Xiaoshi Wang

Professor Pfaffmann

CS 203 Project 3

December 8, 2017

CPU Simulation

Problem Statement and Solution Overview

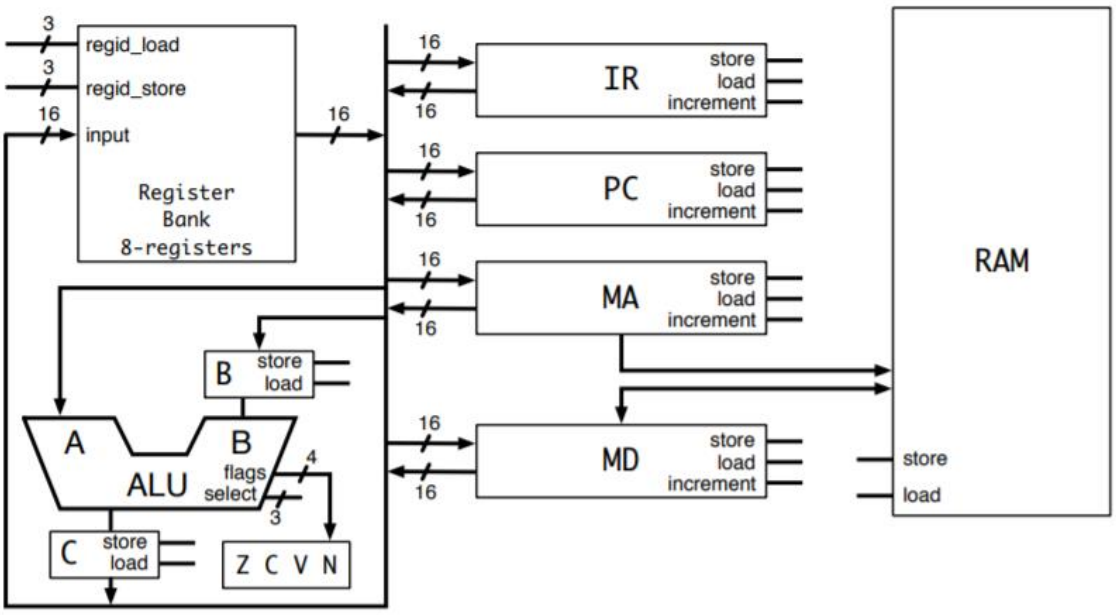
In this project, we need to construct a well-done CPU simulation in Java, which can handle basic load, store, calculation and logical operations. Notice that part of the code has been provided. Here are several important components we need to finish to develop a CPU.

1. ALU, the Arithmetic logic unit, should be added to the existed CPU. The part could achieve simple calculation and logical operations if data are given.
2. Registers B, as one input for our ALU, and register C, the one used to store results from ALU, should be added as well. Notice that both B and C should be connected to the master bus. Also, a 4-bit container for flags should be added to show the result states.
3. The ALU itself needs to connect to the master bus as a path to load the other input.
4. Notice that only Fetch, NOP and LOADI are included in our RTN, the Register Transfer Notation. Thus, we should add more in this project.
5. Since we add the ALU into our CPU, some part of our GUI should be modified.

In conclusion, our goal in this project is adding an ALU and components related to it to our CPU.

ISA design

There are two important part of our ISA: one is RAM, the other is CPU. Graph 1 show all components in our ISA and the bus that connects them together.



Graph 1

To using this simulation to implements a given code, we proceed the code line by line. The process always begins with Fetch. That is, code should first be transfer from RAM to our CPU. First, the PC would store the address we are going to work on. Thus, load the address in PC to master bus, and then load this address to MA by bus. MA is the address register for our main memory RAM. The next step is that increment the address in PC so that we can use the value in PC directly. And then load the corresponding value to MD. Finally, pass the value in MD to IR trough bus. Then, we have already load the code into IR, one component of our CPU, and the counter in PC is ready for the next fetch.

Then, we can proceed the code in IR. First, determine which instruction it refers to. And set our counter to be the corresponding RTN code. Then, proceed the instruction step by step. Notice that for each RTN code, it includes what the CPU should do, and which RTN code it should proceed next.

