Faculty of Engineering and Technology
Department of Electronics and Computer Engineering
ENCS2110
Digital Electronics and Computer Organization Lab
Report #4

Simple Computer Simulation

Papered by:
Student Name: Yousef Injass **ID:** 1200643

Partners:
Student Name: Dana Asfour **ID:** 1211924
Student Name: Omar Hamed **ID:** 1190382

Instructor: Dr. Adnan Yahya

Assistant: Hanan Awawdeh

Section: 9

Date: 21/6/2023

Abstract

The aim of this experiment is to design the Verilog HDL control sequence for a simple computer (SIMCOMP). The SIMCOMP is a very small computer to give the students practice in the ideas of designing a simple CPU with the Verilog HDL notation.

Contents

List of Figures

List of Tables

Theory

Basic Computer Model - Von Neumann Model

Von-Neumann Model

Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture. It consisted of a Control Unit, Arithmetic, and Logical Memory Unit (ALU), Registers and Inputs/Outputs.

Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

A Von Neumann-based computer:

- Uses a single processor.
- Uses one memory for both instructions and data.
- Executes programs following the fetch-decode-execute cycle [1].

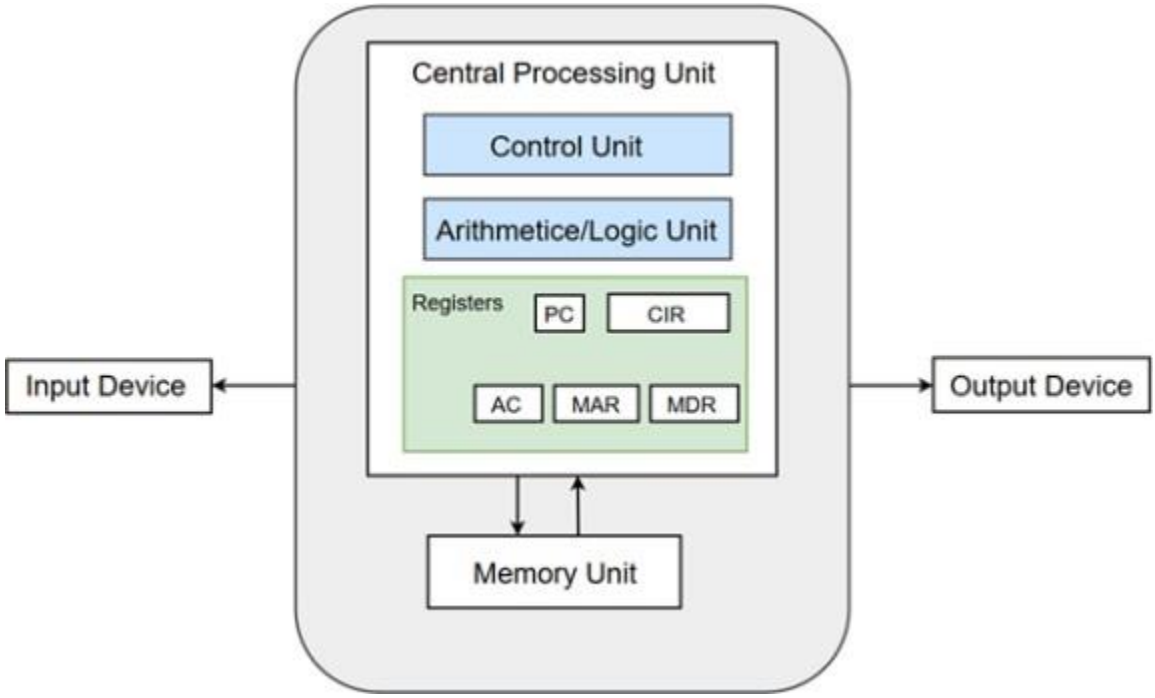

Figure 1: Basic Computer model

Components of Von-Neumann Model:

- Central Processing Unit
- Buses
- Memory Unit

Central Processing Unit:

The part of the Computer that performs the bulk of data processing operations is called the Central Processing Unit and is referred to as the CPU.

The Central Processing Unit can also be defined as an electric circuit responsible for executing the instructions of a computer program.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

The major components of CPU are Arithmetic and Logic Unit (ALU), Control Unit (CU) and a variety of registers.

