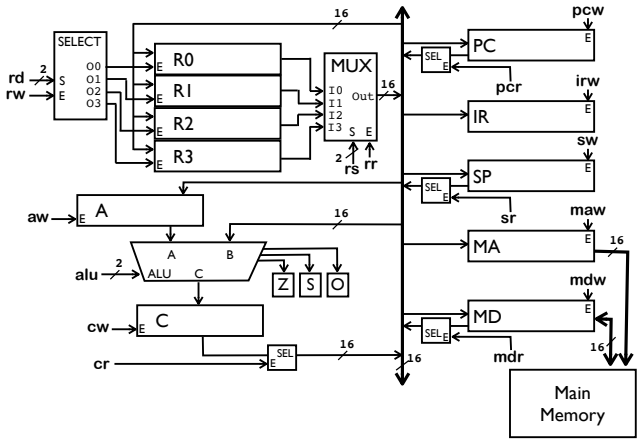
CS203 Project 1 -- Fall 2016
Due: November 8, 2016 @ 11pm
CPU Simulator

The assignment is to create an simulator for the simple CPU and associated main memory, illustrated below. The simulator should be capable of accepting a file Register Transfer Notation that will specify how the instructions of your ISA are implemented. The simulator should be created using Java and use the object-oriented features of that programming language as needed. The simulator should also implement a graphical interface and a set of unit-tests that will demonstrate the functionality associated with each implemented instructions. Unit-tests will require the ability to set and then query the machine state to verify correct functionality.



Configuration:

The program should accept a configuration file that will define the word-size of the computer, bus size, ALU functionality, and number of general registrars. The structure of this file is up to you, but should have an informational structure, so a user will know what line specifies which item. The identified items are examples of what should be included, but does not limit the items that are configurable in the processor design.

Memory Implementation:

In addition to the CPU simulator presented above, you will need to create main memory that is byte addressable. The number addresses should be bounded by the entire addressable space by a specific word-size (such as 16-bits). Consider how the memory worked from the simulator that was used in Labs 3 and 4, and consider how you might compose it. Another consideration is the word-size of your operations, it is convenient to think one memory location consisting of a certain number of byte and the offset is handled appropriately.

Instruction Format:

The description of the instruction format was not discussed in detail previously, but the best thing to do at this point is to go back and re-read the Stalling document that was shared on the Moodle website. For this simulation, you should keep the instruction format simple and use a fix-width instruction format, where each operation is stored in a single memory word. Thus, it is suggested that the simulation use something like the

PDP or ARM instruction format as described by Stalling. Note how you design your instruction format will contribute to your grade and is something to consider early in the design of your project.

For this design you are required to implement the y86 ISA, but you should also note that this is not a fix-width encoding and is instead a variable width encoding. You will need to consider how you will convert the variable-width approach with a fixed-width implementation.

RTN File and Parsing:

The machine should minimally implement the y86 ISA described at the start of Chapter 4 of the course textbook. The RTN should be stored in a file and formatted in a way that was described in class for micro-controlled memory, that is, at the start of the file is the fetch RTN followed by the instruction for each of the different assembly instructions. This is illustrated below and can be seen as the *control memory*, but you should not think of this as something that should be implemented in a binary representation; instead individual RTN instructions can be stored in an object representing each instruction. Additionally, the entire RTN file should be stored in a Java class that represents the control memory, storing an object of each specified set of RTN instructions. Then each operation object can itself store each individual RTN line in a separate object. The file can be parsed with a set of regular expressions using java.util.regex from the Java API.

```
FETCH:
  ...
rrmovq:
  ...
rmmovq:
  ...
mrmovq:
  ...
irmovq:
  ...
jmp:
  ...
```

Fetch-Exec Cycle:

A prime component of the simulator is the Fetch-Execute cycle that will work the individual parts of the control memory. But what does not need to be explicitly implemented is the clock-cycle; instead this can be generally abstracted as performing each line of the RTN from the control memory discussed above. Also to initiate the Fetch-Exec Cycle, you will need to think about memory organization of the simulator and how the assembled program is placed in memory. Use the compiled "**.yo**" files from the lab simulator for inspiration and also be aware of the difference between assembly you created and the machine code that was generated by the **`yas`** program.