For telling what the CPU should do, RTN code may ask CPU to transfer data between registers and main memory. We usually pass data from register to MA and MD through bus, and then use either load or store on MA and MD to modify or obtain values from RAM. If RTN code ask CPU to do some operations, we need to take our ALU into picture. There are usually two inputs. We first load an input through bus to B to temporarily store this piece of data. Then, load the second input into bus, and proceed the needed operations inside the ALU. Before we result a output, we need to check the states of our results and modify it into an valid form if necessary. (The invalid form usually means our number is out of bound.) Finally, put the result to C, and transfer C anywhere required by bus.

Also, the RTN code would tell us which RTN code it should proceed next. Notice that most instruction cannot be finished in one step. Thus, we need multiply of steps to finish one code, and fetch the next line. Therefore, we have three types of commands: go to the next consecutive RTN code (which is the next sub step of the instruction), go to fetch and proceed the current code stored in IR. Then, after we finish the execution in our code, we know what we do next.

ISA Documentation

We include 16 instructions in our RTN code. And Table 1 shows those instructions. This table gives their name, opcode (notice that fetch has no opcode), the number of sub steps, its IR representation and meaning.

Instruction	Opcode	Step#	IR Representation	Meaning
Fetch	-	3	-	Load the code to CPU
NOP	0	1	|5 1|1 0| | op | - |	Go to next line
LOADI	1	1	|5 1|1 8|7 0| | op | dest | immed |	Dest = immed
LDUR	2	3	|5 1|1 8|7 0| | op | reg | addr |	Reg = Memory[addr]

STUR	3	3	|5 1|1 8|7 0| | op | reg | addr |	Memory[addr] = reg
ADDI	4	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 + i
ADD	5	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | x3 |	X1 = x2 +x3
SUBI	6	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 - i
SUB	7	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | x3 |	X1 = x2 +x3
ANDI	8	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 & i
AND	9	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | x3 |	X1 = x2 & x3
ORRI	A	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 | i
ORR	B	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | x3 |	X1 = x2 |x3
LSL	C	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 << i
LSR	D	3	|5 1|1 8|7 4|3 0| | op | x1 | x2 | i |	X1 = x2 >> i
B	E	1	|5 1|1 0| | op | addr |	Go to addr

Table 1

Each types of instructions are detailed explained in the following paragraphs. Each instruction includes two parts: execute and advance.

- Fetch

  The execute and advance operations of Fetch we have already talked about in the *ISA design* part. Fetch would load code to our CPU.

- NOP

  If a NOP code is read, the CPU would do nothing, and then go to the next line. Thus, NOP has only one step. The execute is empty for NOP, and the advance would go to the start of the fetch.

- LOADI

  LOADI is an example instruction provided in the pervious code. The instruction would set register *dest* to be *immed*. (Those symbols are defined from Table 1 above.) There is only one

step: we first load this constant to the master bus, then load this number to the given register from bus. And then we have finished.

- LDUR/STUR

  The LDUR and STUR method are similar. One would load data from main memory to some register, and the other store a value in some register to main memory.

  There are three RTNs for LDUR. LDUR0 would first store address to bus, and then load those address to MA. And proceed the next RTN. Then, CPU would let RAM to transfer the data at the address stored in MA to MD. The last step is transfer the value in MD to bus, and load this value to the corresponding register in bank.

  The instruction STUR is similar with that for LDUR but in the reverse order. It would first load the value from register to MD. Then, get the targeted address from IR. At the end, modify the main memory at address in MA, as the value in MD.

- ADDI/ADD/SUBI/SUB/ANDI/AND/ORRI/ORR/LSL/LSR

  Those ten instructions all involve the use of ALU. Notice that there are operations *add, sub, and*, *orr*, *lsl* and *lsr* in ALU. And the implement of those operations and conditions for flags are detailed discussed in the next part *Correction and Changes.*

  ADD, SUB, AND and ORR are instructions working with two values from registers. For those instructions, we first store the value from the *x2* to bus, and put this value to B. Since we are going to use ALU, initialize all four flags at this step. In the next RTN, load the value from *x3* to the bus, and do the corresponding operation in ALU. In the last RTN, we store the value in C to the master bus. And load this value the *x1*.

