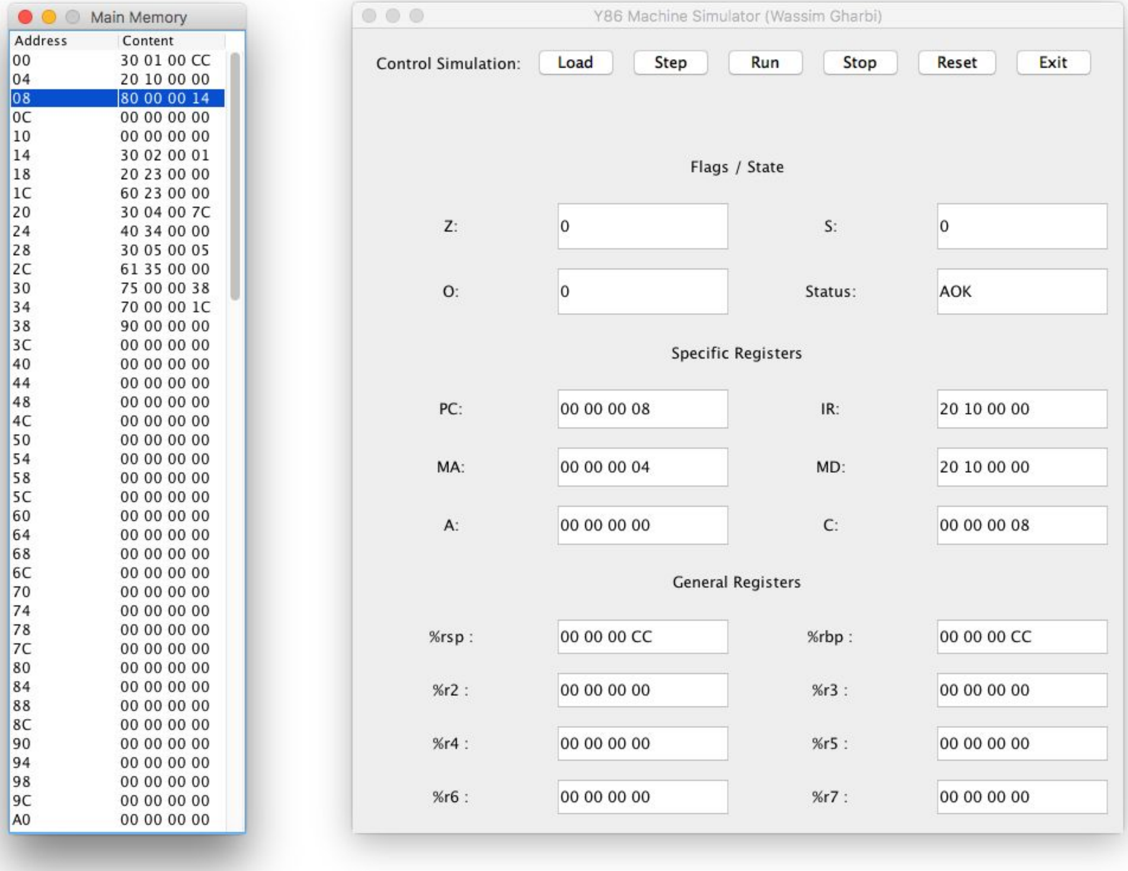
November, 1st 2016

Wassim Gharbi

Professor Pfaffamann

CS 203 : Computer Organization

Project: Y86 ISA Simulator



Address	Content
00	30 01 00 CC
04	20 10 00 00
08	80 00 00 14
0C	00 00 00 00
10	00 00 00 00
14	30 02 00 01
18	20 23 00 00
1C	60 23 00 00
20	30 04 00 7C
24	40 34 00 00
28	30 05 00 05
2C	61 35 00 00
30	75 00 00 38
34	70 00 00 1C
38	90 00 00 00
3C	00 00 00 00
40	00 00 00 00
44	00 00 00 00
48	00 00 00 00
4C	00 00 00 00
50	00 00 00 00
54	00 00 00 00
58	00 00 00 00
5C	00 00 00 00
60	00 00 00 00
64	00 00 00 00
68	00 00 00 00
6C	00 00 00 00
70	00 00 00 00
74	00 00 00 00
78	00 00 00 00
7C	00 00 00 00
80	00 00 00 00
84	00 00 00 00
88	00 00 00 00
8C	00 00 00 00
90	00 00 00 00
94	00 00 00 00
98	00 00 00 00
9C	00 00 00 00
A0	00 00 00 00