Arithmetic and Logic Unit (ALU):

The Arithmetic and Logic Unit (ALU) performs the required micro-operations for executing the instructions. In simple words, ALU allows arithmetic (add, subtract, etc.) and logic (AND, OR, NOT, etc.) operations to be carried out.

Control Unit:

The Control Unit of a computer system controls the operations of components like ALU, memory and input/output devices.

The Control Unit consists of a program counter that contains the address of the instructions to be fetched and an instruction register into which instructions are fetched from memory for execution [2].

Registers:

Registers refer to high-speed storage areas in the CPU. The data processed by the CPU are fetched from the registers.

Following is the list of registers that plays a crucial role in data processing:

Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

Table 1:registers in data processing.

Buses:

Buses are the means by which information is shared between the registers in a multiple-register configuration system.

A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

Von-Neumann Architecture comprised of three major bus systems for data transfer.

Bus	Description
Address Bus	Address Bus carries the address of data (but not the data) between the processor and the memory.
Data Bus	Data Bus carries data between the processor, the memory unit and the input/output devices.
Control Bus	Control Bus carries signals/commands from the CPU.

Table 2: Busses

Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word [3].

Two major types of memories are used in computer systems:

1. RAM (Random Access Memory)
2. ROM (Read-Only Memory)

Generic CPU Instruction Cycle

A program consisting of the memory unit of the computer includes a series of instructions. The program is implemented on the computer by going through a cycle for each instruction.

In the basic computer, each instruction cycle includes the following procedures −

- It can fetch instruction from memory.
- It is used to decode the instruction.
- It can read the effective address from memory if the instruction has an indirect address.
- It can execute the instruction.

After the following four procedures are done, the control switches back to the first step and repeats the similar process for the next instruction. Therefore, the cycle continues until a **Halt** condition is met. The figure shows the phases contained in the instruction cycle.

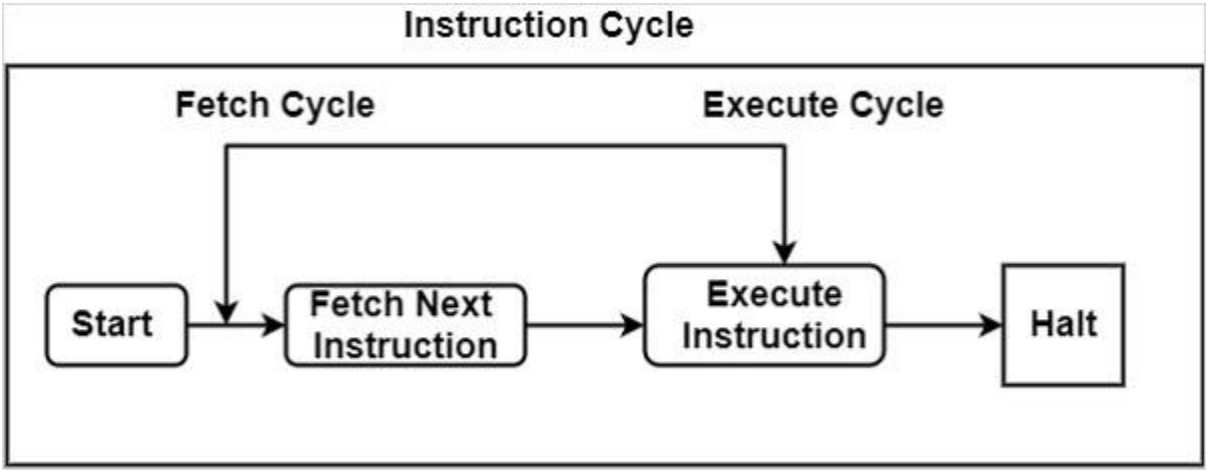

Figure 2:Instruction Cycle

As display in the figure, the halt condition appears when the device receive turned off, on the circumstance of unrecoverable errors, etc.

Fetch Cycle

The address instruction to be implemented is held at the program counter. The processor fetches the instruction from the memory that is pointed by the PC.

Next, the PC is incremented to display the address of the next instruction. This instruction is loaded onto the instruction register. The processor reads the instruction and executes the important procedures.

Execute Cycle

The data transfer for implementation takes place in two methods are as follows −

- **Processor-memory** − The data sent from the processor to memory or from memory to processor.
- **Processor-Input/Output** − The data can be transferred to or from a peripheral device by the transfer between a processor and an I/O device.

