ASSIGNMENT

DESIGNING 4 bit PC using verilog HDL

COURSE ID : 415

ID : 1606038
EEE, Sec A

The following instructions were carried out by the Verilog Code :

Roll XX8

1	ADD A,B
2	SUB A,B
3	XCHG B,A
4	RCL B
5	SHR A
6	MOV [ADDRESS],A
7	XOR A,[ADDRESS]
8	AND A,B
9	OR B,[ADDRESS]
10	OUT A
11	JZ ADDRESS
12	PUSH B
13	POP B
14	CALL ADDRESS
15	RET
16	HLT

The following verilog was first constructed in the EDA playground and the finally simulated in the quartusII software.

```
module amar_4_bit_er_PC (
    clk,AX,BX,uiui,
    A_OUT,B_OUT,
    CARRY_FLAG,ZERO_FLAG,
    instruction,I_WANNA_SHOW_THE_OUTPUT,
    stack_pointer,index_pointer                        );
    input [3:0]AX;
    input [3:0]BX;
    input clk;
    input [3:0] instruction;

    output uiui;
    output reg [3:0]A_OUT,B_OUT,I_WANNA_SHOW_THE_OUTPUT;

    // all the flags
    output reg CARRY_FLAG;
    output reg ZERO_FLAG;
    output reg [3:0]stack_pointer;
    output reg [3:0]index_pointer;

    reg [3:0]SLACK_M[15:0];
    reg [3:0]INTERM;
    reg CARRY_IN;
    reg [3:0]instruction_no;
    reg [3:0]memo[15:0]; // for proper simulation of ram memory in PC
    reg address;




    always @(posedge clk)

      begin
        index_pointer = 4'b 0001;
        stack_pointer = 4'b 0010;
        instruction_no = instruction;
        CARRY_IN = 1'b0;
        address = 1'b 1;
        memo[address]= 4'b 0010;
        SLACK_M [4'b 0001] = 4'b 1001;
        SLACK_M[4'b 0010] = 4'b 1111;

        case(instruction_no)
//-------------------------------------------------
```

```
     case(instruction_no)
//-----------------------------------------------
       4'b 0001 : // 4 bit full adder ADD A,B
         begin
             {CARRY_FLAG,A_OUT} = AX + BX ;
             //CARRY_FLAG = (AX & BX)  |  (AX & CARRY_IN)  |  (BX & CARRY_IN);
             //instruction_no = instruction_no+ 1'b 1;
         end
//-----------------------------------------------
       4'b 0010 : // 4 bit subtraction SUB A,B
         begin
              INTERM = ~BX;
             {CARRY_FLAG,A_OUT} = AX + INTERM ;
             //CARRY_FLAG = (AX & BX)  |  (AX & CARRY_IN)  |  (BX & CARRY_IN);
             //instruction_no = instruction_no+ 1'b 1;
         end
//-----------------------------------------------
       4'b 0011 : // exchange
         begin
           A_OUT = BX;
           B_OUT = AX;
          //instruction_no = instruction_no+ 1'b 1;
         end
//-----------------------------------------------
       4'b 0100 : // 4. RCL B
         begin
            B_OUT[3:0] = {BX[2:0],BX[3]} ;
            CARRY_FLAG = BX[3];
           //instruction_no = instruction_no+ 1'b 1;
         end
//-----------------------------------------------
       4'b 0101 : // 5.SHR A
         begin
            A_OUT[2:0] = AX[3:1];
            A_OUT[3] = 0;
           CARRY_FLAG = AX[0];
           //instruction_no = instruction_no+ 1'b 1;
         end
//-----------------------------------------------
```

```
//-----------------------------------------------
       4'b 0110 : // 6. MOV [address] ,A
        begin
          memo[address] = AX;
          I_WANNA_SHOW_THE_OUTPUT = memo[address];
          //instruction_no = instruction_no+ 1'b 1;

        end
//-----------------------------------------------
       4'b 0111 : // 7. XOR  A,[ADDRESS]
        begin
          A_OUT = AX^memo[address];
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
       4'b 1000 : // 8. AND  A,B
        begin
          A_OUT = AX&BX;
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
       4'b 1001 : // 9. OR  B,[ADDRESS]
        begin
          B_OUT = BX|memo[address];
          //memo[address]= 4'b 0010
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
       4'b 1010 : // 10. OUT  A
        begin
           A_OUT = AX;
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
       4'b 1011 : // 11. JZ  ADDRESS
        begin
          if (ZERO_FLAG == 1) instruction_no = address;
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
       4'b 1100 : // 12. PUSH  B
        begin
          SLACK_M [4'b 0010] = BX;
          I_WANNA_SHOW_THE_OUTPUT = SLACK_M [4'b 0010];
          //instruction_no = instruction_no+ 1'b 1;
        end
//-----------------------------------------------
```

```
129     //-------------------------------------------------
130             4'b 1101 : // 13. pop  B
131             begin
132               B_OUT = SLACK_M [4'b 0010];
133               //SLACK_M[4'b 0010] = 4'b 1111;
134               //instruction_no = instruction_no+ 1'b 1;
135
136             end
137     //-------------------------------------------------
138           4'b 1110 : //14. CALL  ADDRESS
139             begin
140               SLACK_M[stack_pointer] = index_pointer;
141               // index_pointer = 4'b 0001;
142               index_pointer = address;
143               // address = 1'b 1;
144               stack_pointer = stack_pointer + 4'b 0001;
145
146             //instruction_no = instruction_no+ 1'b 1;
147             end
148     //-------------------------------------------------
149           4'b 1111: //ret
150           begin
151             stack_pointer = stack_pointer - 1 ;
152             index_pointer = SLACK_M[stack_pointer];
153             //instruction_no = instruction_no+ 1'b 1;
154           end
155
156     //-------------------------------------------------
157           default:
158           /* nothing happens here, and as the same case
159           every posedge, the outcome will be constant */
160           ;
161         endcase
162
163       end
164   endmodule
```

