Bangladesh University of Engineering and Technology

Department of Electrical and Electronic Engineering

Course Number: *EEE 415*

Course Title: *Microprocessor and Embedded Systems*

Assignment

4-bit PC Using Verilog-HDL

Date of Submission: 1 April, 2021

Table of Contents

Table of Figures

- ## Objective:

The objective of this assignment is to design a 4-bit PC using Verilog-HDL.

- ## Methodology:

A 4-bit computer can work with 4 data-bits for its various operations including addition, subtraction and so on. It consists of several parts to achieve this including program counter, accumulator, instruction registers, general purpose registers, stack registers, etc. The instructions fed into the computer are 8-bit long; with the upper 4-bits i.e. upper nibble represents the operation code, which is basically the program counter for case-based operations. The implemented algorithm is to some extent similar to the Simple As Possible (SAP)-1 architecture, with reduced complexity but enhanced calculative capability.

In this assignment, I was given to perform 16 operations according to the last digit of my student ID. They are:

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

To achieve this goal, the implemented algorithm has been the following:

- Registers A and B (in the code, regs[0] and regs[1], in order) shall contain the values
- All the instructions shall be kept in a 16x8 program registers
- Each instructions will be 8-bit long, with the upper nibble being the opcode
- All the data shall be kept in another 16x8 data registers

- 16 case-based operations shall be performed depending on the opcode
- Zero, sign and carry flags shall contain the updated flag values