  The other 6 instructions have the same process except in the second step, the bus loads number from IR instead of from *x3*.

- B

Notice that the PC in CPU takes care of which line of the main memory we are working on. Thus, we only need to change the value in PC to jump to the target address. There is only on RTN in B. In this RTN, we first get the target address from IR and load into bus. Then, store this value to PC.

Corrections and Changes

The most significant change in the code is that a ALU class is added. This ALU class could handle basic calculation and logical operations in CPU. The ALU is connected with four other parts. Registers B, C, master bus, and flags are connected. For calculation, we add two methods: one for addition and the other for subtraction. And we have and, or, left shift and right shift for logic operations. In all those operations, we regard the value in B as the first input, the value from the master bus as the second input. And put the output in C.

I create a four-entry integer array to achieve the flags for ALU. Those four entries represent N, Z, C and V respectively. As we explained before, we use if statement to set the flags if we meet the corresponding situations. Also, since we set our numbers to be unsigned in the whole CPU, we need to check whether the result is valid to store back in some registers or not. Notice that we will meet invalid cases in add, sub and left shift only. Thus, in these three methods, we set an if statement to check the validness by flags we just set.

N is flagged if the result is negative, and Z is flagged if it is zero. For C, which stands for Carry, it would be flagged if addition is too large, subtraction is positive, or bit shift happens. And V, the overflow, is flagged if the number is out of bound. Since all our numbers are unsigned, if the word size is given the lower bound is 0, and (2^(word size/2)-1) is the upper bound.

One error I found in the previous code is the *setMemoryWord(int address, String value)* in the RAM class. This method used to set the memory of a given address to be given a hexadecimal word. However, the converted hexadecimal word may have a shorter length in store operations, because our input is a hexadecimal number instead of a word. Thus, I change this method to load a value by binary string. Thus, the length would be constant from now on.

Example Programs

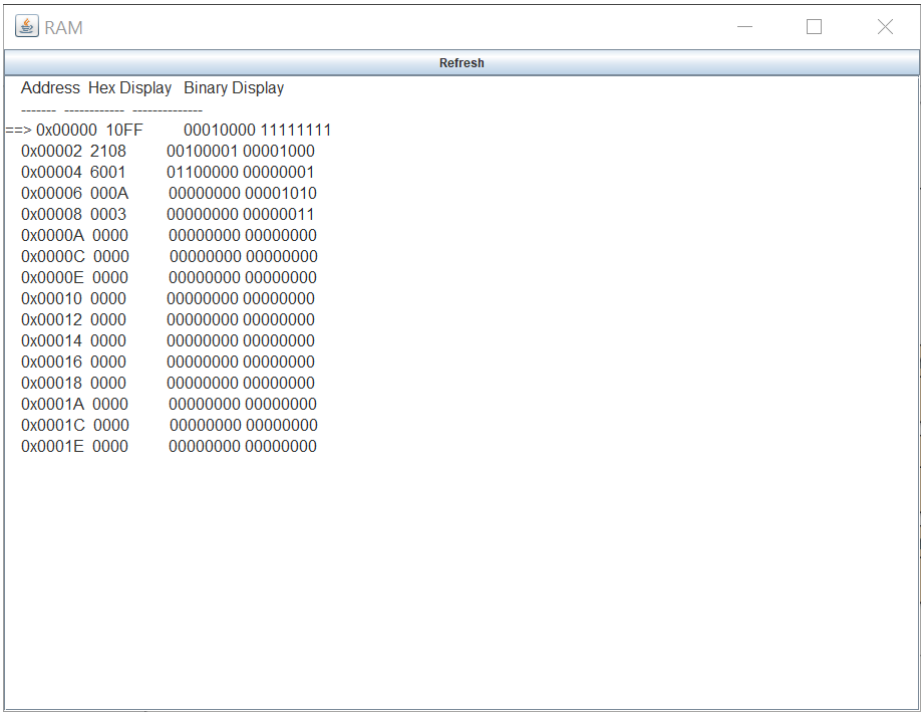
Example 1

This is an example showing how calculations and load work from example_1.as. And table 2 is its code.