Flags / State

Z:	0	S:	0
O:	0	Status:	AOK

Specific Registers

PC:	00 00 00 08	IR:	20 10 00 00
MA:	00 00 00 04	MD:	20 10 00 00
A:	00 00 00 00	C:	00 00 00 08

General Registers

%rsp :	00 00 00 CC	%rbp :	00 00 00 CC
%r2 :	00 00 00 00	%r3 :	00 00 00 00
%r4 :	00 00 00 00	%r5 :	00 00 00 00
%r6 :	00 00 00 00	%r7 :	00 00 00 00

I - Introduction

This project tries to implement a software simulation of a CPU running a simplified version of the Y86 Instruction Set Architecture. It is designed to be able to process and run bytecode based on a dynamic RTN operations set through a simulated, configurable virtual machine. The simulator will take as inputs a configuration file as well as an RTN operations set that describes low-level operations for each implemented assembly instruction. The software will also be able to load a program in the form of bytecode/object code (that is the equivalent of a compiled assembly file) in memory in order to run it. In this report, we will go through the design, implementation and testing of the software simulation that we created.

II - Simulation Design

Our simulation was designed to be based on a simplified model of a CPU, therefore we proceeded to modeling every major component of a CPU using Java classes whose interactions with each other replicate the connections between components in a real-life CPU.

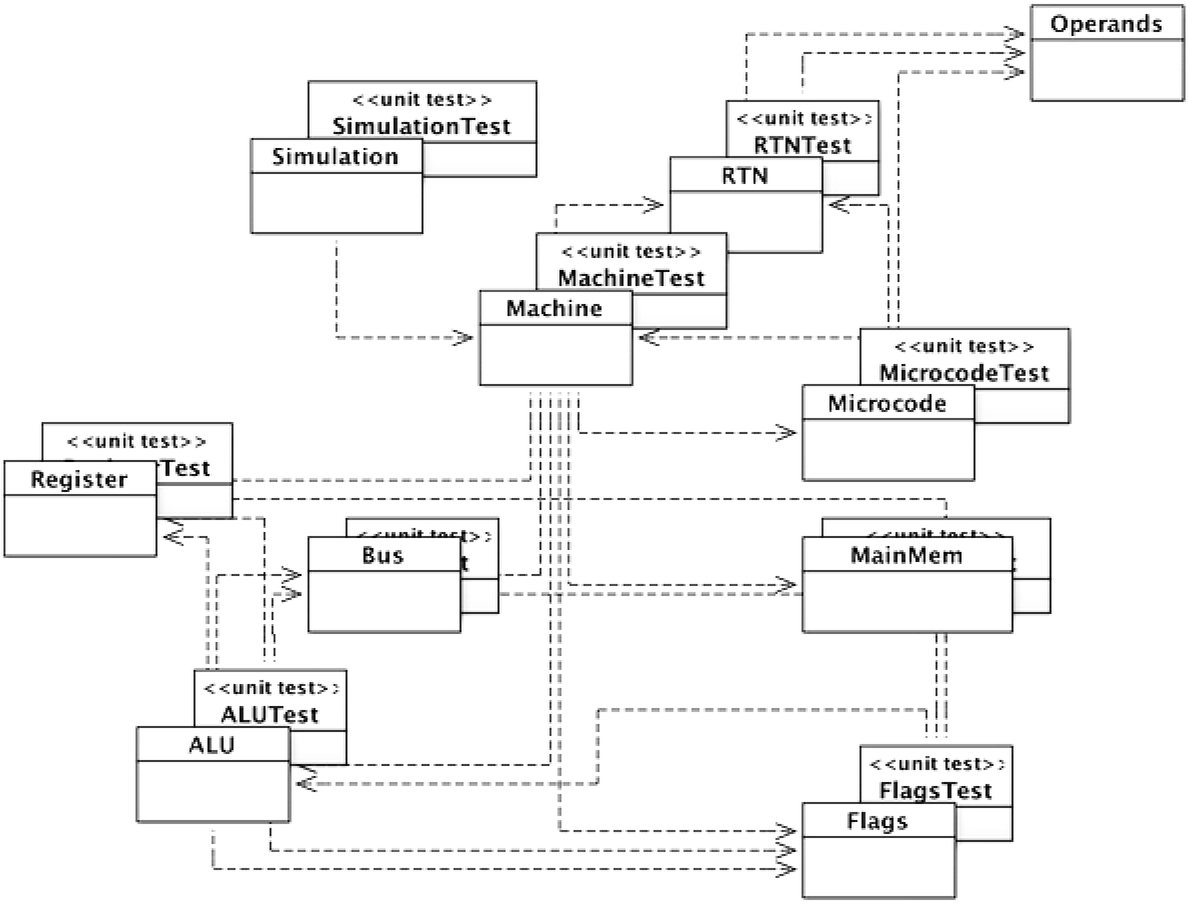


Figure 1. The final simulator classes

Main Memory (MainMem.Java)

The first component we started implementing was the main memory. In fact, determining how the main memory will be modeled has a great impact on how the other components will be implemented since the main memory specifies the structure that will hold our machine's data.

