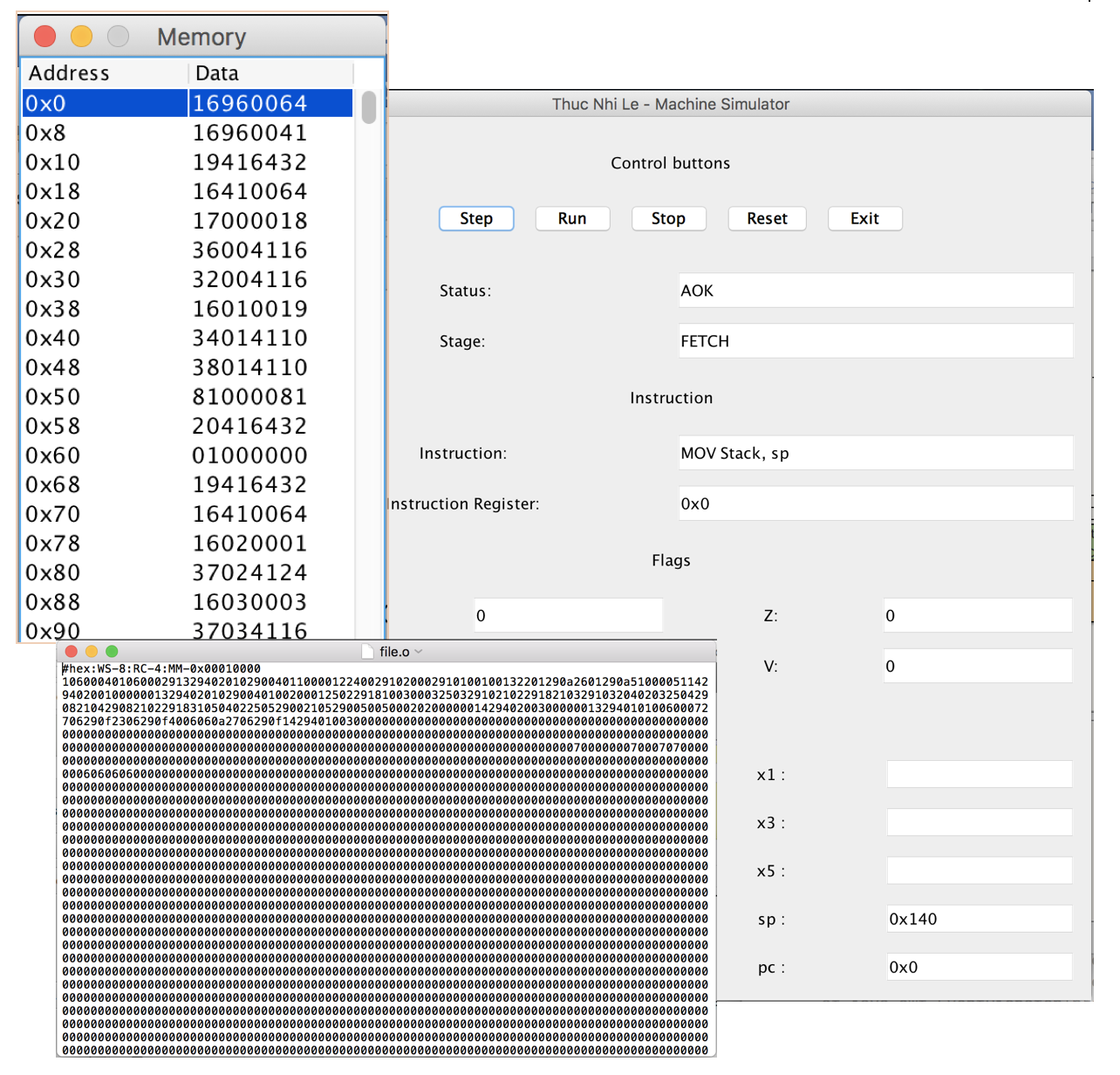
October, 22nd 2016

Thuc Nhi Le

Professor Pfaffmann

CS 203: Computer Organization

**PROJECT 1: ASSEMBLER AND MACHINE SIMULATOR**

****

**I. Introduction**

This project attempts to implement the “tool-change” of the ARM Assembler, CPU Simulator and Memory Image Visualizer. The first component is the simplified version of the ARM Assembler which takes in the ARM instructions from an .as file and create an image file of the memory. The second component is the CPU Simulator that uses the image file as the memory to decode and execute the instructions from the memory. The third component is the Memory Image Visualizer which provides users with the front-end part to interact with the program and the visual display of the process happening in the machine.

