Sworup Bhattarai

ID: 1001093304

TRISC Design Project

12/04/2021

TABLE OF CONTENT

Introduction:

This project consists of logic design activity and successful implementation of logic tasks based on the knowledge acquired throughout the semester. The scope of this project is to design, test and implement a simplified TRISC CPU successfully. The requirements for this project were to use the separate components created throughout the labs and HomeWorks and get them to work together and execute a program that was given for this project. The project will be made using Verilog on Quartus Prime and will be executed on the DE10-Lite FPGA model number 10M50DAF484C7G.

Requirements:

This project requires the creation of a TRISC CPU using components that have been completed in previous laboratory exercises and homework, to implement a functioning processor that can execute programs consisting of INC (Increment), CL(Clear), JMP(Jump), LDA(Load), STA(Store), and ADD(Addition) instructions. Only using the components necessary as shown in Figure 1. This simplifies the integration and testing.

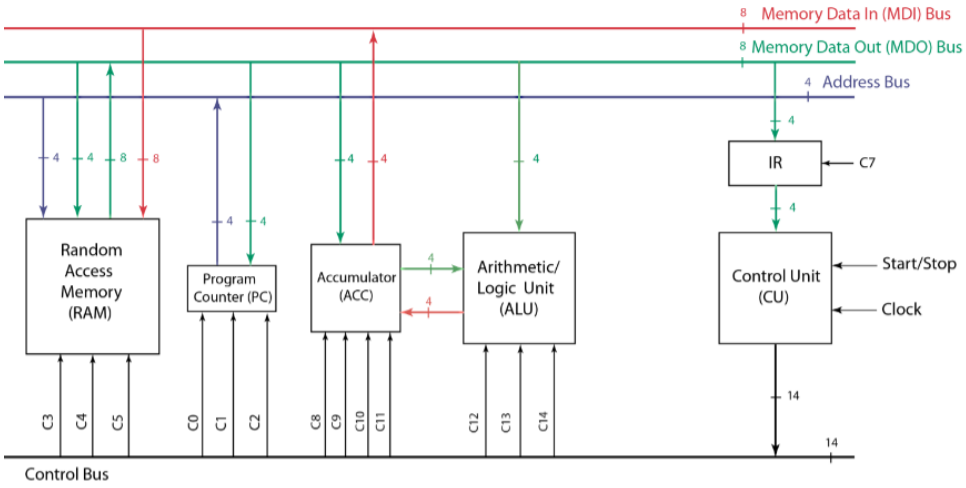

Figure 1:TRISC design for INC, CLR, LDA, STA, ADD, and JMP

Status:

During the design and implementation phase of the project, the FSM(Finite State Machine) in the Control Unit (CU) was not outputting the correct control codes to the rest of the modules and not allowing them to function as they should. The secondary problem that was faced was again with the FSM, code shown in Figure X below and the correct control codes to the BufferRegester, not shown in Figure 1, causing the ADD function to not load the value of the function.

System Design:

The TRIAC processor is made up of six main components, they are the Program Counter (PC), the Accumulator(ACC), the Arithmetic/Logic Unit (ALU), the Instruction Register(IR), Control Unit (CU), and the RAM. The Program Counter stores the current running program number and the memory location to be accessed, the code for the PC is shown in Appendix A below. The ACC is used to pick if the input from the Memory Data Out (MDO) or the ALU and either increments the data or sends the data out via the MDI and to the ALU to be stored and used, the Verilog code for the ACC is shown in Appendix B below. The ALU is the math module of the CPU it is responsible for adding subtracting ANDing and ORing (only add is sued in this CPU) , the Verilog code for the ACC is shown in Appendix C below. The IR is used hold and forward the instruction data that is fetched from the RAM, in function it is the same as the BufferRegester(Br), the IR is shown in Appendix D. The next component of the TRISC CPU is the RAM block it is what stores and provides the data and the control data that the rest of the components use to carry out programs and steps. The final and most important main block of the TRIAC CPU is the CU, the control unit takes the control data from the IR and changes it into control bits that activate different parts of the CPU to make them perform specific tasks shown in Appendix E, the CU is made up of two sub blocks the fourtosixteen decoder that takes the 4 bit

command code and decodes it to the specific INC, CLR, JMP, LDA, STA, and ADD instructions. That data is then sent to the second sub block the Finite State Machine(FSM) which takes the control code from the decoder and creates steps that will control the rest of the different parts of the CPU, the decoder and FSM are shown in Appendix F.

