Faculty of Engineering & Technology

Electrical & Computer Engineering Department

COMPUTER ORGANIZATION AND MICROPROCESSOR

Project#1

Prepared by:

Taleen Bayatneh	1211305	section: 4
Shahd Loai	1211019	section: 2
Miassar Shamla	1210519	section: 2

Instructor:

Ayman Hroub

Abualseoud Hanini

Date: 18-6-2023

Abstract:

Designing an accumulator computer with a 16-bit cell memory involves defining the architecture and functionality of the CPU, memory, and registers by using Verilog HDL.

Theory:

Accumulator computer theory refers to a concept in computer architecture where a central processing unit (CPU) includes an accumulator register as a key component. The accumulator is a special-purpose register that stores intermediate results during the execution of .arithmetic and logical operations

In an accumulator-based computer architecture, the accumulator serves as the primary storage location for data manipulation. It holds the result of arithmetic and logical operations, and other registers interact with it to perform various calculations. The accumulator can be accessed directly by the CPU, and its contents can be modified through arithmetic and logical instructions.

The use of an accumulator register simplifies the design of the CPU by reducing the number of registers required. Instead of having separate registers for each arithmetic operation, the accumulator acts as a temporary storage location for these operations. This approach allows for more efficient instruction coding and streamlines the execution of instructions.

Registers:

In a CPU (Central Processing Unit), registers are small, high-speed storage units used to store and manipulate data during the execution of instructions. They are an essential part of the CPU and play a crucial role in its operation. Here are the common registers found in a CPU:

1- Accumulator (AC):

- The accumulator register is used for arithmetic and logic operations.
- It holds intermediate results and final results of computations.
- Many arithmetic and logic instructions operate directly on the accumulator.

2- Program Counter (PC):

- The program counter register keeps track of the memory address of the next instruction to be fetched.
- It is automatically incremented after each instruction fetch.
- It ensures the CPU fetches instructions in the correct sequence.

3- Instruction Register (IR):

- The instruction register holds the current instruction fetched from memory.
- It is used for decoding the opcode and operands of the instruction.

4- Memory Address Register (MAR):

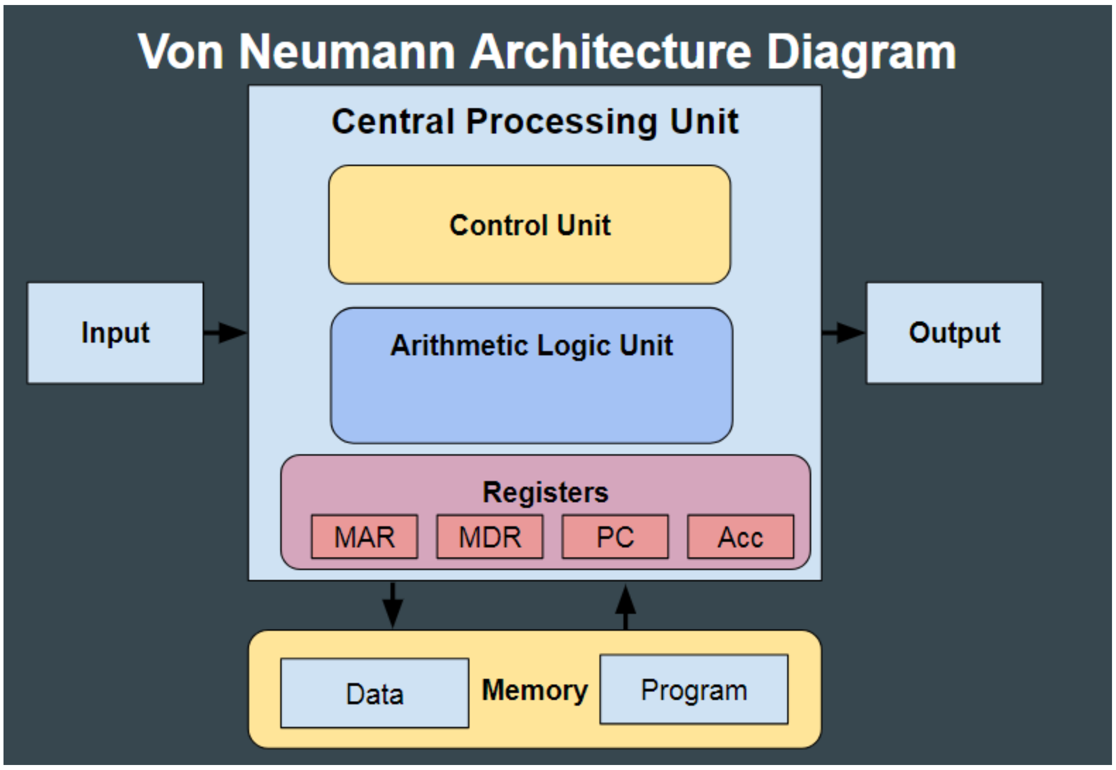
- The memory address register holds the memory address being accessed for read or write operations.
- It is used to specify the address in memory for data transfer.

5- Memory Buffer Register (MBR):

- The memory buffer register temporarily holds data that is read from or written to memory.
- It acts as an intermediary between the CPU and memory during data transfer.

6- General-Purpose Registers:

- General-purpose registers (such as R1, R2, R3, etc.) are used for various purposes, including storing operands, intermediate results, or data during computations.
- They can be used by the programmer to hold data during program execution.



Computer components

CPU Basics: Instruction Cycle

The Instruction Cycle, also known as the Fetch-Decode-Execute cycle, is the basic operation performed by a CPU to process instructions. It consists of several stages that are repeated for each instruction in a program. Let's break down the stages of the Instruction Cycle:

1- Fetch:

- The CPU fetches the next instruction from memory.
- The program counter (PC) contains the memory address of the next instruction to be fetched.
- The CPU transfers the instruction from memory to the Instruction Register (IR).
- The PC is incremented to point to the next instruction in memory.

2- Decode:

- The instruction stored in the IR is decoded to determine the operation to be performed.

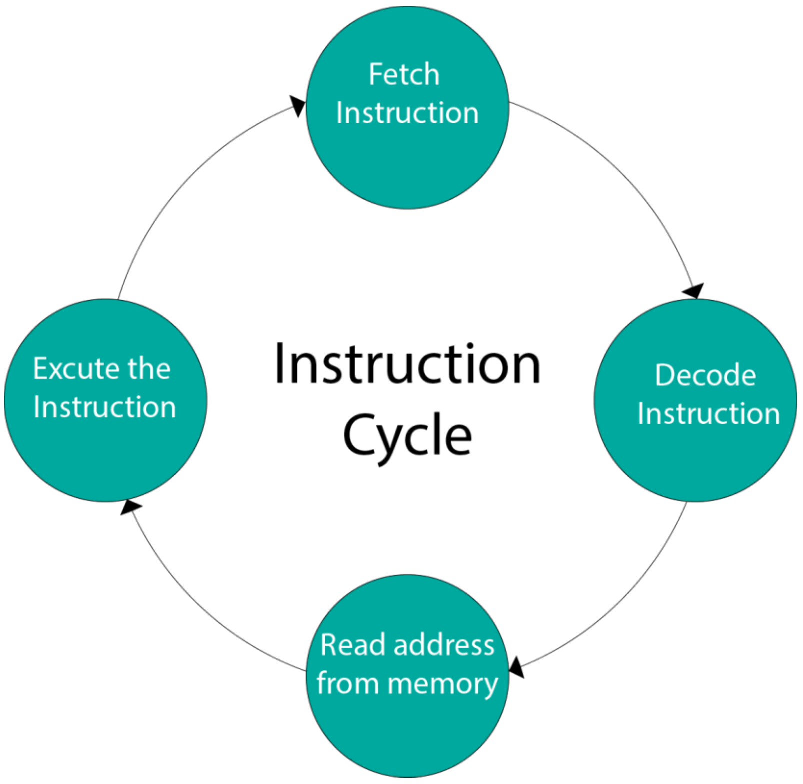
- The control unit of the CPU extracts the opcode (operation code) and any operands or addressing modes from the instruction.