The outputs of the code will depend on the instructions provided through the wvf files in quartus.

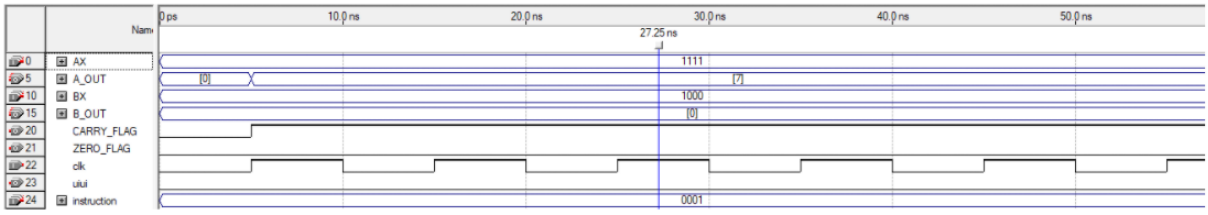
The following outputs are observed for the different instructions :

INSTRUCTION 1:

The code snippet for 1. ADD A,B

```
//------------------------------------------------
        4'b 0001 : // 4 bit full adder ADD A,B
          begin
              {CARRY_FLAG,A_OUT} = AX + BX ;
              //CARRY_FLAG = (AX & BX)  |  (AX & CARRY_IN)  |  (BX & CARRY_IN);
              instruction_no = instruction_no+ 1'b 1;
          end
```

The OUTPUT WAVE SHAPE



Here, the inputs were
A = 1111; B = 1000 ; on adding them , we get
A+B = 10111;
This means there will be a carry of 1 and the sum, being a 4 bit output, will be 0111 or 7.
As we can see in our simulation,
CARRY_FLAG = 1;
A_OUT = 7;
Therefore, the simulation is working properly.

