate the register, 8 to indicate the value).

Note that the literal value is a signed integer ranging between -128 and 128 within the 8-bits. So, it can be negative, and needs to be sign extended. For example,

0010 0001 1111 1111 (MOV R1 -1)

should leave R1 with 32 bits all set to 1!

Implement the functionality of the Move instruction by properly dividing the work among the decode, execute and store stages as necessary.

5) The Jump (JMP) instruction (opcode 0011):

So far, programs fall through sequentially by running one instruction after another until we hit a HLT instruction. We add an unconditional jump, first. You can think of this like a `goto` statement in C. The jump location, representing absolute value of a memory address, must be non-negative. In addition, because of the 2-byte length of instruction, it is not meaningful to jump to odd-valued memory address. Thus, the value of the jump address will be restricted to even unsigned values. The format for jump will be like

JMP 10 – This will jump to the instruction at address 10 (decimal).

To hide the least significant '0' bit of the even jump address in the machine code encoding, we right-shift the actual address by one bit (effectively dividing by 2). Thus, the above is assembled in bits as

0011 0000 0000 0101 // opcode is 3, encoded address is 10/2 = 5 (0000 0000 0101 in binary)

Note that this effectively covers even addresses in the range 0 through $2(2^{12}-1) = 8190$. Recognize this bit pattern in the decode phase and overwrite the PC with the correct jump address in the store phase (such as 10 in this case, and not 5) by unhiding the hidden '0'.

6) The Compare (CMP) instruction (opcode 0100): **Optional with Bonus Credit*****

Unconditional jumps, while necessary, don't let us implement high level concepts like if, for or while, since they have no capability of checking a condition. So, we need to implement conditional jumps, also known as Brach instructions. For our CPU, we do this using a sequence of two instructions. First, using the Compare instruction, we set the Boolean flags ZF and NF, and then take branches based on these two flag values.

We will implement the Compare instruction by performing a subtraction in the ALU. For example,

CMP R1 R2 – compare registers R1 and R2 by computing R1-R2

subtracts R2 from R1, but does not make any attempt to save the result an any register. The only effect is setting the ZF and NF flags in the ALU. Note that CMP used opcode 0100, whereas SUB uses opcode 1100 – they differ only in the leftmost bit. The decoding process for Compare can extract the ALU code exactly in the same way as in ALU-based instructions (i.e., taking remainder after dividing by 8) and invoke the ALU operation with code "100". Note that the ZF flag is set if R1==R2, and the NF flag is set if R1<R2. The net effect of the flag combinations is the following:

ZF=1 and NF=1: Impossible
ZF=1 and NF=0: R1 = R2
ZF=0 and NF=1: R1 < R2
ZF=0 and NF=0: R1 > R2

Because there is no need to store the result of the subtraction, we ignore the bits for the destination register, but keep the operand1 and operand2 registers in the same location as in the ALU-based instructions. The above assembly instruction should be coded as instruction as

0100 XXXX 0001 0010 – XXXX denotes ignored bit, should be assembled as 0000 as a convention

7) The Branch (Bxx) instructions (opcode 0101): **Optional with Bonus Credit*****

There are six possible branch types, based on the six logical comparisons ">", ">=", "<", "<=", "==", and "!=". Six cases require 3 bits for condition codes (with 8 possibilities), leaving two unused codes. One trick that computer architects do to reduce the number of cases is to recognize that the "<" and "<=" cases by instructing assembly language programmers to reverse the order of the operands and use the corresponding greater than cases.

Instead of comparing a>b, we can say b<a; instead of saying a>=b we can say b<=a. Accordingly, we reserve 2 condition code (CC) bits following the opcode. The remaining 10 bits are used as relative branch address (relative to current instruction) with the hidden '0' LSB (just like Jump) and hence can be a signed integer (in 2's notation). The leftmost of these 10 bits serves as the sign-bit of the relative address. Therefore, we have a family of 4 conditional branch instructions with the most significant 6 bits (4 bits opcode and 2 bits condition code) as

```
BNE (0101 00XX XXXX XXXX) – Branch if Not Equal
BLT (0101 01XX XXXX XXXX) – Branch if Less Than
BEQ (0101 10XX XXXX XXXX) – Branch if Equal
BLE (0101 11XX XXXX XXXX) – Branch if Less than or Equal
```

As an example, the instruction BEQ 12 will be encoded as

0101 1000 0000 0110 // Underlined bits are condition codes, encoded address value 12/2 = 6

The format of this family of instructions looks like

0101 CCSA AAAA AAAA

where CC is condition code, S is the sign bit of the address, and A's show the address (relative).

The evaluation of the bits must take place in the execute phase and the overwriting of the PC, if required, must occur in store phase. It is important not to use the ALU for address computation as before so as to not mess up the ALU registers. You may either do the branch target computation in integer mode or may like to add a second instance of ALU object for address computation.

Loading a machine program and testing

We need to be able to write a program into memory. Create a method called

void preload(String[])

in your Computer class. Each string will be bits in the format that we have shown in this document, i.e., 4 groups of 4 0's or 1's. Write these bits into the memory using the write method of the Memory class (creating a long-word for a pair of instructions).

For example, let us assume we have this code (interrupt to dump registers and halt):

0001 0000 0000 0000
0000 0000 0000 0000

We need to build a long-word that holds these. We can then store it into memory. If we have an odd number of instructions, we can either detect that and write 2-bytes or pad the last two bytes with all 0's and write all 4 bytes..

With these in place, we can reasonably test our CPU. Create a new test class TestComputer with the same runTest method as the interface. As with the other tests, these tests must be independent of each other and there must be reasonable coverage. For this, create a new CPU for each test if you have to. Call preload followed by your looping run method. Use MOV to set values in registers, use some ALU based instructions, print out the results using "INT 0" and always halt with HLT. Confirm using "INT 1" that the program is stored correctly in memory. Check the correctness of JMP and Bxx branches by clever placement so that you can skip over a signature action that you have tested (e.g., assignment or arithmetic/logic computation).

You must provide a test file (TestComputer.java) that implements void `runTests` method and call it from your main, along with your existing tests. Submit the files `Memory.java`, `TestMemory.java`, `Computer.java` and `TestComputer.java`. In case you have updated the source code in `LongWord.java` or `ALU.java` files, please make sure to include the compatible updated versions.

You must submit buildable .java files for credit.