**II. Program Design**

1. **Big Picture**

The program should be composed of the components that are similar to the Assembler and the CPU Simulator, namely Memory, Instruction, Register, Flag. The class Assembler will perform as an assembler, class Operation will handle each instruction, update each register and flags and these two components will be controlled by the Machine class. Lastly, the Simulation class will take in the Machine outputs to simulate the GUI for front-end experience.

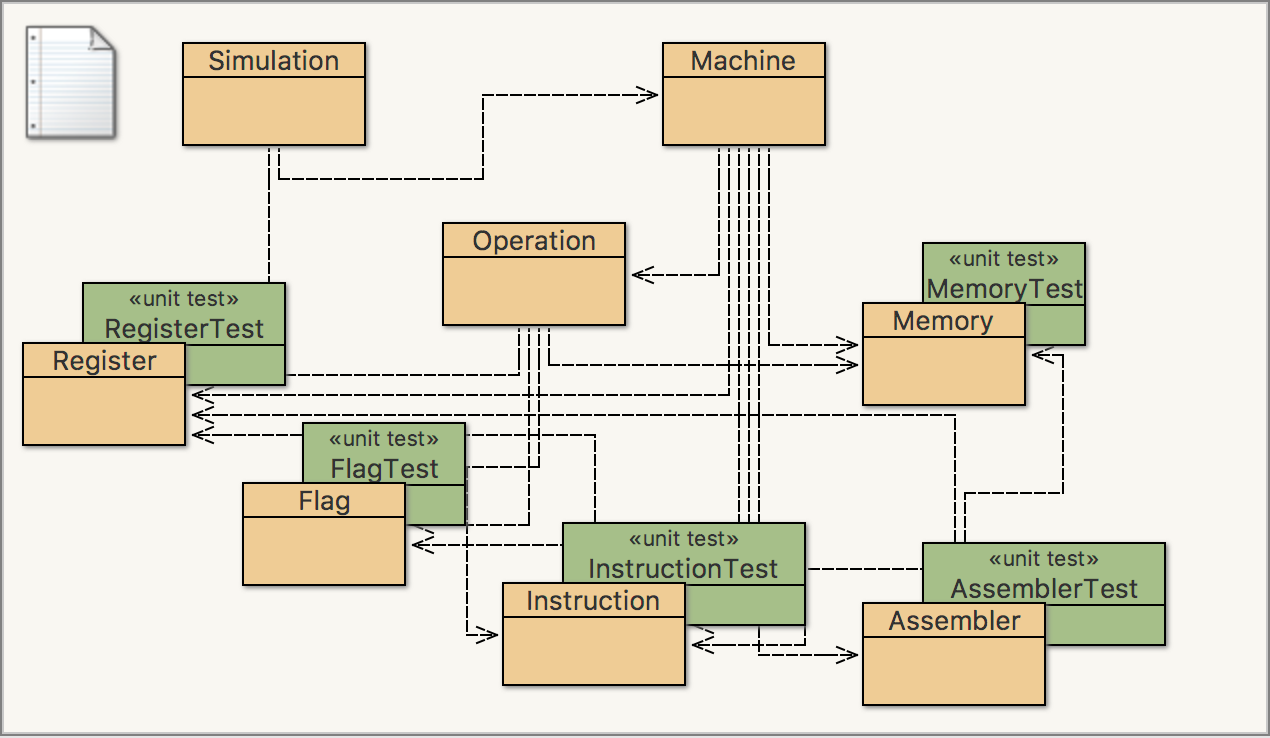


Figure The final classes

1. **Assembler Design**
2. **Memory Class**

The first component that we started to build is the Memory class. This class represents the Memory in the machine by storing the data in the byte type. The reason why we chose the byte type because it is easy to processed in Java. However, the memory is actually composed the byte objects, each of which can be retrieved by a wordsize-byte types array. To find position for an instruction or an element in the memory, we can retrieve the address by the simple calculation address = index \* wordsize. In this project, we chose wordsize 8 and maximum memory 0x00010000. The reason why we did not select larger wordsize and larger maximum memory value is that this project aims at providing a simple understanding of assembler and machine simulation so choosing the small number gives a more discrete picture.