Instruction 2 :

The code snippet for 2. SUB A,B

```
//------------------------------------------------
       4'b 0010 : // 4 bit subtraction SUB A,B
         begin
              INTERM = ~BX;
             {CARRY_FLAG,A_OUT} = AX + INTERM ;
             //CARRY_FLAG = (AX & BX)  |  (AX & CARRY_IN)  |  (BX & CARRY_IN);
             instruction_no = instruction_no+ 1'b 1;
         end
```

The OUTPUT WAVE SHAPE

Here, we observed the subtraction operation where,
A = 1111; B = 1000 ;

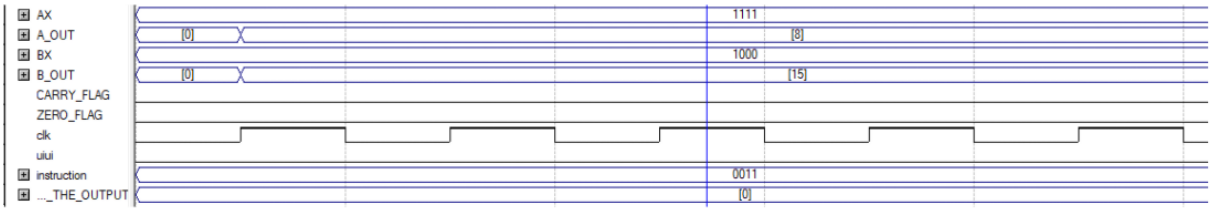
Therefore, Output A = 1111 - 1000 = 0111
Or, the output will be 7 ; and the carry flag will be 0.

Instruction 3:

The code snippet for 3. EXC A,B

```
//------------------------------------------------
        4'b 0011 : // exchange
          begin
            A_OUT = BX;
            B_OUT = AX;
           instruction_no = instruction_no+ 1'b 1;
          end
```

The OUTPUT WAVE SHAPE



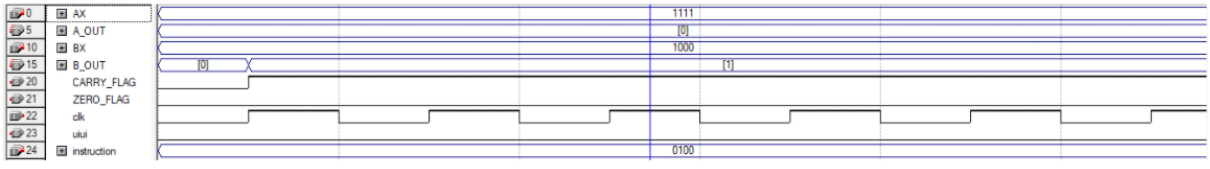
Here, the code was supposed to exchange the values of A and B.
As it can be seen in the waveform, the output
A_OUT = 8 = binary(1000) = B;
B_OUT = 15 = binary(1111) = A;

Instruction 4:

The code snippet for 4. RCL B

```
//------------------------------------------------
        4'b 0100 : // 4. RCL B
          begin
             B_OUT[3:0] = {BX[2:0],BX[3:2]} ;
             CARRY_FLAG = BX[3:2];
            instruction_no = instruction_no+ 1'b 1;
          end
```

The OUTPUT WAVE SHAPE



Here, B = 1000;
On Rotate Left, the left most bit will go to carry flag, and the output B will be replaced with binary(0001) or 1.

Here, from waveform we can see:
B_OUT = 1
CARRY FLAG = 1

Therefore, this is the required output.

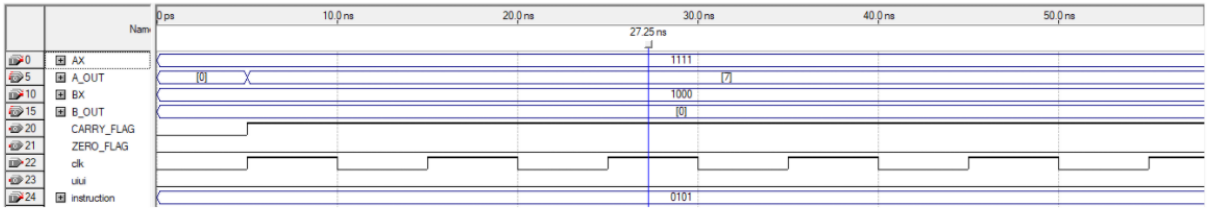
Instruction 5:

The code snippet for 5. SHR A

```
4'b 0101 : // 5.SHR A
  begin
     A_OUT[2:0] = AX[3:1];
     A_OUT[3:2] = 0;
    CARRY_FLAG = AX[0];
    instruction_no = instruction_no+ 1'b 1;

  end
```

The OUTPUT WAVE SHAPE



Here, A = 1111 will be shifter right and the following should be the output
A = 0111; where the right most bit will go to carry.

