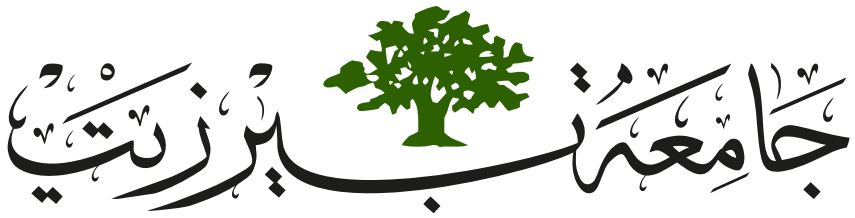


Department of Electrical & Computer Engineering
ENCS2380 - Digital Electronics and Computer Organization Lab

Project I

Prepared by: Mohamad Milhem
ID number: 1211053
Instructor: Dr. Abualseoud Hanani
Date: January 19, 2023

Abstract

In this project, our aim is to improve the accumulator computer provided and described by the instructor in the project file into a more complicated computer. This computer's specifications are a CPU synchronous 128-cell memory with a fixed cell width of 16-bit, PC register, MAR and MBR registers, and 8 general purpose registers from R_0 to R_7.

Contents

1 Design Specifications

1.1 Instruction Set Architecture.

1.1.1 Instruction format

The instructions that our CPU will perform are in the following format.

OpCode (3bits)	Dest. Reg (3bits)	mode (2bits)	Src. Register/ Memory Address (8bits)

Table 1: Instruction format.

Looking at the instruction format we can observe that our machine will:

1. support up to **8 different operations.**
2. support up to **8 general purposes registers** as a destination.
3. support up to **4 addressing modes** (specified in the next subsection.)
4. support up to **512-byte memory** (256 cells with width 16-bit) / support a range of signed constants **from -128 to 127** or unsigned constants **from 0 to 255.**

1.1.2 Addressing Modes bits

As mentioned in the previous subsection, our machine can support up to 4 different modes based on the instruction format. These modes were specified as follows.

I. **Direct Memory access (M = 00)**
In this mode the binary value in the address field in the instruction is the actual memory cell address.

II. **Direct Register access (M = 01)**
In this mode the binary value in the address field in the instruction represents the index of the source register that contains the operand.

III. **Indirect Register access (M = 10)**
In this mode the binary value in the address field in the instruction represents the index of the register that contains the memory address of the cell that contains the operand.

IV. **Immediate/constant access (M = 11)**
In this mode the binary value in the address field in the instruction represents a signed constant represented in the two's complement.

1.1.3 Operations

As mentioned previously, our machine can handle up to 8 different operations the specifications for this project define only 6 of them which are:

I. **LOAD R_i, operand (Opcode = 000)**
This operation loads R_i with the intended value based on the value of the operand and the mode bits.

II. **STORE R_i, operand (Opcode = 001)**
This operation store the content of R_i in the memory cell with address represented by operand.

III. **ADD R_i, operand (Opcode = 010)**
This operation adds the intended value based on the value of the operand and the mode bits to R_i and stores the result in R_i.

IV. **SUB R_i, operand (Opcode = 011)**
This operation subtracts the intended value based on the value of the operand and the mode bits from R_i and stores the result in R_i.

V. **MUL R_i, operand (Opcode = 100)**
This operation multiplies the intended value based on the value of the operand and the mode bits by R_i and stores the result in R_i.

VI. **DIV R_i, operand (Opcode = 101)**
This operation divides the value stored in R_i by the intended value based on the value of the operand and the mode bits and stores the result in R_i.

1.2 Memory

For this project, we need synchronous 128-cell memory with a fixed cell width of 16-bit. Saying this memory is synchronous with the CPU means that the CPU needs one Clock cycle to read or write one cell from the memory.