In order to comply with the linear structure of a real memory, we opted for a one-dimensional array of "bytes" (the Java type). This will therefore make our simulation's memory byte-addressable and will also allow for easy transfer of chunks of data (in particular, words). In fact, to transfer a word from memory to other components, we only have to get the sub-array spanning from the memory address requested until we obtain an array of size **`wordSize`**. Therefore, we use a simple calculation to map the addresses of word-sized chunks of memory to the indices of the array: **`address = index % wordSize;`** For a word size of 4 Bytes, our array implementation will look like this:

Addressable Memory		Array Implementation	
Address	Content	Index	Content (type byte)
0	0x3004001C	0	0x30
		1	0x04
		2	0x00
		3	0x1C
4	0x40340000	4	0x40
		5	0x34
		6	0x00
		7	0x00

Figure 2. Modeling memory with a 1-dimensional array of bytes

As can be determined from *Figure 2*, our design puts the Big Endian ordering format into use. We found Big-Endian to be more intuitive for reading and writing to the bus and registers as will be discussed in the next section.

To visualize the data inside the memory (using the GUI), we had to implement a **`byteToHex()`** method that converts an array of bytes (usually word-sized) to a hex string.

General and specialized registers (Register.Java)

Registers are the next most important component in our CPU simulation. Surprisingly, since we have already determined a design for our memory component, the design of registers seemed to be simple. We therefore used a word-sized array of bytes to hold the register data and implemented **`read`** and **`write`** methods to manipulate the data. We also needed methods to read and write data in the form of an integer so that we can easily perform arithmetic operations in the ALU (will be discussed in the implementation section).

Since specialized registers have the same functionality as general registers (with some limitations in terms of where to read from and write to), we did not feel the need to model them as classes. We therefore opted for implementing specialized registers (as well as general registers) as objects of the **`Register`** class and let the **`Machine`** class handle the distinction between general and specialized registers.

System Bus (Bus.Java)

Although we did not need a system bus for our implementation of the simulation (since we are transferring at most word-sized data), we implemented a **`Bus`** class that can theoretically hold and transfer data whose size is a multiple of the word size.

The **Bus** class is structurally very similar to a **Register**. In fact, it has almost the same methods (reading and writing byte arrays as well as reading and writing integer sized data). The main difference is the size which is a multiple of a register's size.

Flags and Status byte (Flags.Java)

In order to implement assembly jumps as well as ALU operations, we have to have the three basic Y86 flags : SF (Sign Flag), ZF (Zero Flag) and OF (Overflow Flag). We have designed the flags to be implemented inside a **Flags.Java** class as boolean variables. The **Flags.Java** class will play the role of the Status Register.

In addition to status flags, the **Flags.Java** class also holds the program status byte. We chose to implement the status byte as a string variable since it will be shown on the GUI. The status byte can take one of these values :

- AOK: No problem encountered.
- ADR: An addressing error has occurred
- INS: An illegal instruction was encountered
- HLT: A halt instruction was encountered

As will be explained in the "Machine" section, the simulation only runs when the status byte is set to "AOK", any other value will stop the machine from running.

Microcode (RTN.Java, Microcode.Java and Operands.Java)

The Microcode is a very important component in our simulation (as well as in a physical CPU). In fact, the Microcode holds the low-level instruction-set that describes how every Y86 operation will be handled by the CPU. The Microcode implementation was designed to be split into several classes. The **RTN.Java** class has the sole purpose of parsing the RTN file and storing the required RTN operations for each Y86 instruction inside a data structure. We opted to use a **HashMap** with the keys being the byte encoding of each Y86 "opCode" and the values being an **ArrayList** of **Operands** (each "Operands" object being a tuple formed of the left-hand-side operand and right-hand-side operand of each RTN assignment operation).

