Ain Shams University

Faculty of Engineering

Computer Engineering and Software Systems Program

CSE116: Computer Architecture

Mips Project

Submitted By:

Amr Mohamed ElSayed Ali Badawi	16P3034
Ahmed Hesham Mohamed Al-Saey	16P6000
Ahmed Mamdouh Mohammed Ahmed Mounir	16P6020
Ahmed Mohamed Ahmed Elqaffas	16P6012
Sherif Ashraf Ahmed Morsy	16P9033

Submitted to:

Dr. Cherif Ramzi Salama

1.0 IMPLEMENTATION

We've implemented this project by dividing the program into four main parts apart from the bonus features we added.

1.1 Instructions handling

We take the instructions from the user as a 32 bits binary numbers. We then divide these 32 bits to: op code, rs, rt, rd, shift, function code, jump address and 16 bits constant and use what we need from them based on the instruction. After taking all the instructions and storing them in an array, we execute them one by one by comparing the pc (program counter) to a unique instruction counter assigned to each instruction to know which instruction to execute next.

1.2 Registers and Memory

1.2.1 Registers

Registers are implemented using array of size 32, each location represent a register, registers in the array are ordered the same way as mips.

1.2.2 Memory

We used java array lists to implement the memory as it's flexible and doesn't require fixed size like arrays. Each array list element represents a byte. Each element (byte) has a location and value. When we store a word, it's stored in four consecutive places in the list.

1.3 Wires

The execution of each instruction results in changing the values of the wires associated with this instruction. We assume that wires carry over its values from preceding clock cycle so wires having a “Don’t care” are simply not changed and take the values of the preceding instruction. Wires values for each instruction are saved in an array list.

1.4 Outputting Values for Each Clock Cycle

We output values of registers, wires and memory each clock cycle by using extra (Container) array and two array lists respectively. Registers container saves registers arrays for each clock cycle. The same concept goes for memory and wires array lists. We therefore became able to track the values of each clock cycle using a key that represent the clock cycle number.

1.5 Bonus Features

We built the application as a GUI application

Input is taken from user using text areas. Output is displayed to the user using labels.

2.0 DATAPATH

We used the Datapath in lecture 7 slide 10 also shown in the figure below:

