Assignment 3

Due: Nov 3rd, 11:59 pm
Weight: 10% of total class grade

Overview

In this assignment, students will add 6 new instructions to the CPU.

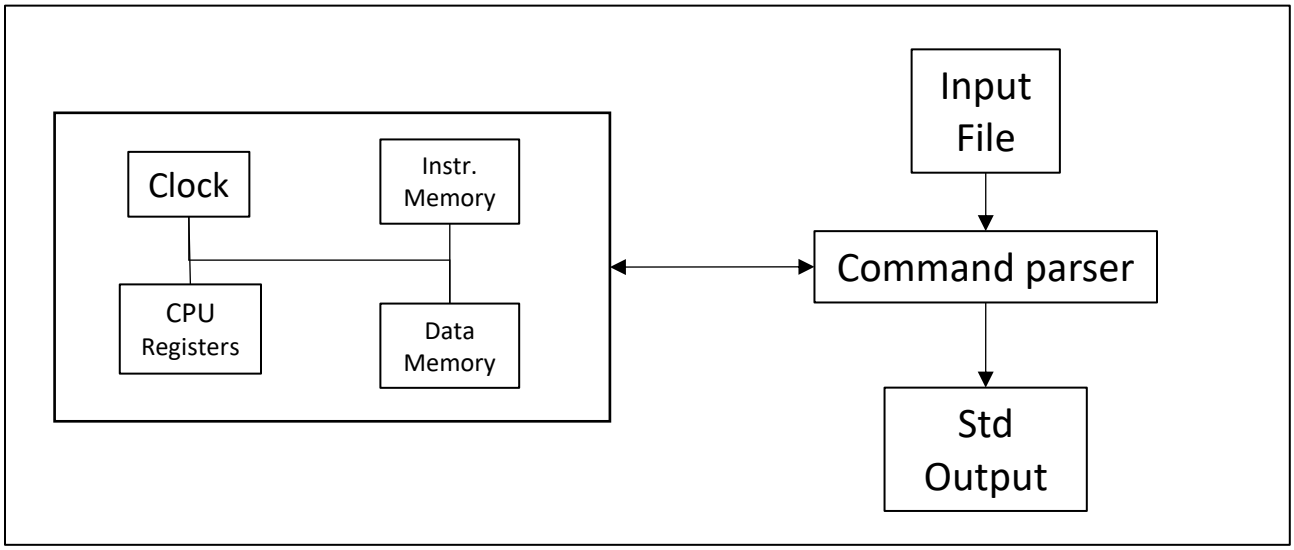
Program Execution

cs3421_emul <data_file>

Input Details

The cs3421_emul program reads an arbitrary number of lines, one line at a time from the data file. Each line will begin with a device identifier. The format of the rest of the line will be device dependent. For this assignment, possible devices are “clock”, “cpu”, “memory”, and “imemory”.

Devices



Clock

The clock device is unchanged from Assignment 2.

Memory

The memory device is unchanged from Assignment 2.

Instruction Memory

The instruction memory device is unchanged from Assignment 2.

CPU

The CPU device is the same as in assignment 2, except the student will add six new instructions, as described below. Note that different instructions require a different number of clock cycles to execute. On each clock cycle, the CPU will evaluate its inputs & state. If not currently executing an instruction, a clock tick will cause the CPU to fetch an instruction from instruction memory, as indicated by the PC. That instruction will be decoded, and begin its first cycle of operation. If an instruction requires multiple clock ticks to complete, the CPU will enter a “wait state”, and will remain in the wait state depending on the number of tick cycles needed to complete. Once the instruction is complete, the PC will be

incremented, and the CPU will return to the “fetch state”, where it will fetch the next instruction on the next clock tick.

CPU commands:

The CPU commands remain unchanged from Assignment 2. Setting the PC will cause the CPU to cancel any instruction currently in process, and return to the “fetch state”.

CPU Instructions:

Entropy's Instruction Set (Credit – James Walker)

Entropy Basics:

- 8-bit words
- 8 data registers (each register is 8 bits)
- 1 Program Counter register (8 bits)
- 8 bits for addressing its data memory
- 8 bits for addressing its instruction memory
- 8 instructions

Entropy stores its instruction and data memory separately, and they are addressed separately. Entropy's registers are RA through RH, and PC for the Program Counter. Entropy supports only one data type: 8-bit two's complement integers.