Code	Translated Code
10FF	LOADI x0, 0xFF
2108	LDUR x1, 0x8
6001	SUBI x0, x0, 0x1
000A	NOP
0003	NOP

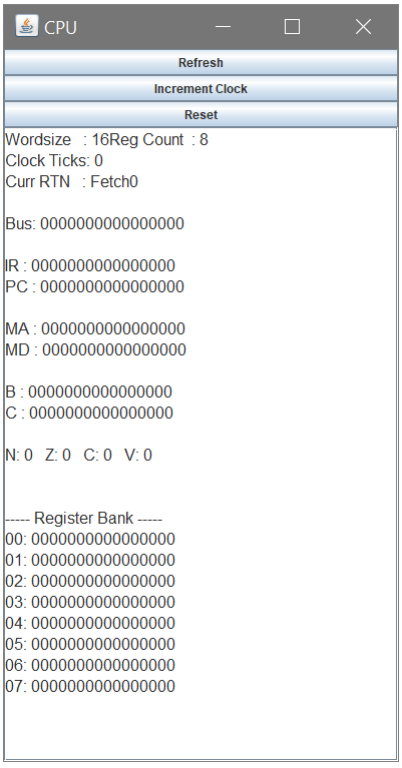
Table 2

Graph 2 and graph 3 show the initial RAM and CPU windows. Notice that RAM is proceeded from the first line, and all registers in CPU is 0.

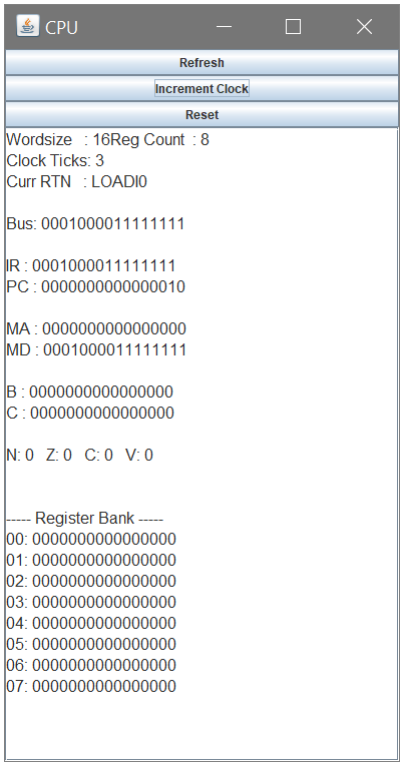


Graph 2

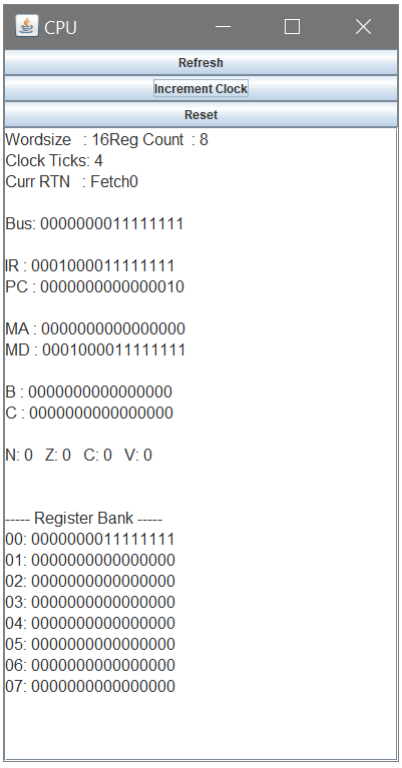
In Graph 4, we just finish the fetch for the first code. And the code is loaded in IR and the PC has been incremented. And Graph 5 shows what happens in register after that. 0Xff is loaded into register 0.