3- Execute:

- Depending on the instruction, the CPU performs various tasks such as data manipulation, arithmetic/logic operations, or control transfers.
- This stage involves accessing registers, performing calculations, and updating the necessary registers or memory locations.

4- Repeat:

- After executing the current instruction, the cycle repeats, and the CPU fetches the next instruction.
- The PC is updated to hold the address of the next instruction, and the process continues until the program completes.



Instruction Cycle

In our project we will design an accumulator computer with a 16-bit cell memory involves defining the architecture and functionality of the CPU, memory, and registers. Here's a basic outline of the components :

1- CPU (Central Processing Unit):

- Accumulator Register (AC): A 16-bit register used to store data and perform arithmetic operations.
- Program Counter (PC) Register: A register that holds the address of the next instruction to be executed.
- Instruction Register (IR): A register used to store the current instruction fetched from memory.

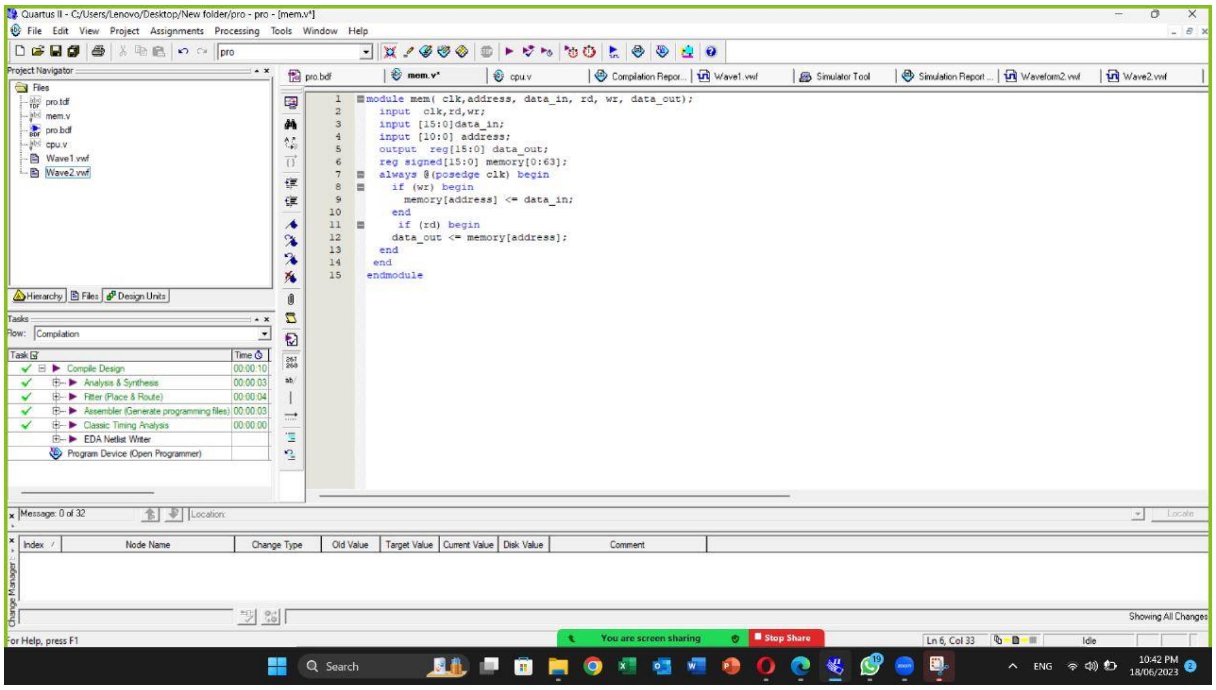
2- Memory:

- 16-bit Cell Memory: The main storage unit of the computer, capable of holding 16-bit data values.
- Memory Address Register (MAR): A register used to hold the memory address being accessed.
- Memory Buffer Register (MBR): A register used to hold the data read from or written to memory.

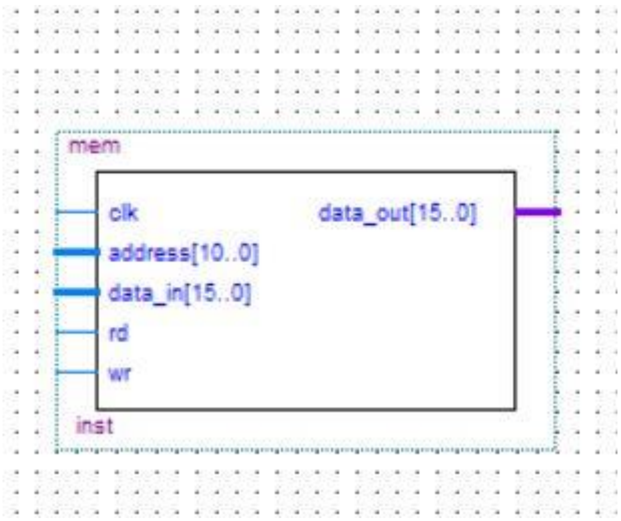
Requirements:

- ### Computer design and implementation:

1) Design and implement main memory as a Verilog module:

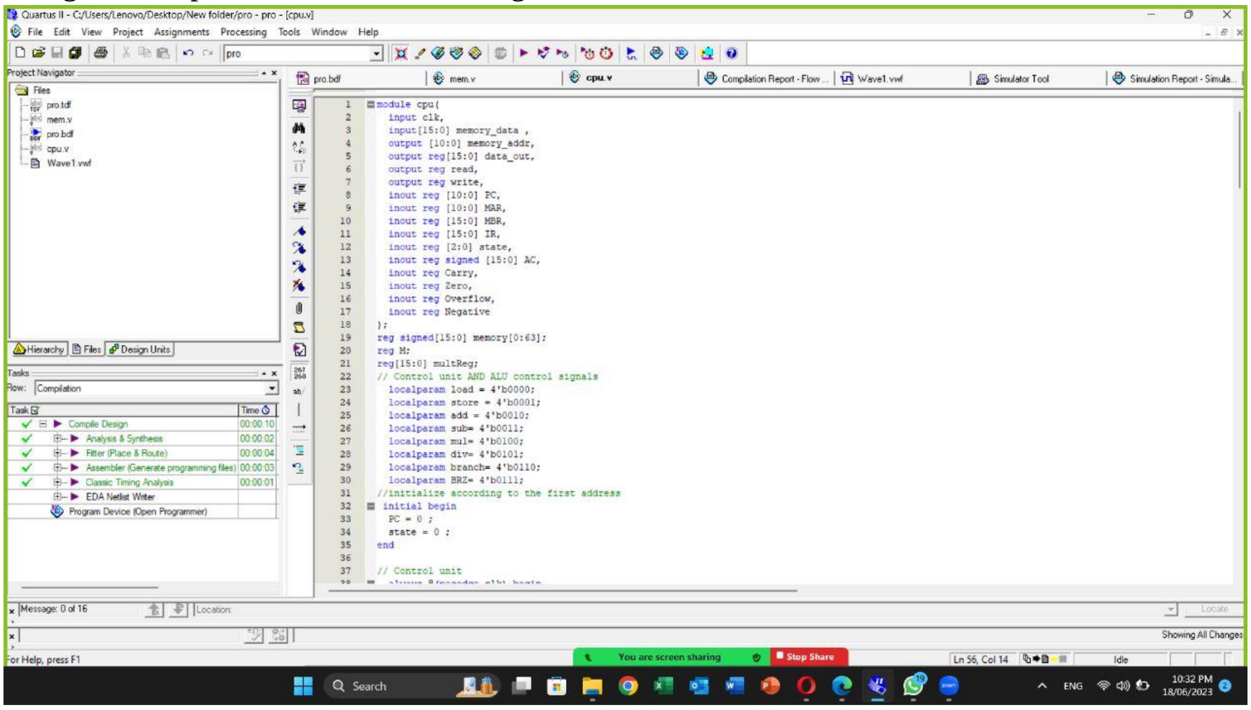


```verilog
module mem( clk,address, data_in, rd, wr, data_out);
  input  clk,rd,wr;
  input [15:0]data_in;
  input [10:0] address;
  output  reg[15:0] data_out;
  reg signed[15:0] memory[0:63];
  always @(posedge clk) begin
    if (wr) begin
      memory[address] <= data_in;
    end
     if (rd) begin
    data_out <= memory[address];
  end
 end
endmodule
```

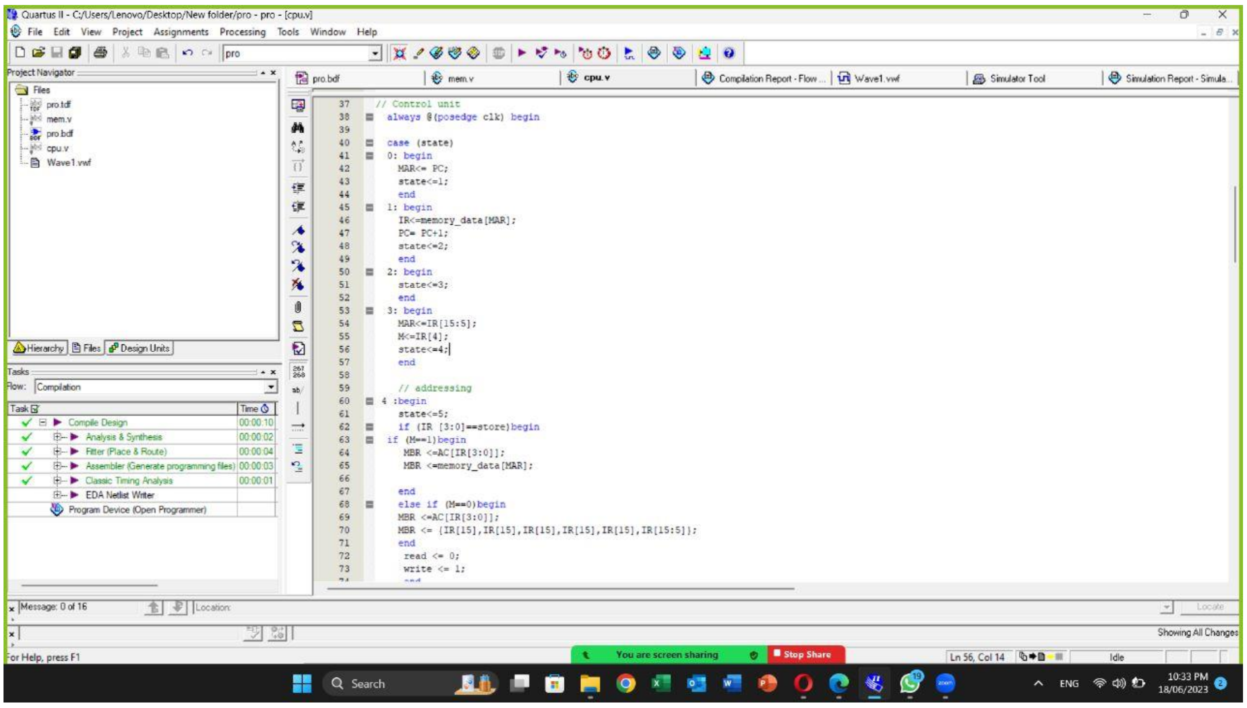


Memory block

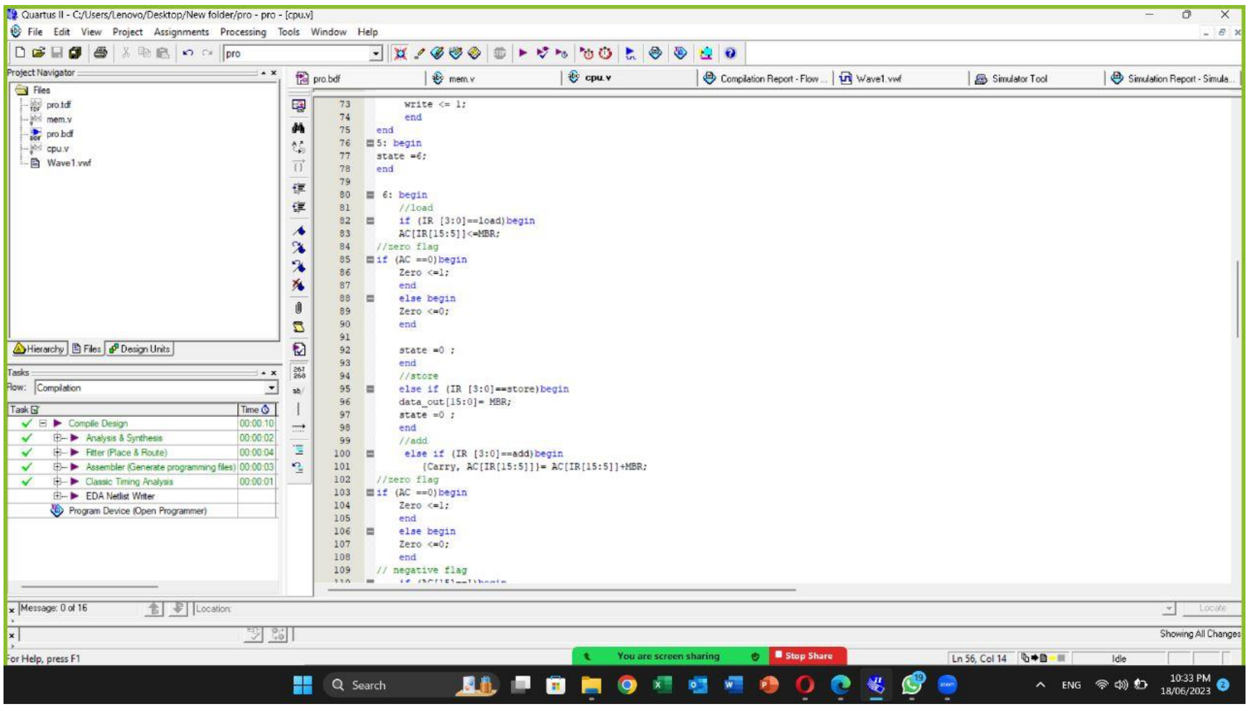
2) Design and implement CPU as a Verilog module:



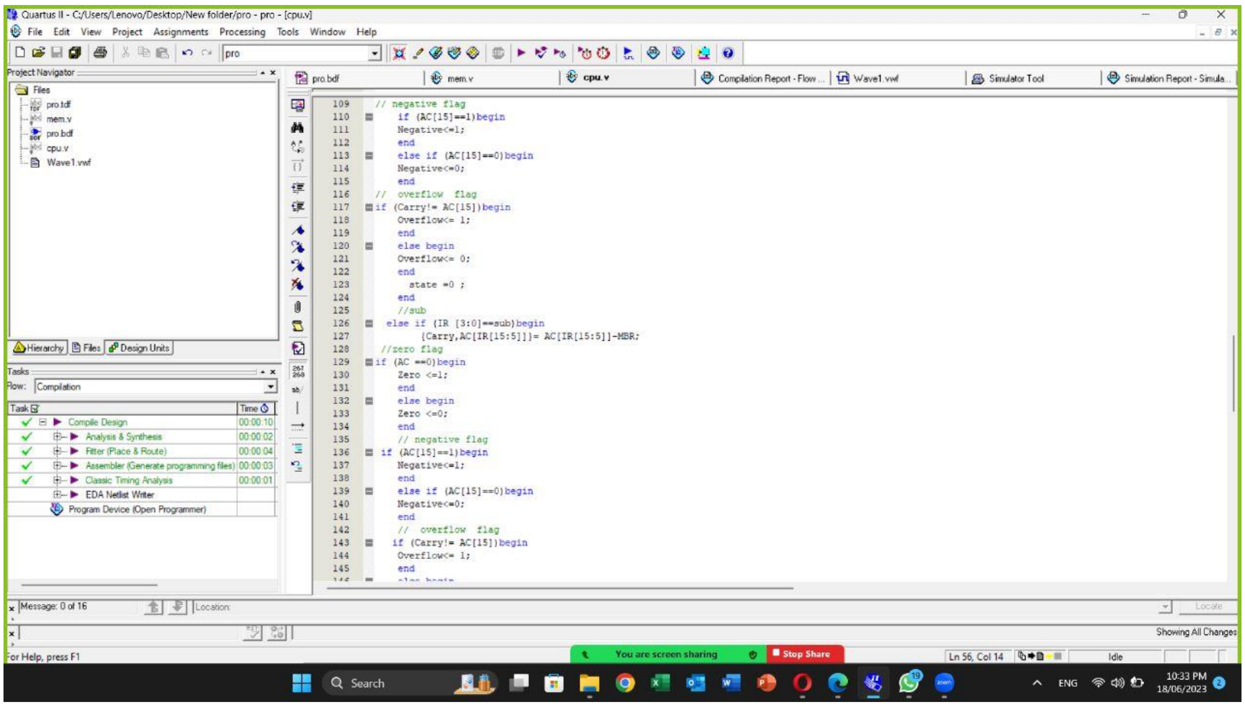
```
module cpu(
  input clk,
  input[15:0] memory_data ,
  output [10:0] memory_addr,
  output reg[15:0] data_out,
  output reg read,
  output reg write,
  inout reg [10:0] PC,
  inout reg [10:0] MAR,
  inout reg [15:0] MBR,
  inout reg [15:0] IR,
  inout reg [2:0] state,
  inout reg signed [15:0] AC,
  inout reg Carry,
  inout reg Zero,
  inout reg Overflow,
  inout reg Negative
);
reg signed[15:0] memory[0:63];
reg M;
reg[15:0] multReg;
// Control unit AND ALU control signals
  localparam load = 4'b0000;
  localparam store = 4'b0001;
  localparam add = 4'b0010;
  localparam sub= 4'b0011;
  localparam mul= 4'b0100;
  localparam div= 4'b0101;
  localparam branch= 4'b0110;
  localparam BRZ= 4'b0111;
//initialize according to the first address
 initial begin
  PC = 0 ;
  state = 0 ;
end

// Control unit
```



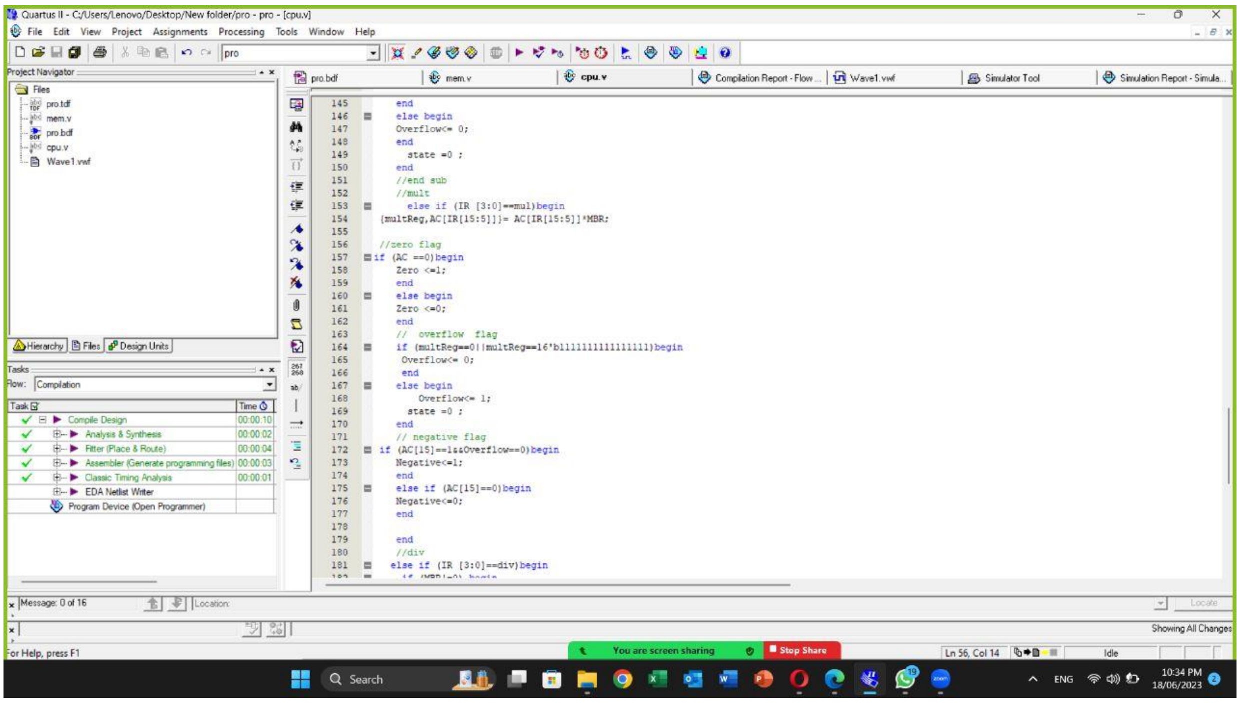
```
// Control unit
 always @(posedge clk) begin

 case (state)
 0: begin
   MAR<= PC;
   state<=1;
   end
 1: begin
   IR<=memory_data[MAR];
   PC= PC+1;
   state<=2;
   end
 2: begin
   state<=3;
   end
 3: begin
   MAR<=IR[15:5];
   M<=IR[4];
   state<=4;
   end

   // addressing
4 :begin
   state<=5;
   if (IR [3:0]==store)begin
 if (M==1)begin
    MBR <=AC[IR[3:0]];
    MBR <=memory_data[MAR];

   end
   else if (M==0)begin
   MBR <=AC[IR[3:0]];
   MBR <= {IR[15],IR[15],IR[15],IR[15],IR[15],IR[15:5]};
   end
    read <= 0;
    write <= 1;
```



