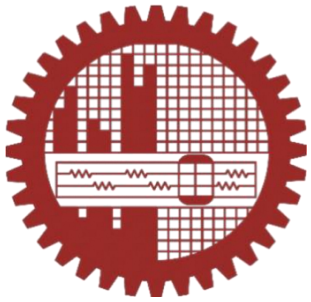
EEE – 415

Microprocessor And Embedded System



Assignment - 01

4-Bit Computer Design

Name: Raisa Bentay Hossain

Student's ID: 1606151

Section: C1

Problem

1. Implement a 4 bit computer in VerilogHDL with the given instruction set. The instruction set will depend on your student ID's last digit (please see the attached PDF file). Your implementation should be done in a single Verilog hdl file.
2. Write appropriate test conditions for your microprocessor to demonstrate its capabilities. You should generate vector waveform files / test bench codes.

Instruction Set

Roll XX1

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

Explanation of the instructions

ADD A, B: This will add the value of A register and B register and store the value in A register.

SUB A, B: It will subtract the value of B register from that of A register and store the value in A register.

XCHG B, A: It will exchange the values stored in A and B registers.

MOV A, [ADDRESS]: This will move the value stored in given address to A register.

OUT A: This will give the value stored in A register as the output.

INC A: It will increase the value stored in A register by 1.

RCR A: RCR means rotate with carry right. The value stored in register A's LSB will be the new value of carry while the old carry will be the new MSB of the value stored and other bits will shift right by 1 bit.

MOV A, BYTE: It will move the current address value to the A register.

JNZ ADDRESS: IF the zero flag has a zero value then the IP will jump to the given address.

PUSH B: Store the value of B register to the stack memory.

POP B: Retrieve the value of B register from the stack memory.

CALL ADDRESS: Storing the current memory address to the stack and changing the current memory to the given address.

RET: Fetching the memory address previously stored in stack memory and changing the current address to that.

OR A, [ADDRESS]: It will perform logical OR operation between the value stored in accumulator and that in the given address.

XOR A, [ADDRESS]: It will perform logical OR operation between the value stored in accumulator and that in the given address.

HLT: Stops all the procedures.

CODE (With Explanation)

```
module computer_4_bit(INPUT_A,INPUT_B,command,ADDRESS,OUTPUT_A,OUTPUT_B,ZF,CF,clk,reset);

input [3:0]INPUT_A,INPUT_B;
//input [3:0]command;
input [3:0]ADDRESS;
output reg [3:0]OUTPUT_A,OUTPUT_B;
output reg ZF, CF;

input clk, reset;
output reg [3:0]command;

reg [3:0]AX;
reg [3:0]BX;
reg [15:0]SI;
reg [3:0]stack_memory[15:0];
reg [15:0] IP, SP;
reg [3:0]temp;
reg hlt;
```

Inputs, outputs and registers:

In this 4 bit computer I have taken two 4 bit inputs. The 4 bit command should be taken as an input to execute specific instruction. But here it is taken as an output for convenience to show the output, as we will vary the value of command from 1 to 16 in order to present all the instructions in one go. The 4 bit

address taken as input will mainly indicate the address the program will start from (org 100h). But it is also used to indicate the given address for some instructions (instruction no 8 and 11).

To show the outputs we have two 4 bit outputs. And for flags there are only zero flag and carry flag. Other inputs are clock and reset.

For registers, we have two 4 bit registers which will temporary store the values of the inputs (AX and BX) and one 16 bit memory register (SI) and one 16 bit 2D stack memory. IP will indicate the current address and SP is the tack pointer.

```
always @(posedge clk)
 begin
     SI[3:0] = 4'b0100;
     SI[7:4] = 4'b0110;
     SI[11:8] = 4'b1000;
     SI[15:12] = 4'b1010;

   if (reset == 1)
    begin
        command = -1;
        hlt = 0;
        IP = ADDRESS;     //pointer for SI
        SP = 15;          //pointer for stack_memory
        AX = INPUT_A;
        BX = INPUT_B;
        ZF = 0;
        CF = 0;
        OUTPUT_A = 4'bZZZZ;
        OUTPUT_B = 4'bZZZZ;
    end
```