Controller design details:

The control for this CPU is done by using a variety of switches and keys, the keys are used to input data to the RAM Key5, Key3, and Key2 and as the clock and clear buttons Key0 and Key1, the input of the data and control data were done using switches 0-7. Hex Displays were also used to show the RAM data and the assignments of the for the keys, switches and HEX displays are shown in Appendix G.

Test Results:

The code provided to run on the TRISC CPU is : 0: 0F, 1: 61, 2: 62, 3: 1E, 4: 74, 5: 0E, 6: 66, 7: 89, 8: 88, 9: 69, A: 2E, B: 7B, C: 6C, D: 88, E: EE, F: FF, this code will Load from RAM location F, then increment the ACC twice then store the value in the ACC to RAM location E, then clear the ACC, Load the ACC from RAM location E increment the ACC by one, jump the program counter to ram location 9, increment the ACC by one, add the ACC value to the value at RAM location E then clear the ACC, increment the ACC, jump the program counter to RAM location 8 and jump back and 9 and back and forth. The results from code provided can be found in the accompanying video called TRISC Demo.mp4.

Conclusion:

This project consisted of creating a TRISC CPU and all the supporting devices such as the ACC, PC, ALU and RAM. The main problem that occurred was that the FSM was not set up to send out the correct control, this was unfortunately only mitigated after scraping the first attempt and re-creating the TRISC CPU from scratch. There were some design errors on the second iterations of the CPU that the accumulator not working, but with the help of the TA Khaled Ahmed the error was traced back to the FSM and the error was fixed.

Lesson Learned:

The main lesson learned during this project was that sometimes the best way to fix a problem/bug is to just start back from scratch and that asking for assistance on things that are not working is also a good way to be pointed in the correct direction. I also learned while creating the CPU that sometimes even though the command that you send may look simple there may be deeper processes that run to achieve that “simple” command.

Appendix A: Program Counter

```
module nbitbinary #(parameter N = 4)
( // Sworup Bhattarai 1001093304
   input COUNT, CLEAR, LOAD,
   input [N-1:0] in,
   output reg [N-1:0] y
);


   always @ (negedge COUNT, negedge LOAD, negedge CLEAR)
   begin
      if (CLEAR==1'b0)
      begin
         y = 0;
      end

      else if (LOAD==1'b0)
      begin
         y <= in;
      end

      else if (COUNT == 1'b0)
      begin
         y <=  y + 1'b1;
      end

   end
endmodule
```

Appendix B: ACC

```
 1   module acc  #(parameter N = 4)
 2   (
 3      input clear, load, inc, AB,
 4      input [N-1:0] A , B,
 5      output [N-1:0] Z
 6   );
 7      wire [N-1:0] D;
 8
 9   //instantiate component modules
10   two2one #(4) two2oneMUX
11   (
12    .A(A) ,      // data input [3:0] A
13    .B(B) ,     // data input [3:0] B
14    .S(AB) ,     // select input  AB.  AB=0 selects A, AB=1 selects B.
15    .Y(D)      // data output [3:0] D
16   );
17   BinUp  #(4) RegCount
18   (
19    .inc(inc) ,    // control input  inc
20    .clear(clear) ,    // input  clear
21    .load(load) ,    // input  load
22    .D(D) ,      // data input from MUX [3:0] D
23    .Q(Z)      // data output from Acc [3:0] Z
24   );
25   endmodule
```

Appendix C: ALU

```
1  module alu (
2
3      input [3:0] A, B, //declare input ports
4      input [1:0] S,
5
6      output reg [3:0] R
7      //declare output ports for sum;
8      );
9
10     wire [3:0] Ad, La, Lx;
11     //wire [3:0] pr;
12     wire Cw, Ow;
13
14     AdderSubtractor ( A, B ,S[0], Ad , Cw, Ow);
15
16     assign La = A & B;
17     assign Lx = A ^ B;
18
19     always @ (S)
20     begin
21        if (S == 2'b00)
22        begin
23        R <= Ad;
24        end
25
26        else if (S == 2'b01)
27        begin
28        R <= Ad;
29
30        end
31
32        else if (S == 2'b10)
33        begin
34        R <= La;
35        end
36
37        else if (S == 2'b11)
38        begin
39        R <= Lx;
40        end;
41
42     end
43
```