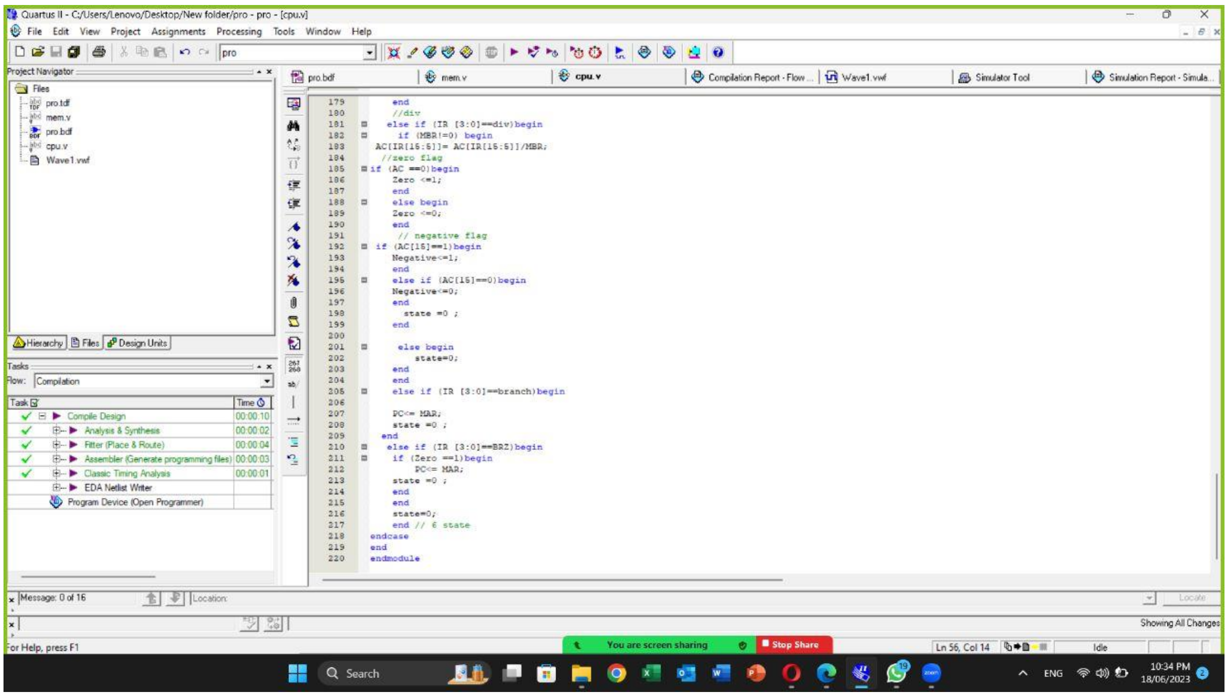
```
        write <= 1;
        end
    end
5: begin
    state =6;
    end

 6: begin
      //load
      if (IR [3:0]==load)begin
      AC[IR[15:5]]<=MBR;
    //zero flag
if (AC ==0)begin
      Zero <=1;
      end
      else begin
      Zero <=0;
      end

      state =0 ;
      end
      //store
      else if (IR [3:0]==store)begin
      data_out[15:0]= MBR;
      state =0 ;
      end
      //add
       else if (IR [3:0]==add)begin
          {Carry, AC[IR[15:5]]}= AC[IR[15:5]]+MBR;
    //zero flag
if (AC ==0)begin
      Zero <=1;
      end
      else begin
      Zero <=0;
      end
    // negative flag
```



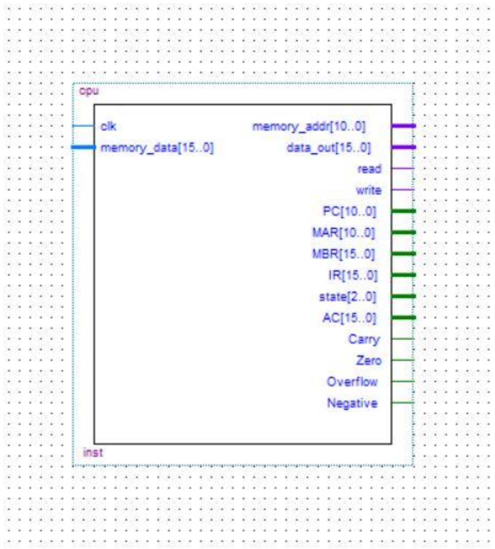
```
// negative flag
      if (AC[15]==1)begin
      Negative<=1;
      end
      else if (AC[15]==0)begin
      Negative<=0;
      end
//  overflow  flag
if (Carry!= AC[15])begin
      Overflow<= 1;
      end
      else begin
      Overflow<= 0;
      end
        state =0 ;
      end
      //sub
  else if (IR [3:0]==sub)begin
          {Carry,AC[IR[15:5]]}= AC[IR[15:5]]-MBR;
 //zero flag
if (AC ==0)begin
      Zero <=1;
      end
      else begin
      Zero <=0;
      end
      // negative flag
 if (AC[15]==1)begin
      Negative<=1;
      end
      else if (AC[15]==0)begin
      Negative<=0;
      end
      //  overflow  flag
    if (Carry!= AC[15])begin
      Overflow<= 1;
      end
```


Quartus II - C:/Users/Lenovo/Desktop/New folder/pro - pro - [cpu.v]
File Edit View Project Assignments Processing Tools Window Help
pro.bdf
mem.v
cpu.v
Compilation Report - Flow ...
Wave1.vwf
Simulator Tool
Simulation Report - Simula...
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
Classic Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)
You are screen sharing
Stop Share
Ln 56, Col 14
Idle
10:34 PM
18/06/2023

```
    end
    else begin
    Overflow<= 0;
    end
      state =0 ;
    end
    //end sub
    //mult
      else if (IR [3:0]==mul)begin
 {multReg,AC[IR[15:5]]}= AC[IR[15:5]]*MBR;

 //zero flag
if (AC ==0)begin
    Zero <=1;
    end
    else begin
    Zero <=0;
    end
    //  overflow  flag
    if (multReg==0||multReg==16'b1111111111111111)begin
     Overflow<= 0;
     end
    else begin
       Overflow<= 1;
      state =0 ;
    end
    // negative flag
 if (AC[15]==1&&Overflow==0)begin
    Negative<=1;
    end
    else if (AC[15]==0)begin
    Negative<=0;
    end

    end
    //div
   else if (IR [3:0]==div)begin
```