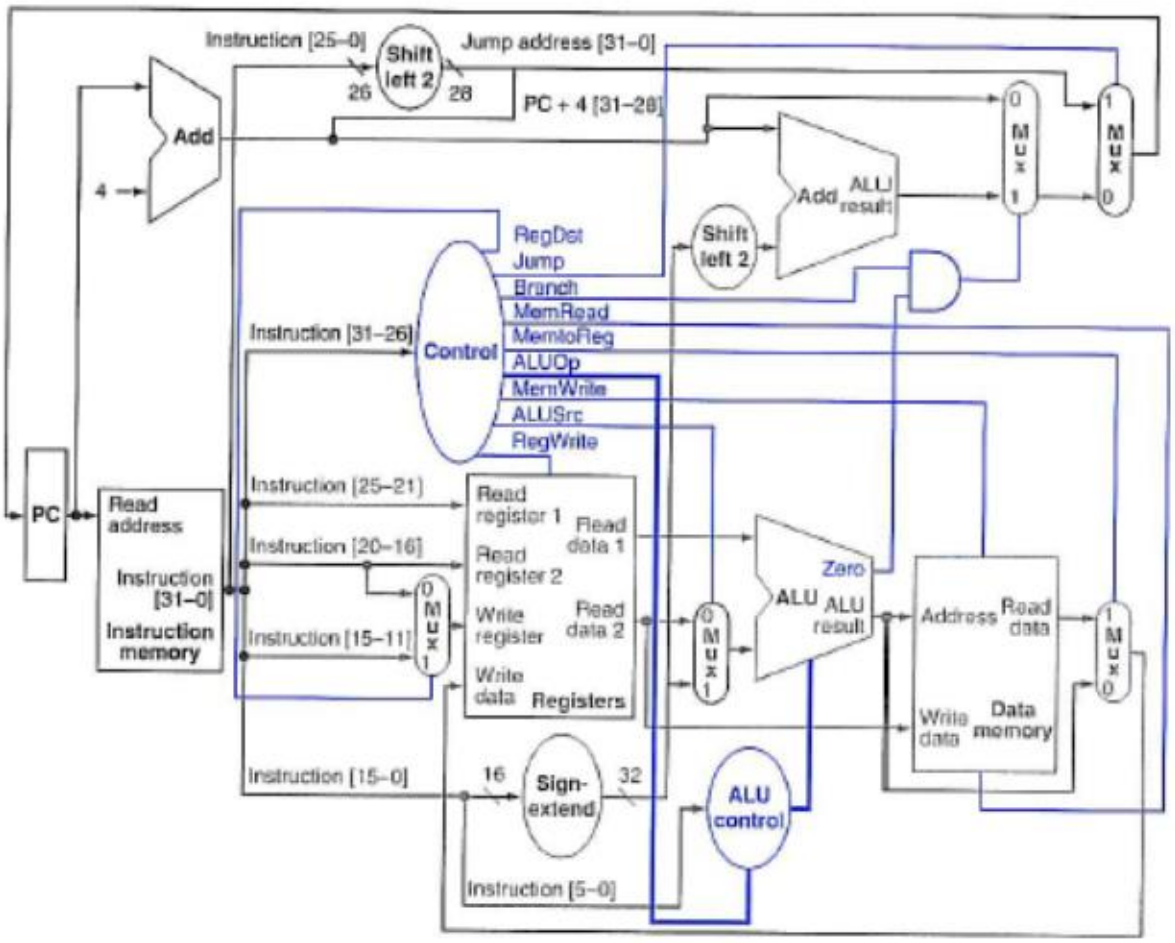

Figure 1: Main Datapath

We then made some modifications to support the rest of the unsupported instructions.

2.1 JAL

We replaced the RegDst mux with another that has 3 options (0,1,2) where 2 means that we will write in $ra register, we therefore replaced the RegDst control signal to be 2 bits instead of 1.

We also added a JAL mux that takes the pc+4 and MemtoReg mux output as inputs. The output is connected to the registers to be written. This mux is controlled by a new JAL signal from the control unit.

2.2 JR

We added a JR control signal from ALU control (1 if instruction is JR and 0 otherwise) which is inverted and anded with RegWrite signal and the output is connected to the register so that we don't write in the register if instruction is JR.

We also added a mux that takes its inputs from pc+4 wire and read data 1 wire. The mux is controlled by ALU control which selects "1" for register output or "0" for pc+4

2.3 SLT and SLTI

We added a SLT signal in ALU control which is "1" if instruction is SLT and "0" in case we want to write the regular ALU output and don't care if we don't write back to registers. This signal controls a mux which chooses between the regular ALU output or the output of a negative flag (which we assumed it exists) to write back in the register.

2.4 LB and LBU

We replace the least significant two bits from the ALU output with zeroes and then read the word in that address normally. We then divide the memory output to 4 different 8 bits wires (divide word to 4 bytes) which enter a mux that chooses which byte will be loaded based on the 2 bits that was replaced from the ALU output. The loaded byte is then appended to 24 bits of zeroes (in case it was LBU) or 24 bits of zeroes or ones based on the sign of the loaded byte (in case it was LB).

Then a mux chooses whether to load these 32 bits or load the regular word outputted from the memory based on a control signal that we added which is 1 in case of LB and LBU or 0 in case of lw and don't care otherwise.

2.5 SB

We assumed there is a store byte signal in the data memory which is 1 only in case of storing byte. We added a control signal to the control unit which controls this memory signal.

3.0 TRUTH TABLE AND DIAGRAM

Below are the truth table and logic diagram for our project. We assumed we only need to take care of op-codes of the instructions we support so we set the other combinations of op-code to don't care.

3.1 Truth Table

In the table below, we used the 6 bits op code and the 1 bit of jr ALU control signal as inputs to the control unit.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
op1	op2	op3	op4	op5	op6	jr ALU control	ALUop1	ALUop2	MemToReg	RegDst1	RegDst2	RegWrite	Branch	Jump	MemWrite	MemRead	StoreByte	LoadByte	JAL
0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	x	0
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	x	0
1	0	0	0	1	1	0	0	0	1	0	0	1	0	0	0	1	0	0	x
1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	x
1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	x
1	0	1	0	1	1	0	0	0	X	X	X	0	0	0	1	0	0	x	x
1	0	1	0	0	0	0	0	0	X	X	X	0	0	0	1	0	1	x	x
0	0	0	1	0	0	0	0	1	X	X	X	0	1	0	0	0	0	x	x
0	0	1	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	x	0
0	0	0	0	1	0	0	x	x	X	X	X	0	X	1	0	0	0	x	x
0	0	0	0	1	1	0	x	x	X	1	0	1	X	1	0	0	0	x	1
0	0	0	0	0	0	1	1	0	0	X	X	0	X	1	0	0	0	x	x
							A'C'D'	A'E + A'D	A	A'F	A'C'E'	A'F + A'C + AC' + A'D'E'G'	A'D	G + A'C'E	AC	AC'	ACE'	E'	F