The RTN definitions of Y86 functions should be stored in a text file in this format :

```
y86op:
      X←Y
      Y←Z
      ...
```

If the RTN definition requires a conditional (conditional jumps as an example), the condition has to be enclosed between parenthesis and a right-facing arrow should precede the conditioned assignment , for example :

```
y86op:
       (SF&OF)→X←Y
```

As an example, let's look at the "**addq**" operation (encoded as **0x60**). Let's assume our rtn.txt file contains the following definition :

```
addq:
      A←R[arg1]
      C←A+R[arg2]
      R[arg2]←C
```

The **RTN.Java** class will therefore parse this code using Regex matching and return a HashMap that contains the separated operands. For the **addq** operation, **RTN.Java** will return the following HashMap :

```
instructionList = {
0x60 : [new Operands(“A”, “R[arg1]”),
new Operands(“C”, “A+R[arg2]”),
new Operands(“A[arg2]”, “C”)]
}
```

In the case where the operation requires a condition then the condition will be added (as a string) to a separate **conditionList** implemented as a **HashMap** where the keys represent the byte encoding of the opCode and the values are the parsed condition strings.

Both **instructionList** and **conditionList** will be therefore interpreted by the **Microcode.Java** class as the program is being executed instruction by instruction. To do so, we built methods that first read the value from the right hand operand (fetch the value from the appropriate medium depending on the interpretation of the right-hand-side operand) and write that value to the appropriate medium as determined by the left-hand-side operand.

When a Y86 instruction is to be executed, **Microcode.Java** executes the fetch cycle as described by the RTN, then queries the instruction register to obtain the instruction. Then decomposes the instruction to its core components :

0x6	0x2	0x2	0x3	0x0	0x0	0x0	0x0	0x0
opCode (1 byte)		reg. specifier (1 byte)		data (address, immediate, etc.)				

Figure 3. Design of an instruction (similar to Y86-32)

The opCode is therefore used to obtain the RTN definition and conditions for the instruction from **RTN.Java** (since the RTN definition **HashMap** is indexed by opCode). Once we obtain the condition and the operations **ArrayList**, we verify whether the condition is met or not (using status flags) then proceed to interpreting the RTN operations one by one. For each operation, we are given an "Operands" object consisting of the parsed right-hand-side and left-hand-side operators (ex. **`Operands("C", "A+R[arg2]")`**). Therefore **`Microcode.Java`** will interpret the right-hand-side first as follows :

- If there exists any of the allowed arithmetic operations (ex : "+", "-", "*", etc) in the right-hand-side operand then handle the right hand side by the ALU and get the value (split the string by the arithmetic operator then determine the operation to be executed by the ALU).
- If the right-hand side operand is "**`M[MA]`**" then there is no need to look at the left-hand side since the left-hand side will always be **`MD`**. In this case, directly read the memory content and place it in **`MD`**.
- Otherwise determine the register to read from using the value of the right-hand side, read it and place the value in the System Bus. If the register to read from is a general register then the right hand side will be either "**`R[arg1]`**" or "**`R[arg2]`**", in this case we look at the instruction's second byte to determine which register to read from.

Once we obtained the value to be assigned, we take a look at the left-hand-side of the operation and determine where to assign the value. There exist three cases:

- Either we handled the assignment when we encountered "**`M[MA]`**" in the right-hand-side.
- Or the left-hand-side is "**`M[MA]`**" in which case we read data from **`MD`** then write directly to the memory (data written to the memory can't come from anywhere other than **`MD`**)
- Or we determine the register to write to using the value of the left-hand side, read from the bus then place the value in the target register. If the register to write to is a general register then the left hand side will be either "**`R[arg1]`**" or "**`R[arg2]`**", in this case we look at the instruction's second byte to determine which register to write to.

The ALU (ALU.Java)

The ALU class models the CPU's Arithmetic and Logic Unit, it has elementary arithmetic and logic operations that can be executed after storing data in the "A" register and the Bus. The results from the ALU operations are always stored in "C", thus the methods in the ALU do not take any parameters (since the inputs and outputs are constant).

We have implemented several operations that the configuration file can choose from to make available to the RTN definition file :

1. Add (adds the bus value and the value stored in A)
2. Increment (increments the bus value by an immediate value)
3. Sub (subtracts the bus value from the value stored in A)
4. Decrement (decrements the bus value by an immediate value)
5. Multiply (multiplies bus and A)
6. AND (bitwise and between A and bus)
7. OR
8. XOR
9. NOT

To implement these methods, we used the **`readInt()`** and **`writeInt()`** methods of the bus and registers in order to use standard Java integer operations on the values (that were stored as byte arrays). After each arithmetic/logic operation the sign, zero and overflow flags are set accordingly.