Quartus II - C:/Users/Lenovo/Desktop/New folder/pro - pro - [cpu.v]
File Edit View Project Assignments Processing Tools Window Help
pro.bdf
mem.v
cpu.v
Compilation Report - Flow ...
Wave1.vwf
Simulator Tool
Simulation Report - Simula...
You are screen sharing
Stop Share
Ln 56, Col 14
Idle
10:34 PM
18/06/2023

```
    end
    //div
   else if (IR [3:0]==div)begin
     if (MBR!=0) begin
 AC[IR[15:5]]= AC[IR[15:5]]/MBR;
  //zero flag
if (AC ==0)begin
    Zero <=1;
    end
    else begin
    Zero <=0;
    end
     // negative flag
 if (AC[15]==1)begin
    Negative<=1;
    end
    else if (AC[15]==0)begin
    Negative<=0;
    end
      state =0 ;
    end

     else begin
        state=0;
    end
    end
    else if (IR [3:0]==branch)begin

    PC<= MAR;
    state =0 ;
  end
   else if (IR [3:0]==BRZ)begin
    if (Zero ==1)begin
        PC<= MAR;
    state =0 ;
    end
    end
    state=0;
    end // 6 state
endcase
end
endmodule
```



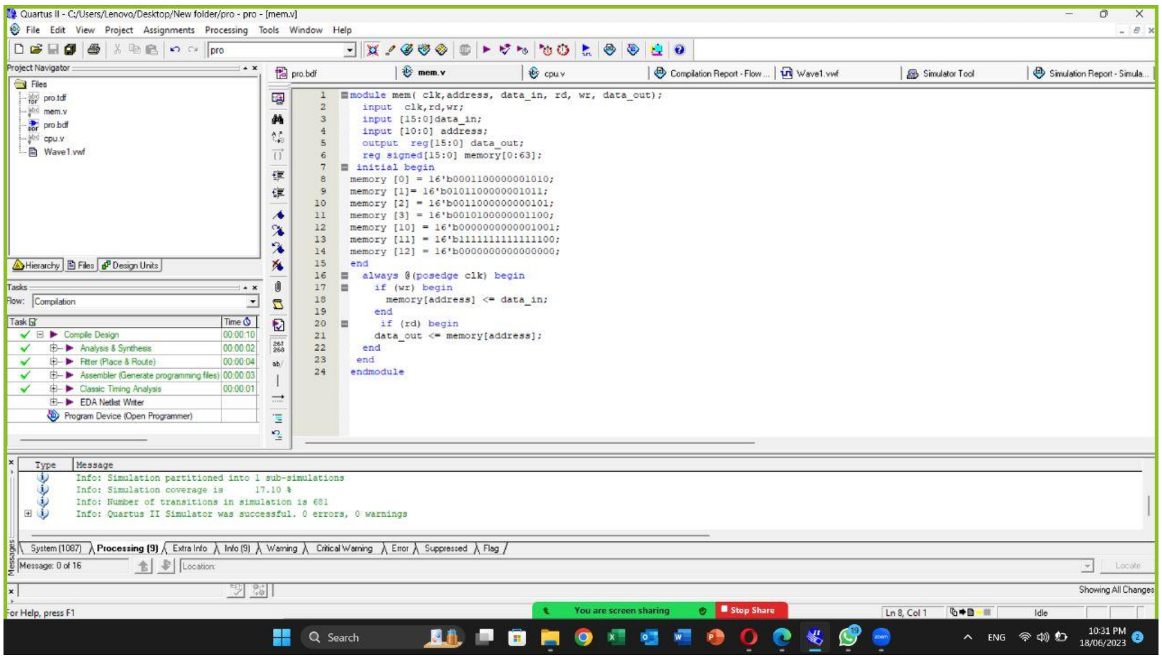
CPU block

- **Simulation:**

1- Initialize the memory with the following four instructions at memory address 0-3, and data at memory address 10-12, as shown in the following table.

   a. Interpret each instruction into assembly instruction and add it to its corresponding instruction in the table. Also, interpret the integer's data into decimal and add them into the table.

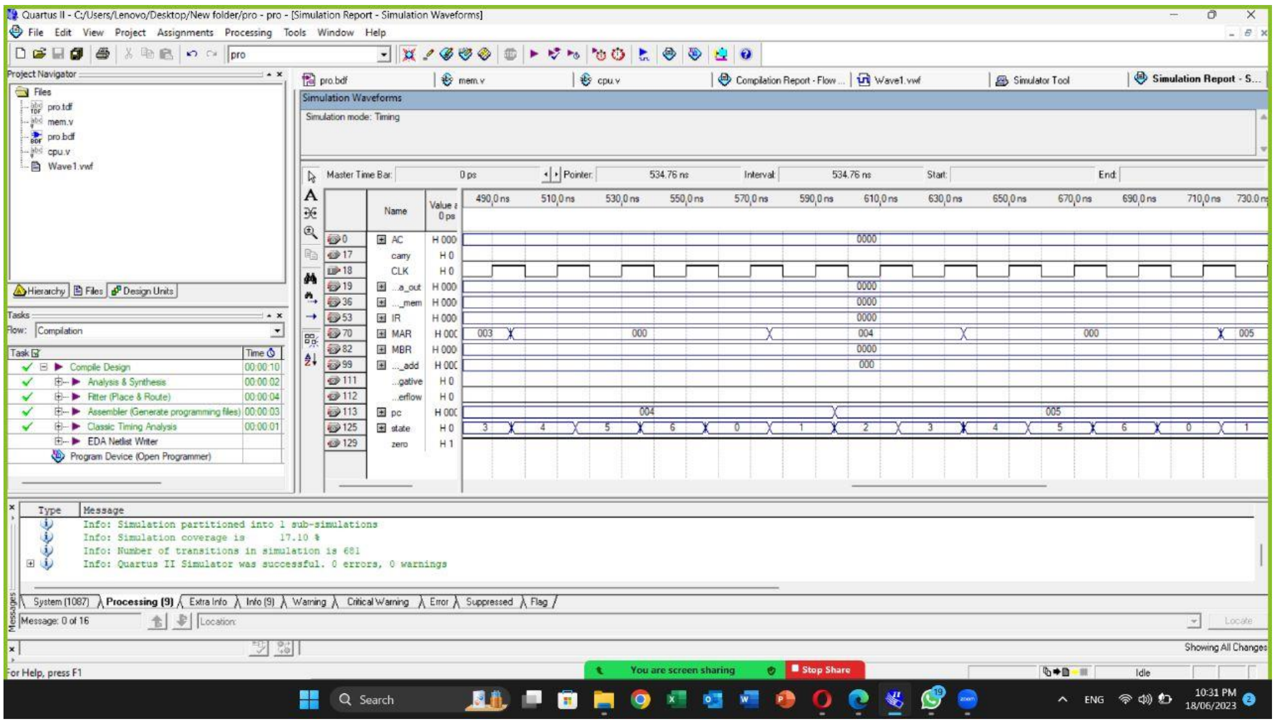
   b. Simulate the four instructions at address 0 by initializing PC =0. Provide a snapshot of your resulted waveform. Verify that it works correctly and the also verify that the result stored at address 12 is correct. Attach simulation waveform and the Verilog source file.


Quartus II - C:/Users/Lenovo/Desktop/New folder/pro - pro - [mem.v]
File Edit View Project Assignments Processing Tools Window Help
pro
Project Navigator
Files
pro.tdf
mem.v
pro.bdf
cpu.v
Wave1.vwf
Hierarchy
Files
Design Units
Tasks
Flow: Compilation
pro.bdf
mem.v
cpu.v
Compilation Report - Flow...
Wave1.vwf
Simulator Tool
Simulation Report - Simula...