Figure 2: Control unit truth table

3.2 Logical Diagram

The following is the logical diagram for the control unit

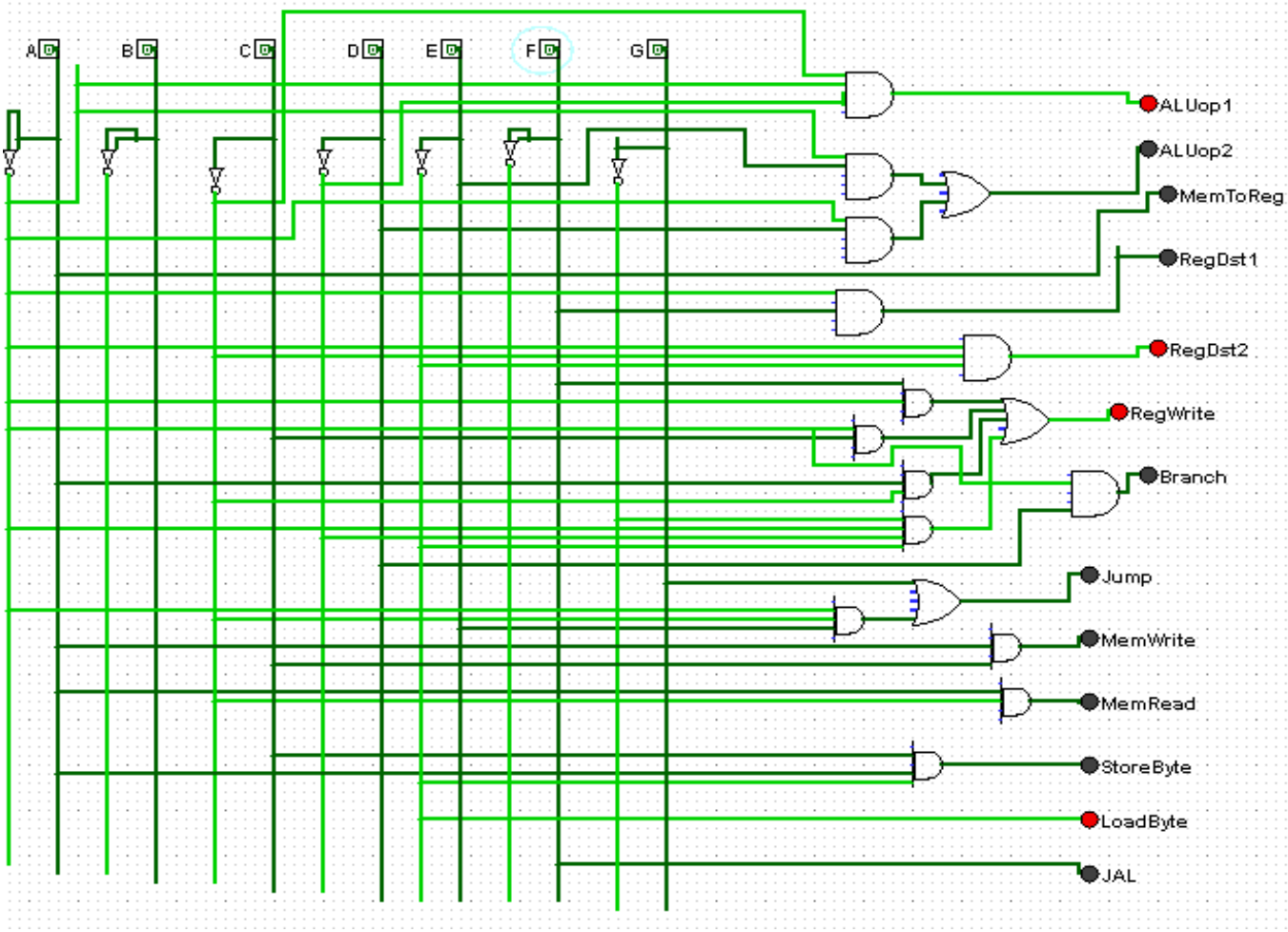

Figure 3: Control unit logic diagram

Both the truth table and the logic diagram are included in the project folder.

4.0 ASSUMPTIONS

We assumed the following

- We assume that wires carry over its values from preceding clock cycle so wires having a “Don’t care” are simply not changed.
- We assume that there is a negative flag in the ALU that is 1 when the ALU result is negative and 0 otherwise.
- We assume that the memory has a store byte signal that is 1 when the memory has to store a byte instead of a word and 0 otherwise.