In the execute cycle, the processor implements the important operations on the information, and consistently the control calls for the modification in the sequence of data implementation. These two methods associate and complete the execute cycle.

State Diagram for Instruction Cycle

The figure provides a large aspect of the instruction cycle of a basic computer, which is in the design of a state diagram. For an instruction cycle, various states can be null, while others can be visited more than once.

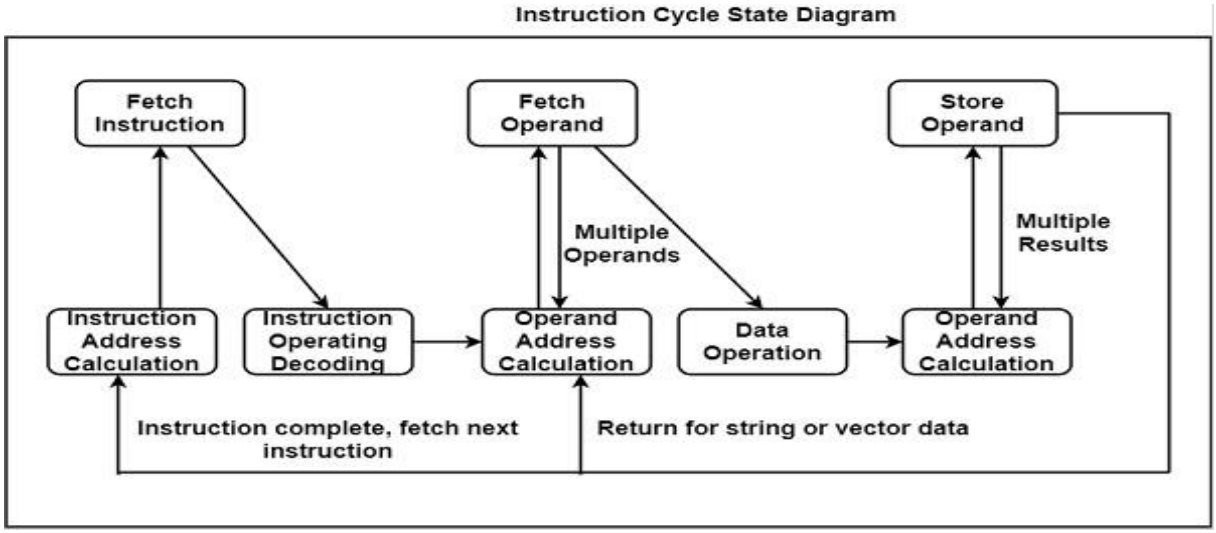

Figure 3:Instruction Cycle State Diagram

- **Instruction Address Calculation** − The address of the next instruction is computed. A permanent number is inserted to the address of the earlier instruction.
- **Instruction Fetch** − The instruction is read from its specific memory location to the processor.
- **Instruction Operation Decoding** − The instruction is interpreted and the type of operation to be implemented and the operand(s) to be used are decided.
- **Operand Address Calculation** − The address of the operand is evaluated if it has a reference to an operand in memory or is applicable through the Input/Output.
- **Operand Fetch** − The operand is read from the memory or the I/O.
- **Data Operation** − The actual operation that the instruction contains is executed.
- **Store Operands** − It can store the result acquired in the memory or transfer it to the I/O [4].

Addressing Modes:

The term addressing modes refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand [5].