In addition, the class also has the hex array to represent the memory in hexa-decimal for printing the image file and for the GUI part. For each instruction, we decode the hex as figure 2. Instruction Class will show how we separate element to decode and the Implementation and Testing part will provide the Table for decoding instruction.

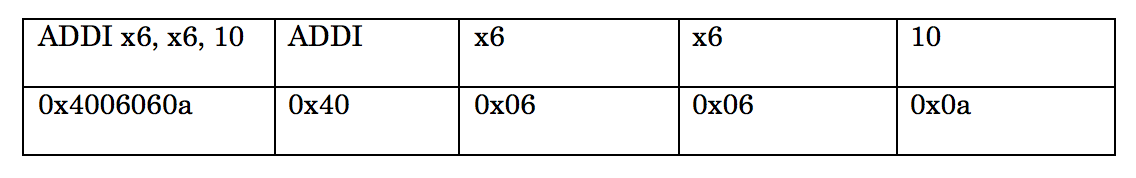


Figure Example of hex decoding ADDI instruction

1. **Instruction Class**

The second class that we built is Instruction class. This class represents an ARM Assembly Instruction. Since ARM Instruction usually has from 1-4 elements, figure 2 shows the example of how we separate the element to decode. For the positions that do not hold any value, we just decode to 0. For the first position, as we all know it is the opcode, we decode the opcode with the OpcodeMap, for other position, if we detect the register, we will create the register class (which will be discussed in details in Register class) and if we detect the immediate value, we just decode into both hexadecimal form for displaying and binary form for arithmetic and logical instruction. Table for decoding instruction will be provided in the Implementation part for further understanding.