Graph 3

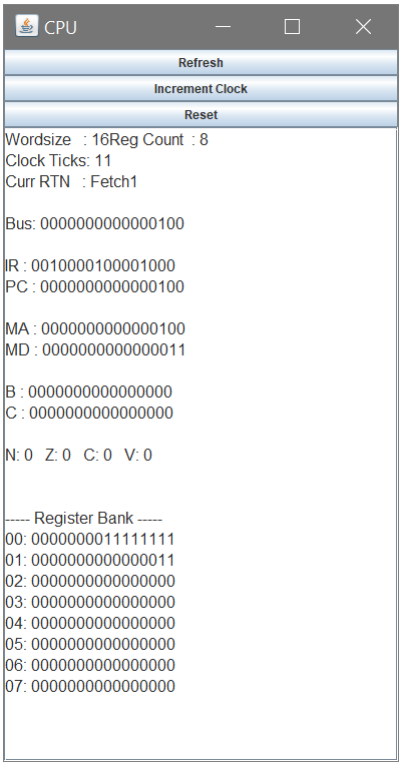


Graph 4

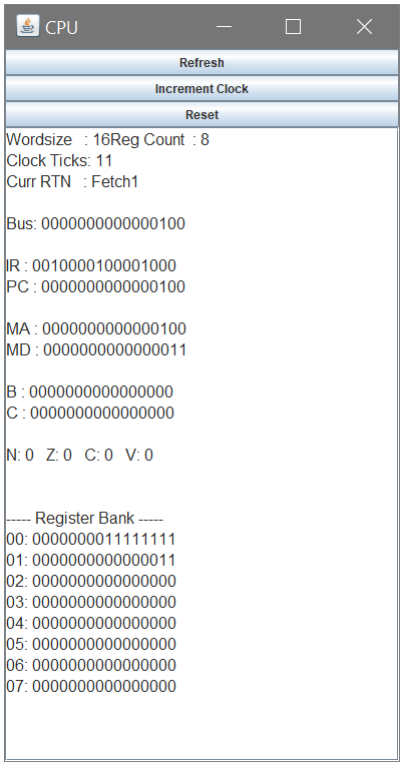


Graph 5

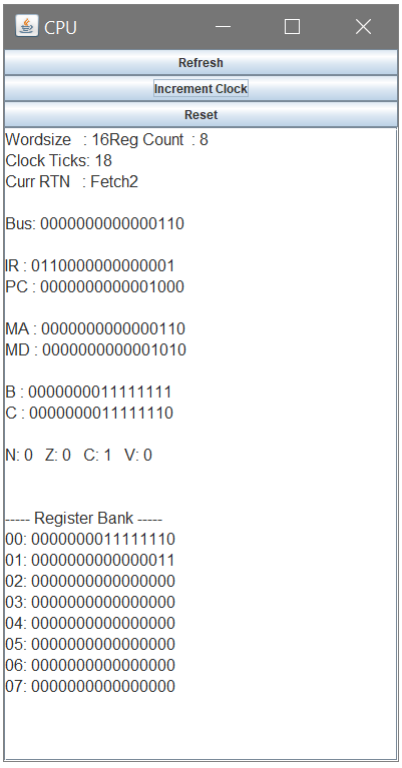
Then, graph 6 is the CPU after proceeding the LDUR method and 3 is loaded from the main memory. An CPU in Graph 7 and 8 do the SUBI operation. Since our result is positive, C, the carry, is flagged.