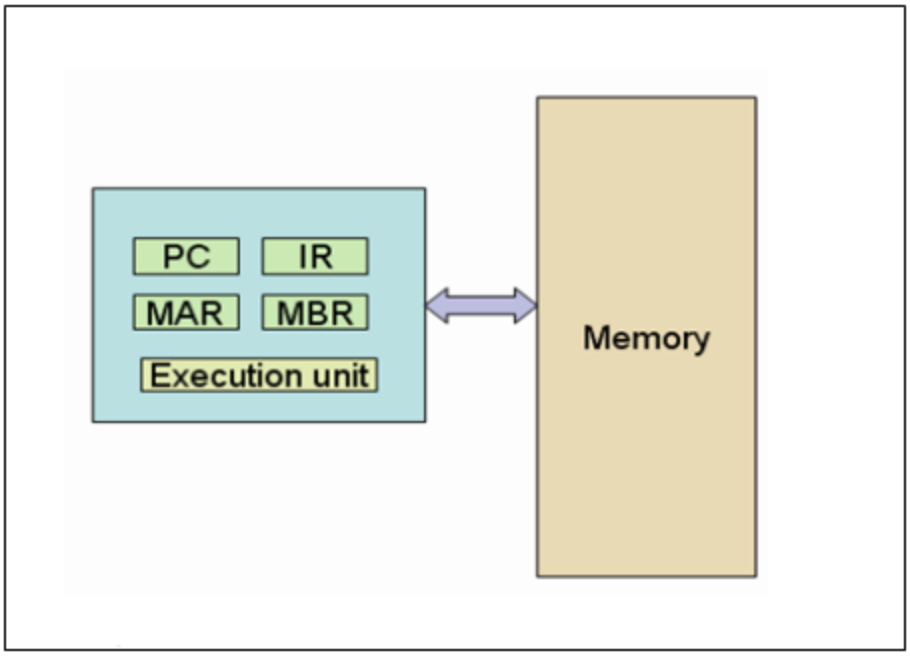

Figure 4: Addressing Modes

Our Simple Computer

SIMCOMP has a two byte-addressable memory with size of 128byte. The memory is synchronous to the CPU, and the CPU can read or write a word (or cell) in single clock period. The memory can only be accessed through the memory address register (MAR) and the memory buffer register (MBR). To read from memory, you use:
MBR <= Memory [MAR];
And to write to memory, you use:
Memory [MA] <= MBR;

1- The CPU has three registers: an accumulator (AC), a program counter (PC) and an instruction register (IR).

2- The SIMCOMP has only three instructions: Load, Store, and Add.

3- The size of all instructions is 16 bits; all the instructions are single address instructions and 5 access a word in memory.

Figure 5: Instruction Format

Op-code	Instruction	Description
0011	Load M	Loads the contents of memory location **M** into the accumulator.
1011	Store M	Stores the contents of the accumulator in memory location **M**.
0111	Add M	Adds the contents of memory location **M** to the contents of the accumulator.

Figure 6: The opcodes

Pre Lab ,Task 1,Task 2 and Task 3:

1- You have to write the basic code shown in Fig 1 at the last page of this experiment in your own Quartus II file.
2- You have to trace the basic code manually so that you can understand what does the code do. Use the following table:

```
1  module SIMCOMP (clock,PC,IR,MAR,MBR,AC);
2
3  input clock;
4  output PC,IR,MAR,MBR,AC;
5  reg [15:0]IR,MBR,AC;
6  reg [11:0]PC,MAR;
7  reg [15:0] Memory [0:63];
8  reg [2:0]state;
9
10 reg [15:0] R[0:3];
11
12 parameter load = 4'b0011 , store = 4'b1011 , add=4'b0111 , jump=4'b0001;
13
14 initial begin
15     Memory[10] = 16'h3020;
16     Memory[11] = 16'h7021;
17     Memory[12] = 16'hB014;
18     Memory[13] = 16'h100B;
19
20     Memory[32] = 16'd7;
21     Memory[33] = 16'd5;
22
23 PC =10; state = 0;
24 end
25
26 always @(posedge clock)
27 begin
28
29 case(state)
30
31 0:begin
32   MAR<=PC;
33   state = 1;
34   end
```

Figure 7: Task 1,2,3

```
1:begin
   IR <= Memory[MAR];
   PC<= PC + 1;
   state =2;
   end

2:begin
   MAR <= IR[11:0];
   state = 3;
   end

3:begin
   state = 4;
   case(IR[15:12])
     load:  MBR <= Memory[MAR];
     add:   MBR <= Memory[MAR];
     store: MBR <= AC;
    endcase
   end

4:begin
   if(IR[15:12] == 4'h7)
    begin
     AC <= AC + MBR;
     state = 0;
    end

    else if (IR[15:12] == 4'h3)
     begin
      AC <= MBR;
      state = 0;
     end

    else if (IR[15:12] == 4'hB)
     begin
      Memory[MAR] <= MBR;
      state = 0;
     end

    else if (IR[15:12] == 4'h1)
     begin
         PC <= MAR;
         state = 0;
     end
   end

endcase
end
endmodule
```

Code Explanation

Variables

Clock: Simulates computer internal clock.
General registers: PC, IR, MBR, MAR, AC.
Memory: 64 *16 size.
State: Manage instruction cycle steps.