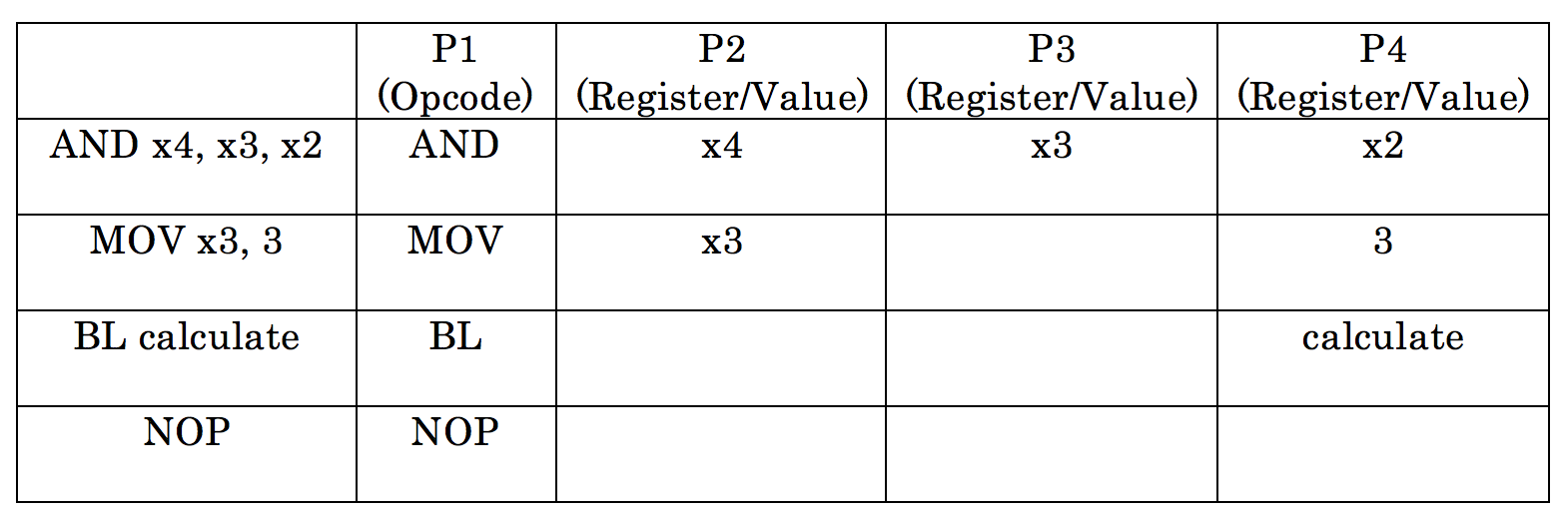


Figure 3 Example of separating Instruction

1. **Register Class**

Another important component of the machine is the Register class. This class represents the general and special registers in the computer. For this program, we implement four special register: Stack Pointer (sp), Frame Pointer (fp), Instruction Register (ir) and Program Counter (pc). In addition, we implement 7 general registers (x0 – x6). Since this is a simplified version of the Assembler and CPU Simulator, we tried to simplify many parts (i.e reduce the number of unused general register) as long as the users can still understand the big picture of how Assembler and CPU runs in the machine. For general registers, each has the id (0-6) and the value (in binary String for addition, subtraction, logical and arithmetic instructions) that the register holds. For special registers, the id for each register is its index in the register list (sp – 7, fp – 8, pc – 9, ir – 10), the value for each register is in String and the data which is an Integer value (address which it points to) for easier usages. For example, Program Counter has the integer value of the address it will point to so that the program can find to decode and execute the next instruction. The Table for decoding register will be provided in the Implementation and Testing part.

1. **Assembler Class**

One of the 3 biggest components for this project is the Assembler class. This class represents the Assembler and parses the file .as which is composed of ARM Assembly Instruction line by line to construct the memory and create the output of an .o file. It sets the wordsize, the regcnt, maximum memory value for the program by parsing the directives from the file. The first line of the image output file will display the form of number, the word size, regcnt and maximum memory to guide the users. For the memory display, at first, the image file is filled with maxmem number of 0s because the real memory is composed of 0s and 1s. However, we choose to display the memory in hexadecimal form instead of binary form for users to read the memory more easily. Figure 4 shows the top part of the memory image file.