In my waveform diagram:

A_OUT = 7 = binary(0111);
CARRY FLAG = 1;

Therefore, this is the required output.

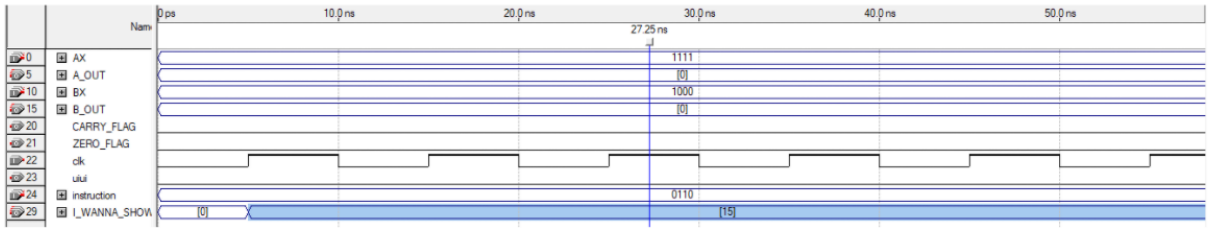
Instruction 6:

The code snippet for 6. MOV [address, A]

```
//------------------------------------------------
        4'b 0110 : // 6. MOV [address] ,A
         begin
           memo[address] = AX;
           I_WANNA_SHOW_THE_OUTPUT = memo[address];
           instruction_no = instruction_no+ 1'b 1;

         end
```

The OUTPUT WAVE SHAPE



To observe the validity of the command, the following values were taken for memo[address]:
Address = binary(1)
Memo[1] = binary(0010);
But, as the address was taken as a register, the output was shown through an output variable "I_WANNA_SHOW_THE_OUTPUT"

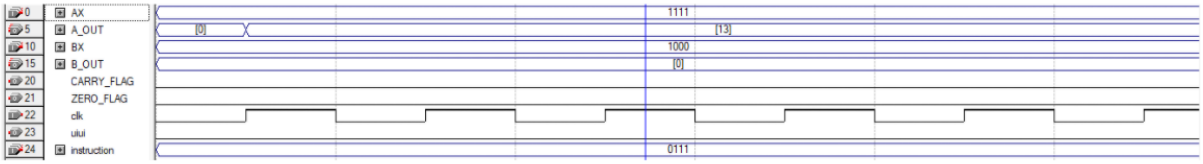
The following output was obtained:
I_WANNA_SHOW_THE_OUTPUT = 0010 (if we didn't execute the code)
I_WANNA_SHOW_THE_OUTPUT = binary (1111) = 15 = A

Instruction 7:

The code snippet for 7. XOR A, [address]

```
//------------------------------------------------
        4'b 0111 : // 7. XOR  A,[ADDRESS]
         begin
           A_OUT = AX^memo[address];
           instruction_no = instruction_no+ 1'b 1;
         end
```

The OUTPUT WAVE SHAPE



The truth table for XOR operation :

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Here,
A = 1111
Address = 0001

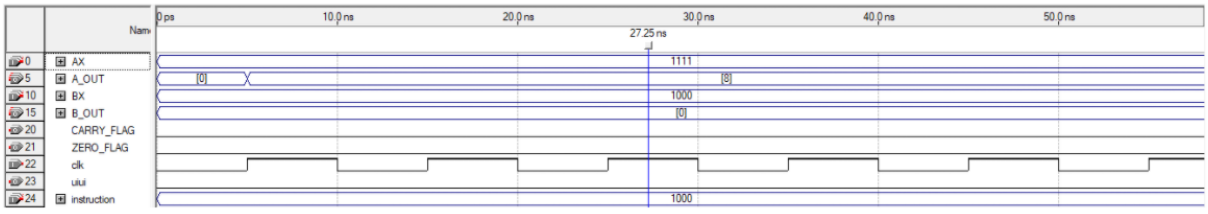
Output should be = 1110;
Which corresponds with the obtained output.
Therefore the code seems valid.