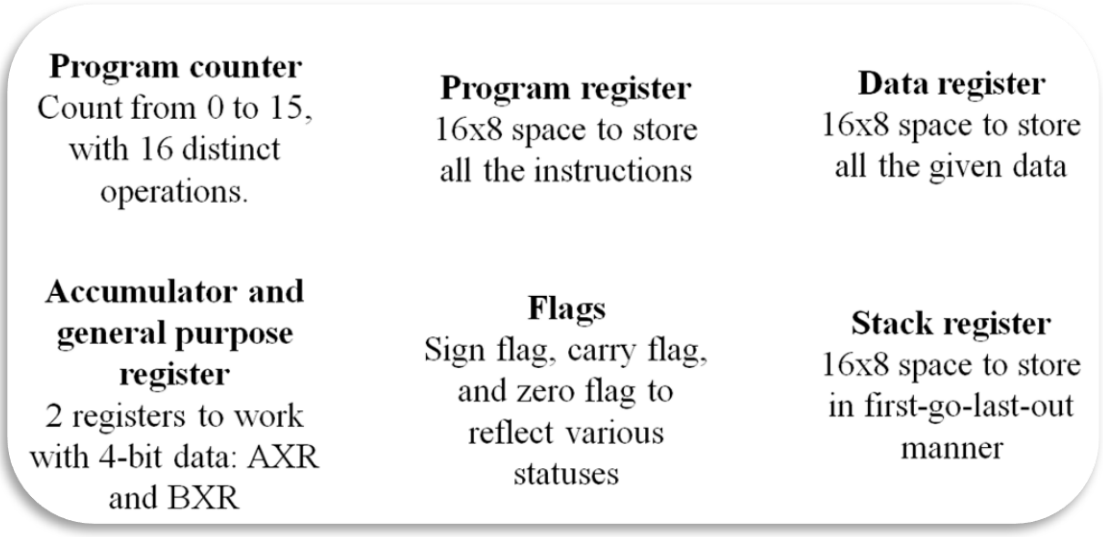

Figure 1 Various components of the applied method

- **Code:**

```
module FourBitCPU(clk_load, reset, clk, myprogram, myinput,
progReg, dataStore, stackReg, myoutput, HLT, s_flag, z_flag,
c_flag);

     parameter n=4;

     input clk_load, reset, clk; // program, data shall be
loaded with clk_load and the computer shall

                                          // with clk


     input [7:0] myprogram;               // Instructions and
memory addresses that

                                                    // are
clk_loaded before a computer run

     input [n-1:0] myinput;                         // Data
that are clk_loaded before a computer run
```

```
      reg [n-1:0] regs[1:0]; // registers, where regs[0]=A and
regs[1]=B; In this way it can be easily extended.

      output reg [7:0] progReg[0:15]; // instructions shall be
loaded here.

      output reg [n-1:0] dataStore[0:15]; // data shall be loaded
here.

                                                              //
responsible for address-based operations

      output reg [n-1:0] stackReg[0:15]; // stack registers shall
contain the push operations



      output reg [n-1:0] myoutput; // output shall be shown here
under OUT operation

      output reg HLT, s_flag, z_flag, c_flag;     // status flags:
sign, zero and carry



      reg [n-1:0] stackPoint, addPoint, temp;     // temporary
storage, opcode, address and stack pointers



      integer progCount = 0; // Program counter, counting from 0
to 15

      integer i = 0;



      // initializing

      initial

      begin

            {s_flag, c_flag, z_flag, HLT} = 0;

            stackPoint = 4'b1110;

            regs[0] = 4'b0111;

            regs[1] = 4'b0011;

      end
```

```
      // instructions, memory addresses are being loaded --
before the computer starts

      always @(posedge clk_load)

      begin

            if(i < 16)

            begin

                  progReg[i] = myprogram;

                  dataStore[i] = myinput;

                  i = i + 1;

            end

      end


      // Computer shall operate at this portion, after all the
instructions are loaded.

      always @(posedge clk)

      begin

            if(progCount < 16 && !HLT) // for 16 operations,
counter shall count up to 16 or until HLT is given


                  addPoint = progReg[progCount][3:0];    // lower
nibble represents the address


                  begin

                        casez(progReg[progCount]) //casez is
considered since the lower nibble are don't care here.


                              //ADD regs[0],regs[1]

                              8'b0000????:

                              begin
```

```
                              {c_flag, regs[0]} = regs[0] +
regs[1];
                              z_flag = (regs[0] == 0)? 1:0;

                        end

                        //SUB regs[0],regs[1]
                        8'b0001????:
                        begin
                              if (regs[0] < regs[1]) s_flag =
1;
                              else s_flag = 0;
                              regs[0] = regs[0]+(~regs[1]+1);
                              z_flag = (regs[0] == 0)? 1:0;

                        end

                        //XCHG regs[1],regs[0]
                        8'b0010????:
                        begin
                              temp = regs[0];
                              regs[0] = regs[1];
                              regs[1] = temp;
                        end

                        //MOV regs[0],[ADDRESS]
                        8'b0011????:
                        begin
                              regs[0] = dataStore[addPoint];
// lower nibble is in addPoint, which will indicate the value
from dataStore.
```

```
                              z_flag = (regs[0] == 0)? 1:0;
                        end

                        //RCR regs[1]
                        8'b0100????:
                        begin
                              temp[3] = c_flag;
                              {regs[1], c_flag} = {regs[1],
c_flag} >> 1;
                              regs[1][3] = temp[3]; //z_flag
is not considered since accumulator is not changing

                        end



                        //IN regs[0]
                        8'b0101????: regs[0] = myinput;

                        //OUT regs[0]
                        8'b0110????: myoutput = regs[0];



                        //AND regs[0],regs[1]
                        8'b0111????:
                        begin
                         regs[0] = regs[0]&regs[1]; //AND
operation
                         if (regs[0] == 0) z_flag = 1;
                         end
```

```
                        //TEST
                        8'b1000????:
                        begin
                              c_flag = 0; // TEST operation
                               temp = regs[0];
                               regs[0] = (regs[0]&regs[1]);
                               s_flag = regs[0][3];
                               regs[0] = temp;
                        end

                        //OR B, ADDRESS
                        8'b1001????:

                        begin
                              regs[1] = regs[1] |
dataStore[addPoint]; //flags will not update as its not
accumulator
                        end

                        //XOR regs[0], ADDRESS
                        8'b1010????:

                        begin
                              regs[0] = regs[0] ^
dataStore[addPoint];
                              if (regs[0] == 0) z_flag = 1;
                              else z_flag = 0;
                        end
```

```
                        //PUSH
                        8'b1011????:

                        begin
                              stackPoint = stackPoint - 1;
                              stackReg[stackPoint] = regs[1];
                        end




                        //POP
                        8'b1100????:

                        begin
                              regs[1] = stackReg[stackPoint];
                              stackPoint = stackPoint + 1;
                        end

                        //HLT
                        8'b1111????:
                        begin
                              HLT = 1;
                        end
                  endcase
            progCount = progCount + 1;
            end
      end
endmodule
```