Appendix D: IR

```
1
2   module IR #(parameter N = 4)
3   (
4       input [N-1:0] D,
5       input CLK, CLR,
6       output reg [N-1:0] Q
7   );        //declare N-bit data output
8
9
10          always @ (negedge CLK, negedge CLR)
11          begin
12              if (CLR==1'b0) Q <= 0;
13              else if (CLK==1'b0) Q <= D;
14          end
15  endmodule
16
```

Appendix E: CU

```
1   module triscPCU
2   (
3       input SysClock,SysReset,
4       input [0:3]Switch,    // 1000
5       output [0:14] C,
6       output clock, reset,
7       output [0:3]Sout
8   );
9       wire [0:15] Y;
10      assign Sout = Switch;
11      assign clock = SysClock;
12      assign reset = SysReset;
13
14      fourtosixteen(Switch,Y);
15
16      triscFSM FSM(
17      SysClock,SysReset,Y[0],Y[1],Y[2],Y[3],Y[4],Y[5],Y[6],Y[7],Y[8],Y[9],Y[10],
18      C[0],C[1],C[2],C[3],C[4],C[7],C[8],C[9],C[5],C[10],C[11],C[12],C[13],C[14] );
19
20  endmodule
21
```

Appendix F: Decoder & FSM

```
module fourtosixteen
    (
    input [0:3] S,
    output reg [0:15] Y
    );

    always @ (S)
    case({S})
        4'b0000: Y = 16'b1000000000000000; //LDA
        4'b0001: Y = 16'b0100000000000000; //STA
        4'b0010: Y = 16'b0010000000000000; //ADD
        4'b0011: Y = 16'b0001000000000000; //SUB - NOT USED
        4'b0100: Y = 16'b0000100000000000; //XOR - NOT USED
        4'b0110: Y = 16'b0000010000000000; //INC
        4'b0111: Y = 16'b0000001000000000; //CLR
        4'b1000: Y = 16'b0000000100000000; //JMP
        4'b1100: Y = 16'b0000000010000000; //JPN - NOT USED
        4'b1001: Y = 16'b0000000001000000; //JPZ - NOT USED
        4'b1111: Y = 16'b0000000000100000; //HLT - NOT USED
        default: Y = 16'b0000000000000000; //Defaults to all 0s
    endcase
endmodule
```

```
module triscFSM
(
    input SysClock,StartStop,LDA,STA,ADD,SUB,XOR,INC,CLR,JMP,JPZ,JPN,HLT,
    output reg C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14
);
    reg [4:0] state,nextstate;
    parameter A=5'b00000,B=5'b00001,C=5'b00010,D=5'b00011,E=5'b00100,F=5'b00101,G=5'b00110,H=5'b00111,I=5'b01000,J=5'b01001,K=5'b01010,L=5'b01011,
          M=5'b01100,N=5'b01101, O=5'b01110,P=5'b01111,Q=5'b10000,R=5'b10001,S=5'b10010,T=5'b10011,U=5'b10100;

    always @ (negedge SysClock,negedge StartStop)
       if (StartStop==1'b0) state <= A; else state <= nextstate;

    always @ (state,INC,CLR,JMP,LDA,STA,ADD)
       case (state)
          A: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b10000000000000; nextstate = B; end //INITIALIZE
          B: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00010000000000; nextstate = C; end //FETCH
          C: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00011000000000; nextstate = D; end //FETCH
          D: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00011000000000; nextstate = E; end //FETCH
          E: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00110100000000;        //DECODE
             if (INC)nextstate = F;
                else if (CLR) nextstate = G;
                   else if (JMP) nextstate = H;
                      else if (LDA) nextstate = I;
                         else if (STA) nextstate = M;
                            else if (ADD) nextstate = P;
          end
          F: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000001000000; nextstate = B; end //INC 0110
          G: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000010000000; nextstate = B; end //CLR 0111
          H: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b01000000000000; nextstate = B; end //JMP 1000
          I: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = J; end //LDA 0000
          J: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = K; end //LDA
          K: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = L; end //LDA
          L: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000001000; nextstate = B; end //LDA
          M: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = N; end //STA 0001
          N: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000100000; nextstate = O; end //STA
          O: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000100000; nextstate = B; end //STA
          P: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000000000; nextstate = Q; end //ADD 0010
          Q: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = R; end //ADD
          R: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00001000000000; nextstate = S; end //ADD
          S: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000010000; nextstate = T; end //ADD
          T: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000011001; nextstate = B; end //ADD
          //U: begin {C0,C1,C2,C3,C4,C7,C8,C9,C5,C10,C11,C12,C13,C14} = 14'b00000000001001; nextstate = B; end //ADD
       endcase
endmodule
```

