ICSI404 – Assignment 7 – implement more instructions

This assignment builds on the previous assignment. Remember to use bit comparison, not getSigned() or getUnsigned() to check opcodes.

Part 1 – HALT!!!

Recall that we create a bit in our computer that indicates if the computer should keep running. When we encounter a halt instruction (0000), that bit should be altered to indicate that computer should not continue looping. This can be done in the execute() or store() function. The halt instruction takes no parameters, so decode() and store() should not alter the registers. The bits of halt after the 0000 are ignored.

Implement the halt instruction.

Part 2 – Move

One of the problems with our computer is that it has no ability to set the registers to any values. Move will fix that. It works a little different than the ALU related instructions:

move R2 10 – moves the literal value 10 into R2.

The bits for this instruction would look like this:

0001 0010 0000 1001 (4 for the opcode, 4 to indicate the register, 8 to indicate the value).

Note that the value can be negative. If that is the case, it needs to be sign extended.

0001 0001 1111 1111 (move R1 -1) should leave R1 with 32 bits all set to ON!

Implement the move instruction by modifying decode(), execute() and store() as necessary.

Part 3 – Interrupt

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). They do this so that the processor and the device can work independently most of the time. A soft interrupt allows the program to change its state. These are used, for example, to call the kernel in an operating system. Here, we are going to SIMULATE using interrupts 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 this project, we will define two interrupts:

0010 0000 0000 0000 – print all of the registers to the screen

0010 0000 0000 0001 – print all 1024 bytes of memory to the screen.

The printing can be done in the usual way (system.out.print and friends) and can be done in the execute() method. Note that decode() and store() don't do anything for interrupts.

Part 4 - Initialization

We need to be able to write a program into memory. Create a method called "void preload(string[])" on your CPU. Each string will be bits in the format that we have shown in these documents – 4 groups of 4 0's or 1's. Write these bits into the memory using the memory store method (creating pairs of longwords).

For example, let us assume we have this code (two halts):

0000 0000 0000 0000

0000 0000 0000 0000

We need to build a longword that holds these. We can then store it into memory.

If we have an odd number of instructions, we can pad the last write with all 0's.

With these in place, we can reasonably test our CPU. Create a new test class (cpu_test1) with the same runTest() interface. Create a new CPU for each test. Call preload and your loop method. Use move to set values in registers, use the ALU, print out the results using interrupt 0 and halt. Confirm using interrupt 1 that the program is stored correctly in memory.

<u>You must submit buildable .java files for credit.</u>

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Unit Tests	None (0)	Partial Coverage (7)	All methods covered, needs more cases (13)	All methods/cases covered (20)
Halt	None (0)	Attempted (5)		Completely working (10)
Move	None (0)	Attempted (5)		Completely working (10)
Interrupt 0	None (0)	Attempted (5)		Completely working (10)
Interrupt 1	None (0)	Attempted (5)		Completely working (10)
Preload	None (0)	Attempted (10)		Completely working (20)