- **Relating code with the operations:**

The progReg holds all the instructions, dataStore holds all the input values, stackReg holds the pushed values where the stackPoint is at 14 (b'1110). For the case-based operations, *casez* is utilized as the lower 4-bit of the instructions are don't care values. The regs[0] and regs[1] are declared as reg, each 4-bit long, as this is a 4-bit PC. They are declared in this way so that more registers can be easily accessed.

- ADD: regs[0] and regs[1] values are added, and if regs[0] becomes 0 after addition, then z_flag (zero flag) will be 1, otherwise 0.
- SUB: regs[1] is subtracted from regs[0], and if regs[0] becomes 0 after addition, then z_flag (zero flag) will be 1, otherwise 0. If regs[0] is less than regs[1], then s_flag (sign flag) will be 1.
- XCHG: Using a temporary register, regs[0] and regs[1] interchanges their values. If regs[0] becomes zero, then z_flag=1.
- MOV A, [ADDRESS]: According to the lower nibble (4-bits) of the instruction, data is fetched from the dataStore and put in A.
- RCR B: The carry flag is stored and later put in the MSB of B, whereas the rest of the values are shifter to the right.
- IN A: Input is put in A
- OUT A: A is put in myoutput.
- AND A,B: And operation is performed, which is by default bitwise and.
- TEST B, [ADDRESS]: Here and operation is performed without altering B. To do that, temp stores the value of B primarily, and later it is put in B again. The lower nibble of the instruction is important, to fetch the desired value from dataStore.
- OR B, [ADDRESS]: Or operation is performed after fetching the value from dataStore.
- XOR A, [ADDRESS]: XOR operation is after fetching the value from dataStore.
- PUSH B: The value of B is pushed into the stack. The stackPoint was at 14, since it works in LIFO (Last-in-First-out) method, the stackPoint becomes 13 and there the value of B is stored.
- POP B: The value of B is returned from the stack. stackReg[stackPoint] holds the desired value, whose size is also 4-bit.
- CALL ADDRESS: Performing the call operation, this particular value of progCount shall be stored, as it needs to perform the later operations from this point. This is similar to PUSH, but I couldn't implement it.
- RET: The program starts with the stored progCount value to bring that stored value. I couldn't implement it.

- HLT: Hlt variable becomes 1, stopping the whole program, i.e., the case-based operations.