1.3 Sign representation

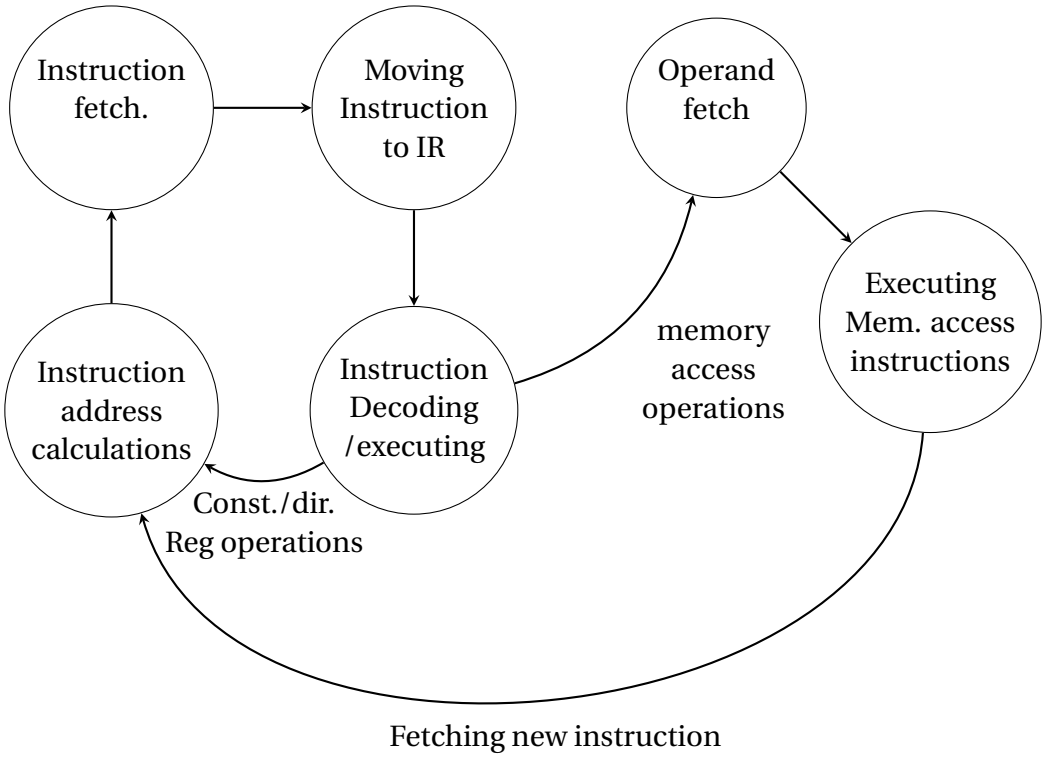
The original sign representation of this project was a 16-bit Sign-Magnitude representation, but due to the inconvenient implementation of the add and subtraction operations, the instructor replace the Sign-Magnitude representation with the two's complement representation which is more convenient and more usable.

2 Implemented Design

The design was implemented, simulated, and verified by Quartus II 9.0 Web Edition software.

2.1 States and Design Flow

The simulated design flows as shown in the next state diagram.



1. **Instruction address calculations**

```
if (state == 0)begin

    MAR <= PC;
    state = 1;

end
```

Figure 2.1: state 0

The address of the next instruction is sent to the Memory Address Register (MAR) to fetch it in the next state.

2. **Instruction fetch**

```
else if (state == 1)begin

    MBR <= Memory[MAR];
    PC <= PC + 1;
    state = 2;

end
```

Figure 2.2: state 1

The new instruction is stored in the Memory Buffer Register (MBR) and we add one to PC so it contains the address of the next instruction.

3. **Moving Instruction to IR**

```
else if (state == 2)begin

    IR <= MBR;
    state = 3;

end
```

Figure 2.3: state 2

We move the instruction to the Instruction register (IR) to prepare to decode it.

