Assignment 2

Due: Oct 6th, 11:59 pm
Weight: 10% of total class grade

Overview

In this assignment, students will add a new memory device – instruction memory, add wait states to existing (data) memory, and implement load & store instructions.

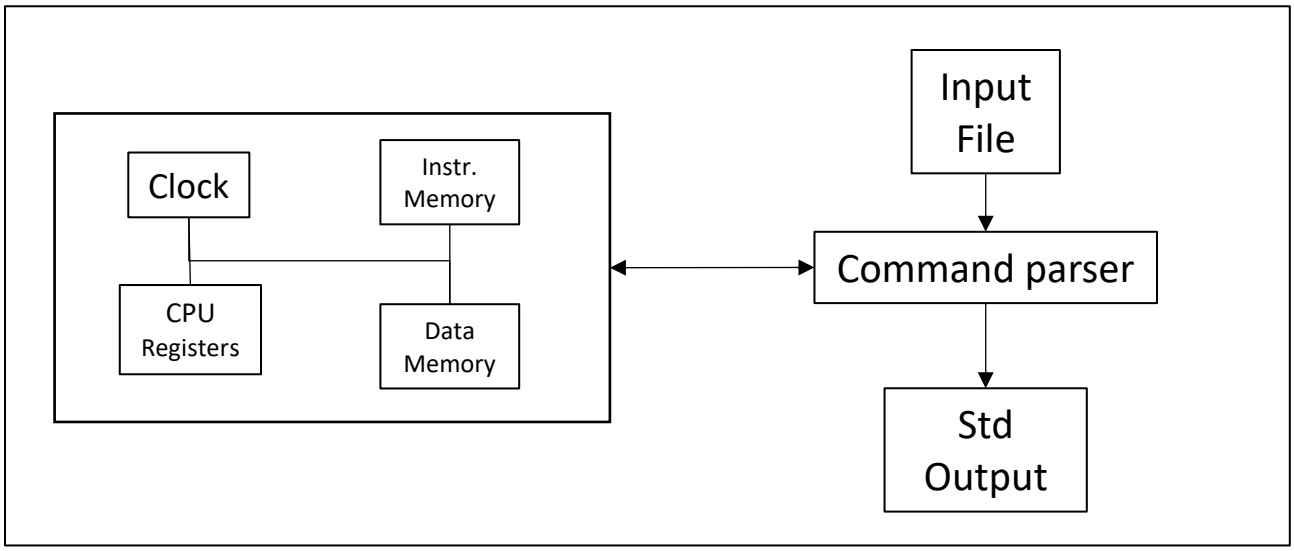
Program Execution

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”. For example:

Devices



Clock

The clock device is unchanged from Assignment 1, other than potentially sending ticks to the memory device (depending on implementation).

Memory

The memory device commands are unchanged from Assignment 1, though will now be referred to as “data memory”. For Assignment 2, every access to data memory by the CPU will require 5 clock ticks to complete. For example, if a load instruction begins execution on clock cycle 0, that load will complete after the CPU receives the 5th “tick command”, and at the end of the cycle, the clock would have a value of 5. The same 5 cycles are needed to complete store instructions.

Instruction Memory

The instruction memory device (imemory) provides instruction storage for the CPU. Unlike data memory which has a word size of 8 bits/1 byte, instruction memory has a word size of 20 bits. This 20 bits matches the size of a CPU instruction. Each fetch would retrieve a word (20 bits), and is it indexed in word-size increments. For instance “instr_memory[0]” indexes the first CPU instruction, and identifies a 20 bit quantity. The next instruction would be at “instr_memory[1]”, and again identifies a 20 bit

quantity. Conceptually, it is an array of 20 bit words. The actual implementation need not match that model, and the student should consider an array of 32 bit words, and simply ignore the extra 12 bits.

Prior to use, it must be created with the “create” command, which specifies the size (in hex) of the number of 20 bit words to create. At that point, memory is in an undefined condition. The “reset” command is used to initialize all of memory to zeros. The contents of memory can be displayed via the “dump” command, and assigned with the “set” command.