Define supported Opcodes

```
parameter load = 4'b0011 , store = 4'b1011 , add=4'b0111 , jump=4'b0001;
```

Initialization

Code: Load data @ address 20 into AC
Add data @ address 21 into AC
Store AC content
Data: [32] = d7
[33] = d5
PC to 10 [The program counter]
State to 0 [Initial state/step]

Instruction Cycle

Instruction Fetch	Operand Fetch	Instruction Execute

always @(posedge clock)
begin
 	
case(state)
 	
0:begin
MAR<=PC;
state = 1;
end
 	
1:begin
IR <= Memory[MAR];
PC<= PC + 1;
state =2;
end
	2:begin
	MAR <= IR[11:0];
	state = 3;
	end
	
	3:begin
	state = 4;
	case(IR[15:12])
	load: MBR <= Memory[MAR];
	add: MBR <= Memory[MAR];
	store: MBR <= AC;
	jump: MBR <= MAR;
	endcase
	end
		4:begin
		if(IR[15:12] == 4'h7)
		begin
		AC <= AC + MBR;
		state = 0;
		end
		
		else if (IR[15:12] == 4'h3)
		begin
		AC <= MBR;
		state = 0;
		end
		
		else if (IR[15:12] == 4'hB)
		begin
		Memory[MAR] <= MBR;
		state = 0;
		end
		
		else if (IR[15:12] == 4'h1)
		begin
		PC <= MAR;
		state = 0;
		end
		end

The figure below explains what we edited on the code.

```
parameter load = 4'b0011 , store = 4'b1011 , add=4'b0111 , jump=4'b0001;
```

```
3:begin
   state = 4;
   case(IR[15:12])
     load:  MBR <= Memory[MAR];
     add:   MBR <= Memory[MAR];
     store: MBR <= AC;
     jump:  MBR <= MAR;
    endcase
   end
```

Load data:

The first positive edge the value of the pc was copied to the MAR. In the next positive edge, the instruction from the memory was stored in the IR, and increment for pc happened. The third positive edge, the address for the operand stored in the MAR. And the last one the operand was moved from the memory and stored in the MBR.

Add data:

The first positive edge the value of the pc was copied to the MAR. In the second positive edge, the instruction was stored in the instruction register, and the pc was increment by one. In the third positive edge the address from the instruction stored in the MAR. The next positive edge, the 5 operand from the memory stored in the MAR. Moreover, the last positive edge the execute for the add was done and the MBR was added to the accumulator.

Store data:

The first positive edge the value of the pc was copied to the MAR. In the second step the instruction stored in the MAR and the pc increment by one. The third positive edge, the address that found in the instruction register will copied into the MAR. The fourth positive edge, the accumulator is copied into the MBR. Furthermore, in the memory the value of accumulator was stored.