5.0 USER GUIDE

Running the program, an input screen is presented to the user

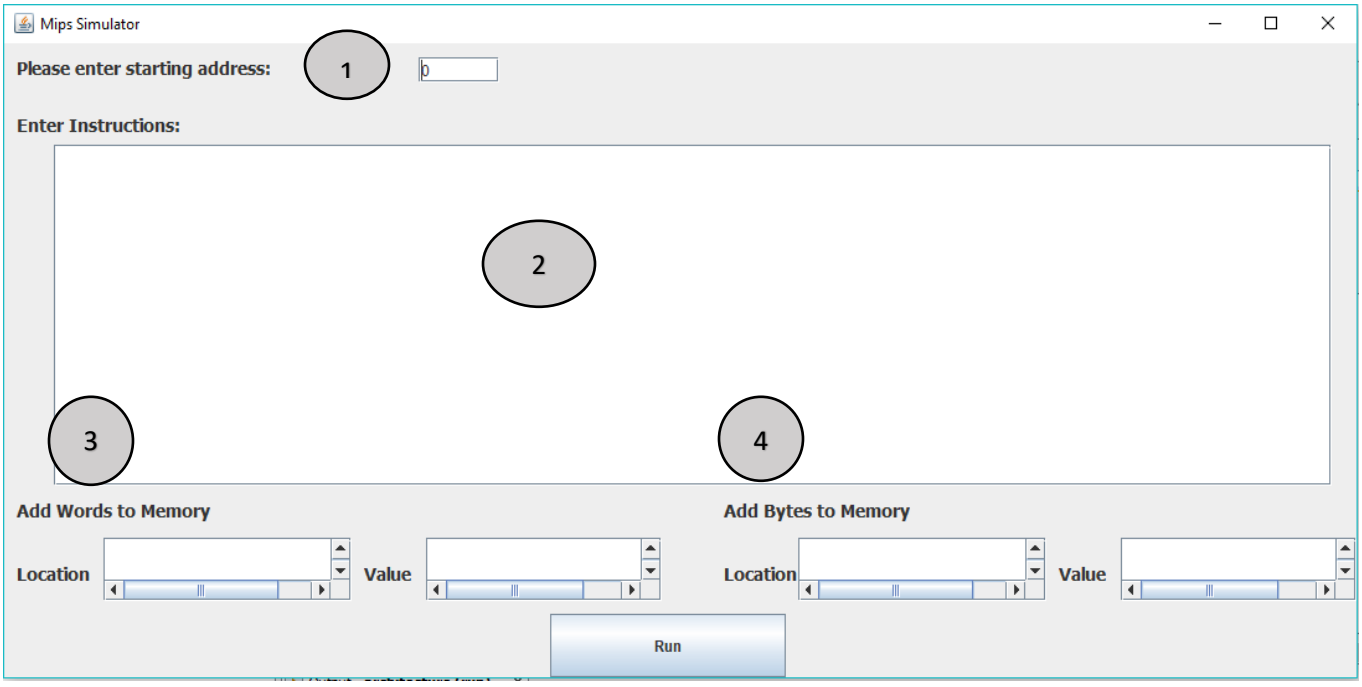

Figure 4: Input Screen

5.1 Starting Address

The user can enter the starting address of the instructions “initial pc” in the text box which must be divisible by 4, default value for it if user doesn’t enter a value is 0.

5.2 Instructions Area

This is the area where the user can enter his 32 bits instructions. Only one instruction can be present in a line. Instructions mustn’t be preceded with spaces.

5.3 Add Word and Add Byte

The first two text areas under the instructions area are for adding a word to the memory. The first text area is for entering the location of the word in the memory (This address should be divisible by four and in decimal). The other one is for the value the user wants to add also in decimal.

The two text areas to the right are for adding a byte to the memory. Like adding a word, the inputs should be in decimal format. However, the address doesn’t have to be divisible by four.

Every line should only contain one address in case of “location” or one value in case of “value”. Data entered shouldn’t be precede with spaces.