The Machine (Machine.Java)

`Machine.Java` is the class that orchestrates all of the operations of our simulation. When a Machine object is created, it initializes multiple variables (components) as follows :

- An RTN definition (an **`RTN`** object which initializes parsing the RTN file)
- A Microcode (that has access to the RTN definition and the status flags)
- A number of general registers (array of **`Register`** objects of size determined by the configuration file) **`%r0`** and **`%r1`** are reserved as **`%rsp`** and `%rbp` respectively
- Specialized registers (also **`Register`** objects) : PC, IR, MD, MA, A and C
- A Main Memory, an ALU, a System Bus and Flags

The machine provides one simple method "**`run()`**" which prompts the Microcode to run the fetch-execute cycle in case the status of the machine is "AOK" (no exception occured).

III - Implementation Features and Limitations

Our implementation of this design tries to create an instruction set that is very similar to Y86. We first started by choosing a word size of 4 bytes that allows us to perform arithmetic operations by treating each word-sized byte chunk as a Java integer inside the ALU.

We also opted for the simplification of taking the bus size as equal to the word size (although our bus supports transferring byte chunks bigger than the word size, we did not feel the need to do so since our instructions are word-sized). This also allows us to cast word-sized byte arrays to integers to be able to operate on them easily inside the ALU class (both the word size and the integer size are 4 bytes).

To encode the instructions, we tried to follow the Y86 instruction format however we introduced some changes to simplify the implementation. As described in the Microcode section, the first byte of the instruction is reserved to the opCode while the second byte is occupied by either the register specifiers or is part of an address that spans till the end of the instruction (3 bytes). If this is the case (usually in case of a jump), our maximum addressable space is $2^{8\cdot(wordSize-1)} = 2^{8\cdot 3} = 2^{24}$ addresses.

The instructions (which are similar to Y86-32) are encoded as follows :

instruct.	opCode		Reg. specifiers		Data		Data	
halt	0	0	-	-	-	-	-	-
nop	1	0	-	-	-	-	-	-
rrmovq	2	0	src	dst	-	-	-	-
irmovq	3	0	-	dst	val	val	val	val
rmmovq	4	0	src	dst	-	-	-	-
mrmovq	5	0	src	dst	-	-	-	-
addq	6	0	-	-	-	-	-	-
subq	6	1	-	-	-	-	-	-
andq	6	2	-	-	-	-	-	-
xorq	6	3	-	-	-	-	-	-
jmp	7	0	dst	dst	dst	dst	dst	dst
jle	7	1	dst	dst	dst	dst	dst	dst
jl	7	2	dst	dst	dst	dst	dst	dst
je	7	3	dst	dst	dst	dst	dst	dst
jne	7	4	dst	dst	dst	dst	dst	dst
jge	7	5	dst	dst	dst	dst	dst	dst

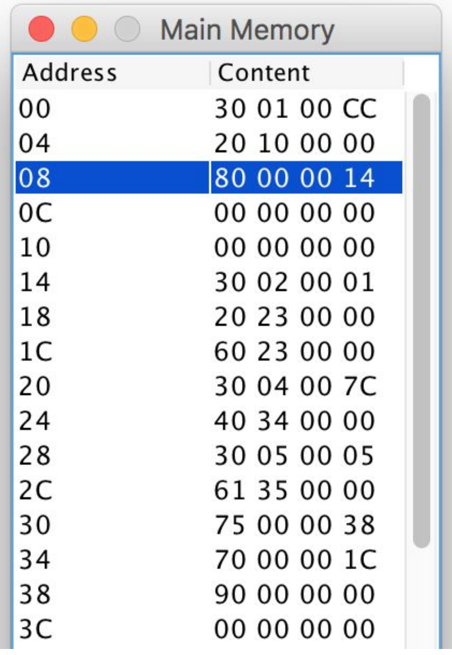
jg	7	6	dst	dst	dst	dst	dst	dst
call	8	0	dst	dst	dst	dst	dst	dst
ret	9	0	-	-	-	-	-	-
pushq	A	0	src	-	-	-	-	-
popq	B	0	dst	-	-	-	-	-

Figure 4. opCodes and arguments for each implemented instruction

Graphical User Interface (Simulation.Java)

The front-end part of our simulations consists of a simple Graphical User Interface (GUI) that provides control over the simulation as well as show the status/content of each simulated component of the CPU. The GUI is divided into two separate windows : a main control/status window and a memory content panel.

The memory content panel shows the current data that the Main Memory holds and is refreshed every time an instruction is executed to reflect the changes to the memory. The panel is divided into two columns : An Address column and a Content column. The Address column shows the hex-encoded memory address that indicates the beginning of a word-sized block (thus in our simulation addresses increase by 4 since the word size is 4 bytes). The Content column shows the hex-encoded aggregated content of the 4 bytes starting at the indicated address. The memory panel also serves to show which instruction is currently being executed (the instruction that is currently in the IR will be highlighted in blue).