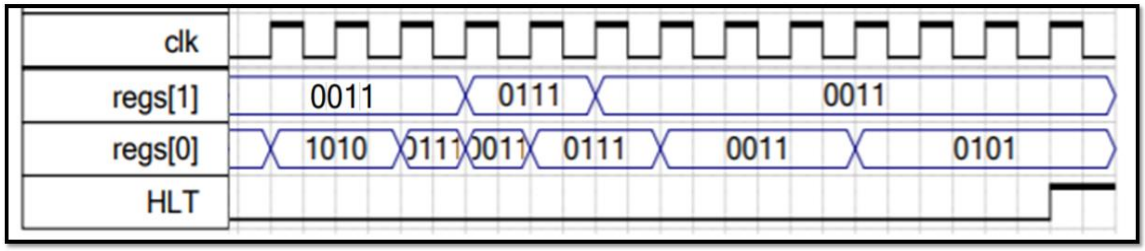

Figure 2 Waveform for the two registers

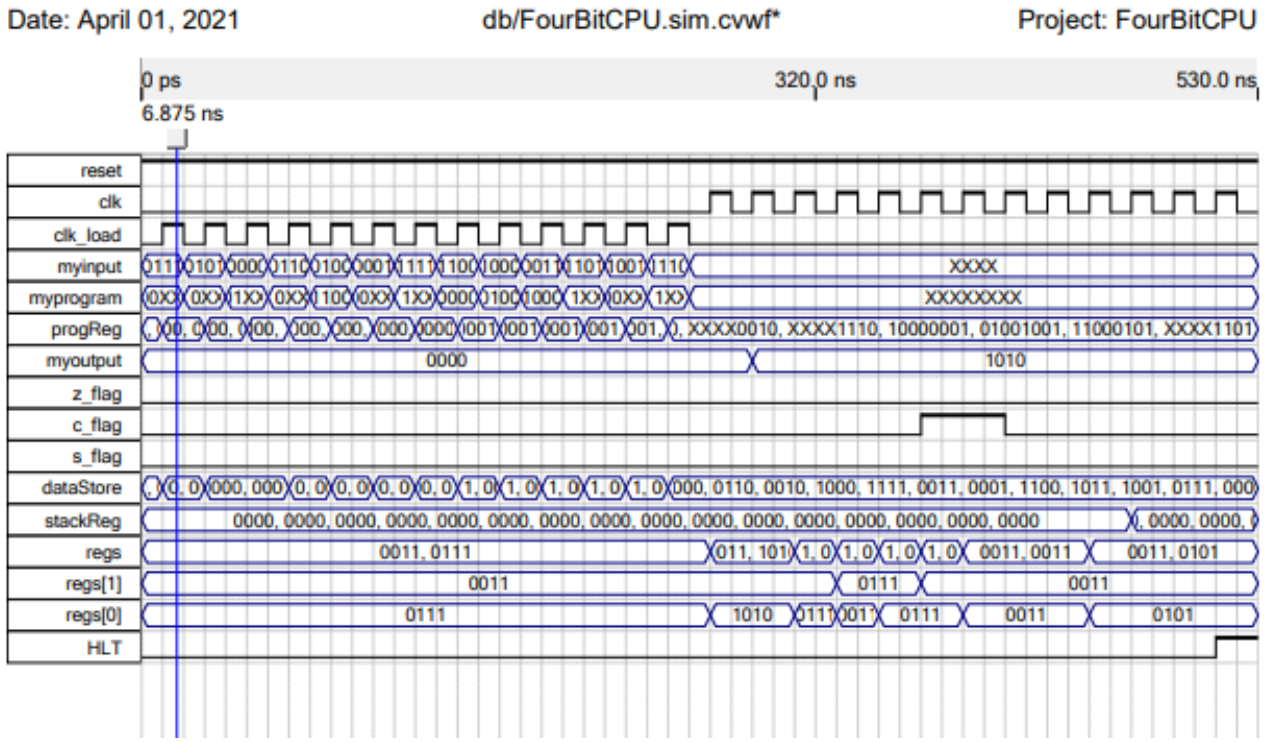

Figure 3 The total waveform. In the fetching cycle, dataStore and progReg are being loaded. In the execution cycle, regs[0] and regs[1] are being operated.

- ## Discussion:

The experiment's goal is to understand the basic architecture of a 4-bit PC and build one using Verilog-HDL. The implemented algorithm is similar to the Simple As Possible (SAP)-1 architecture; although for reducing complexity, case-based operations were performed, but they reflected the basic properties of a general PC. The whole procedure has been simulated using Verilog-HDL. Among the 16 commands, CALL and RET couldn't be implemented in the waveform result.

- ## References

1. *SAP-1 Architecture* by Education for ALL. Available [online], url: https://deeprajbhujel.blogspot.com/2015/12/sap-1-architecture.html. Retrieved March 31 2021.
2. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z. Available [online], url: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf . Retrieved March 31 2021.
3. Malvino AP, Brown JA. Digital Computer Electronics third edition, GLENCOE Macmillan.