Assignment 1

Due: See Canvas

Overview

In this assignment, students will code a model of a system clock, memory, and the fetch instruction cycle of a CPU. These models will be controlled by a command parser reading a data file containing commands. The parser will read each line, parse the command, and apply it to the appropriate device. The device will execute the command by emulating the appropriate behavior. Upon reaching end of file, the program will exit. To improve readability, all "dump" commands should output an extra blank line after showing the appropriate output.

Program Execution

cs3421_emul <data_file>

Input Details

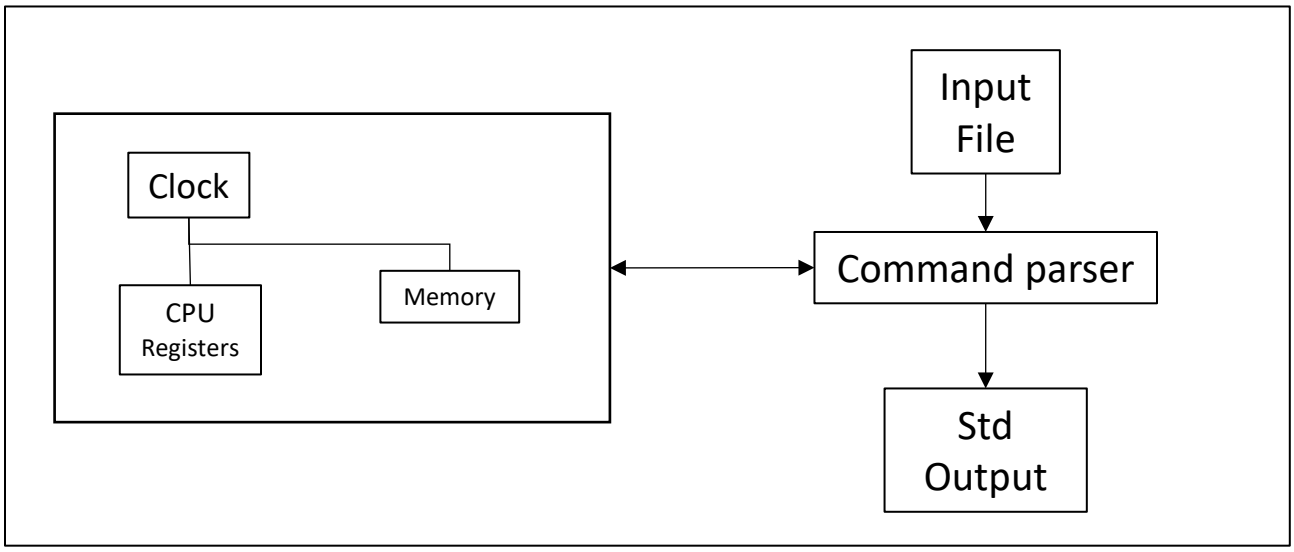
The cs3421_emul program reads an arbitrary number of lines, one line at a time from the data file. Each line will begin with a device identifier. The format of the rest of the line will be device dependent. For this assignment, possible devices are "clock", "cpu", and "memory". For example:

```
clock reset
memory create 0x100
memory reset
memory set 0x0 0x8 0x08 0x07 0x06 0x05 0x04 0x03 0x02 0x01
cpu reset
clock tick 8
cpu dump
```

Output Details

The cs3421_emul should write its output to standard output (stdout).

Devices



Clock

The clock device is the heart of the system, providing synchronization between all other devices. On command, it provides a “tick pulse” to all other devices that need a clock. For diagnostic purposes, the clock will maintain a monotonically increasing unsigned 16 bit count that can be displayed.

Clock commands:

reset

The reset command sets the internal counter to zero. Example: “clock reset”.

tick <decimal integer>

The tick command accepts a positive decimal integer indicating how many clock ticks should be issued to attached devices. The internal counter is incremented by the specified number of ticks.

dump

The dump command shows “Clock: “ followed by the value of the internal clock in decimal. Example: “clock dump” might show:

Clock: 148

Memory

The memory device provides storage for the system. Prior to use, it must be created with the “create” command, which specifies the size in bytes. 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.

Memory commands:

create <size in hex bytes>

The “create” command accepts a single integer parameter which indicates the size of the memory in bytes. Example: “memory create 0x10000”.

reset

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

dump <hex address> <hex count>

The dump command shows the contents of memory starting at address <address>, and continuing for <count> bytes. The output should begin with a header showing “Addr”, 1 space, and then 00-0F with a single space between each. Subsequent lines are formatted such that each line begins with a 1 byte address (0x and 2 digit upper case hex characters) that is an even multiple of 16, followed by a single space. Following that, up to 16 bytes of data, in 2 digit upper case hex is printed. If the address is not a multiple of 16, blank spaces should be printed until the address is reached. Example: “memory dump 0x04 0x20”

```
Addr 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x00             11 44 23 93 50 22 AE DE AD BE EF FF
0x10 AE DE AD C0 DE FA CE FE ED CA FE BE EF 30 A8 EE
0x20 55 AA 10 20
```

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

The set commands initializes memory to a user supplied set of values. The “hex address” specifies where to begin setting memory values, “hex count” is how many bytes will be assigned, and is followed by “hex count” number of hex bytes, separated by a single space. For this assignment, the command will never be used with more than 100 hex bytes. Example: “memory set 0x10 0x05 0x08 0xDE 0xAD 0xBE 0xEF”