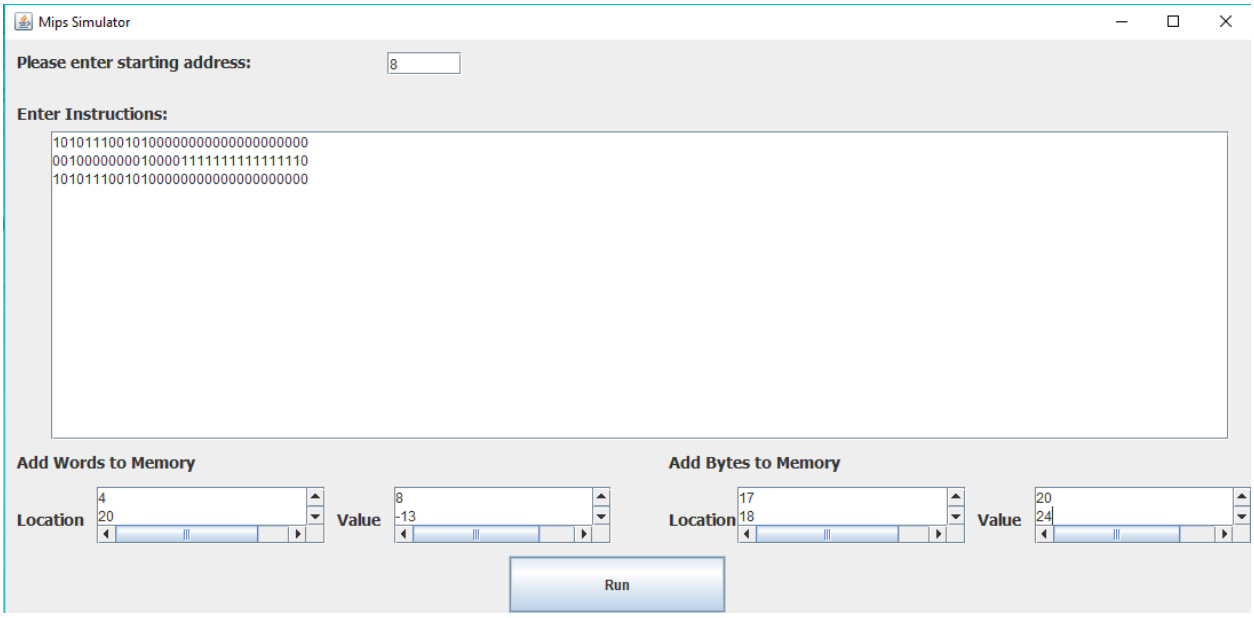

Figure 5: An example for user input

The input in the figure is a basic testing where register $s0 is saved in the memory in address 0, decremented by 2 then stored again in memory.

- sw $s0, 0($s2)
- addi $s0, $s0, -2
- sw $s0, 0($s2)

After entering the required inputs and clicking “Run”. An output screen is displayed. The initial screen displays the values of wires, registers and memory after the first clock cycle.

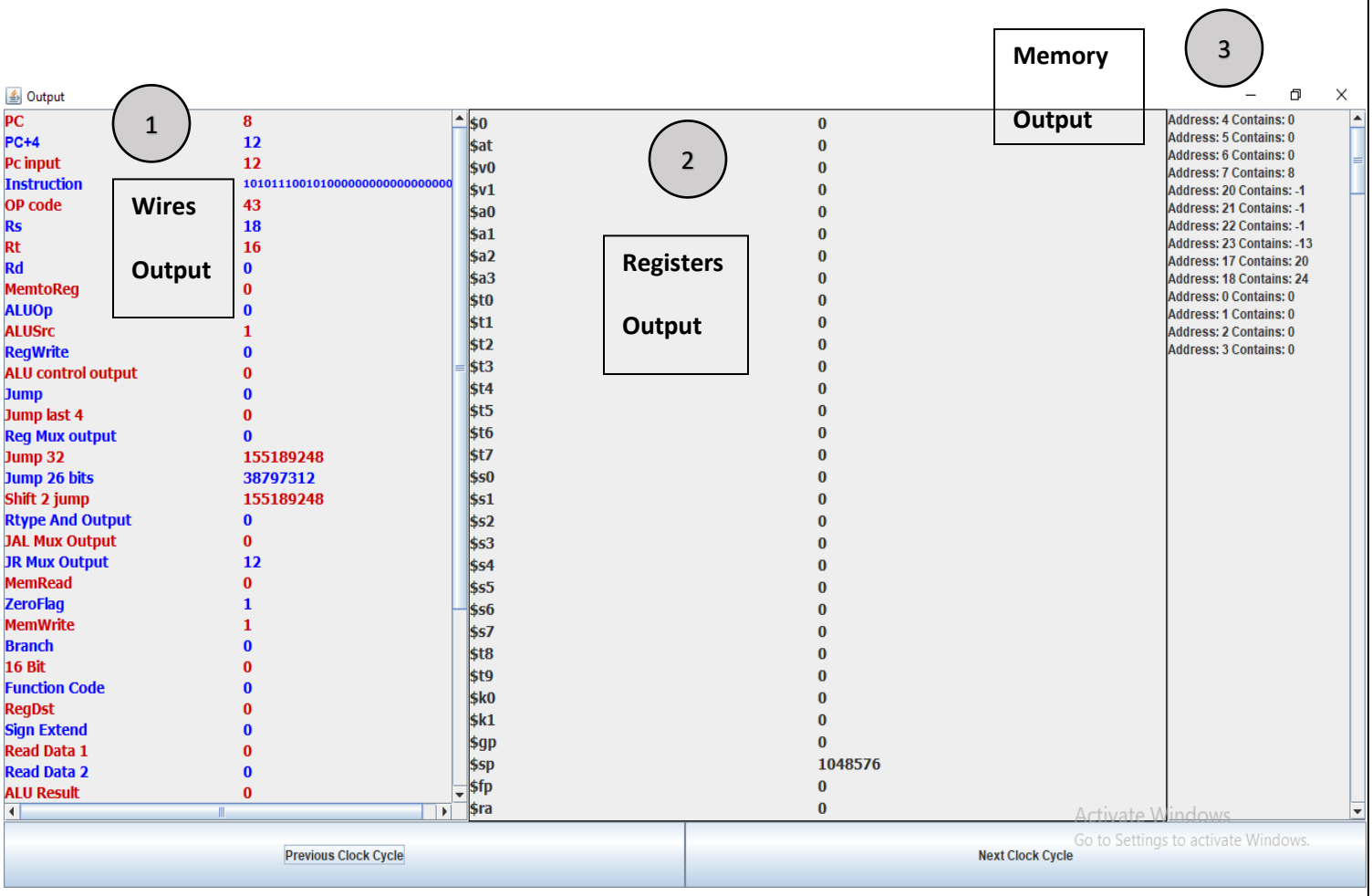

Figure 6: Output for first clock cycle

5.4 Wires Output

At the left in the output screen, the user can see the values of the wires of this clock cycle represented in decimal (except instructions wire which made sense to be represented in binary instead).