Instruction 8:

The code snippet for 8. AND A,B

```
4'b 1000 : // 8. AND  A,B
 begin
   A_OUT = AX&BX;
   instruction_no = instruction_no+ 1'b 1;
 end
```

The OUTPUT WAVE SHAPE



Since
A = 1111
B= 1000
Result should be = 1000
In decimal = 8

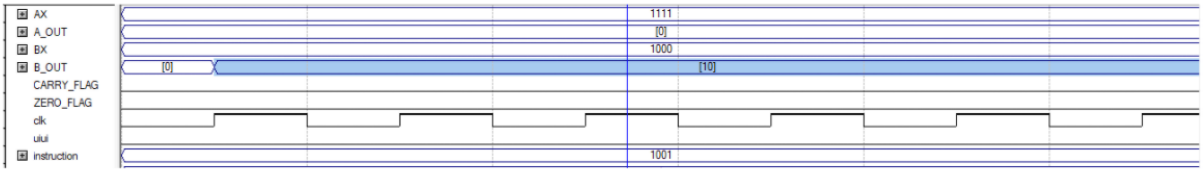
As it can be seen from output,A_OUT= 8, the code can be considered valid for this case.

Instruction 9:

The code snippet for 9. OR B, [ADDRESS]

```
4'b 1001 : // 9. OR  B,[ADDRESS]
 begin
   B_OUT = BX | memo[address];
   //memo[address]= 4'b 0010
   instruction_no = instruction_no+ 1'b 1;
 end
```

The OUTPUT WAVE SHAPE



Here,
B = 1000;
Memo[address] = 0010;
Output will be = 1010;
Output in decimal = 12.

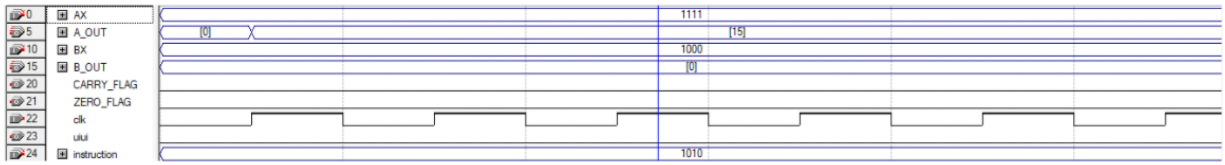
As it can be seen from the output waves, A_OUT = 12, the code can be considered valid.

Instruction 10:

The code snippet for 10. OUT A

```
4'b 1010 : // 10. OUT  A
 begin
    A_OUT = AX;
   instruction_no = instruction_no+ 1'b 1;
 end
```

The OUTPUT WAVE SHAPE



Here, the values in A is assigned to output.
A = 1111;
Output sould be = 1111 or 15 in decimal, which is obtained in wave diagram.

Instruction 11:

```
//------------------------------------------------
        4'b 1011 : // 11. JZ  ADDRESS
         begin
           if (ZERO_FLAG == 1) instruction_no = address;
           //instruction_no = instruction_no+ 1'b 1;
         end
//------------------------------------------------
```

The following code is made to make the next command execute the command 1.
Due to simulation device limitations, the execution time was taking too long and the results were not showable.On running the code, the remaining cases could not have been shown, as the command will perpetually move through the command 1 to 11.