Graph 6

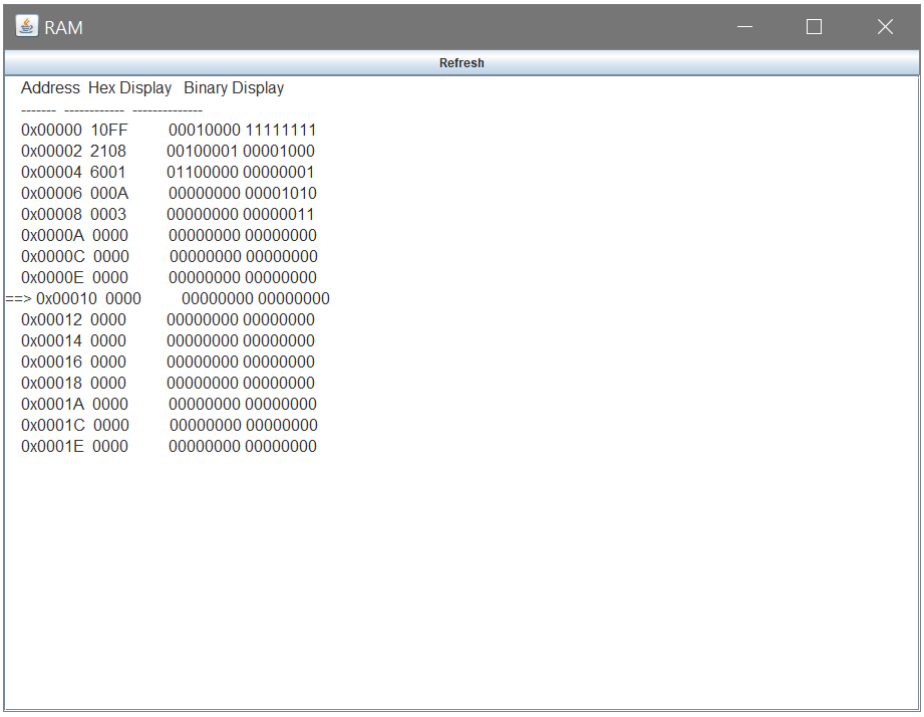


Graph 7



Graph 8

Graph 9 shows after line 4, since the opcode the CPU gets is always NOP, the CPU would keep proceed the next line.



Graph 9

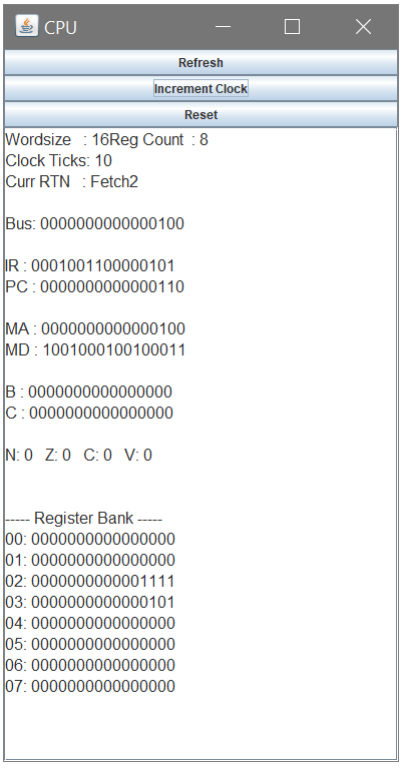
Example 2

The table Table 3 below shows the second example from *example_2.as*. This example includes some logic operations and the store method.

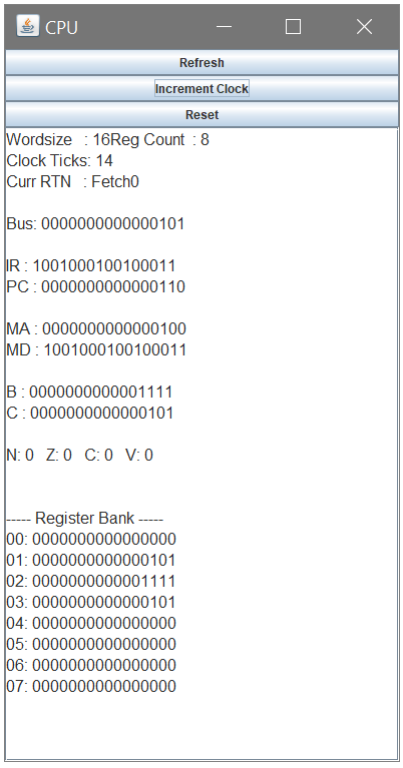
Code	Translated Code
120F	LOADI 2, 0x0F
1305	LOADI 3, 0x5
9123	AND x1, x2, x3
D202	LSL x2, x2, 0x2
320A	STUR, x2, 0xA