Appendix G: Pin Assignments

To	Assignment Name	Value	Enabled
MAR[0]	Location	PIN_F18	Yes
MAR[1]	Location	PIN_E20	Yes
MAR[2]	Location	PIN_E19	Yes
MAR[3]	Location	PIN_J18	Yes
MAR[4]	Location	PIN_H19	Yes
MAR[5]	Location	PIN_F19	Yes
DataIn[1]	Location	PIN_C11	Yes
DataIn[2]	Location	PIN_D12	Yes
DataIn[3]	Location	PIN_C12	Yes
DataIn[4]	Location	PIN_A12	Yes
DataIn[5]	Location	PIN_B12	Yes
DataIn[6]	Location	PIN_A13	Yes
DataIn[7]	Location	PIN_A14	Yes
DataIn[0]	Location	PIN_C10	Yes
C[0]	Location	PIN_AB7	Yes
C[1]	Location	PIN_AB8	Yes
C[2]	Location	PIN_AB9	Yes
C[3]	Location	PIN_Y10	Yes
C[4]	Location	PIN_AA11	Yes
C[5]	Location	PIN_AA12	Yes
C[7]	Location	PIN_AB17	Yes
C[8]	Location	PIN_AA17	Yes
C[9]	Location	PIN_AB19	Yes
C[10]	Location	PIN_AA19	Yes
C[11]	Location	PIN_E14	Yes
C[12]	Location	PIN_D14	Yes
C[13]	Location	PIN_A11	Yes
C[14]	Location	PIN_B11	Yes
RW	Location	PIN_AB20	Yes
Reset	Location	PIN_B8	Yes
SysClock	Location	PIN_A7	Yes
ClearAddGen	Location	PIN_AB6	Yes
ClockIn	Location	PIN_AB5	Yes
Mode	Location	PIN_F15	Yes
MDIout[0]	Location	PIN_C14	Yes
MDIout[1]	Location	PIN_E15	Yes
MDIout[2]	Location	PIN_C15	Yes
MDIout[3]	Location	PIN_C16	Yes
MDIout[4]	Location	PIN_E16	Yes
MDIout[5]	Location	PIN_D17	Yes
MDIout[6]	Location	PIN_C17	Yes
MDIout[8]	Location	PIN_C18	Yes
MDIout[9]	Location	PIN_D18	Yes
MDIout[10]	Location	PIN_E18	Yes
MDIout[11]	Location	PIN_B16	Yes

MDIout[12]	Location	PIN_A17	Yes
MDIout[13]	Location	PIN_A18	Yes
MDIout[14]	Location	PIN_B17	Yes
MDOout[8]	Location	PIN_F21	Yes
MDOout[9]	Location	PIN_E22	Yes
MDOout[10]	Location	PIN_E21	Yes
MDOout[11]	Location	PIN_C19	Yes
MDOout[12]	Location	PIN_C20	Yes
MDOout[14]	Location	PIN_E17	Yes
MDOout[13]	Location	PIN_D19	Yes
MDOout[0]	Location	PIN_B20	Yes
MDOout[1]	Location	PIN_A20	Yes
MDOout[2]	Location	PIN_B19	Yes
MDOout[3]	Location	PIN_A21	Yes
MDOout[4]	Location	PIN_B21	Yes
MDOout[5]	Location	PIN_C22	Yes
MDOout[6]	Location	PIN_B22	Yes
PC[0]	Location	PIN_J20	Yes
PC[1]	Location	PIN_K20	Yes
PC[2]	Location	PIN_L18	Yes
PC[3]	Location	PIN_N18	Yes
PC[4]	Location	PIN_M20	Yes
PC[5]	Location	PIN_N19	Yes
PC[6]	Location	PIN_N20	Yes
MAR[6]	Location	PIN_F20	Yes
MDO[5]	Location	PIN_A9	Yes
MDO[6]	Location	PIN_A10	Yes
MDO[7]	Location	PIN_B10	Yes
MDO[4]	Location	PIN_A8	Yes