Task	Time
Compile Design	00:00:10
Analysis & Synthesis	00:00:02
Fitter (Place & Route)	00:00:04
Assembler (Generate programming files)	00:00:03
Classic Timing Analysis	00:00:01
EDA Netlist Writer	
Program Device (Open Programmer)	

```
module mem( clk,address, data_in, rd, wr, data_out);
  input  clk,rd,wr;
  input [15:0]data_in;
  input [10:0] address;
  output  reg[15:0] data_out;
  reg signed[15:0] memory[0:63];
 initial begin
memory [0] = 16'b0001100000001010;
memory [1]= 16'b0101100000001011;
memory [2] = 16'b0011000000000101;
memory [3] = 16'b0010100000001100;
memory [10] = 16'b0000000000001001;
memory [11] = 16'b1111111111111100;
memory [12] = 16'b0000000000000000;
end
  always @(posedge clk) begin
    if (wr) begin
      memory[address] <= data_in;
    end
     if (rd) begin
    data_out <= memory[address];
  end
 end
endmodule
```

```
Type   Message
       Info: Simulation partitioned into 1 sub-simulations
       Info: Simulation coverage is      17.10 %
       Info: Number of transitions in simulation is 681
       Info: Quartus II Simulator was successful. 0 errors, 0 warnings
```



```
Type  Message
      Info: Simulation partitioned into 1 sub-simulations
      Info: Simulation coverage is      17.10 %
      Info: Number of transitions in simulation is 681
      Info: Quartus II Simulator was successful. 0 errors, 0 warnings
```

Memory Address	Content	Content interpretation: assembly instruction + data in decimal
0	0x180A	Load [10]
1	0x580B	Mull [11]
2	0x3005	Add 5
3	0x280C	Store [12]
10	0x0009	9
11	0xFFFC	-4
12	0x0000	0

2- Assume A,B,C,D,E and Y are memory cells with addresses 20,21,22,23,24, and 25, respectively. Given ,

$Y = \frac{A+B*C-5}{D+E+1}$,

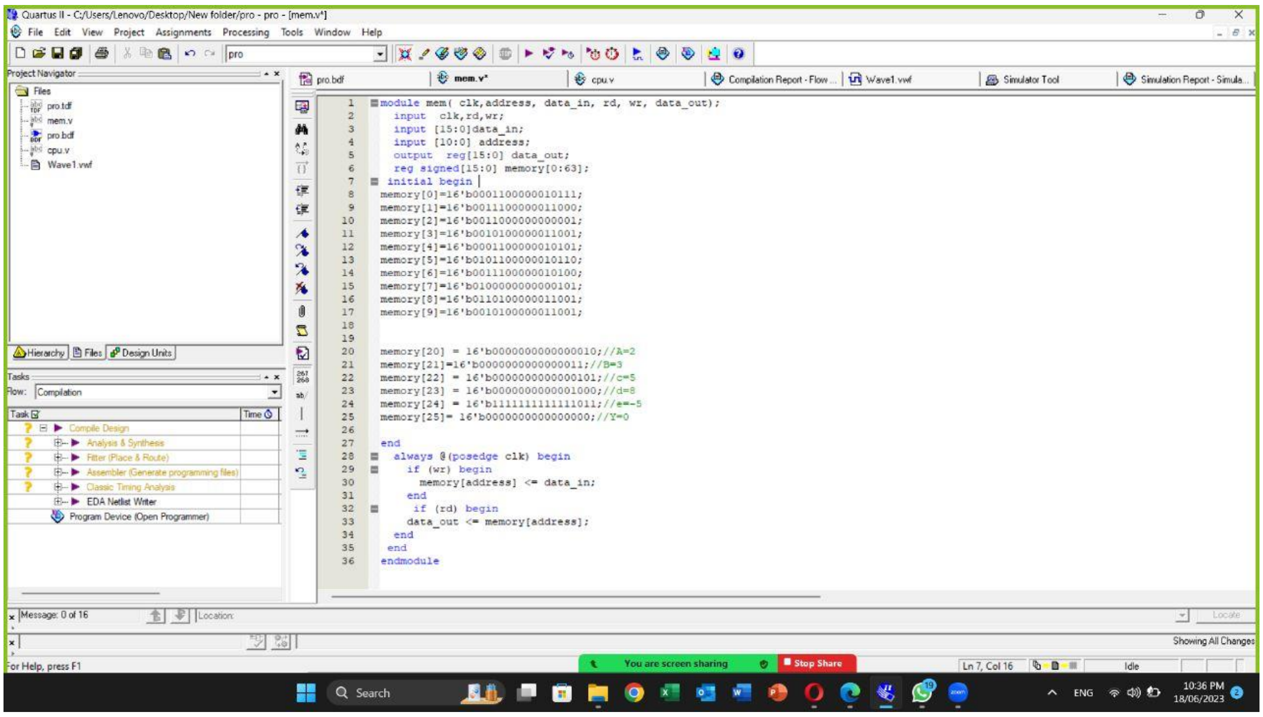
a. Write assembly code for implementing the above arithmetic expression?

```
Load [23]
Add [24]
Add 1
Store [25]
Load [21]
Mul [22]
Add [20]
Sub 5
Div [25]
Store [25]
```

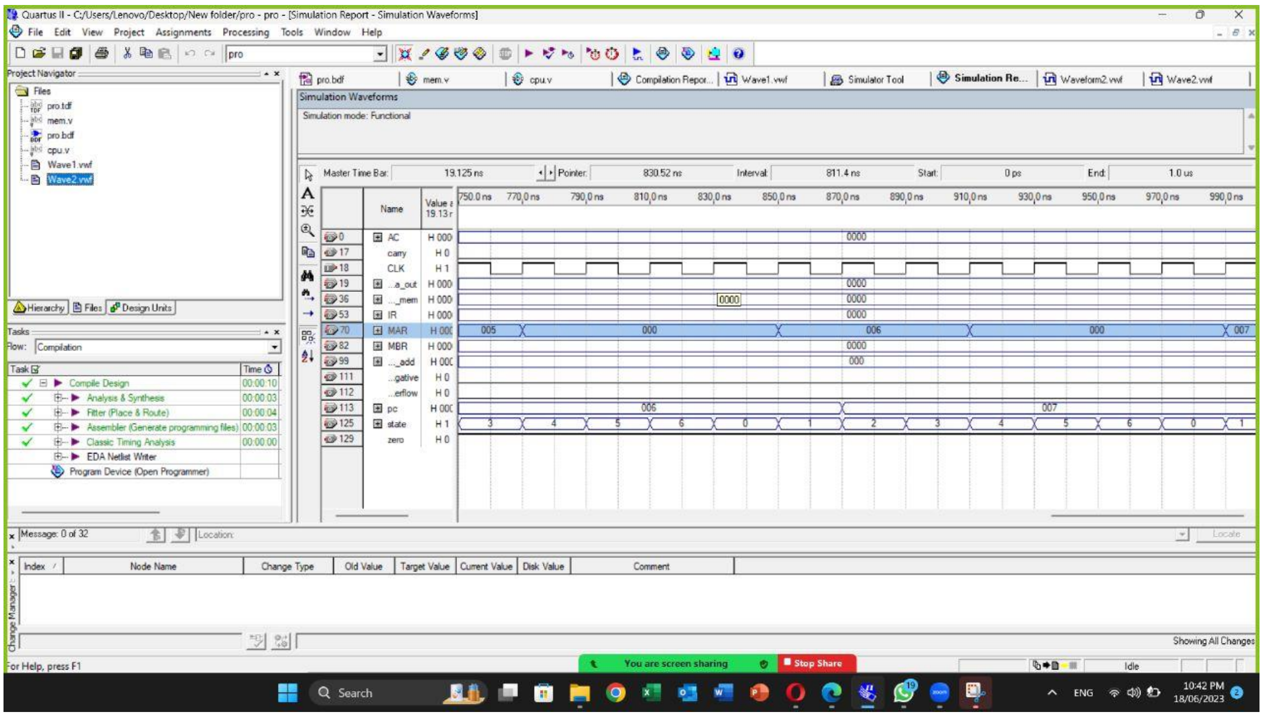
b. Convert the above assembly instructions into machine code and store them in the memory starting at address 0.