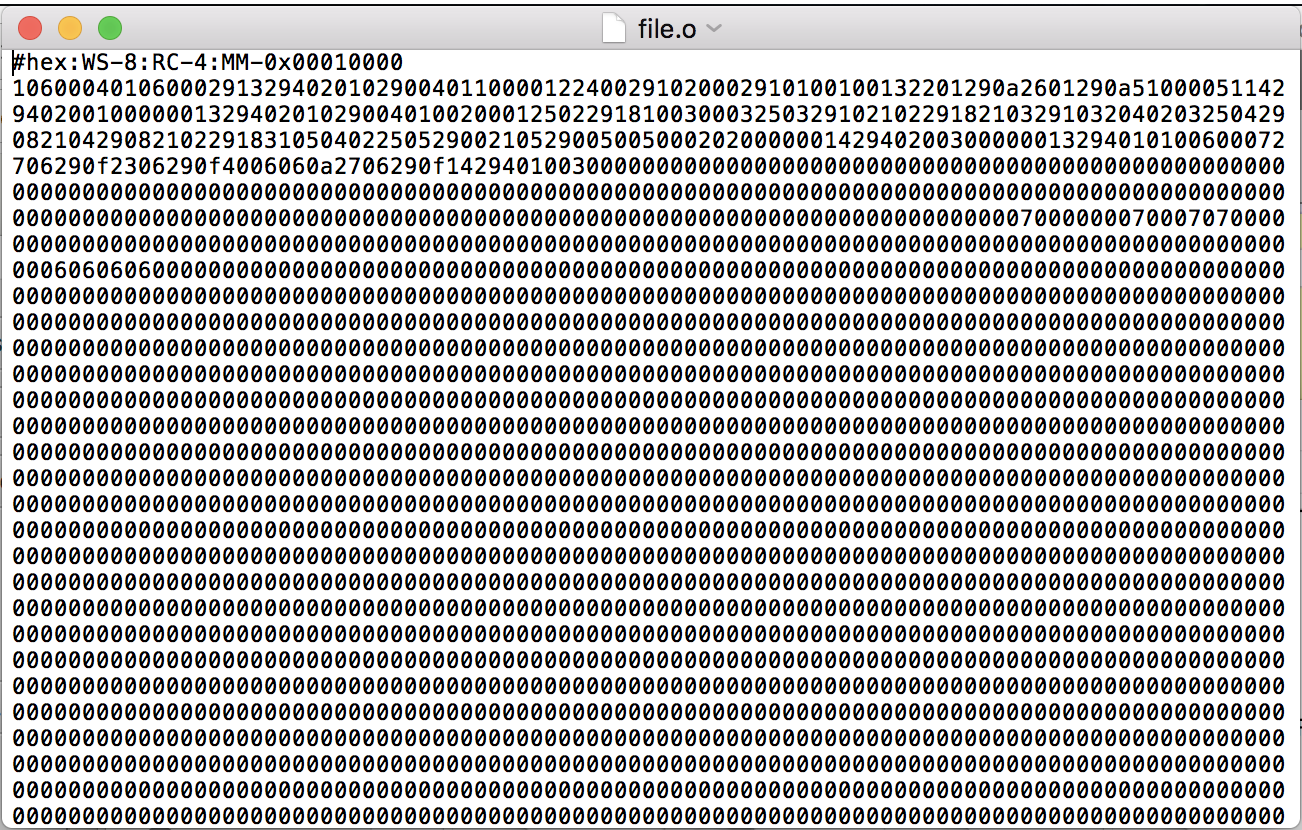
****

Figure 4 The image output from the Assembler

1. **CPU Simulator Design**
2. **Flag Class**

For the CPU, one significant component is the Flag. The Flag class represents the flag in the machine. We implemented 4 basic flags:

* Negative flag (N) – This flag is on if the register holds a negative value
* Zero flag (Z) – This flag is on if the register holds a zero value
* Carry on flag (C) – This flag is on if the register holds a carry-on value after the operation ADD/SUB
* Overflow flag (V) – This flag is on if the register holds an overflow value after the operation ADD/SUB

For example, in figure 5, ADDI x6, x6, 10, with x6 holds 0x11, will create the overflow and carry-on and makes those flags turn 1.

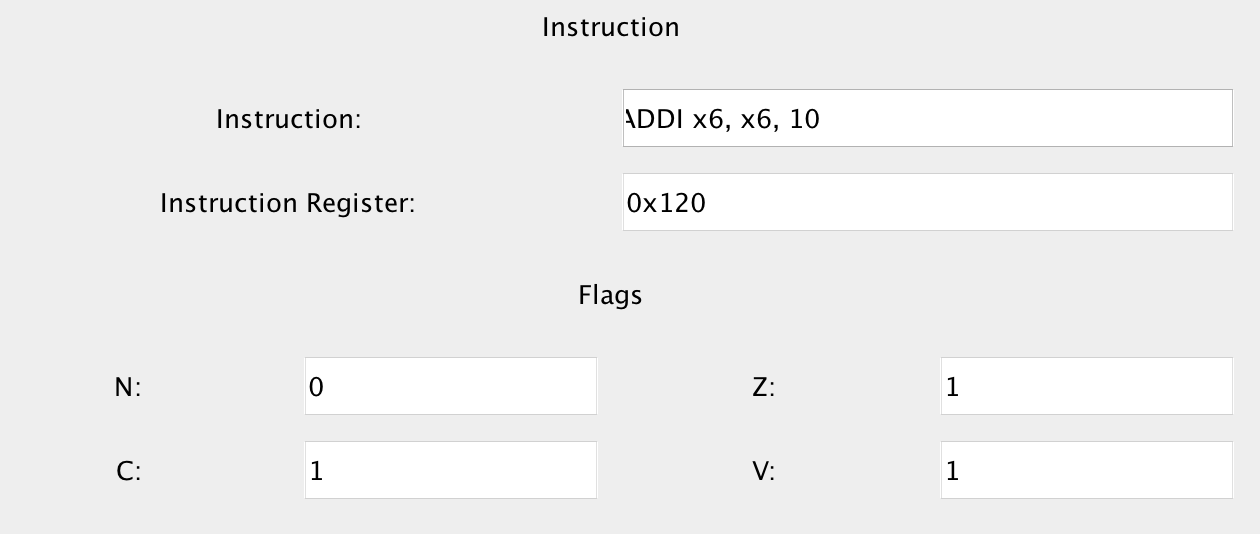


Figure 5 Flags update for ADDI x6, x6, 10

In addition, we also implement the status flag which shows the status of the instruction.

* AOK: The program can run
* HALT: The program comes to a HALT and stops running
* IL: The instruction is illegal (not true) and the program stops running

The status flag will keep update as the program runs. Figure 6 shows “AOK”, which means the program can run.