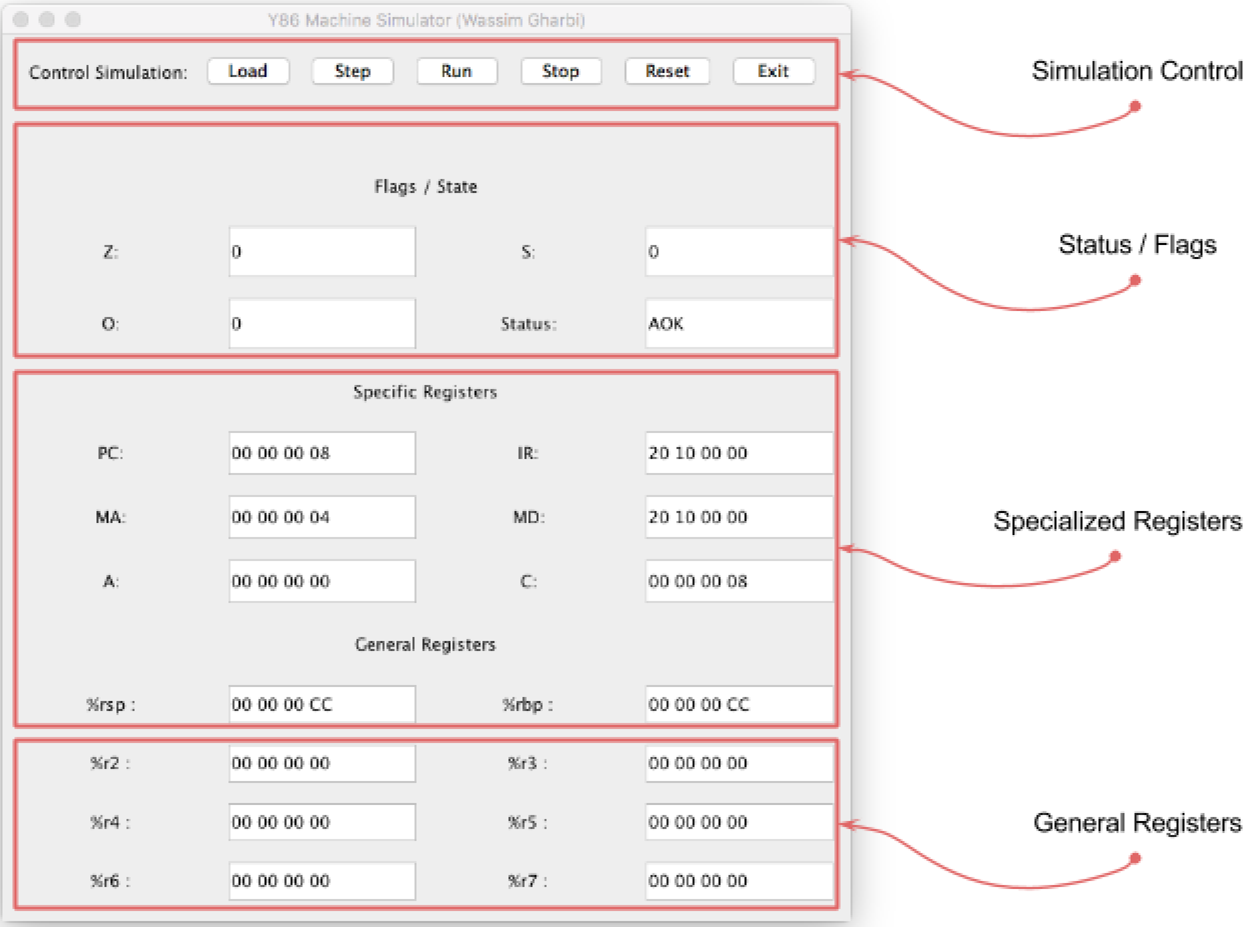


Main Memory

Address	Content
00	30 01 00 CC
04	20 10 00 00
08	80 00 00 14
0C	00 00 00 00
10	00 00 00 00
14	30 02 00 01
18	20 23 00 00
1C	60 23 00 00
20	30 04 00 7C
24	40 34 00 00
28	30 05 00 05
2C	61 35 00 00
30	75 00 00 38
34	70 00 00 1C
38	90 00 00 00
3C	00 00 00 00

The main window is divided into multiple sections : Control, Status/Flags, Specialized Registers and General Registers. To control the simulation use the following commands :

- Load : Loads the demonstration program into memory (will be discussed later)
- Step : Runs a single Fetch-Execute cycle every time the button is clicked
- Run : Runs a Fetch-Execute cycle every 0.5 seconds
- Stop : Stops the timer that controls the clock
- Reset : Clears memory content and resets all registers
- Exit : Closes the simulation