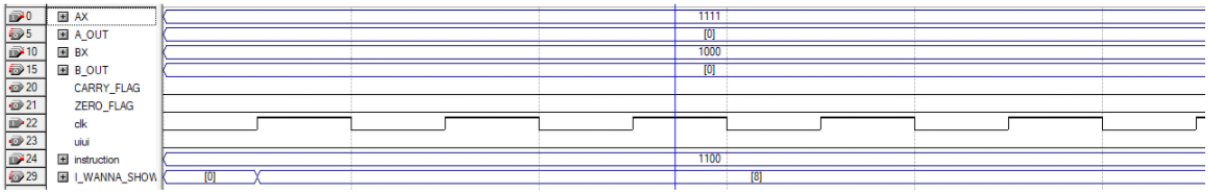
Instruction 12:

The code snippet for 12. PUSH B

```
4'b 1100 : // 12. PUSH  B
 begin
   SLACK_M [4'b 0010] = BX;
   I_WANNA_SHOW_THE_OUTPUT = SLACK_M [4'b 0010];
   instruction_no = instruction_no+ 1'b 1;

 end
```

The OUTPUT WAVE SHAPE



Here,
Input B = 1000;
This value was stored in the slack memory. To validate the assumption, we see the output of the slack memory in the same address in another variable

Here, I_WANNA_SEE_THE_OUTPUT = 8 = binary (1000) = B
Thus, this is the required output.

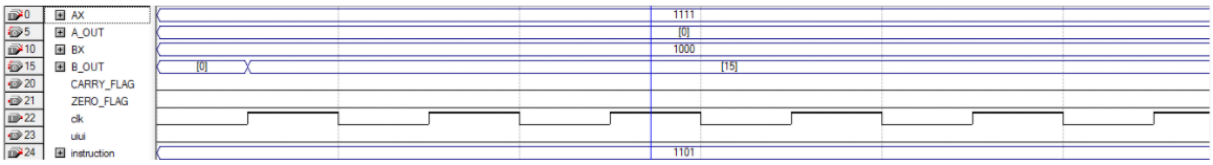
Instruction 13:

The code snippet for 13. POP B

```
4'b 1101 : // 13. pop  B
 begin
   B_OUT = SLACK_M [4'b 0010];
   //SLACK_M[4'b 0010] = 4'b 1111;
   instruction_no = instruction_no+ 1'b 1;

 end
```

The OUTPUT WAVE SHAPE



In the slack memory , the value at the certain address was defined beforehand to the see effectivity of the code.
Here,
B_OUT = 15 = binary(1111) = value stored in SLACK_M (4'b 0010)

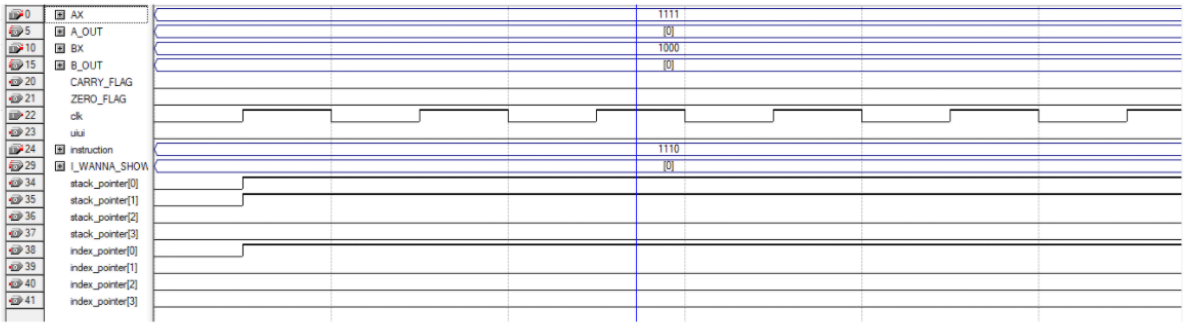
Instruction 14:

The code snippet for 14. CALL ADDRESS

```
4'b 1110 : //14. CALL  ADDRESS
  begin
    SLACK_M[stack_pointer] = index_pointer;
    // index_pointer = 4'b 0001;
    index_pointer = address;
    // address = 1'b 1;
    stack_pointer = stack_pointer + 4'b 0001;

  instruction_no = instruction_no+ 1'b 1;
  end
```

The OUTPUT WAVE SHAPE



Here, the index pointer is stored in slack memory and the address is stored in the index pointer.
Here, index pointer = 0001 (which is same as the address)