Figure 6 Status flag example

1. **Operation Class**

The Operation class plays an important role since it represents the part in the machine that executes the code. This class will implement the fetch-decode-execute-update cycle in the CPU. For this program, we decode and execute some instructions:

* Stack push/pop (STP/LDP): We changed the value of frame pointer and the stack pointer to illustrate how the stack works in a general way.
* MOV/FMOV: We just assign the value to the register for this instruction
* Data Transfer (LDUR/ LDURSW/ LDURB/ LDURH/ STUR/ STURW/ STURB/ STURB): We stored and loaded the value to the memory. However, while working on this part, we encountered the problem to transfer double because the size of double is bigger than word size which is the size of each register. We will discuss this difficulty more in the implementation and testing part.
* ADD, SUB, ADDI: We do the math on binary form of the value. We also had function to add binary (bit by bit), find 2’s compliment to facilitate the binary math.
* Arithmetic and Logical Instruction (AND): We do the bitwise operation on this instruction.
* Unconditional Branch(BL): We changed the program counter to the destination.
* Conditional Branch (CBZ): We did similar to unconditional branch, but in this case, we added the condition.
* No Operation (NOP): We just continue on the next instruction.
* Return (RET): We saved the last position before branching into a variable so whenever we call RET, we only needed to change the program counter to the last position.
* Halt (HALT): This instruction will stop the program and set the status flag to “HALT”.

In the operation class, we also updated the flags if negative, zero, carry-on or overflow value existed.

1. **Machine Class**

The Machine class is the second important part in the three components of our tool-chain. This class represents the CPU. After the memory is created and the instruction is parsed, this class will run the simulation and process the operations and instructions. Machine objects will have a Memory object, Assembler object, lists and maps of instructions and addresses. In addition, if the status flag is “AOK”, the machine will run the operations, otherwise it will stop.

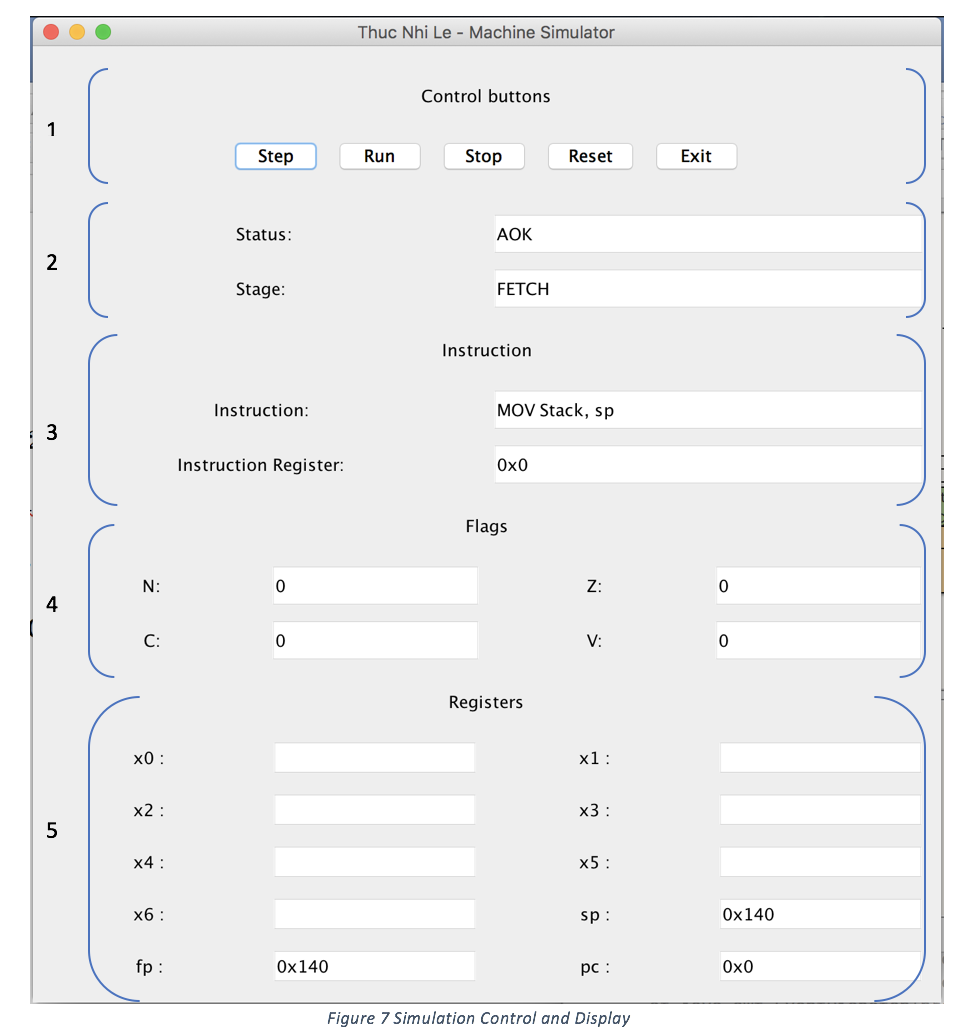
1. **Image Visualizer Design (Simulation Class)**

The Simulation class is the front-end part of the program, even though a lot of low-level coding is not displayed in the GUI part, we try to display the main elements for users to understand the process of the machine. The Image Visualizer is composed of two windows: Simulation Control and Display, and Memory Window. The GUI is built with Java Swing and the timer of the simulation is built with Java.util.Timer that allows the program to schedule the tasks.