5.5 Registers Output

In the middle of the page, the registers values are displayed with the same ordering as in lectures and mars. The first instruction doesn't overwrite registers (store word) so all the registers are still zeroes (except stack pointer).

5.6 Memory Output

At the right, the user can find all the memory locations and currently used and the stored values. The memory is displayed as a byte by byte not by a word

So bytes 0 to 4 represent the first word in the memory which Is used by the instructions while bytes 4 to 7 and also 20 to 23 represent the two words added by the user. The bytes 17 and 18 represent the two bytes also added by the user.

The user can use “Next Clock Cycle” and “Previous Clock Cycle” to go through the clock cycles “instructions” executed. The figure below shows the next clock cycle

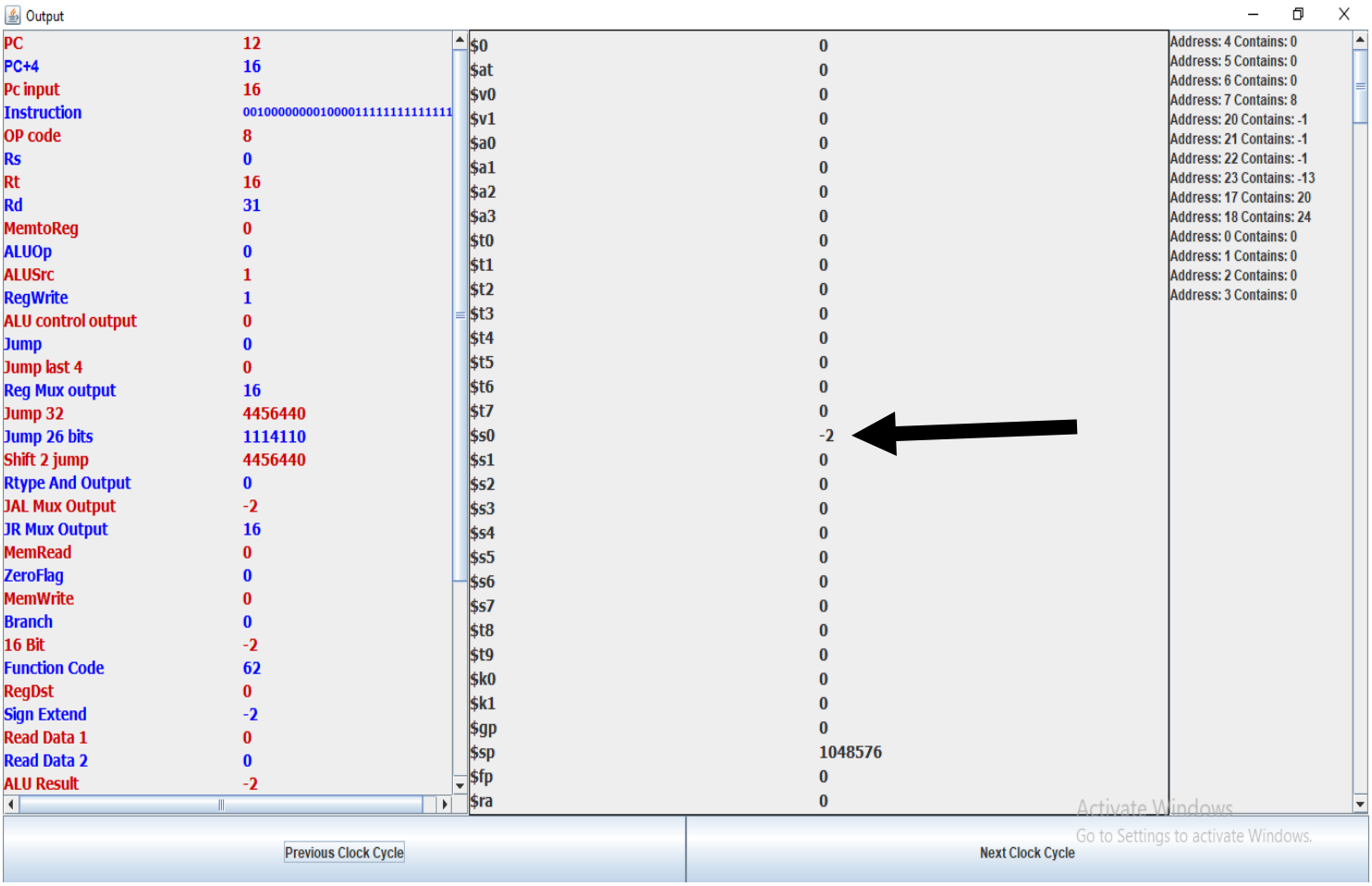

Figure 7: Second clock cycle output

The second instruction is a addi instruction, so no changes were made to the memory. Values of wires is updated and as can be seen in the figure, register $S0 is decremented by 2.