Appendix H: Main Code

```
module triscCPU
(

    input Reset, SysClock, Mode, ClockIn, ClearAddGen, RW, //Mode = SW9, ClockIn = Key2, ClearAddGen = Key3, RW = Key5
    input [7:0] DataIn, //DataIn = {SW7,SW6,SW5,SW4,SW3,SW2,SW1,SW0}

    output [0:14] C,
    output [14:0] MDIout, MDOout,
    output [6:0] PC, MAR


);


    wire [3:0] AddIn, AddGen, AddBus, IRin, ACCbits, Buffer, ALUbits;
    wire RAMin, RAMwrite, toggle;
    wire [7:0] RAMdata, RAMadd, MDO, MDI;


    assign RAMadd = C[3] == 1'b0 ? MDO[3:0] : AddBus ;
    assign AddIn = Mode == 1'b0 ? RAMadd : AddGen;
    assign RAMin = Mode == 1'b0 ? SysClock*C[4] : ClockIn;
    assign RAMdata = Mode == 1'b0 ? MDI : DataIn;
    assign RAMwrite = Mode == 1'b0 ? C[5] : ~RW;

IR InstructionRegester
(
    .D(MDO[7:4]) , // input [N-1:0] D_sig
    .CLK(~C[7]) ,  // input  CLK_sig
    .CLR(Reset) ,  // input  CLR_sig
    .Q(IRin)    // output [N-1:0] Q_sig
);

triscPCU PCU
(
    .SysClock(~SysClock) ,  // input  SysClock_sig
    .SysReset(Reset) ,   // input  SysReset_sig
    .Switch(IRin) ,   // input [0:3] Switch_sig
    .C(C)    // output [0:14] C_sig
);

nbitbinary ProgramCounter
(
    .COUNT(~C[2]) ,   // input  COUNT_sig
    .CLEAR(~C[0]) ,   // input  CLEAR_sig
    .LOAD(~C[1]) , // input  LOAD_sig
    .in(MDO[3:0]) ,   // input [N-1:0] in_sig
    .y(AddBus)  // output [N-1:0] y_sig
);


acc Accumulator
(
    .clear(~C[8]) ,   // input  clear_sig
    .load(~C[11]) ,   // input  load_sig
    .inc(~C[9]) ,  // input  inc_sig
    .AB(C[10]) ,   // input  AB_sig
    .A(MDO[3:0]) , // input [N-1:0] A_sig
    .B(Buffer) ,   // input [N-1:0] B_sig
    .Z(MDI[3:0])   // output [N-1:0] Z_sig
);

IR BufferRegester
(
    .D(ALUbits) ,  // input [N-1:0] D_sig
    .CLK(~C[14]) , // input  CLK_sig
    .CLR(Reset) ,  // input  CLR_sig
    .Q(Buffer)  // output [N-1:0] Q_sig
);

alu ArithmeticLogicUnit
(
    .A(MDI[3:0]) , // input [3:0] A_sigA
    .B(MDO[3:0]) , // input [3:0] B_sig
    .S({C[13],C[12]}) ,  // input [1:0] S_sig
    .R(ALUbits)    // output [3:0] R_sig
);


binary2seven (AddBus, PC);  //  PC (HEX5)
binary2seven (AddIn, MAR);  //  MAR (HEX4)??
binary2seven (MDO[7:4], MDOout[14:8]);  //  MDOut (HEX3)
binary2seven (MDO[3:0], MDOout[6:0]);  //  MDOut (HEX2)
binary2seven (RAMdata[7:4],  MDIout[14:8]);  //  MDOIn (HEX0)
binary2seven (RAMdata[3:0],  MDIout[6:0]);  //  MDOIn (HEX1)
```

```
90
91    OnOffToggle DivideX2
92    (
93        .OnOff(ClockIn) , // input OnOff_sig
94        .IN(1'b1) , // input IN_sig
95        .OUT(toggle) // output OUT_sig
96    );
97
98
99    BinUp AddressGen
100   (
101       .inc(toggle) , // input inc_sig
102       .clear(ClearAddGen) , // input clear_sig
103       .load(1'b1) , // input load_sig
104       .D(4'b0) , // input [N-1:0] D_sig
105       .Q(AddGen) // output [N-1:0] Q_sig
106   );
107
108
109   triscRAM RAM
110   (
111       .address ( AddIn ),
112       .clock ( ~RAMin ),
113       .data ( RAMdata ),
114       .wren ( RAMwrite ),
115       .q ( MDO )
116   );
117
118
119   endmodule
120
```

Appendix I :