4. **Instruction Decoding/executing**

This state has three parts.

```
else if (state == 3)begin

    if (IR[9:8] == constant)begin // Execution, MODE is constant.

        if (IR[15:13] == load)begin
            GeneralRegs[IR[12:10]] <= {{8{IR[7]}}, IR[7:0]};
        end

        if (IR[15:13] == store)begin
            // there is no store with mode.
        end

        if (IR[15:13] == add)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] + {{8{IR[7]}}, IR[7:0]};
        end

        if (IR[15:13] == sub)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] - {{8{IR[7]}}, IR[7:0]};
        end

        if (IR[15:13] == mul)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] * {{8{IR[7]}}, IR[7:0]};
        end

        if (IR[15:13] == div)begin
            GeneralRegs[IR[12:10]] <= {{8{IR[7]}}, IR[7:0]} / GeneralRegs[IR[12:10]];
        end
        state = 0;

    end
```

Figure 2.4: state 3.1

The first part is for constant operations, when the mode is immediate/constant there is no need to fetch anything from memory so we can execute the instruction directly without any more extra clock cycles.

```
else if (IR[9:8] == dirReg)begin // Execution, MODE is Direct Register.

    if (IR[15:13] == load)begin
        GeneralRegs[IR[12:10]] <= GeneralRegs[IR[7:0]];
    end

    if (IR[15:13] == store)begin
        // there is no store with this mode.
    end

    if (IR[15:13] == add)begin
        GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] + GeneralRegs[IR[7:0]];
    end

    if (IR[15:13] == sub)begin
        GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] - GeneralRegs[IR[7:0]];
    end

    if (IR[15:13] == mul)begin
        GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] * GeneralRegs[IR[7:0]];
    end

    if (IR[15:13] == div)begin
        GeneralRegs[IR[12:10]] <= GeneralRegs[IR[7:0]] / GeneralRegs[IR[12:10]];
    end
    state = 0;

end
```

Figure 2.5: state 3.2

The second part is for direct register operations, when the mode is direct register there is no need to fetch anything from memory so we can execute the instruction directly without any more extra clock cycles.

```
else begin // Memory access is needed, calculation for operand address.

    if (IR[9:8] == indirReg)
        MAR <= GeneralRegs[IR[7:0]];

    if (IR[9:8] == dirMem)
        MAR <= IR[7:0];

    state = 4;
end
```

Figure 2.6: state 3.3

The third part is for other modes, all other modes need memory access so in this part we are doing the operand address calculations to prepare to fetch the operand in the next state.

5. **Operand Fetch**

```
else if (state == 4)begin

    if (IR[15:13] == store)
        MBR <= GeneralRegs[IR[12:10]];


    else
        MBR <= Memory[MAR];

    state = 5;

end
```

Figure 2.7: state 4

After calculating the address of the operand, we can fetch it using the MAR and MBR.

6. **Executing the memory access instructions**

```
else if (state == 5)begin // executing instruction with memory access.

        if (IR[15:13] == load)begin
            GeneralRegs[IR[12:10]] <= MBR;
        end

        if (IR[15:13] == store)begin
            Memory[MAR] <= MBR; // to test store in TestR
        end

        if (IR[15:13] == add)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] + MBR;
        end

        if (IR[15:13] == sub)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] - MBR;
        end

        if (IR[15:13] == mul)begin
            GeneralRegs[IR[12:10]] <= GeneralRegs[IR[12:10]] * MBR;
        end

        if (IR[15:13] == div)begin
            GeneralRegs[IR[12:10]] <= MBR / GeneralRegs[IR[12:10]];
        end
        state = 0;

end
```

Figure 2.8: state 5

After fetching the operand we can do the operation needed and store the result and finish the instruction cycle.

3 Testing and verification

Using the example provided by the instructor.