1. **Simulation Control and Display**

The Simulation Control and Display allows users to control the Simulation by using the control buttons. In each step, the window will display the current instruction, the value of both special and general registers and the status of the flags as well as the status of the instruction. The figure 7 shows the front-end look of the simulator. This window is composed of five parts:

* Control buttons (1): Five buttons allow users to look at the process of simulation in various ways. For “Step” button, the window will display one in the four stages (Fetch/Decode/Execute/Update) of one instruction in the fetch-execute cycle. For the “Run” button, the simulation will run all the operations that are in the instructions input. The “Stop” button will pause the simulation. The “Reset” button will create the new machine to simulate. In this project, since we only had one input file, the “Reset” button will reset all flags and registers and starts from the position 0x0 in the memory. Lastly, the “Exit” button will exit the simulation program.



* Status and Stage (2): This part displays the status flag of the instruction whether the program can run or not. If the status is not “AOK”, the program will stop running. Besides, it shows the stage in the fetch-execute cycle that the simulation is currently in.
* Instruction (3): This part displays and updates the information about the current Instruction. It shows the Instruction and the Instruction Register which shows the address of the instruction code in the memory.
* Flags (4): This part displays and updates the current status of the four flags after the Update stage of each instruction.
* Registers (5): This part displays and updates the value of 7 general registers, stack pointer, frame pointer and program counter after each stage.

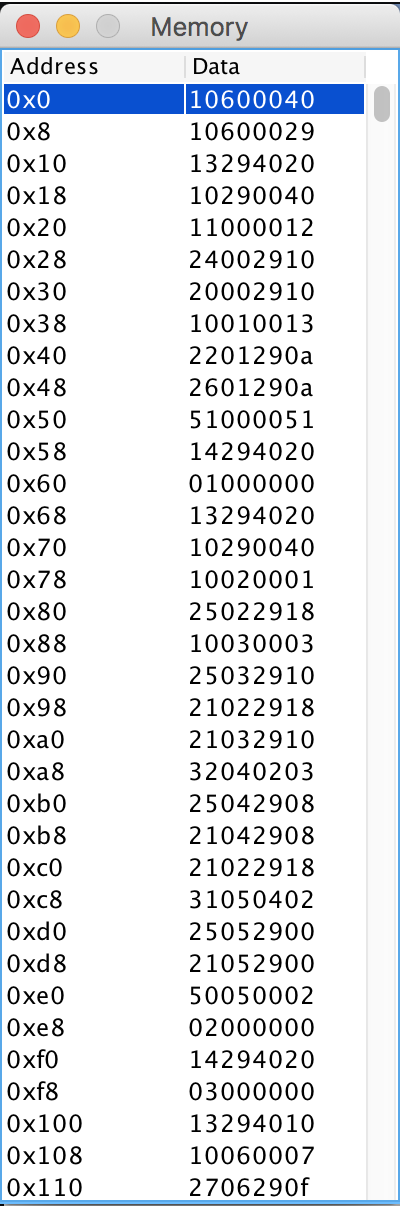
****

Figure 8 Memory Visualizer

1. **Memory Window**

The second part of the Image Visualizer is the Memory Window. This displays the address and the data in hexadecimal value. In order to build this, we created the two-dimensional array in the Memory class and passed into the Simulation class to visualize the memory. Besides, the instruction that is currently executed will be highlighted so that users can keep track of the simulation.

**III. Implementation and Testing**

1. **Implementation**

Since we tried to implement the ARM Simulator, we create the set of instructions that is quite similar to the ARM Instruction. However, we chose word size 8 for more concise visualization.