Table 3

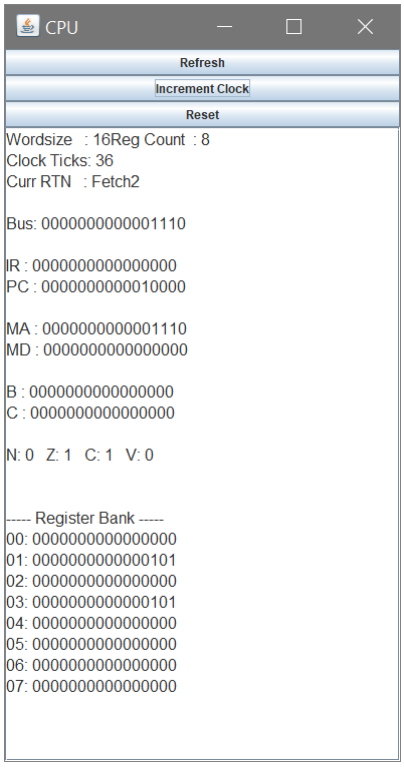
The 0xf and 0x5 are loaded into register 2 and 3. And the resulted CPU are shown in Graph 10. Then, in Graph 11 the CPU add the x2 and x3 and store it into x1. And the LSL happens in Graph 12. Notice that the flags show that the result is zero and with carry.



Graph 10



Graph 11



Graph 12

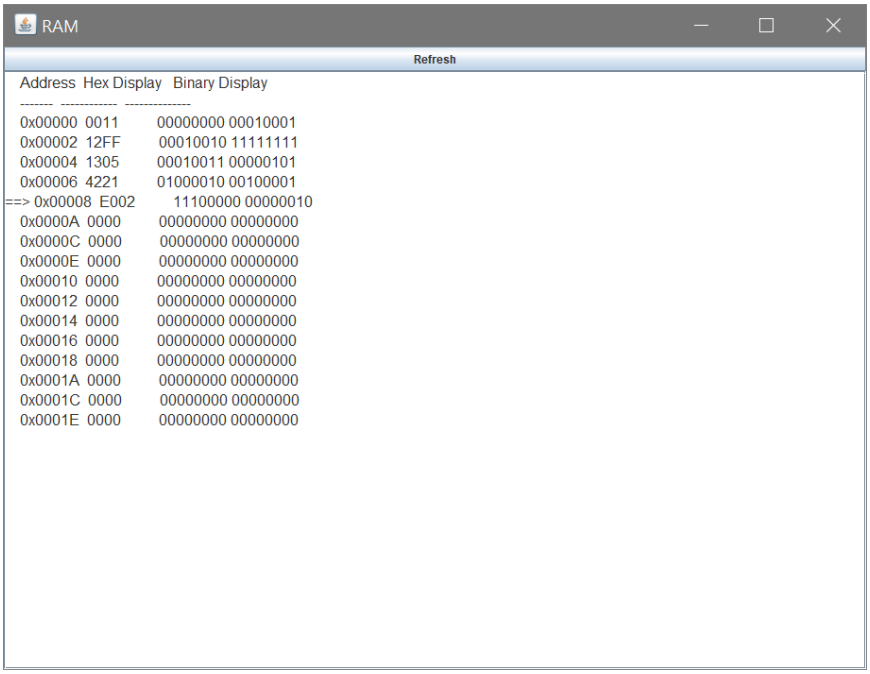
Example 3

The last example shows how B, the branch, works in our CPU simulation. Table 4 shows the code and the translated code in *example_3.as*.

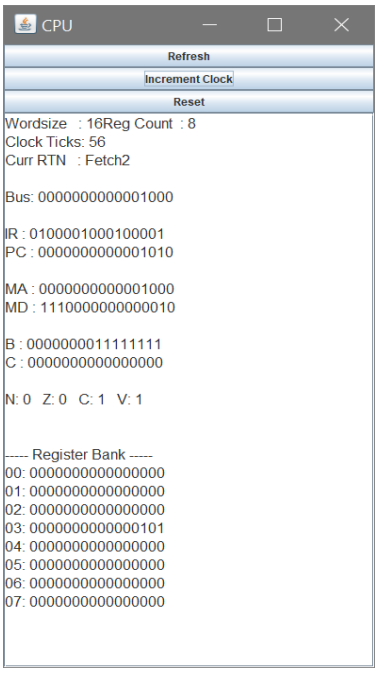
Code	Translated Code
0011	NOP
12FF	LOADI 2, 0xFF
1305	LOADI 3, 0x5
4221	ADDI x2, x2, 0x1
E002	B. 0x2