Example

To add memory cell of address 10 with memory cell pointed by register R1, and store the result in memory cell of address 12, we use the following program:

Assembly	Machine code
Load R_0, [10]	000 000 00 00001010
Load R_1, 11	000 001 11 00001011
Add R_0, [R_1]	010 000 10 00000001
Store R_0, [12]	001 000 00 00001100

Table 2: Example instructions

Simulation

we start by initializing memory with the instruction and data, also we have to set the PC register to address of the first instruction. by the given data the answer should be 12.

```
13
14  initial begin
15
16      // instructions
17      Memory[32] = 16'b000_000_00_00001010; // load R0, [10]
18      Memory[33] = 16'b000_001_11_00001011; // load R1, 11
19      Memory[34] = 16'b010_000_10_00000001; // add R0, [R1]
20      Memory[35] = 16'b001_000_00_00001100; // store R0, [12]
21
22      // data
23      Memory[10] = 16'd5;
24      Memory[11] = 16'd7;
25
26      // set the program counter to the start of the program
27      PC = 32; state = 0;
28  end
29
```

Figure 3.1: initial block.

after initializing the memory we run a waveform simulation to test the design. To be able to see the output we store the final value in Test Register called TestR. (All value are in Hexadecimal).

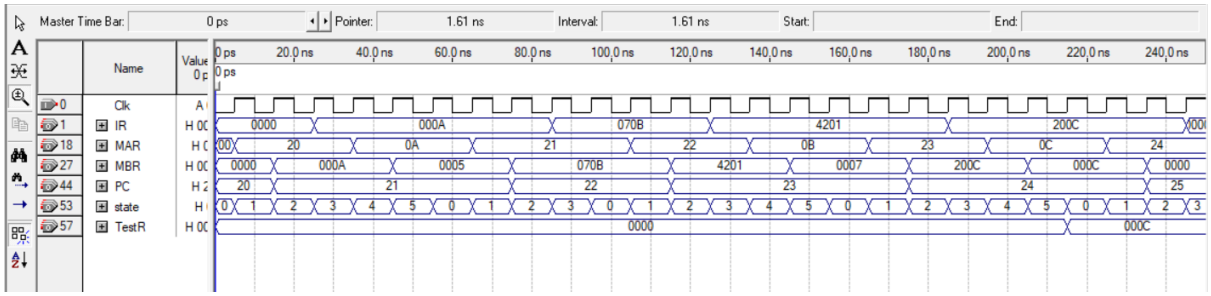


Figure 3.2: waveform simulation for Example (Hex).

4 Requirements

Q1. a) **How many instructions this machine can support?**

As we mentioned in subsection 1.1.1, and since we have 3 bits for Opcode our machine can support up to **8 different operations.**

b) **What is the range of unsigned and signed constant numbers this machine supports?**

As we also mentioned in subsection 1.1.1, and since we have 8 bits for representing the constant, our machine supports signed constants in range **from -128 to 127** and a range of unsigned constants **from 0 to 255.**

c) **What is the length (in bits) of the following registers in this machine?**

PC:

The PC stores address of the next instruction and since we have a 128 cells memory so we have 128 different address, **7 bits will be enough** to cover all the memory.

But, since the address field in the instruction format is 8 bits and to leave a room for possible improvements (adding more memory), the PC register was set in the implementation to 8 bits register.

IR:

Based on the instruction format the instruction have 16-bit so the Instruction Register **(IR) should have 16-bit** to contain the instruction.

MAR:

The Memory Address Register (MAR) is used to fetch data from memory, so it will store addresses. So As we mentioned for the PC register **7 bit are enough** but we prefer to use 8 bit in the implementation for the same reasons mentioned for the PC register.

MBR:

The Memory Buffer Register (MBR) is used to store the data fetched from memory. and since the cell of the data is 16-bit length. **MBR is 16-bit length.**

Q2. **Simulate the following program by converting each instruction to the corresponding machine code. Then store the machine code in memory starting from location 10:**

Memory Address	Contents
10	Load R0, [20] (instruction)
11	Load R1,21 (instruction)
12	Add R0, [R1] (instruction)
13	load R1, [22] (instruction)
14	Sub R1, +8 (instruction)
15	Add R0,R1 (instruction)
16	Store R0, [23] (instruction)
...	...
20	6 (Data)
21	4 (Data)
22	13 (Data)
23	0 (Data)
24	0 (Data)

Table 3: Q2 instructions

First, we convert the Assembly to machine code.