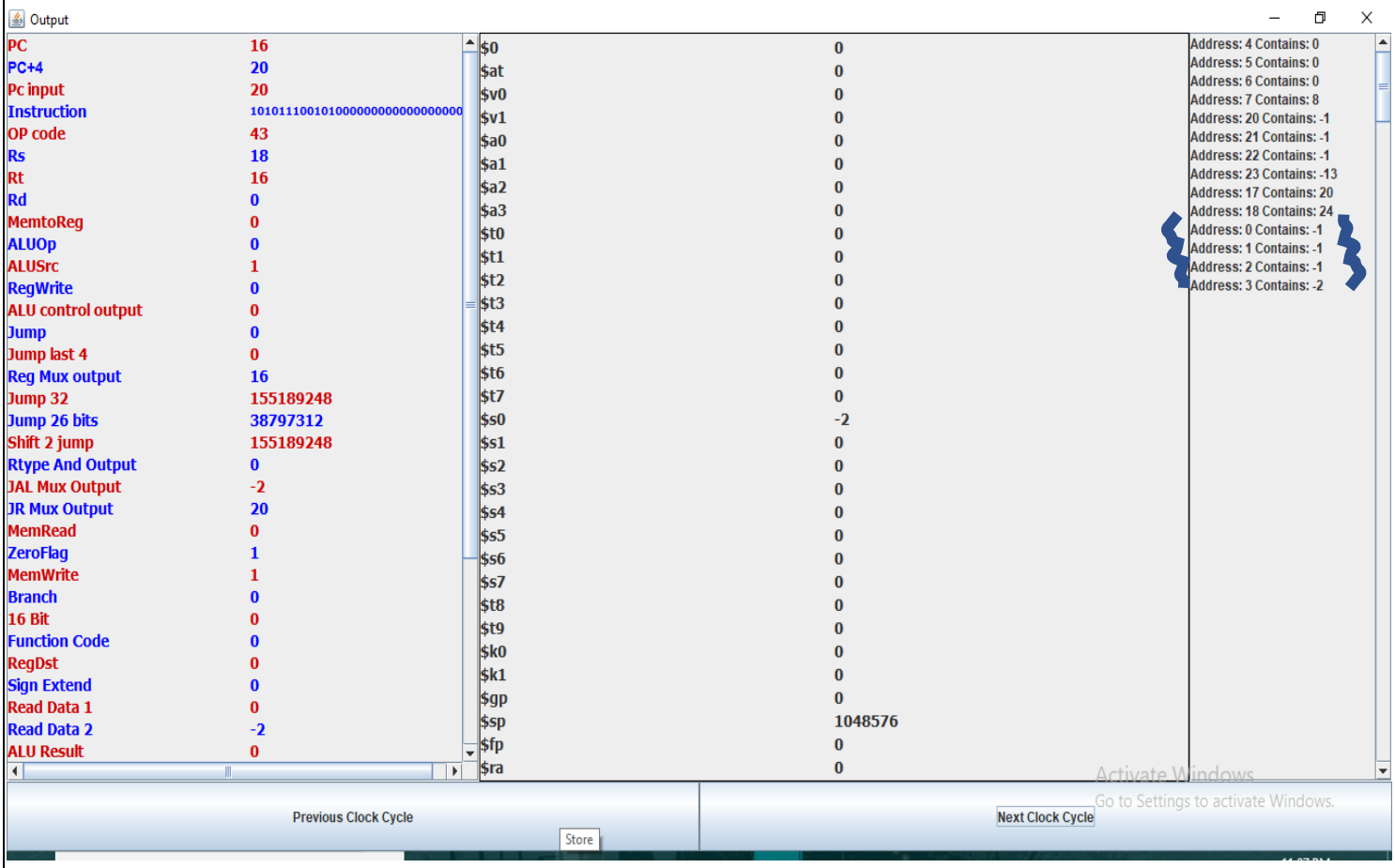

Figure 8: Third clock cycle output

The third clock cycle is the same as the first one (store word) it updates the value of the first 4 bytes in the memory with the new value of $s0, also wires values is updated. No changes are made to the registers.

6.0 TEST CASES

We've included five different text files containing programs we simulated using this program. The test cases assume that pc starts with zero. Here is the list of the programs:

- "Bigger Number" which test function calls by returning the bigger of 2 numbers.
- "Summation of Positive Number" which includes a loop to get the summation of a positive number.
- " Initialize array" which tests store and load word and initialize a 10 elements array of values from 0 to 9 in the memory.
- "SLTI,NOR" which tests slti and nor instructions.
- "Load and Store Byte" which stores a byte then calls a function that loads the stored byte twice (once as a signed number and the second as unsigned).