Initializing the values:

Some garbage values are kept in the stack memory initially. When reset is 1 all the register values will be initialized. Initially the IP pointer will be at the input address, the stack pointer SP will be at the last bit of the stack memory, AX and BX registers will store the given inputs.

```
else if (reset == 0 && hlt == 0)
   begin
     IP = IP + 1;
     command = command + 1;
        case(command)
        00: begin                          // ADD AX,BX
             {CF,OUTPUT_A} = AX+BX;
             AX = OUTPUT_A;
             ZF = (OUTPUT_A==0)? 1:0;
            end
        01: begin                          // SUB AX,BX
             {CF,OUTPUT_A} = AX-BX;
             AX = OUTPUT_A;
             ZF = (OUTPUT_A==0)? 1:0;
            end
        02: begin                          // XCHG AX,BX
             temp = AX;
             AX = BX;
             BX = temp;
             OUTPUT_A = AX;
             OUTPUT_B = BX;
             ZF = (OUTPUT_A==0)? 1:0;
            end
        03: begin                          // MOV AX, [ADDRESS]
             OUTPUT_A = SI[IP];    // Using IP to indicate the current address
             AX = OUTPUT_A;
             ZF = (OUTPUT_A==0)? 1:0;
            end
```

Executing the INSTRUCTIONS:

If the value of reset is 0 with hlt ==0, the code will start executing the instructions. IP was initially at input address, which will increase by 1 with each clock cycle, value of 'command' will increase by 1 (from -1 to 15) so that we can excess each instruction.

Inside the case index,

When command = 0, it will add the values of **AX=INPUT_A** and **BX=INPUT_B**. the carry will be stored in CF(carry flag). If the output is zero, the ZF(zero flag) will be 1. Then, **AX=INPUT_A + INPUT_B.**

When command = 1, the subtraction of **AX=INPUT_A + INPUT_B** and **BX=INPUT_B** will be happened, the CF and ZF will work as same. Finally, **AX=INPUT_A.**

For command = 2, the values of AX and BX will be exchanged. So after executing the command the values will be, **AX=INPUT_B and BX=INPUT_A.**

For command = 3, output_A will take the value from stack_memory, IP indicates the current address, which will be input address + number of previous instruction executed. **AX=SI[IP_STEP_3].**

```
04: begin                       // OUT AX
     OUTPUT_A = AX;
     ZF = (OUTPUT_A==0)? 1:0;
    end
05: begin                       // INC AX
     {CF, OUTPUT_A} = AX+1;
     AX = OUTPUT_A;
     ZF = (OUTPUT_A==0)? 1:0;
    end
06: begin                       // RCR AX
     OUTPUT_A[3:0]={CF, AX[3:1]};
     CF = AX[0];
     AX = OUTPUT_A;
     ZF = (OUTPUT_A==0)? 1:0;
    end
07: begin
     AX = IP;                 // MOV AX, BYTE
     OUTPUT_A = AX;
     ZF = (OUTPUT_A==0)? 1:0;
    end
08: begin                       // JNZ ADDRESS
     if (ZF == 0)
      IP = ADDRESS;
      OUTPUT_B = IP;            // using B to show the value of IP
    end
09: begin                       //PUSH BX
     stack_memory[SP] = BX;
     OUTPUT_A = stack_memory[SP]; // using A to show the value of stack_memory
     SP = SP - 1;
     OUTPUT_B = SP;               // using B to show the value of SP
    end
```

Command values of 4 will give **OUTPUT_A = AX=SI[IP]** and command 5 will increase the value of AX by 1, or **OUTPUT_A = AX= SI[IP_STEP_3]+ 1.**

For command = 6, the value stored in **AX** which is **SI[IP_STEP_3]+ 1** will be rotated to right.

For command 7, the value stored in register AX will be changed to the **current address** and the OUTPUT_A will show the value of AX. So now, **AX = ADDRESS + no of instructions** executed previously.

Command=8, if the zero flag = 0, the IP will be set to a specific address, in this case it will be the input address, ADDRESS. To make sure the process has taken place successfully, we have used OUTPUT_B to show the value of IP.

