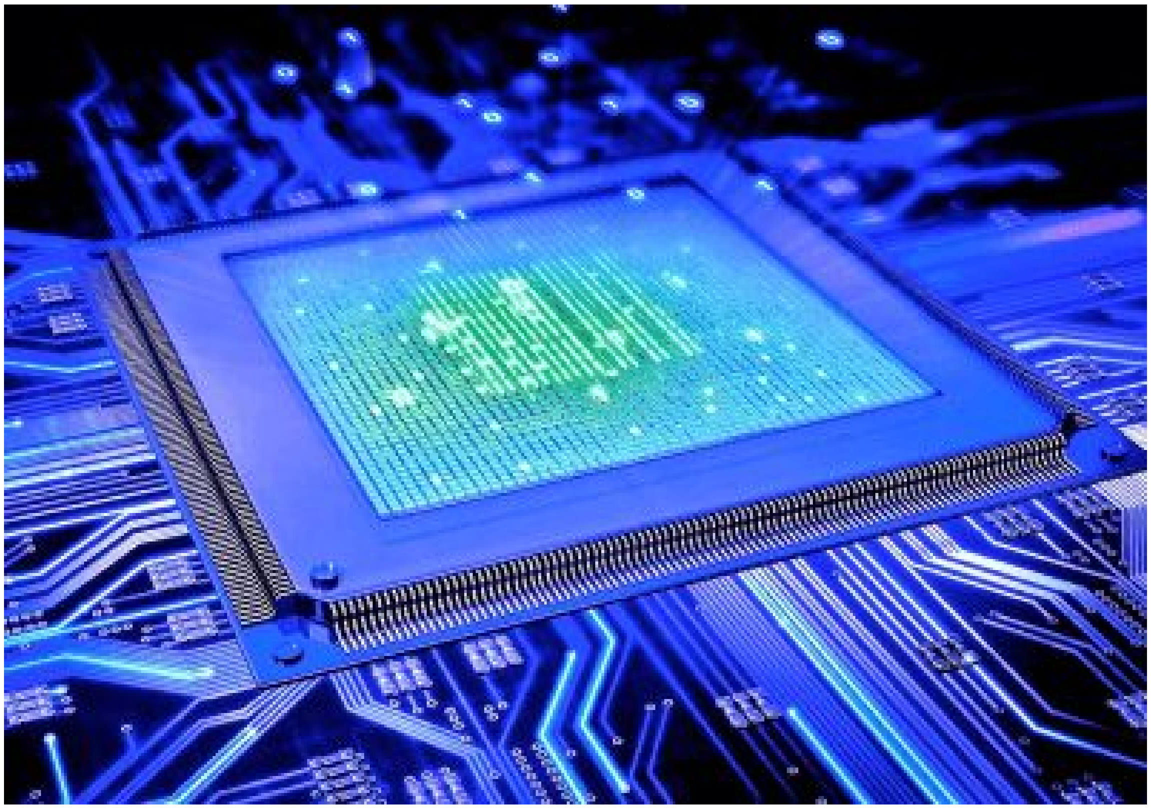
CPU Project Report



CS 203: Computer Organization

Luis Lopez-Bonilla

10/22/17

User Manual:

Compiling:

To compile the program all that has to be done is to run the make command from the project root(CS203Project). This will run a script that will execute compile commands for all three components of the **tool chain**. The commands they run are:

```
javac Simulator/src/*.java
javac Assembler/src/*.java
javac Viewer/src/*.java
```

Running:

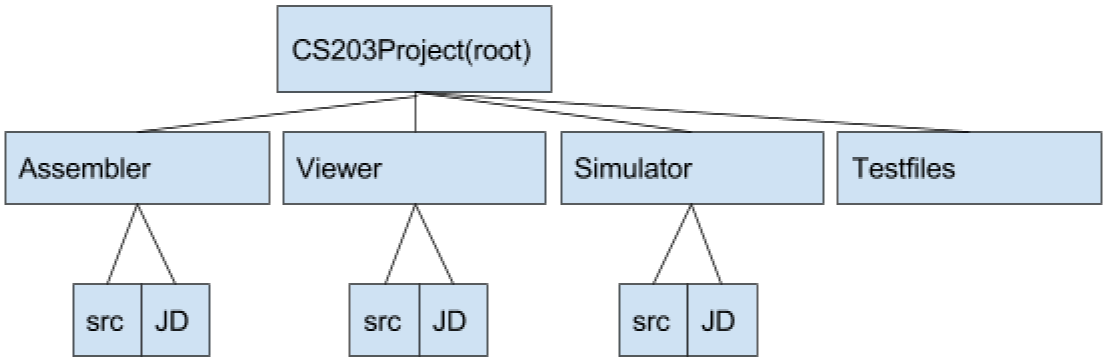
To run the components of the tool chain just call make run from the root directory. Otherwise the general running commands are as follows:

```
java -cp Simulator/src Main filepath filename noisy hex
java -cp Assembler/src Main filepath filename
java -cp Viewer/src Main filepath filename hex start end
```

Javadoc:

Each component of the tool chain has it's own directory and stored in each directory is a javadoc for each component. The javadoc is stored in directory named JavaDoc in each directory.

Project Directory Hierarchy:



Introduction:

The purpose of this project was to design and implement our own version of the assembly language and create a tool chain. The tool chain contains and Assembler, Visualizer, and a Simulator that would work with the self created assembly code. This report will walk through the uses of the tool chain, design choices, and also flaws.

Walkthrough:

Assembler:

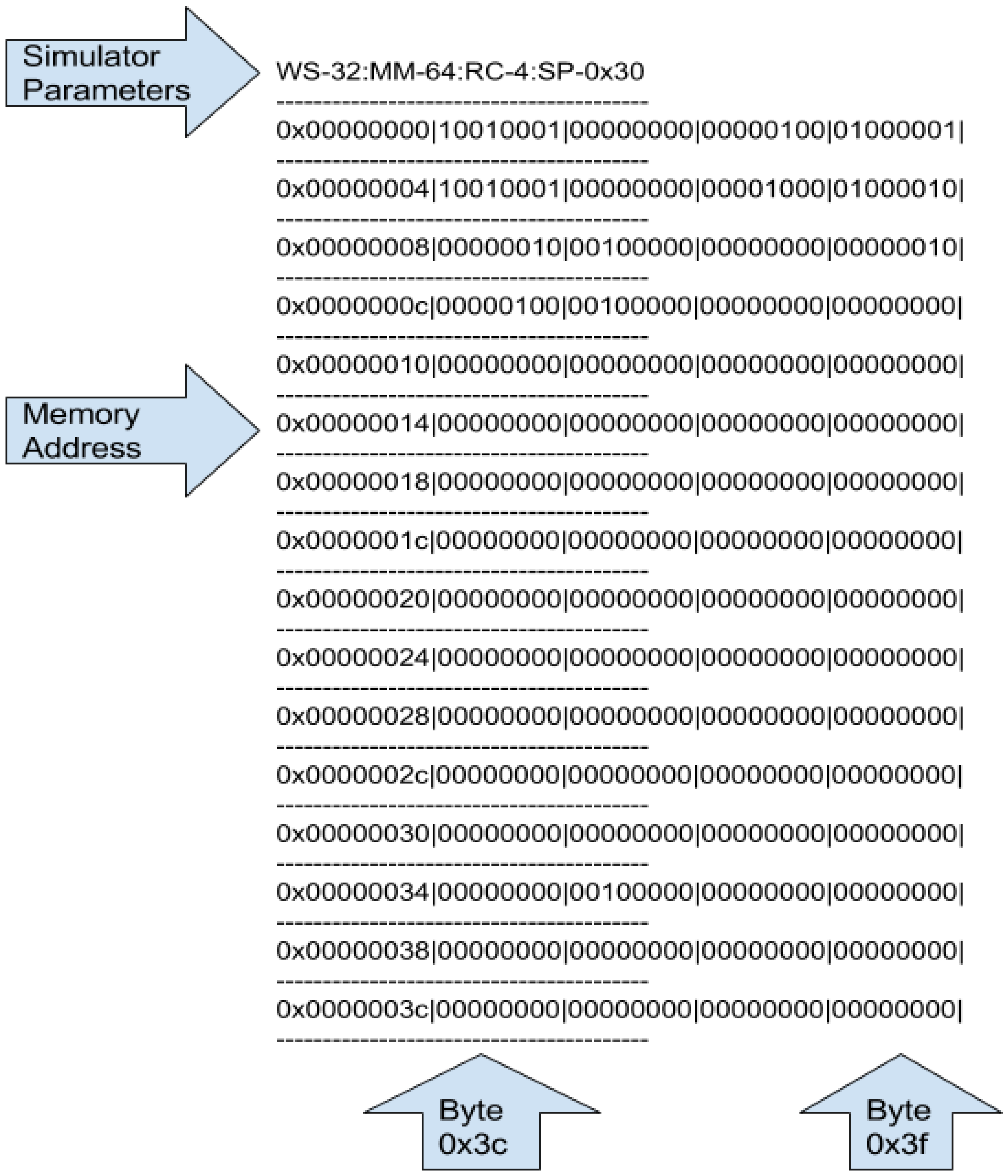
Assembler takes in an .as file. An example assembly file is provided below.

```
.wordsize 32            ; sets the machine wordsize
.regcnt    4            ; identifies number of gen purpose registers
.maxmem   0x40          ; max memory size is 64 bytes

ADDI x1, x2, #1;
ADDI x2, x2, #2;
PUSH x2;
POP  x0;
.pos 0x30
stack:
.pos 0x34
HALT
```