Memory Address	Contents	machine code
10	Load R0, [20] (instruction)	000 000 00 00010100
11	Load R1,21 (instruction)	000 001 11 00010101
12	Add R0, [R1] (instruction)	010 000 10 00000001
13	load R1, [22] (instruction)	000 001 00 00010110
14	Sub R1, +8 (instruction)	011 001 11 00001000
15	Add R0,R1 (instruction)	010 000 01 00000001
16	Store R0, [23] (instruction)	001 000 00 00010111
...	...	...
20	6 (Data)	0000000000000110
21	4 (Data)	0000000000000100
22	13 (Data)	0000000000001101
23	0 (Data)	0000000000000000
24	0 (Data)	0000000000000000

Table 4: Q2 instructions with machine code

Initialize the memory with the instructions and data. Initialize PC to 10.

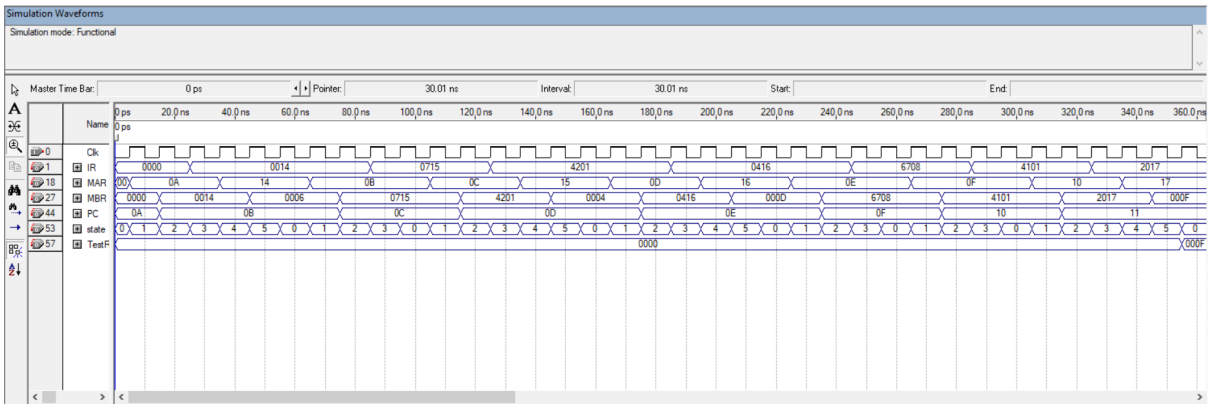


Figure 4.1: waveform simulation for Q2.

As we can see the value at the 23 address in the memory is 15 in decimal.

Q3. **Assume A, B, C, D, E, and Y are memory cells with addresses 30,31,32,33,34, and 35, respectively. Given ($Y = \frac{A+B*C-1}{D-E}$)**

a) **Write assembly code for implementing the above arithmetic expression?**

load R_0, [31]

Mul R_0, [32]

Add R_0, [30]

Sub R_0, +1

load R_1, [33]

Sub R_1, [34]

Div R_1, R_0

Store R_1, [35]

b) **Convert the above assembly instructions into machine code and store them in the memory starting at address 10.**

Memory Address	Contents	machine code
10	load R_0, [31]	000 000 00 00011111
11	Mul R_0, [32]	100 000 00 00100000
12	Add R_0, [30]	010 000 00 00011110
13	Sub R_0, +1	011 000 11 00000001
14	load R_1, [33]	000 001 00 00100001
15	Sub R_1, [34]	011 001 00 00100010
16	Div R_1, R_0	101 001 01 00000000
17	Store R_1, [35]	001 001 00 00100011