Command 9, value in **BX = INPUT_A** will be stored in stack_memory and stack poinyter SP will decrease by 1. To show that the process taken place we will show the value stored in OUTPUT_A and the value of SP at OUTPUT_B.

```
10: begin
     SP = SP + 1;         // POP BX
     BX = stack_memory[SP];
     OUTPUT_B = stack_memory[SP];
     OUTPUT_A = SP;             // using OUTPUT_A to show the value SP
     ZF = (OUTPUT_B==0)? 1:0;
    end
11: begin                       // CALL ADDRESS
     OUTPUT_B = IP;             // CURRENT IP IN B
     stack_memory[SP] = IP;
     OUTPUT_A = stack_memory[SP]; // VALUE STORED IN SP
     IP = ADDRESS;
     SP = SP - 1;
    end
12: begin                       // RET
     SP = SP + 1;
     OUTPUT_A = stack_memory[SP]; // CHECK THE VALUE IN MEMORY
     IP = stack_memory[SP];
     OUTPUT_B = IP;               // CHECK THE VALUE IS TRANSFERRED
    end
13: begin                     // OR AX, [ADDRESS]
     OUTPUT_B = IP;
     OUTPUT_A= AX| SI[IP];   // Using IP to indicate the current address
     AX = OUTPUT_A;
     ZF = (OUTPUT_A==0)? 1:0;
    end
14: begin
     OUTPUT_B = IP;                    // XOR AX, [ADDRESS]
     OUTPUT_A = AX^ SI[IP];   // Using IP to indicate the current address
     AX = OUTPUT_A;
     ZF = (OUTPUT_A==0)? 1:0;
    end
15: hlt = 1;                     // HLT
```

Command 10, the stack pointer will reduce by one and the BX register will have its previous value, **BX = INPUT_A.** The value will be shown in OUTPUT_B. OUTPUT_A will show the value of stack counter.

Command 11, store the current memory address (IP) to the stack and changing the current memory to the given address. OUTPUT_B will show the current IP and OUTPUT_A will show the stored value; these two should be the same.

Command 12, it will fetch the memory address previously stored in stack memory (shown in pervious OUTPUT_A) and changing the current address to that.

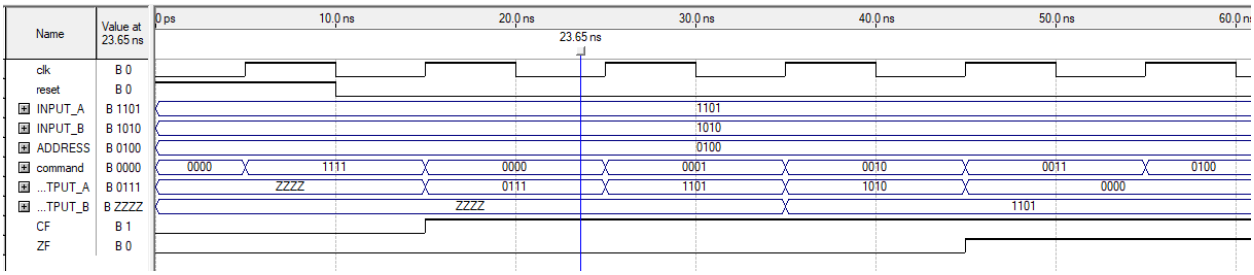
Command 13, logical OR operation between **AX= RCR(ADDRESS + no of previous instructions_step_7)** and value stored in stack memory for value shown by OUTPUT_B. The new AX will be,

AX = (ADDRESS + no of instructions) | value stored in stack memory.

Command 14, logical XOR operation between **AX in previous step** and value stored in stack memory for value shown by OUTPUT_B.

Command 15, the code will be halted.

Output (With Explanation)



Here INPUT_A = 1101, INPUT_B = 1010 and ADDRESS= 0100.