Figure 1: Assembly Example

The assembler checks for either directives, labels, or assembly instructions. Every instruction should end with a semicolon and the instruction name should be entirely capitalized. The only exception to this rule is the HALT instruction. HALT does not require a semicolon at the end of the instruction. All directives should start with a period and have a space between the directive and any argument included with the directive. Also, the commas and spaces shown above for normal instructions must always be included(ie. ADD a, b, c;). Labels are declared as “somename:” they must always have a colon at the end. The resulting memory image from running the above assembly file through the assembler is shown below.



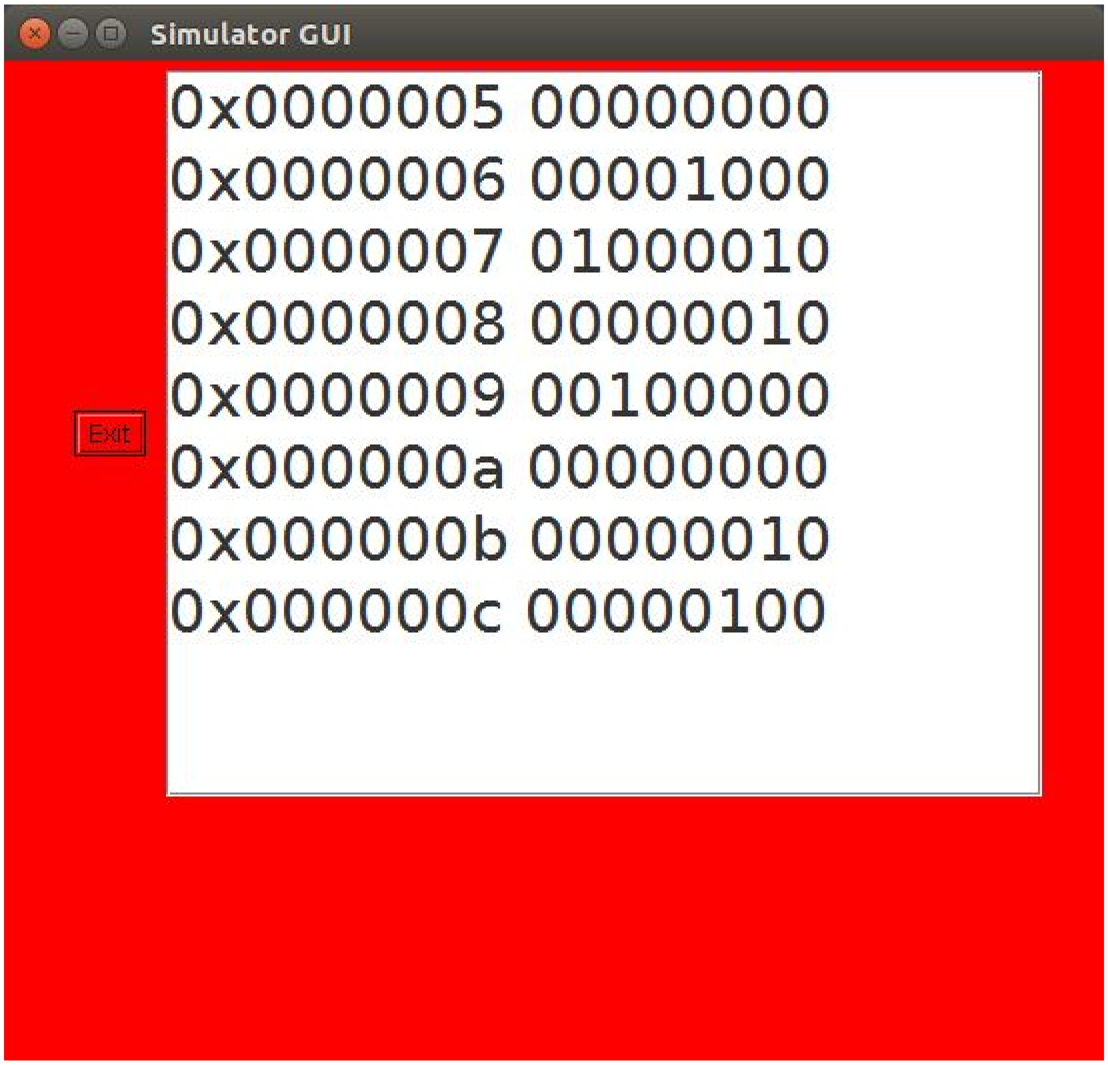
```
WS-32:MM-64:RC-4:SP-0x30
----------------------------------------
0x00000000|10010001|00000000|00000100|01000001|
----------------------------------------
0x00000004|10010001|00000000|00001000|01000010|
----------------------------------------
0x00000008|00000010|00100000|00000000|00000010|
----------------------------------------
0x0000000c|00000100|00100000|00000000|00000000|
----------------------------------------
0x00000010|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000014|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000018|00000000|00000000|00000000|00000000|
----------------------------------------
0x0000001c|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000020|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000024|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000028|00000000|00000000|00000000|00000000|
----------------------------------------
0x0000002c|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000030|00000000|00000000|00000000|00000000|
----------------------------------------
0x00000034|00000000|00100000|00000000|00000000|
----------------------------------------
0x00000038|00000000|00000000|00000000|00000000|
----------------------------------------
0x0000003c|00000000|00000000|00000000|00000000|
----------------------------------------
```

Figure 2: Memory Image

As the above diagram shows, the first line of the image file contains wordsize, max mem, number of registers,memory starts at 0x00000000 and goes until the max memory's hex equivalent. Each line in the memory image contains 4 bytes because that is the word size of the program. The word size can not be changed. I created a 32-bit ISA and decided to design my program around a word size of 32 bits. The number of registers and max memory is completely able to be manipulated.