Flags / State

Z: 0　S: 0

O: 0　Status: AOK

Specific Registers

PC: 00 00 00 08　IR: 20 10 00 00

MA: 00 00 00 04　MD: 20 10 00 00

A: 00 00 00 00　C: 00 00 00 08

General Registers

%rsp : 00 00 00 CC　%rbp : 00 00 00 CC

%r2 : 00 00 00 00　%r3 : 00 00 00 00

%r4 : 00 00 00 00　%r5 : 00 00 00 00

%r6 : 00 00 00 00　%r7 : 00 00 00 00

Simulation Control

Status / Flags

Specialized Registers

General Registers

Figure 5. Components of the Graphical User Interface

Testing the simulation

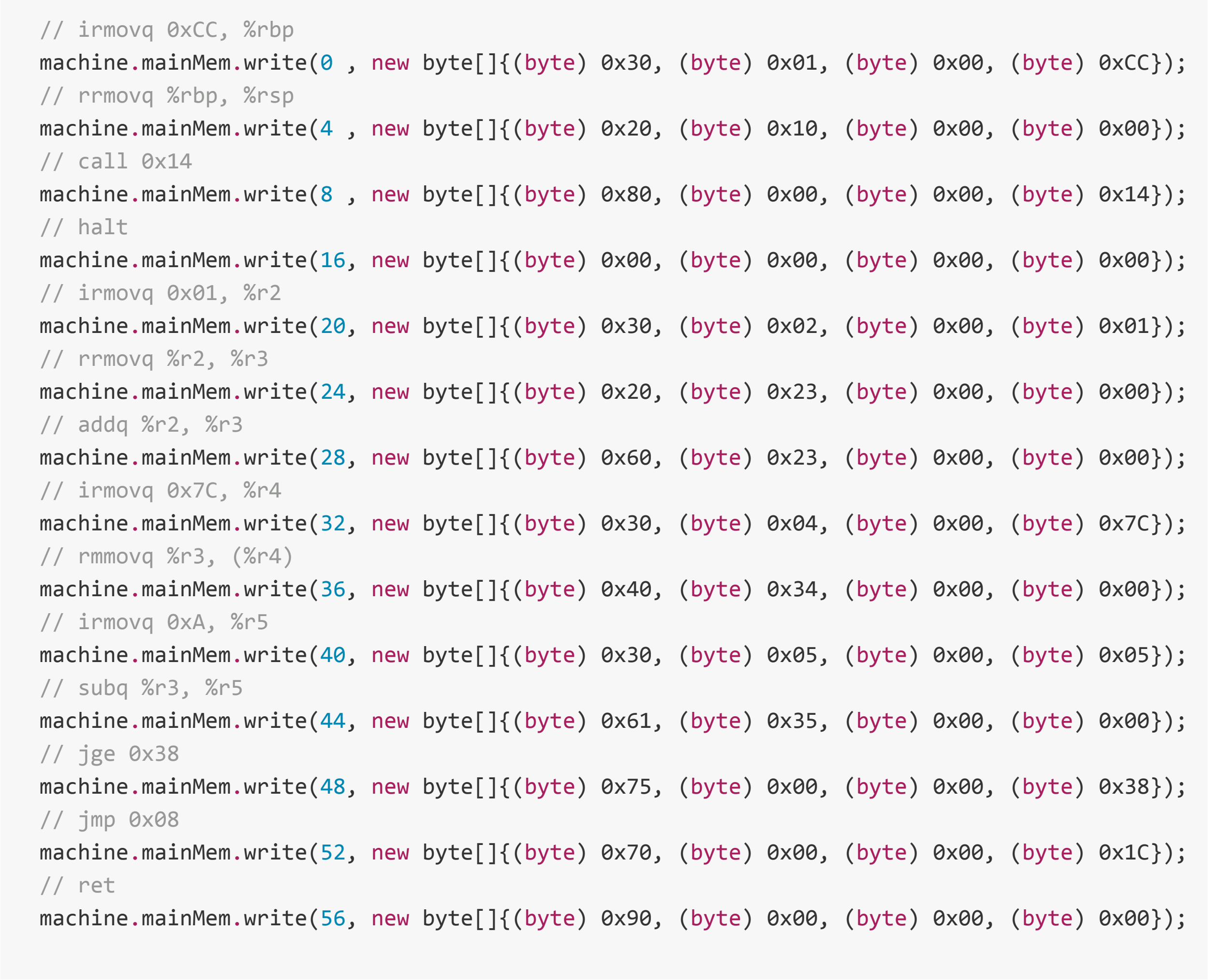
Testing played a big role in implementing the simulation since the components are extremely connected to each other and the correctness of the implementation of one component relied heavily on the assumption that other components work correctly.