Entropy's instructions are 20 bits each, and are divided up as follows:

```
NNN DDD SSS TTT IIIIIIII
```

Where:

- NNN is the three bit instruction encoding
- DDD is the three bit destination register selector
- SSS is the three bit source register selector
- TTT is the three bit target register selector
- IIIIIIII specifies an immediate value

For Assignment 3, the student will implement the following eight instructions (six of which are new from Assignment 2). A “-” indicates an unused/ignored bit.

add

Adds the source & target register words, storing the result in the destination register. All values should be treated as 8 bit two’s compliment numbers.
Operation: $d = $s + $t; PC++
Assembly syntax: add $d $s $t
Binary encoding: 000 ddd sss ttt --------
Timing: 1 cycle

addi

Adds the source register & immediate value words, storing the result in the destination register. All values should be treated as 8 bit two’s compliment numbers.
Operation: $d = $s + imm; PC++
Assembly syntax: addi $d $s imm
Binary encoding: 001 ddd sss --- iiiiiiii
Timing: 1 cycle

mul

Takes the upper 4 bits and lower 4 bits of the source register, multiplies those values together, and stores the result in the destination register. Interprets its inputs as UNSIGNED. I.e., 1001 is interpreted as decimal 9, not decimal -7. The fact that the multiplier is NOT a two's complement multiplier can yield mathematically incorrect results. For example, 1111 * 1111, or 15x15, would yield a result of 11100001, which the CPU will interpret as -31. This is fine; it is the intended behavior.
Operation: $d = $s[0:3] * $s[4:7]; PC++
Assembly syntax: mul $d $s
Binary encoding: 010 ddd sss --- --------
Timing: 2 cycles

inv

Inverts all the bits in the source register word, storing the result in the destination register
Operation: $d = !($s); PC++
Assembly syntax: inv $d $s
Binary encoding: 011 ddd sss --- --------
Timing: 1 cycle

beq

If the words in the source & target registers are equal, assign the PC to the immediate-specified imemory address, otherwise increment the PC
Operation: if $s == $t PC=imm else PC++
Assembly syntax: beq $s $t label
Binary encoding: 100 --- sss ttt iiiiiiii
Timing: 1 cycle if registers not equal, 2 cycles if branch taken

lw

Loads a word into the destination register, from data memory at the address specified in the target register
Operation: $d=DMEM[$t]; PC++
Assembly syntax: lw $d $t
Binary encoding: 101 ddd --- ttt --------
Timing: greater of 1 cycle or speed of data memory (5 ticks for Assignment 3)

sw

Stores the source register word into data memory at the address specified in the target register
Operation: DMEM[$t]=$s; PC++
Assembly syntax: sw $s $t
Binary encoding: 110 --- sss ttt --------
Timing: greater of 1 cycle or speed of data memory (5 ticks for Assignment 3)

halt

Halts execution of the processor after incrementing PC. After halting, the CPU will ignore all future clock ticks, but will cooperate in supporting all parser commands such as “cpu dump”.
Operation: halt; PC++
Timing: 1 cycle
Binary encoding: 111 --- --- --- --------
Timing: 1 cycle

Data Representation:
For the “add”, and “addi” instructions, all values are treated as 8 bit two’s complement numbers. For instance if RA had the value 1, and RB had the value 0xFF (-1 in 8 bit two’s complement), the instruction “add $c $a $b” would result in RC getting the value 0. As specified for the “mul” instruction, the four bit values are treated as unsigned. The “inv” instruction simply inverts all bits.

Instruction Timing:
Instructions timing is given in number of clock cycles needed to complete executing the instruction. Instructions that rely upon external devices such as memory (e.g. lw &sw) will have a lower limit on the number of cycles needed by the CPU itself, and may potentially be limited by the number of cycles need by the external device. For instance, a lw instruction takes 1 cycle of CPU time to finish. If connected to memory that can also complete the request in a single cycle, the time for a lw from that memory would by 1 cycle (since CPU & memory run in parallel). If however that memory requires 5 cycles to complete, a lw from that memory would take 5 cycles.

Assembler

An assembler that converts Entropy assembly into binary machine code is available. The assembler is written as a Java .class file. To run it, pass in the name of the assembly file as a command-line argument. The assembler prints the resulting machine code to stdout (so it can be redirected to a file).