These programs are also included in the appendix.

7.0 WORK SPLITTING

The Work was split between us as follows

- Taking instructions as 32 bits binary numbers, splitting them to op code, rs, rt, rd,….etc. setting pc and initial memory values and the input frame was made by Amr Mohamed.
- The identifying of instruction from op or function code and its execution was made by Sherif Ashraf.
- The setting of wires values in each instruction was made by Ahmed Mamdouh
- The operation of memory and registers and some facilitating binary functions were made by Ahmed Hesham.
- The Outputting of registers, wires and memory each clock cycle beside report was made by Ahmed Mohamed.

8.0 EXPERIENCE

This project helped us gain a better knowledge of how instructions that are not supported in the Datapath by default are implemented.

APPENDIX

Here are the 5 programs mentioned in section 6.0

“Bigger number”:

C++:

```
int bn (int a, int b) {
if(a>b)
return (a);
else
return (b);
}
void main(){
int a=10; // We can change a and b  values for testing
int b=-20;
 bn(a,b);
}
```

Mips:

```
j main
bn: slt $t0, $a1, $a0
beq $t0, $zero, L1
add $v0, $0, $a0
j L2
L1: add $v0, $0, $a1
L2: jr $ra
main:addi $a0, $0, -20
addi $a1, $0 , 10
jal bn
```

00001000000000000000000000000111
00000000101001000100000000101010
00010001000000000000000000000010
00000000000001000001000000100000
00001000000000000000000000000110
00000000000001010001000000100000
00000011111000000000000000001000
00100000000001001111111111101100
00100000000001010000000000001010
00001100000000000000000000000001

Summation of positive number:

MIPS:

```
addi $s0 $s0 20
addi $t1 $s0 -1
add $t0 $0 $s0
L:beq $t0 $0 out
addi $t0 $t0 -1
add $s0 $s0 $t1
addi $t1 $t1 -1
j L
out: add $v0 $0 $s0
```

00100010000100000000000000010100
00100010000010011111111111111111
00000000000100000100000000100000
00010001000000000000000000000100
00100001000010001111111111111111
00000010000010011000000000100000
00100001001010011111111111111111
00001000000000000000000000000011
00000000000100000001000000100000

c++:

```
int sum(){
int a=20; // we can change this for testing
int b=a-1
for(int i=a;i>0;i--){
 a+=b;
 b--;

  }
 return(a);
}
```

Initializing array:

c++:

```
void in(int a[], int size) {
for(int i=0;i<size;i++)
  a[i]=i;
}
void main(){
int a[10];
 in(a,10);
}

// assume i in $s0
```

MIPS:

```
j main
in: addi $sp, $sp, -4
sw $s0, 0($sp)
addi $s0, $zero, 0
L1: slt $t0, $s0, $a1
beq $t0, $zero, Exit
sll $t1, $s0, 2
add $t1, $a0, $t1
sw $s0, 0($t1)
addi $s0, $s0, 1
j L1
Exit: lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
main:addi $a0, $zero, 2000
addi $a1, $zero, 10
jal in
```

00001000000000000000000000001110
00100011101111011111111111111100
10101111101100000000000000000000
00100000000100000000000000000000
00000010000001010100000000101010
00010001000000000000000000000101
00000000000100000100100010000000
00000000100010010100100000100000
10101101001100000000000000000000
00100010000100000000000000000001
00001000000000000000000000000100
10001111101100000000000000000000
00100011101111010000000000000100
00000011111000000000000000001000
00100000000001000000011111010000
00100000000001010000000000001010
00001100000000000000000000000001

SLTI , NOR:

C++:

```
void main(){
int a=5;
int b=1;
if (b<10)
   a = ~(a+b);
}
```

MIPS:

```
addi $s0, $s0, 5
addi $s1, $s1, 1
slti $t0, $s1, 10
beq $t0, $zero, exit
nor $s0, $s0, $s1
exit:
```

```
00100010000100000000000000000101
00100010001100010000000000000001
00101010001010000000000000001010
00010001000000000000000000000001
00000010000100011000000000100111
```

Load and Store Byte:

MIPS:

```
j main
load:
lb $t0, 0($a1) # prints -1
lbu $t1 , 0($a1) # prints 255
jr $ra
main:
addi $a0 ,$zero, 255
addi $a1 , $t1 , 5
sb $a0 , 0($a1)
jal load
```

00001000000000000000000000000100
10000000101010000000000000000000
10010000101010010000000000000000
00000011111000000000000000001000
00100000000001000000000011111111
00100001001001010000000000000101
10100000101001000000000000000000
00001100000000000000000000000001