Code Simulation

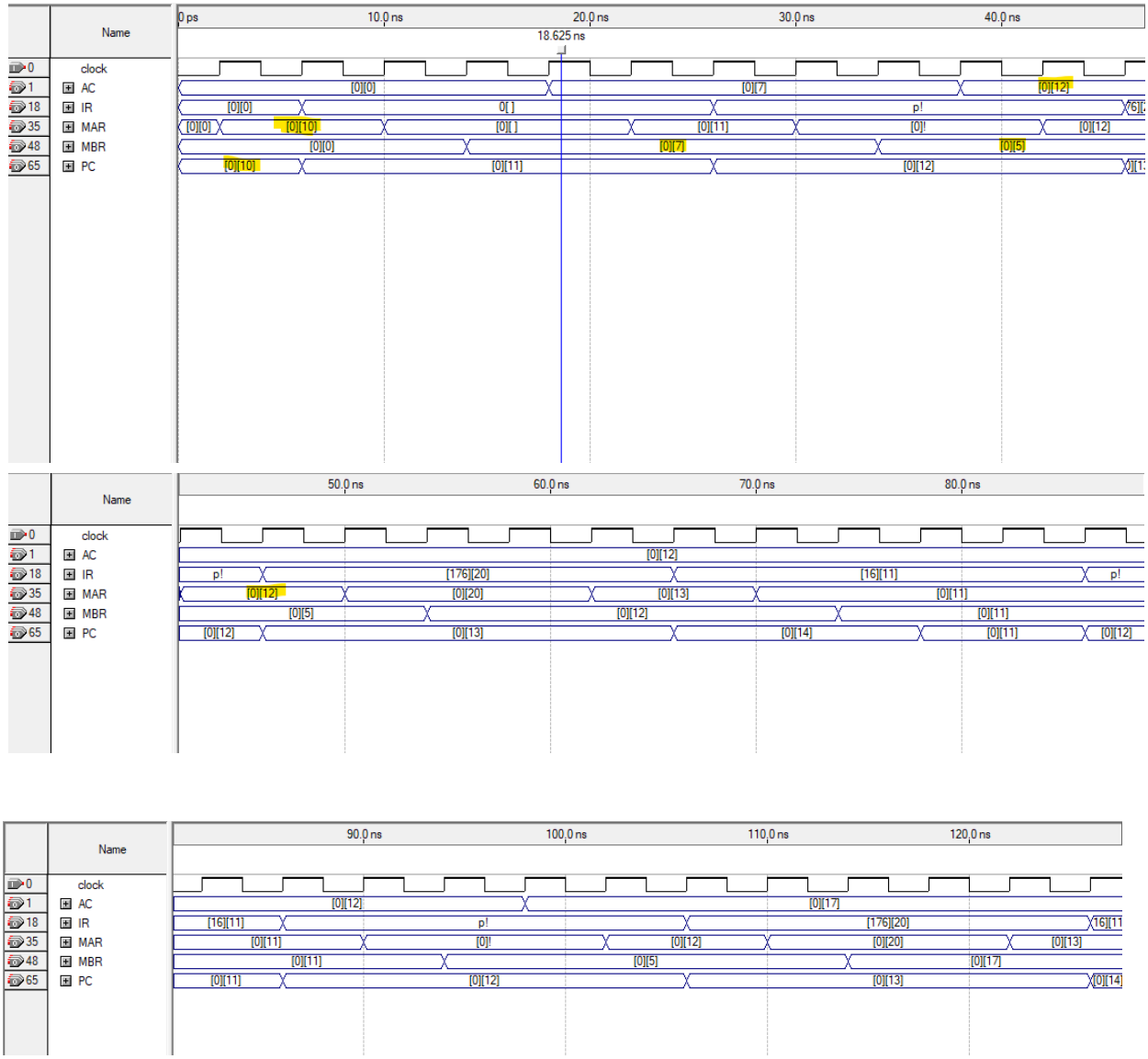

Figure 8: Simulation

Task 4: Modify the four instructions of the old SIMCOMP

The new instruction should follow the new form:
1- **LOAD R[i], M** loads the contents of memory location M into R[i].
2- **STORE R[i], M** stores the contents of R[i] in memory location M.
3- **ADD R[i], R [j], R[k]** adds contents of R[j] and R[k] and places result inR[i].
To test my design, I perform the following program where:
1- PC starts at 10,
2- Suppose **IR [9:8]** is used to specify the register number.
3- In the "add instruction" **IR [11:10]** destination register, **IR [9:8], IR [7:6]** source1, source2 respectively.

Task 5:

Write the General form of the instruction set for SIMCOMP2.

Code:

```
1  module SIMCOMP3 (clock,PC,IR,MAR,MBR,R0,R1,R2,R3);
2
3  input clock;
4  output PC,IR,MAR,MBR,R0,R1,R2,R3;
5  reg [15:0]IR,MBR,R0,R1,R2,R3;
6  reg [11:0]PC,MAR;
7  reg [15:0] Memory [0:63];
8  reg [2:0]state;
9
10 reg [15:0] R[0:4]; //  declaration of registers bank
11
12 parameter load = 4'b0011 , store = 4'b1011 , add=4'b0111 , jump=4'b0001;
13
14 initial begin
15      Memory[10] = 16'h3103;  //  load memory 3 to R1
16      Memory[11] = 16'h3204;  //  load memory 4 to R2
17      Memory[12] = 16'h7580;  //  add R1, R2
18      Memory[13] = 16'hb105;  //  store
19     // Data
20      Memory[03] = 16'hA;
21      Memory[04] = 16'h6;
22
23 PC =10; state = 0;
24 end
25
26 always @(posedge clock)
27 begin
28
29 case(state)
30
31 0:begin
32   MAR<=PC;
33   state = 1;
34   end
```

```
1:begin
   IR <= Memory[MAR];
   PC<= PC + 1;
   state =2;
   end

2:begin
   MAR <= IR[11:0];
   state = 3;
   end

3:begin
   state = 4;
   case(IR[15:12])
     load:  MBR <= Memory[MAR];
     add:   MBR <= Memory[MAR];
     store: MBR <= R[IR[9:8]];
     jump:  MBR <= Memory[MAR];
    endcase
   end

4:begin
   if(IR[15:12] == 4'h7)
    begin
     R[IR[11:10]] <= R[IR[9:8]] + R[IR[7:6]];
     state = 0;
    end

    else if (IR[15:12] == 4'h3)
     begin
      R[IR[9:8]] <= MBR;
      state = 0;
     end
```