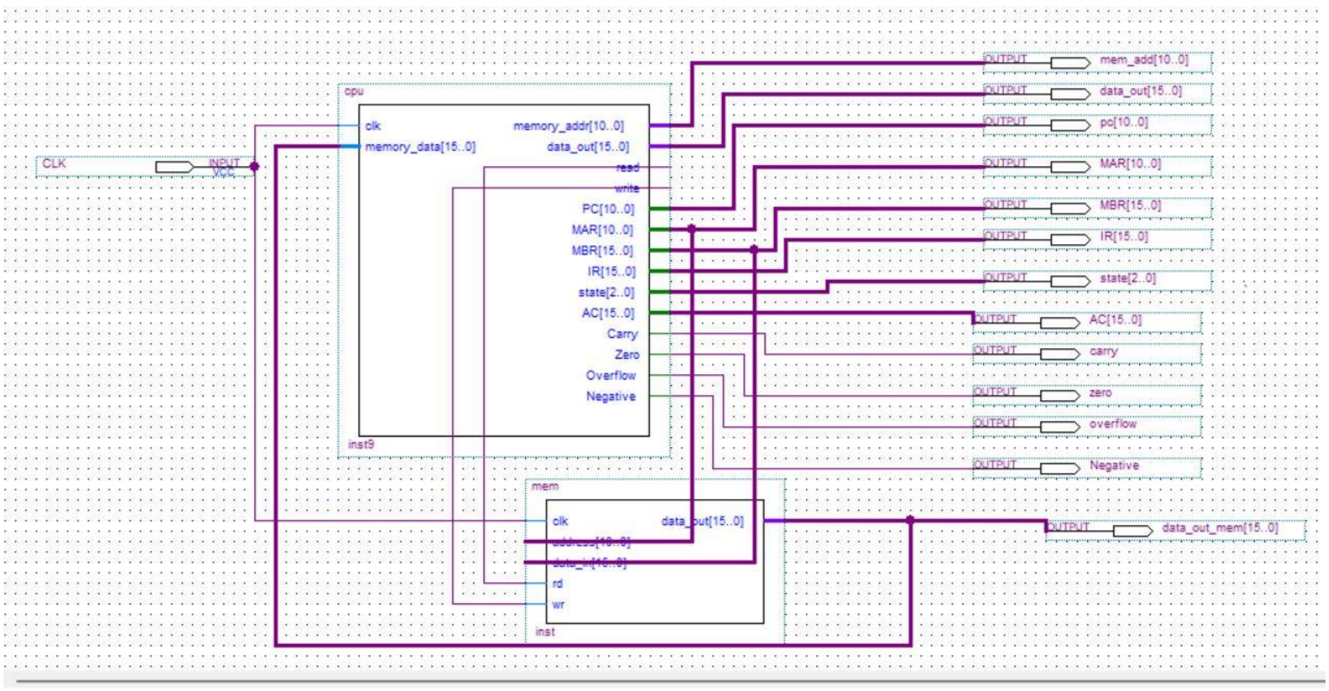
address	content
0	**0001 1 00000010111**
1	**0011 1 00000011000**
2	**0011 0 00000000001**
3	**0010 1 00000011001**
4	**0001 1 00000010101**
5	**0101 1 00000010110**
6	**0011 1 00000010100**
7	**0100 0 00000000101**
8	**0110 1 00000011001**
9	**0010 1 00000011001**
10	

c. Set PC=0 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 2, 3, 5, 8, and -5, respectively.


Quartus II - C:/Users/Lenovo/Desktop/New folder/pro - pro - [mem.v*]
File Edit View Project Assignments Processing Tools Window Help
Project Navigator
pro.tdf
mem.v
pro.bdf
cpu.v
Wave1.vwf
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
Classic Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)
mem.v*
cpu.v
Compilation Report - Flow ...
Wave1.vwf
Simulator Tool
Simulation Report - Simula...
Message: 0 of 16
Ln 7, Col 16
Idle
10:36 PM
18/06/2023

```verilog
module mem( clk,address, data_in, rd, wr, data_out);
  input  clk,rd,wr;
  input [15:0]data_in;
  input [10:0] address;
  output  reg[15:0] data_out;
  reg signed[15:0] memory[0:63];
 initial begin
memory[0]=16'b0001100000010111;
memory[1]=16'b0011100000011000;
memory[2]=16'b0011000000000001;
memory[3]=16'b0010100000011001;
memory[4]=16'b0001100000010101;
memory[5]=16'b0101100000010110;
memory[6]=16'b0011100000010100;
memory[7]=16'b0100000000000101;
memory[8]=16'b0110100000011001;
memory[9]=16'b0010100000011001;


memory[20] = 16'b0000000000000010;//A=2
memory[21]=16'b0000000000000011;//B=3
memory[22] = 16'b0000000000000101;//c=5
memory[23] = 16'b0000000000001000;//d=8
memory[24] = 16'b1111111111111011;//e=-5
memory[25]= 16'b0000000000000000;//Y=0

end
  always @(posedge clk) begin
    if (wr) begin
      memory[address] <= data_in;
    end
     if (rd) begin
    data_out <= memory[address];
  end
 end
endmodule
```


Quartus II - C:/Users/Lenovo/Desktop/New folder/pro - pro - [Simulation Report - Simulation Waveforms]
Simulation Waveforms
Simulation mode: Functional
Master Time Bar: 19.125 ns
Pointer: 830.52 ns
Interval: 811.4 ns
Start: 0 ps
End: 1.0 us
AC
carry
CLK
MAR
MBR
IR
pc
state
zero
Message: 0 of 32
10:42 PM
18/06/2023


CLK
INPUT
VCC
cpu
clk
memory_data[15..0]
memory_addr[10..0]
data_out[15..0]
read
write
PC[10..0]
MAR[10..0]
MBR[15..0]
IR[15..0]
state[2..0]
AC[15..0]
Carry
Zero
Overflow
Negative
inst9
OUTPUT
mem_add[10..0]
data_out[15..0]
pc[10..0]
MAR[10..0]
MBR[15..0]
IR[15..0]
state[2..0]
AC[15..0]
carry
zero
overflow
Negative
mem
clk
address[10..0]
data_in[15..0]
rd
wr
data_out[15..0]
inst
data_out_mem[15..0]

The codes for the module were written, the project was designed, and the simulation questions were solved collectively through Zoom meetings.