Example of running the assembler:

```
java EntropyCompiler assembly_file.s > memfile
```

The assembler is quite minimalistic. It can detect and report some errors, but some errors might cause it to fail with little or no explanation. Carefully check your syntax to make sure it is correct.

Other details about Entropy assembly:

- Registers are specified by a dollar sign $ followed by the letter a through h in lowercase ($a $b $c $d $e $f $g and $h).
- Text which appears on a line after a semicolon (“;”) is considered a comment and is ignored by the assembler.
- Following is a toy example Entropy assembly program which demonstrates the functionality of all the different instructions. The code is commented to explain what it is doing at each line.

Sample

```
; execution of this program proceeded by following emulator commands
; cpu reset – all registers including PC set to zero
; memory create 0x100
; memory reset – all memory set to zero
;
; before execution, relevant memory is
; Addr   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
; 0x0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
;
; after execution, relevant memory is
; Addr   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
; 0x0000 00 00 00 00 00 00 00 00 00 00 00 00 00 F3 00 00 00
;
; after execution, registers are
; PC: 0x0A
; RA: 0x06
; RB: 0x06
; RC: 0x0C
; RD: 0xF3
; RE: 0xF3
; RF: 0x00
; RG: 0x00
; RH: 0x00
;
; Entropy sample program (RA-RH start with zero)
addi $a $a 35      ; stores 0x23 in RA
mul  $a $a         ; 0x2 x 0x3 = 0x6, store in RA
.loop:
addi $b $b 1       ; increment RB by 1
beq  $a $b endloop ; when RA==RB (both 0x6) break out of loop
beq  $h $h loop    ; effective jump/goto command
.endloop:
add  $c $a $b      ; 0x6+0x6=0xC, store 0xC in RC
inv  $d $c         ; invert RC to 0xF3 and store that in RD
sw   $d $c         ; store RD at address specified by RC (0xC)
lw   $e $c         ; load from address in RC (DMEM[0xC]=0xF3) into RE
halt               ; stop execution
;
; The above code produces the following 10 instructions
; addr instr
; 0000 20023
; 0001 40000
; 0002 24801
; 0003 80105
; 0004 83F02
; 0005 08100
; 0006 6D000
; 0007 C1A00
; 0008 B0200
; 0009 E0000
```

Important: If your CPU passes the above test, that is an excellent start. However, it does not guarantee that your CPU is 100% correct. Be sure to write your own tests, hitting all of your registers, every

address in DMEM, and various combinations of commands and loop structures. Try hard to break your CPU. **Do not rely exclusively on the example program to test your processor.**

Language

The developer may use either Java or C/C++.

Submission Details

Submission will be via Canvas. Create a directory matching your username, and place all files you plan to submit with that directory. To submit your assignment, create a zip file of that directory (<your_username>.zip), and submit the zip file via Canvas. You should verify that the zip file has the correct directory structure, meaning when extracted without any special options, it WILL create a directory corresponding to the username of the student.

The zip file for Assignment 1 (<your_username>.zip) should contain the following files:

<your_username> - directory corresponding to your username which holds all files
- Author.txt – a text file with a single line containing your first & last name
- Language.txt – a single line containing the word "C" or "JAVA" (all upper case, no quotes) indicating the language used for the assignment
- Readme.txt – Any comments you'd like to share regarding the submission
- Makefile – an optional file that will compile your code into the cs3421_emul executable
- cs3421_emul source & header files necessary for your project

Note, if you have known issues with your program, DOCUMENT THAT in the Readme.txt file. Any use of non-standard libraries or packages should also be documented, as well as how to retrieve & install them.

Grading

Programs will be primarily graded on the correctness of the output, **and ability to follow submission guidelines**. Given the large number of submissions, the evaluation of the output will be automated. This means it is **CRITICAL** to follow the sample output above. Sample data & output will also be provided to test your programs. The assignment grade may also be based on style, comments, etc. Programs that crash, fail to produce the correct output, or don't compile/run may lose up to 100% of the points, depending upon severity of the error. Some effort may be made during grading to correct errors, but students should not depend upon this. For grading, programs will be compiled and run on Ubuntu Linux 18.04 LTS.