Instruction Memory commands:

create <size in hex bytes>

> The “create” command accepts a single integer parameter which indicates the size of the memory in 20 bit words. Example: “imemory create 0x10000”.

reset

> The “reset” command causes all of the allocated memory to be set to zero. Example: “imemory reset”.

dump <hex address> <hex count>

> The dump command shows the contents of memory starting at address <address>, and continuing for <hex count> words of 20 bits. The output should match the format shown below, with a header showing the offset (0-7), and each subsequent line showing the address, and up to 8 20 bit words. As in assignment 1, memory not within the requested range should be shown as blanks. Example: “imemory dump 0x1A01 0x14”

```
Addr        0     1     2     3     4     5     6     7
0x1A00        ABCDE FFFFF ABEEF FEED1 00000 11111 22222
0x1A08 DEADB EEFFE EDABC 00000 98765 43210 01234 56789
0x1A10 ABCDE F0123 44444 67341 AB3C9
```

set <hex address> <hex count> <hex word 1> <hex word 2> … <hex word N>

> The set commands initializes instruction memory to a user supplied set of values. The “hex address” specifies the word address to begin setting memory values, “hex count” is how many 20 bit words will be assigned, and is followed by “hex count” number of 20 bit hex words, separated by a single space. For this assignment, the command will never be used with more than 1,000 words. Example: “imemory set 0x20 0x05 0xABCDE 0x01234 0x56789 0xFFFFF 0xABEEF”.

set <hex address> file <datafile>

> This set command variant is the same as above, except the data is contained in a file. The file must contain a set of hex word values separated by a single space. The count is determined by the number of entries. Example: “imemory set 0x20 file /my/data/file”.

CPU

The CPU device has 8 one byte data registers labeled RA through RH. The CPU also has a 1 byte program counter (PC) that holds the address in instruction memory of the next CPU instruction to fetch. On each clock cycle, the CPU will advance its 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 enter the "fetch state", where it will fetch the next instruction on the next clock tick.

Note – the "register shifting" behavior from Assignment 1 no longer occurs.

CPU commands:

The CPU commands remain unchanged from Assignment 1. 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 (2 implemented for Assignment 2)

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 IIIIIII
```

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
- IIIIIII specifies an immediate value

For Assignment 2, the student will implement the following two instructions (a "-" indicates an unused/ignored bit):

lw

Loads a word into register $d, from data memory at the address specified in register $t
Operation: $d=DMEM[$t]; PC++
Assembly syntax: lw $d $t
Binary encoding: 101 ddd --- ttt --------
Timing: Driven by speed of memory (5 ticks for Assignment 2)

sw

Stores the value in register $s into data memory at the address specified in register $t
Operation: DMEM[$t]=$s; PC++
Assembly syntax: sw $s $t
Binary encoding: 110 --- sss ttt --------
Timing: Driven by speed of memory (5 ticks for Assignment 2)

Assembler

An assembler that compiles 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 compiler.

Sample

```
; execution of this program proceeded by following emulator commands
; cpu reset
; memory create 0x100
; memory set 0x0 0x8 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8
;
; before execution, memory is
; Addr   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
; 0x0000 01 02 03 04 05 06 07 08 00 00 00 00 00 00 00 00
;
; after execution, memory is
; Addr   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
; 0x0000 01 08 07 06 05 04 03 02 01 00 00 00 00 00 00 00
;
lw $a $h ; h is 0, load a with what's at data offset 0 (which is 1)
lw $b $a ; a now 1, load b with what's at data offset 1 (which is 2)
lw $c $b ; b now 2, load c with what's at data offset 2 (which is 3)
lw $d $c ; c now 3, load d with what's at data offset 3 (which is 4)
lw $e $d ; d now 4, load e with what's at data offset 4 (which is 5)
lw $f $e ; e now 5, load f with what's at data offset 5 (which is 6)
lw $g $f ; f now 6, load g with what's at data offset 6 (which is 7)
lw $h $g ; g now 7, load h with what's at data offset 7 (which is 8)
sw $a $h ; store contents of a (1) where h points (8) mem[8]=1
sw $b $g ; store contents of b (2) where g points (7) mem[7]=2
sw $c $f ; store contents of c (3) where f points (6) mem[6]=3
sw $d $e ; store contents of d (4) where e points (5) mem[5]=4
sw $e $d ; store contents of e (5) where d points (4) mem[4]=5
sw $f $c ; store contents of f (6) where d points (3) mem[3]=6
sw $g $b ; store contents of g (7) where b points (2) mem[2]=7
sw $h $a ; store contents of h (8) where a points (1) mem[1]=8
;
```

```
; The above code produces the following 16 instructions
; addr instr
; 0000 A0700
; 0001 A4000
; 0002 A8100
; 0003 AC200
; 0004 B0300
; 0005 B4400
; 0006 B8500
; 0007 BC600
; 0008 C0700
; 0009 C0E00
; 000A C1500
; 000B C1C00
; 000C C2300
; 000D C2A00
; 000E C3100
; 000F C3800
```

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.