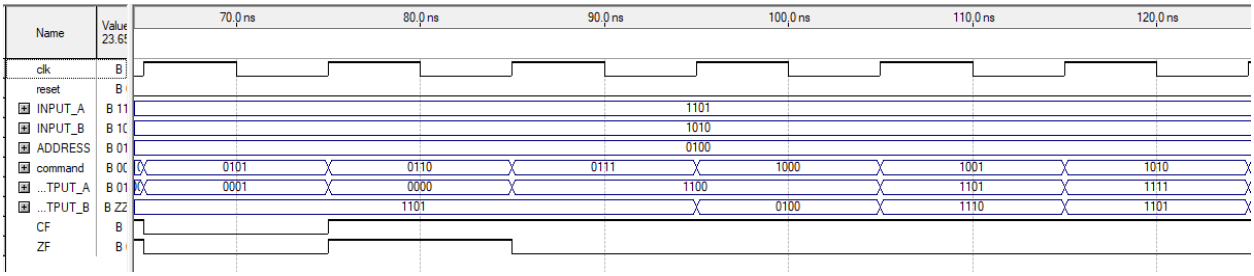
Command 0000, INPUT_A + INPUT_B = 1101 + 0100 = 1 0111. So, AX = OUTPUT_A = 0111, CF = 1;

Command 0001, OUTPUT_A = AX – BX = 0111 – 0100 = 1101.

Command 0010, OUTPUT_A = AX = INPUT_B = 1010, OUTPUT_B = BX = 1101.

Command 0011, OUTPUT_A = AX= SI[IP] = SI[0111]= SI[7] =0000. ZF =1;

Command 0100, OUTPUT-A = AX =0000. AX and BX will not change. AX= 0000 and BX = 1101. ZF =1.



Command 0101, AX = OUTPUT_A = AX +1 = 0000 +1 = 0001, BX=1101, same as before. CF= 0. ZF=0.

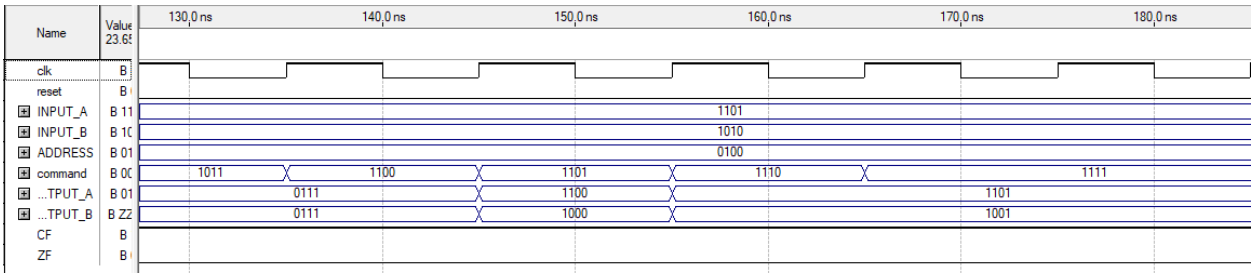
Command 0110, AX = OUTPUT_A = {CF, AX[3:1]} = 0000, CF= AX[0] = 1; ZF =1; BX=1101.

Command 0111, AX = OUTPUT_A = IP = 0100B + 8D = 1100B. ZF=0.

Command 1000, OUTPUT_B = IP = ADDRESS= 0100.

Command 1001, OUTPUT_A = stack_memory[SP] = stack_memory[15] = BX =1101 ;OUTPUT_B = SP -1 = 14D = 1110.

Command 1010, OUTPUT_A= SP =15 =1111; OUTPUT_B = stack_memory[SP] = stack_memory[15] =1101.



Command 1011, OUTPUT_A = OUTPUT_B = 0111.

Command 1100, same value of both output proves the process is executed properly.

Command 1101, OUTPUT_B = IP = 1000; OUTPUT_A= AX| SI[IP] = 1100 | SI[8] = 1100 | 0000 =1100.

Command 1110, OUTPUT_B = IP = 1000+1 = 1001; OUTPUT_A= AX ^ SI[IP] = 1100 ^ SI[9] =

1100 ^ 0001=1101. As 1^ 0 =1, 0^ 0 = 0 ; 0^1 = 1.

Special Feature

The zero and carry flag is the extra added features, also the steak memory and the steak pointer can be considered something other than required.