To encode the instructions and simplify the implementation, we create our own Table for decoding Instruction. As being briefly discussed in the Instruction and Assembler class, we cut the instructions into 2-4 part and encoded into hexadecimal form separately, then merged into one code and stored in the Memory.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Register | Byte Type | Hex |
| General | X0 | 01 | 0x01 |
| X1 | 02 | 0x02 |
| X2 | 03 | 0x03 |
| X3 | 04 | 0x13 |
| X4 | 05 | 0x14 |
| X5 | 06 | 0x10 |
| X6 | 07 | 0x11 |
| Special | sp | 19 | 0x13 |
| fp | 20 | 0x14 |
| pc | 22 | 0x16 |
| ir | 23 | 0x17 |

*Figure 9 Table for decoding Register*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Opcode | Register/Value | Register/Value | Register/Value |
| HALT | HALT | x | x | x |
| No Operation | NOP | x | x | x |
| Return | RET | x | x | x |
| Stack Push | STP | Register | Register | Value |
| Stack Pop | LDP | Register | Register | Value |
| MOV | MOV | Register | x | Register/Value |
| FMOV | FMOV | Register | x | Register/Value |
| D type | LDUR | Register | Register | Value |
| LDURSW | Register | Register | Value |
| LDURH | Register | Register | Value |
| LDURB | Register | Register | Value |
| STUR | Register | Register | Value |
| STURW | Register | Register | Value |
| STURH | Register | Register | Value |
| STURB | Register | Register | Value |
| R type | ADD | Register | Register/Value | Register/Value |
| SUB | Register | Register/Value | Register/Value |
| I type | ADDI | Register | Register/Value | Value |
| ALU | AND | Register | Register | Register |
| Conditional Branch | CBZ | Register | x | Value |
| Unconditional Branch | BL | x | x | Value |

*Figure 10 Table of Instruction Structure*

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Opcode | Byte Type | Hex |
| HALT | HALT | 01 | 0x01 |
| No Operation | NOP | 02 | 0x02 |
| Return | RET | 03 | 0x03 |
| Stack Push | STP | 19 | 0x13 |
| Stack Pop | LDP | 20 | 0x14 |
| MOV | MOV | 16 | 0x10 |
| FMOV | FMOV | 17 | 0x11 |
| D type | LDUR | 32 | 0x20 |
| LDURSW | 33 | 0x21 |
| LDURH | 34 | 0x22 |
| LDURB | 35 | 0x23 |
| STUR | 36 | 0x24 |
| STURW | 37 | 0x25 |
| STURH | 38 | 0x26 |
| STURB | 39 | 0x27 |
| R type | ADD | 48 | 0x30 |
| SUB | 49 | 0x31 |
| I type | ADDI | 64 | 0x40 |
| ALU | AND | 50 | 0x32 |
| Conditional Branch | CBZ | 80 | 0x50 |
| Unconditional Branch | BL | 0x51 | 0x51 |

*Figure 11 Table for Decoding Opcode*

1. **Testing**

For the Testing part, we use the unit tests to test the correctness of several functions for each class. To Test the whole program, we use the file.as that contains different types of instructions: MOV, D-type, R-type, I-type, ALU type, Stack instructions, Branch instructions, RET, NOP, HALT to make it similar to the real CPU. In addition, we also print out a double, a single, a half and a byte to see how padding works in the machine.

1. **Challenges**

While developing the program, we encounter some limitations of Java. The first problem is that Java addition and subtraction do not support the 2s’ complement form. As a result, we have to write our own bit-wise function to add and find 2s’ complement (to support subtraction) to add and subtract binary numbers. In addition, the biggest problem is to deal with double whose size is twice as much as the wordsize. As a result, in practice, a register of word size cannot hold the double. In the program, we decide to find the substitution of word size for the double and mark the register that holds that value as actually holding a double.

**IV. Conclusion**

In conclusion, building the tool-chain of the Assembler, CPU Simulator and Image Visualization is an interesting and valuable experience. Thanks to this project, we acquired and strengthened our understanding on computer organization, especially on how instructions, registers, assembler and the machine work together in low level, which we barely see when programming and working with high-level language.

**V. References**

1. Computer Science 203 Computer Organization, Professor Pfaffmann, Project 1 Description, http://139.147.9.182/~pfaffmaj/courses/f17/cs203/lectures/w4\_d1/#(1)

2. ARM, InfoCenter, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/

3. Patterson, David A. and Hennessy, John L., Computer Organization and Design Arm Adition. MKP, 2017. PDF

4. Oracle, Java Swing Tutorial, https://docs.oracle.com/javase/tutorial/uiswing/start/index.html