CPU

This simple CPU device has 8 one byte data registers labeled RA through RH. The CPU also has a 1 byte program counter (PC) register that holds the address in memory of the next CPU instruction to fetch. For this assignment, CPU “instructions” will simply be 1 byte of data. On each clock cycle, the CPU will shift the contents of each register to the next higher register, such that RH receives the content of RG, RG receives what is in RF, and so on, until finally, RB receives RA. The CPU will then fetch the byte from the Memory Device pointed to by the PC, place it into RA, and increment the PC by 1. While the CPU doesn’t do anything useful, this behavior is a means to test the I/O & memory fetch capabilities of the code.

CPU commands:

reset

The “reset” command causes all CPU registers (including PC) to be zero. Example: “cpu reset”.

set reg <reg> <hex byte>

The “set reg” command sets the value of the specified CPU register. The “reg” parameter can be the value RA-RH, or PC for the program counter. Example: “cpu set reg RE 0xAA”.

dump

The “dump” command shows the value of all of the CPU registers. A sample output would be as follows:

PC: 0xAA
RA: 0x23
RB: 0x14
RC: 0xFF
RD: 0x44
RE: 0xAA
RF: 0x00
RG: 0x09
RH: 0x18

Implementation & APIs

The implementation of this system is expected to follow good software coding and development practices. Each device must have its own source and header files. Clear application programming interfaces (APIs) should exist for all public interactions with the devices. All private and internal implementation details must never be used by external components. In C, private functions & data members should utilize the “static” declaration modifier to restrict their scope to that source file. In C++, classes should use the private & public declarations modifiers to control access.

Memory Application Programming Interface (API)

In future assignments, the memory device will take more than a single cycle to finish. The following API separates the act of requesting the data from delivering it. For Assignment 1, this function will not only start the fetch, but will also complete it (setting the Boolean pointed to by fetchDonePtr to true). This will not be the case in future assignments. The student must implement the following API:

```
// This API is called by memory clients to initiate a fetch (read) or store (write) from/to memory
// address – the offset in memory where the read/write should begin
// count – the number of bytes that should be read/written
// dataPtr – a pointer where data should be placed (read), or source of data (write)
// memDonePtr – a pointer to a boolean that the Memory Device will set to true when
//                  the data transfer has completed (possibly multiple cycles after request)
//


memStartFetch(unsigned int address, unsigned int count,
                  unsigned char *dataPtr, bool *memDonePtr);
```

For assignment 1, the "count" will always be 1. The CPU would use the above API as follows:

```
struct CPU {
   unsigned char regs[8];  // CPU registers
   unsigned char PC;       // program counter
   …
};

unsigned char fetchByte;
bool          fetchDone;
memStartFetch(cpu.PC, 1, &fetchByte, &fetchDone);
```

Clock Application Programming Interface (API)

As described above, the clock is the "master device" that drives the operation of the system. Each such driven device will implement a "clock client" API to receive clock ticks called by the clock. For Assignment 1, that API is as follows:

doCycleWork();
: Called by clock to give the device a chance to do some (or all) of the work during that clock cycle.

In C, the name of the device should be prepended to the function name to avoid future name conflict (e.g. cpuDoCycleWork). In C++, this could be a public member function of a CPU class. For Assignment 1, only a single device (the CPU) must be a client of the clock, and be called by it to do work.

Language

The developer must use C or C++.

Submission Details

Submission will be via Canvas. Create a directory matching your username, and place all files you plan to submit within 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 the assignment (named <your_unix_username>.zip) should contain the following files:

<your_unix_username> - directory corresponding to your username which holds all files
: Author.txt – a text file with a single line containing your first & last name
: 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.

Hints

Floating Point Numbers

If you are tempted to use any floating point numbers for any assignments, rethink your approach. You should never need to use functions such a "floor". The same is true for dividing & remainder. Instead try and implement the function using binary math, masking, shifts, etc. See the slides for details.

Byte

In C/C++, a byte is stored using the type "char" or when they might be numbers as "unsigned char". Your Memory Device should contain a pointer such as "unsigned char *memPtr". The CPU has byte registers which would also be declared as an "unsigned char". To make accessing those registers easier, declare them as an array such as "unsigned char CpuRegs[8]".

Strings

Strings in C must include enough space to hold a null terminator, and are typically declared as character arrays such as "char myString[11]", which has enough space to hold 10 characters.

scanf

If writing your code in C, a number of features of scanf can be used to make life easy. Don't try and read in the entire file at once, or even the entire line. Work on one symbol (called a "token") at a time, and let that token drive your next action. To read hexadecimal, use the "%x" format specifier. To read a string, use "%s", which will skip over all white space, and read in all characters up to the next white space (which included an end of line). The scanf function returns how many input arguments were processed, so if it returns <=0, that's a good sign you've reached the end of file. On a Linux system, type "man 3 scanf" to see the manual page.

printf

The printf function can be told to add leading zeros, and make things upper case. For instance printf("%02X", i)" will print the variable i in upper case hex, use leading zeros, and be at least two characters wide. Type "man 3 printf" to see the manual page.