Unit-Testing Individual Instructions:

In addition to the configuration file and the RTN file, there should also be a set of unit-tests to determine that each assembly instruction correctly works. To facilitate this unit-testing, you need to also make your simulator flexible enough to set it at a default state, execute the instruction out of memory, then be able to examine the system resources to verify the machine state has changed. This unit-testing can be integrated as either a single unit-test class or through a command-line mechanism.

These unit-tests should be in addition to any small programs you might construct for your simulator.

GUI Implementation:

To help in the debugging, be sure to implement a Graphical UI to allow you to visualize what the program is doing. It is suggested that all the core functionality be contained in a single simulation object that a graphical interface object can interact with. Containing the entire simulation to a single object will simplify the design, and then use methods for the GUI to query the state of the simulator.

Requirements and Advice:

- Do not use Boolean logic and a full-fledged clock, keep everything as just the fetch-execute cycle abstraction and think about what happens at each time step to the simulation state.
- Use objects for everything, but keep inheritance to a minimum, as there probably is not too much need for it. But inheritance is also a design choice and ultimately a decision of the developer.

Report:

Start your process with the creation of a report describing your design. Also components should be detailed, but also why certain design-choices were made should also be presented. The following points are important for discussion. Then after completion of the simulator implementation, complete the report and submit it with the rest of your materials.

- Project organization.
- General simulation design, including implementation of: CPU, Main Memory, RTN Control Memory, GUI and control interface.
- Design of the instruction format.
- Organization of the unit-test.
- Discussion of test programs.

Submission:

Once complete, submit all your materials to the Moodle. These materials should be organized in such a way that the simulation is easily understood and used by the person using your tool.

Appendix A – Register Transfer Notation

The primary reason for Register Transfer Notation (RTN) is to specify how the hardware works when a specific assembly instruction stored in memory as machine code is executed. RTN has a very simple format; the format is the source of data on the left is moved to the destination where the data should be stored. Thus the following is a meta-representation of this.

<destination> ← <source>

So how this might be used is in the following, where the Instruction Register is moved to the Memory Data register.

MD ← IR

Typically, each RTN operation is stored on a separate line, but also several transfer instructions can be performed in parallel. This is specified by a single line of several operations separated by a semicolon, shown below. Note as many instructions can be strung together on a line as needed, but the operations should not conflict in their manipulation of the bus.

<destination> ← <source> ; <destination> ← <source>; <destination> ← <source>; <etc>

An example of this is seen in the well specified fetch for that simple CPU. In this example the program counter uses the central bus, which is the stored in both the memory address register and incremented by the Arithmetic Logic Unit (and stored in register c.)

C ← PC + 2; MA ← PC

The source and destination are generally registers or memory locations, with the functionality constrained by the rules of the bus and other control mechanisms. But also, the source can be an ALU operations, demonstrated in the RTN just above to increment the program counter. For the registers and memory, they come in three flavors illustrated below: special register, general-purpose register, and memory location.

SP
R[R0]
M[MA]

These different identifiers can be either a source or destination, but must follow the rules identified for the simple CP. For example, the Memory Data (MD) register can be either a source or destination as in can either write or read to/from the bus. However, the Memory Address (MA) can only read from the bus and thus can only be a destination. One RTN that is unique is the representation for storing information to memory or reading it out of memory, given below. These must exist separately from manipulation of either the MA or MD registers.

MD → M[MA]
MD ← M[MA]

Finally, where does the RTN reside? It is stored in the control unit, as illustrated in the "processor game" slides, because it is what controls the computer and "implements" the functionality for individual operations. How is it loaded, it is stored in a file and loaded in as part of the initialization of the simulation.