Viewer:

The viewer takes in a machine code file. It takes in a file similar to figure 2. The viewer simplies visualizes a certain region of the memory image into a Gui. The region is inputted through the command line through the parameters start and end. An example is shown below. The figure 3 below shows the viewer displaying memory from a range of 5-12 so in total 8 bytes are being displayed on the Gui.



```
0x0000005 00000000
0x0000006 00001000
0x0000007 01000010
0x0000008 00000010
0x0000009 00100000
0x000000a 00000000
0x000000b 00000010
0x000000c 00000100
```

Figure 3: Viewer

Simulator:

The simulator takes in a machine code file. The machine code is parsed and placed into the simulator's memory image. The simulator showcases the fetch and execute cycle.

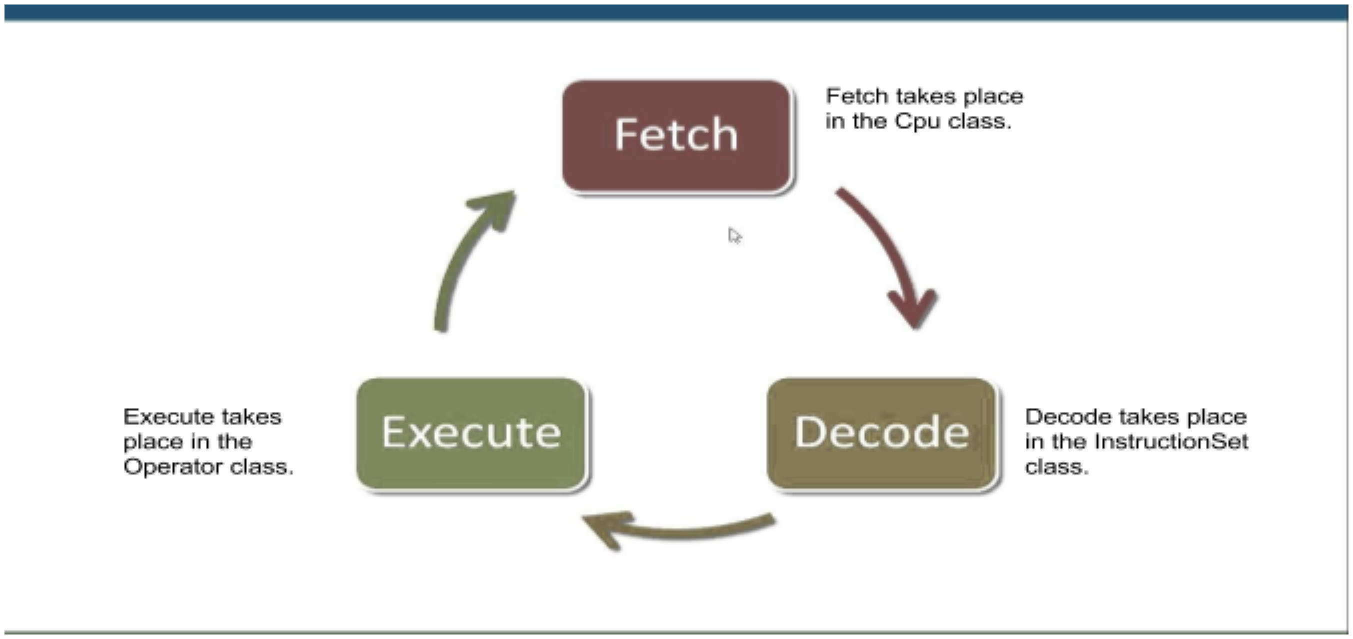
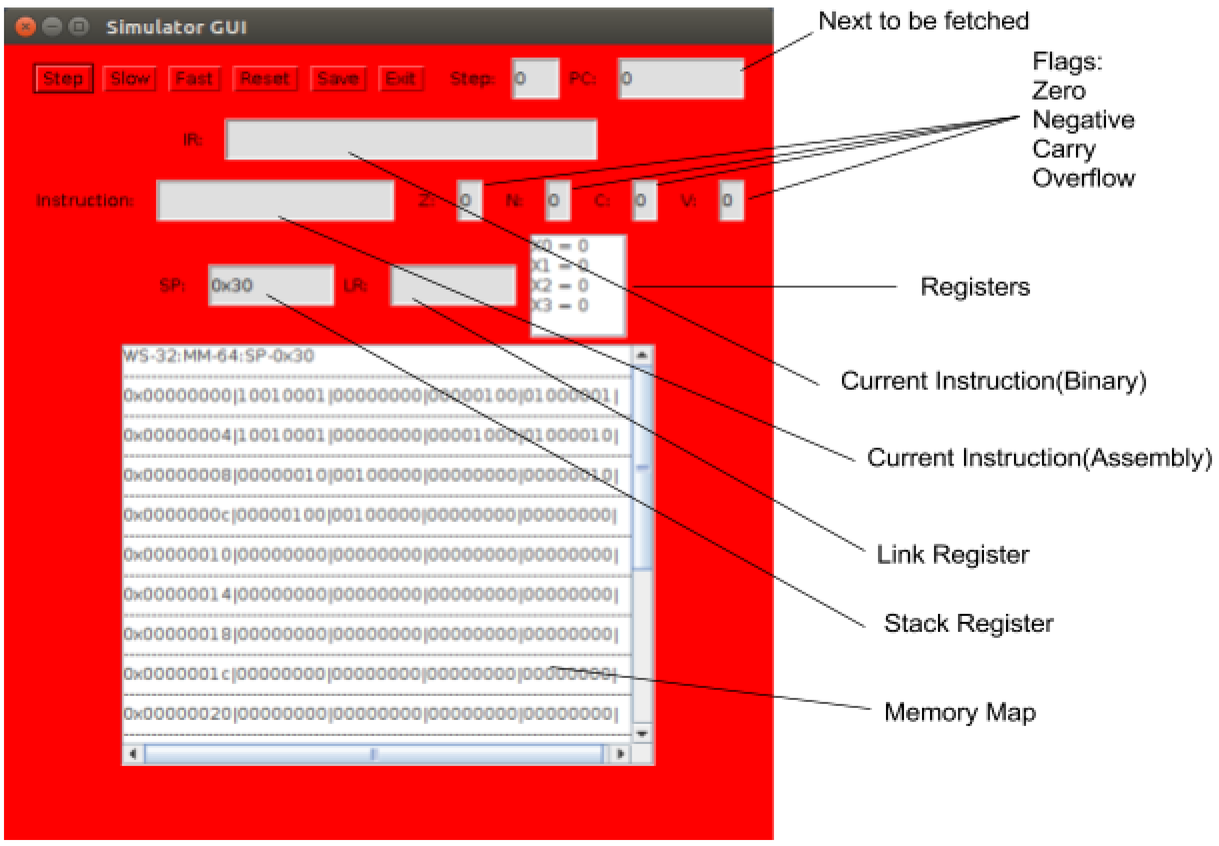