Table 5: Q3 instructions with machine code

c) **Set PC=10 and simulate the above program. Verify that it works correctly and the result stored at memory variable Y is correct. Attach simulation waveform and the Verilog source file. Assume A, B, C, D and E have the values -1, 3, 5, 8, and 4, respectively.**

Initialize the memory with the instructions and data. Initialize PC to 10.

```
14  initial begin
15
16      // instructions
17      Memory[10] = 16'b000_000_00_00011111; // load R0, [31]
18      Memory[11] = 16'b100_000_00_00100000; // Mul R0, [32]
19      Memory[12] = 16'b010_000_00_00011110; // Add R0, [30]
20      Memory[13] = 16'b011_000_11_00000001; // Sub R0, +1
21      Memory[14] = 16'b000_001_00_00100001; // load R1, [33]
22      Memory[15] = 16'b011_001_00_00100010; // Sub R1, [34]
23      Memory[16] = 16'b101_001_01_00000000; // Div R1, R0
24      Memory[17] = 16'b001_001_00_00100011; // Store R1, [35]
25
26
27      // data
28      Memory[30] = 16'b1111111111111111;  //A = -1
29      Memory[31] = 16'b0000000000000011;  //B = 3
30      Memory[32] = 16'b0000000000000101;  //C = 5
31      Memory[33] = 16'b0000000000001000;  //D = 8
32      Memory[34] = 16'b0000000000000100;  //E = 4
33
34      // set the program counter to the start of the program
35      PC = 10; state = 0;
36  end
37
```

Figure 4.2: Initialization for Q3.

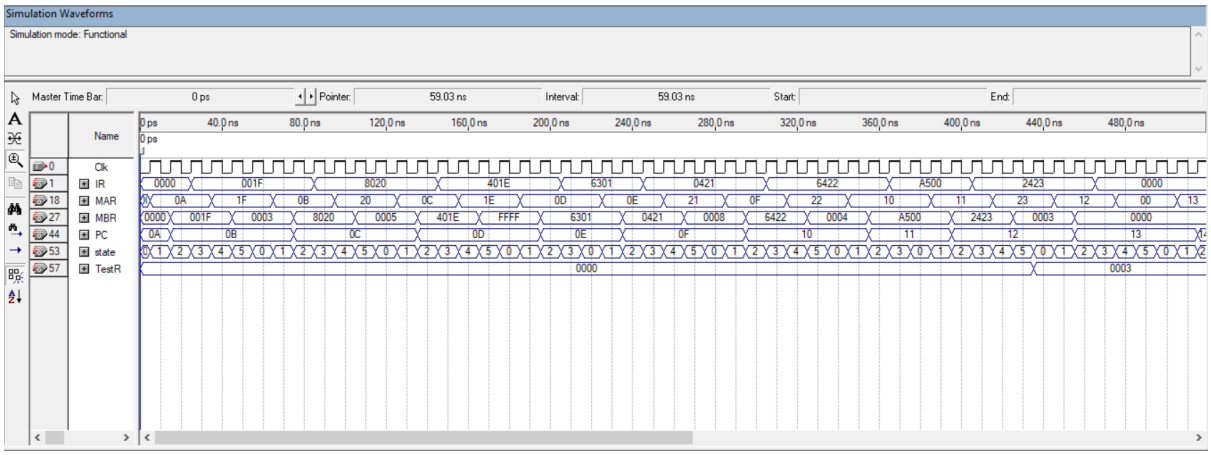


Figure 4.3: waveform simulation for Q3 (Hex).

All values in the simulation waveform are in Hexadecimal representation.

5 Conclusion

In this project, we have implemented a simple computer using Verilog, We have studied how to implement the different parts of the computer like the CPU and memory. And also, worked with the two's complement representation and went through the process of the cycle instruction stage by stage.

In addition, we gained some experience with Verilog and Quartus software. For example, you can write a function in Verilog but it's not synthesizable most of the time, it can be used for combinational circuits since it has no delay time also it's used in building test benches.

List of Figures

List of Tables