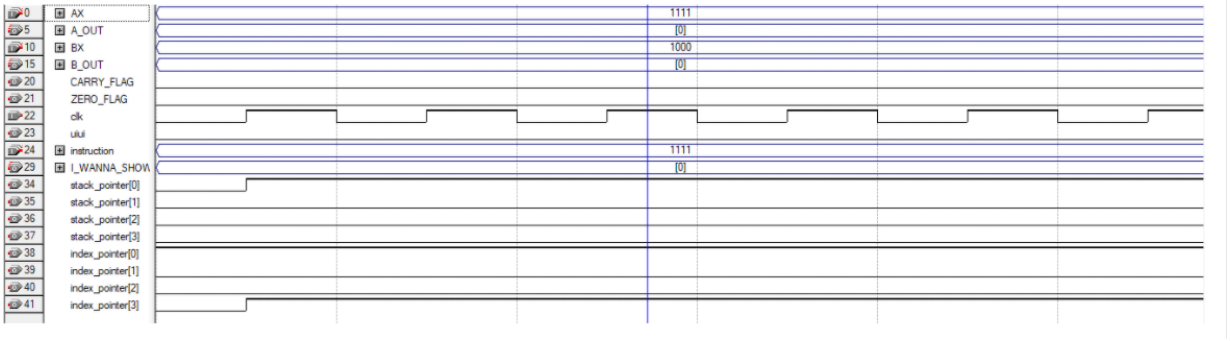
Slack pointer = 0011 which one more than the pre-defined slack pointer value 0010.

Instruction 15 :

The code snippet for 15. RET

```
4'b 1111: //ret
begin
  stack_pointer = stack_pointer - 1 ;
  index_pointer = SLACK_M[stack_pointer];
  instruction_no = instruction_no+ 1'b 1;
end
```

The OUTPUT WAVE SHAPE



Here, the stack pointer is retracted,
Stack pointer value = 0001 , which is 1 bit less than the pre-defined value 0010.
Index pointer stores the memory in the SLACK addressed by the slack pointer.
Here, data at slack memory in address 0001 is 1001.
As seen in waveform

Index_[pointer = 1001;
Slack pointer = 0001;
Hence, the outputs are what we expected.

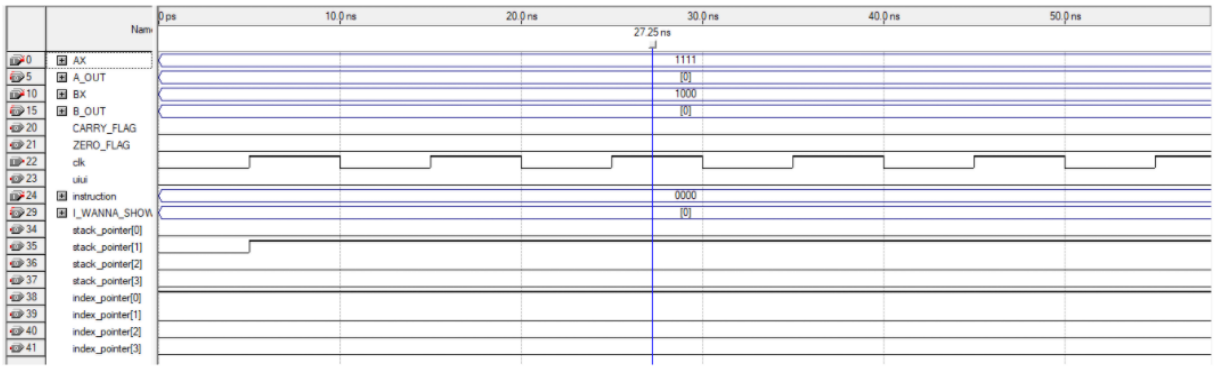
Instruction 16:

This instruction was executed through the default case. Here, the value sustains throughout the future iterations,

The code snippet for 16. HLT

```
default:
/* nothing happens here, and as the same case
every posedge, the outcome will be constant */
;
endcase
```

The OUTPUT WAVE SHAPE



The code executed all the commands asked in the requirements, separately, based on the command calls.