Fetch
Fetch takes place in the Cpu class.
Decode
Decode takes place in the InstructionSet class.
Execute
Execute takes place in the Operator class.

Figure 4: FDE classes



Step: 0 PC: 0

IR:

Instruction: Z: 0 N: 0 C: 0 V: 0

SP: 0x30 LR:

```
X0 = 0
X1 = 0
X2 = 0
X3 = 0
```

```
WS-32:MM-64:SP-0x30
0x00000000|10010001|00000000|00000100|01000001|
0x00000004|10010001|00000000|00001000|01000010|
0x00000008|00000010|00100000|00000000|00000010|
0x0000000c|00000100|00100000|00000000|00000000|
0x00000010|00000000|00000000|00000000|00000000|
0x00000014|00000000|00000000|00000000|00000000|
0x00000018|00000000|00000000|00000000|00000000|
0x0000001c|00000000|00000000|00000000|00000000|
0x00000020|00000000|00000000|00000000|00000000|
```

Figure 5: Simulator

The simulator has two options that are passed through the command line. It has noisy mode that prints to the command line whenever a register is altered and a hex mode that represents the memory image in hex rather than binary.

The simulator offers five options to the user. The step method which carries out one fetch and execute cycle. Two options(fast and slow) for continuous fetch and execute cycles. A reset method which restores the memory image back to before the simulator was started. Finally, a save method which writes the memory image and simulator status to two separate files. The two files are called mem”filename”.txt and sim”filename”.txt. Example files are shown below.

```
WS-32:MM-64:SP-0x30
-------------------------------------------------------------------------------------------
0x00000000|10010001|00000000|00000100|01000001|
-------------------------------------------------------------------------------------------
0x00000004|10010001|00000000|00001000|01000010|
-------------------------------------------------------------------------------------------
0x00000008|00000010|00100000|00000000|00000010|
-------------------------------------------------------------------------------------------
0x0000000c|00000100|00100000|00000000|00000000|
-------------------------------------------------------------------------------------------
0x00000010|00000000|00000000|00000000|00000000|
-------------------------------------------------------------------------------------------
0x00000014|00000000|00000000|00000000|00000000|
-------------------------------------------------------------------------------------------
```

```
Registers:
X0 = 0
X1 = 0
X2 = 0
X3 = 0
SP: 0x30
LR: null
Flags:
Z = 0
N = 0
C = 0
V = 0
PC: 0x4
```

Figure 6: Mem and Sim Output Files

Design:

Instruction Set:

At the start of the designing process, I decided to copy the Arm Instruction Set that we were doing in class, but as I changed my mind as I continued designing and implementing. The Arm Set became to complicated and large scale and so I decided it was not wise for me to continue implementing it. So my Instruction Set became a pseudo Arm Set. The types are shown below.

R	opcode(11)	rm(5)	shamt(6)	rn(5)	rd(5)
I	opcode(11)	alu(12)	rn(5)	rd(5)	
D	opcode(11)	dtadd(9)	opp(2)	rn(5)	rt(5)
B	opcode(11)	addr(21)			
CB	opcode(11)	addr(16)	rt(5)		
S	opcode(11)	rn(21)			

Figure 7: Instruction Types

HALT ADD ADDI ADDS ADDIS SUB SUBI SUBS SUBIS LDUR STUR	AND ORR EOR LSL LSR B BL CBNZ CBZ PUSH POP

Figure 8: Implemented Commands

Directives:

Directives are instructions that affect the assembler and are not technically project code.

- .align: A directive that moves the address in memory to a number divisible by the value of the argument. The value of the argument must be positive.
- .pos: A directive that moves the address in memory to the value of the argument. The value must be in the memory range.
- .double: A directive that inserts a 64-bit value into the memory image. The value is padded to fit 64 bits if necessary.
- .single: A directive that inserts a 32-bit value into the memory image. The value is padded to fit 32 bits if necessary.
- .half: A directive that inserts a 16-bit value into the memory image. The value is padded to fit 16 bits if necessary.
- .byte: A directive that inserts a 8-bit value into the memory image. The value is padded to fit 8 bits if necessary.

Labels:

A label acts as a placeholder for an address in memory. The address that the label references is parsed from the .pos directive that is required above the label.

Decisions:

In the process of implementing this project, there were a couple of design decisions that were fundamental to the program that were made. First, any label or stack directive should have .pos immediately before it. This is due to the fact that the position of the label or stack may be miscalculated if an Instruction is set before it in the code. Second, there should be nothing on the same line as a label or the stack. The was decided on for ease of parsing. Third, LDUR and STUR work with chunks of memory that are wordsize. This decision just made the instructions flow more effectively in my program since it was centered around 32-bits. Fourth, I did not technically implement a nop command. The reason for not doing so is that my program naturally treats non command entries as a nop.

Flaws:

The flaws in my program mostly stem from lack of experimental time that I gave myself. My first flaw is that I did not implement sufficient D type commands to make my program more efficient with working with smaller memory chunks. The second one is that I implemented flags, but I did not implement conditionals based on flags.