Figure 9: Task 5

```
    else if (IR[15:12] == 4'hB)
     begin
      Memory[MAR] <= MBR;
      state = 0;
     end

    else if (IR[15:12] == 4'h1)
     begin
         PC <= MAR;
         state = 0;
         end
         R0 <= R[0];
         R1 <= R[1];
         R2 <= R[2];
         R3 <= R[3];
   end

endcase
end
endmodule
```

Note that, this new SIMCOMP2 has four 16-bit general purpose registers, R[0], R[1], R[2] and R[3] which replace the AC. In Verilog, we declare R as a bank of registers much like we do Memory. The highlighted above explains what we edited on the code.

Task 6:

Wave form:

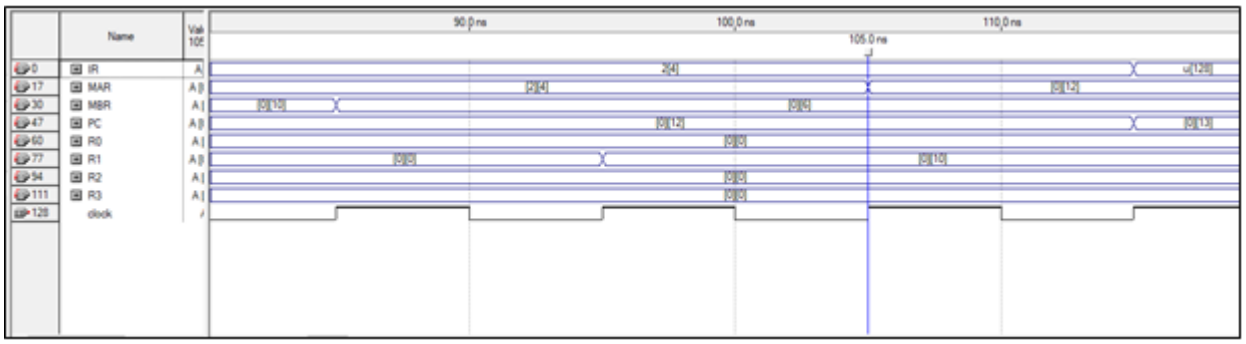

Figure 10: Task 6

Task 7:

Add immediate addressing to the SIMCOMP2

If bit (IR [11]) is a one in a Load, the last eight bits are not an address but an operand. The operand is in the range -128 to 127.
If immediate addressing is used in a LOAD, the operand is loaded into the register.
Load R1, 8
R1 ← 8

Code:

```
1  module SIMCOMP4 (clock,PC,IR,MAR,MBR,R0,R1,R2,R3);
2
3  input clock;
4  output PC,IR,MAR,MBR,AC,R0,R1,R2,R3;
5  reg [15:0]IR,MBR,R0,R1,R2,R3;
6  reg [11:0]PC,MAR;
7  reg [15:0] Memory [0:63];
8  reg [2:0]state;
9
10 reg [15:0] R[0:4]; //  declaration of registers bank
11
12 parameter load = 4'b0011 , store = 4'b1011 , add=4'b0111 , jump=4'b0001;
13
14 initial begin
15      Memory[10] = 16'h3903;  //  load memory 3 to R1
16      Memory[11] = 16'h3AFC;  //  load memory 4 to R2
17      Memory[12] = 16'h7580;  //  add R1, R2
18      Memory[13] = 16'hb105;  //  store
19     // Data
20      Memory[03] = 16'hA;
21      Memory[04] = 16'h6;
22
23 PC =10; state = 0;
24 end
25
26 always @(posedge clock)
27 begin
28
29 case(state)
30
31 0:begin
32   MAR<=PC;
33   state = 1;
34   end
```