We therefore had to implement multiple unit tests that demonstrated that every component did what it is supposed to do correctly. We created read/write tests for registers, memories and system buses of multiple sizes and contents, we created arithmetic and logic tests to evaluate the functionalities of the ALU and we made sure that each flag (zero, sign and overflow) is set appropriately after each arithmetic or logic operation.

We also heavily tested the encoding/decoding of instructions and the fetch-execute cycle by making sure conditions are met when flags are set and making sure the instructions affect the target components in the processor.

The Provided Demo Program

The built-in demo program consists of the implementation of for loop that increments a counter and writes it to a memory location (0x7C) until the counter is equal to 5. The program demonstrates most implemented Y86 instructions such as **`mrmovq`**, **`rrmovq`**, **`irmovq`**, **`addq`**, **`jmp`** and **`jge`** (conditional jump). The demo program also makes use of the **`call`**, return (**`ret`**) and **`halt`** instructions to demonstrate the memory stack implementation. The program is hard-coded in **`Simulation.java`** and is written to memory through byte arrays.



```
// irmovq 0xCC, %rbp
machine.mainMem.write(0 , new byte[]{(byte) 0x30, (byte) 0x01, (byte) 0x00, (byte) 0xCC});
// rrmovq %rbp, %rsp
machine.mainMem.write(4 , new byte[]{(byte) 0x20, (byte) 0x10, (byte) 0x00, (byte) 0x00});
// call 0x14
machine.mainMem.write(8 , new byte[]{(byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x14});
// halt
machine.mainMem.write(16, new byte[]{(byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00});
// irmovq 0x01, %r2
machine.mainMem.write(20, new byte[]{(byte) 0x30, (byte) 0x02, (byte) 0x00, (byte) 0x01});
// rrmovq %r2, %r3
machine.mainMem.write(24, new byte[]{(byte) 0x20, (byte) 0x23, (byte) 0x00, (byte) 0x00});
// addq %r2, %r3
machine.mainMem.write(28, new byte[]{(byte) 0x60, (byte) 0x23, (byte) 0x00, (byte) 0x00});
// irmovq 0x7C, %r4
machine.mainMem.write(32, new byte[]{(byte) 0x30, (byte) 0x04, (byte) 0x00, (byte) 0x7C});
// rmmovq %r3, (%r4)
machine.mainMem.write(36, new byte[]{(byte) 0x40, (byte) 0x34, (byte) 0x00, (byte) 0x00});
// irmovq 0xA, %r5
machine.mainMem.write(40, new byte[]{(byte) 0x30, (byte) 0x05, (byte) 0x00, (byte) 0x05});
// subq %r3, %r5
machine.mainMem.write(44, new byte[]{(byte) 0x61, (byte) 0x35, (byte) 0x00, (byte) 0x00});
// jge 0x38
machine.mainMem.write(48, new byte[]{(byte) 0x75, (byte) 0x00, (byte) 0x00, (byte) 0x38});
// jmp 0x08
machine.mainMem.write(52, new byte[]{(byte) 0x70, (byte) 0x00, (byte) 0x00, (byte) 0x1C});
// ret
machine.mainMem.write(56, new byte[]{(byte) 0x90, (byte) 0x00, (byte) 0x00, (byte) 0x00});
```

Figure 6. The provided demonstration program and its interpretation in Assembly

IV - Conclusion

In conclusion, implementing a CPU simulation and making design decisions on what approach to use to model each component as well as designing an Instruction Set Architecture was a very instructive and rewarding experience. We learned that small decisions make a great impact on the architecture of a processor and we also learned that thorough unit testing of each component makes scaling up an easy task.

V - References

1. Bryant, Randal E., and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Boston: Prentice Hall, 2011. Print.
2. Computer Science 203 Computer Organization, Professor Pfaffmann, *Project Description*, http://www.cs.lafayette.edu/~pfaffmaj/courses/f16/cs203l/labs/lab01/
3. Carnegie Mellon University, R. E. Bryant, D. R. O'Hallaron, *CS:APP2e Guide to Y86 Processor Simulators*. Retrieved November 5, 2016 from http://csapp.cs.cmu.edu/2e/simguide.pdf
4. Michigan Technology University, Murat M. Köksal, *GNU Assembler Directives* http://cs.mtu.edu/~mmkoksal/blog/?x=entry%3Aentry120116-130037