Table 4

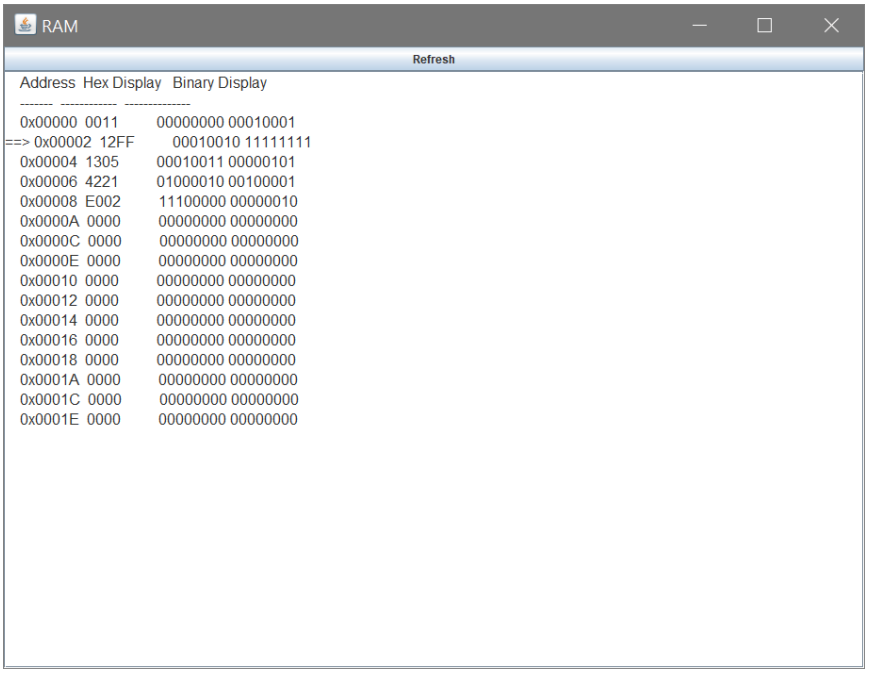
Graph 13 and Graph 14 show RAM and CPU before the branch. And Graph 15 and 16 show them after the branch. Notice that the arrow in RAM points to the address 0x2 in Graph15, which implies our branch does work well.



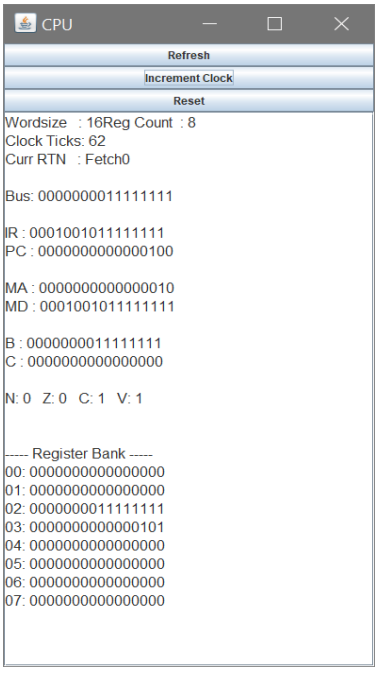
Graph 13



Graph 14



Graph 15



Graph 16

User Manual

To use this CPU simulation provided in the file, the user should first modify the document “test_file.as” by typing the wanted code. (Notice that the three examples should in this report are also included in the submitted file. But their names are “example1.as”, “example2.as” and “example3.as”. To verify these examples, just copy the code in the file, and paste it in to “test_file.as”.) Notice that the code is typed in hexadecimal digits. The decode rules, or their representation is shown in table 1 in the report.

Then, type “make run” in your terminal to compiling all java files. And type “make build” to run the simulation. Two windows would pop-up. And One would show states of our RAM, and the other would show the states of all components in CPU. (Notice that both commands are typed without the quotation marks.)