Figure 11: Task 7

```
35
36  1:begin
37    IR <= Memory[MAR];
38    PC<= PC + 1;
39    state =2;
40    end
41
42  2:begin
43    MAR <= IR[11:0];
44    state = 3;
45    end
46
47  3:begin
48    state = 4;
49    case(IR[15:12])
50      load:  MBR <= Memory[MAR];
51      add:   MBR <= Memory[MAR];
52      store: MBR <= R[IR[9:8]];
53      jump:  MBR <= Memory[MAR];
54     endcase
55    end
56
57  4:begin
58    if(IR[15:12] == 4'h7)
59     begin
60      R[IR[11:10]] <= R[IR[9:8]] + R[IR[7:6]];
61      state = 0;
62     end
63
64     else if (IR[15:12] == 4'h3)
65      begin
66       R[IR[9:8]] <= MBR;
67       state = 0;
```

```
67          state = 0;
68         end
69
70           else if (IR[11] == 1'b1)
71           if (IR[7]==1'b0)
72        begin // load if
73         R[IR[9:8]] <= IR[7:0];
74         state = 0;
75        end
76
77        else if (IR[7]==1'b1)
78        begin // load if
79         R[IR[9:8]] <= -(~(IR[7:0]+1);
80         state = 0;
81        end
82
83       else if (IR[15:12] == 4'hB)
84        begin
85         Memory[MAR] <= MBR;
86         state = 0;
87        end
88
89       else if (IR[15:12] == 4'h1)
90        begin
91            PC <= MAR;
92            state = 0;
93            end
94            R0 <= R[0];
95            R1 <= R[1];
96            R2 <= R[2];
97            R3 <= R[3];
98      end
99
100   endcase
```

Task 8:

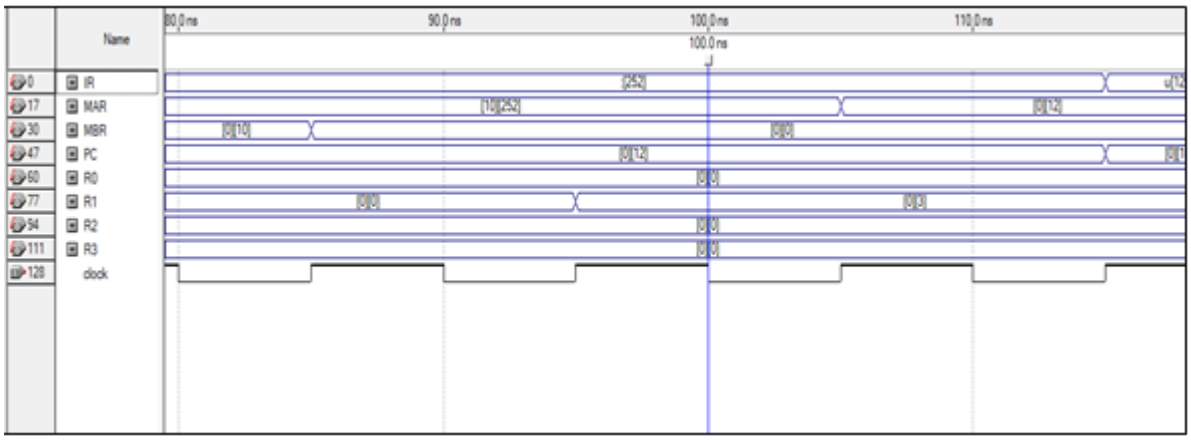

Figure 12: Task 8

Conclusion

In this experiment we learned about basic computer model how the CPU works (Generic Instruction Cycle) also about addressing mode, and how to write a complete Verilog HDL code that implement a simple computer, we also practiced on Quartus and test our code by using wave form, and learned new skills.

Reference

[1] https://www.javatpoint.com/von-neumann-model Accessed on 21/6/2023 at 10:52 PM.

[2] https://www.javatpoint.com/von-neumann-model Accessed on 21/6/2023 at 10:58 PM.

[3] https://www.javatpoint.com/von-neumann-model Accessed on 21/6/2023 at 11:05 PM.

[4] https://www.tutorialspoint.com/what-is-instruction-cycle-in-computer-architecture Accessed on 21/6/2023 at 11:13 PM.

[5] lab manual Accessed at different time