5/4/2018

TRISC

Design and Implementation

CSE 2441 Term Project

PARAS SHARMA

ID: 1001403338

paras.sharma@mavs.uta.edu

1001403338

Contents

List of Figures and Tables

Introduction

Project Overview

This is the term project of CSE 2441 Introduction to Digital Logic class where I designed a miniature CPU based on TRISC. The CPU carries out 8 bits instructions (first four opcode and last four data). The final CPU of this project carries out 6 instructions properly: CLR, INC, STA, LDA, JMP and ADD. This project utilizes what I learned in the class and labs.

Project Status

The CPU was successfully designed and implemented which carries out instructions LDA, STA, JMP, ADD, INC and CLR. So far, there is no unresolved bugs. Other functions such as SUB, XOR, etc. are yet to be added in the future. I also plan to change the architect to support 16 bits instructions later during my free times.

Report Overview

The report consists of descriptions of system, its various subsystems or components, control unit Verilog code, state diagram of control unit, tests output, etc. and the figures and some tables that relates to them.

System Design

System-level Description

The TRISC is made by integration of various components such as RAM, PC, ACC, ALU, BR, IR and CU. They are all integrated together to form a system and their interaction is controlled by various control signals produced by CU. Four buses, Memory Data In (MDI), Memory Data Out (MDO), Address (ADR or ADD) and Control (C) are used to transport large bits of data between those components such as RAM and ACC.

The system was broken into three different parts: A, B and C.

Part A

The first part required integrating of basic components such as ACC, RAM, PC and CU together using buses and wires. Following is the model diagram of the system built in part A. It was expected to implement two instructions: CLR and INC. INC would increase the accumulator values while CLR would clear it.

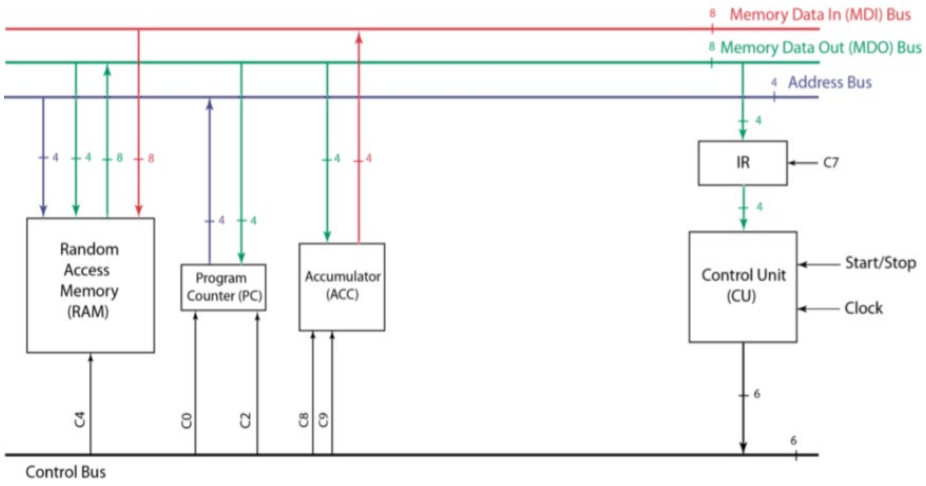


Figure 1 Part A Design Diagram

Part B

Part B consisted of adding multiplexer on the address bus and accumulator so that the address on the RAM's address bus can be selected from either program counter or MDO bus (last four bits). Also, the accumulator was tweaked so that it could load values and the RAM can store values on MDI bus on giving signals to its clock and wren. The instructions implemented were INC, CLR, LDA, STA. The design diagram for part B is:

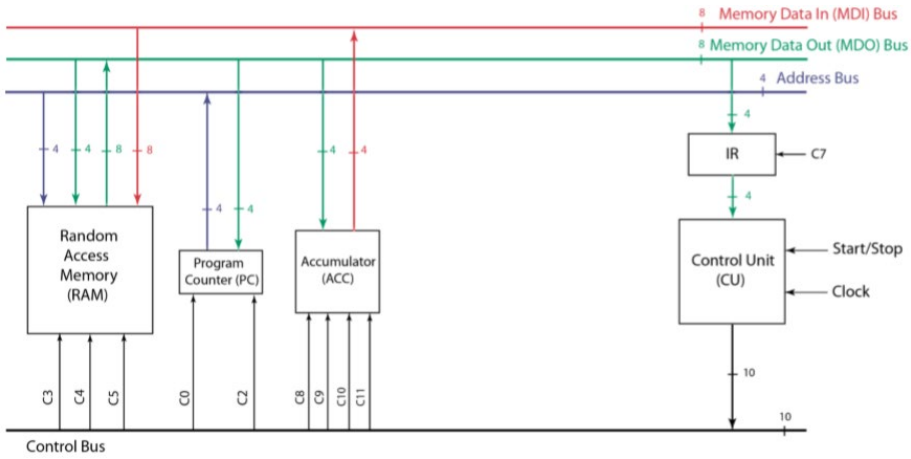


Figure 2 Part B Design Diagram

Part C

Here, I extended part B by adding ALU, BR, multiplexer to ACC, etc. I was able to successfully run instructions such as ADD, JMP, CLR, LDA, STA, INC.

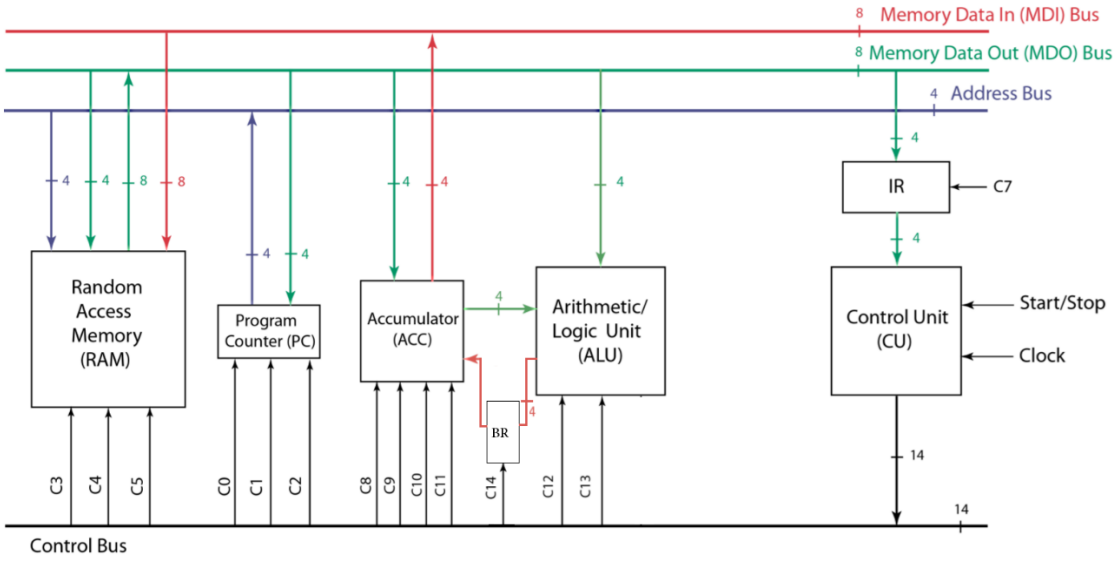
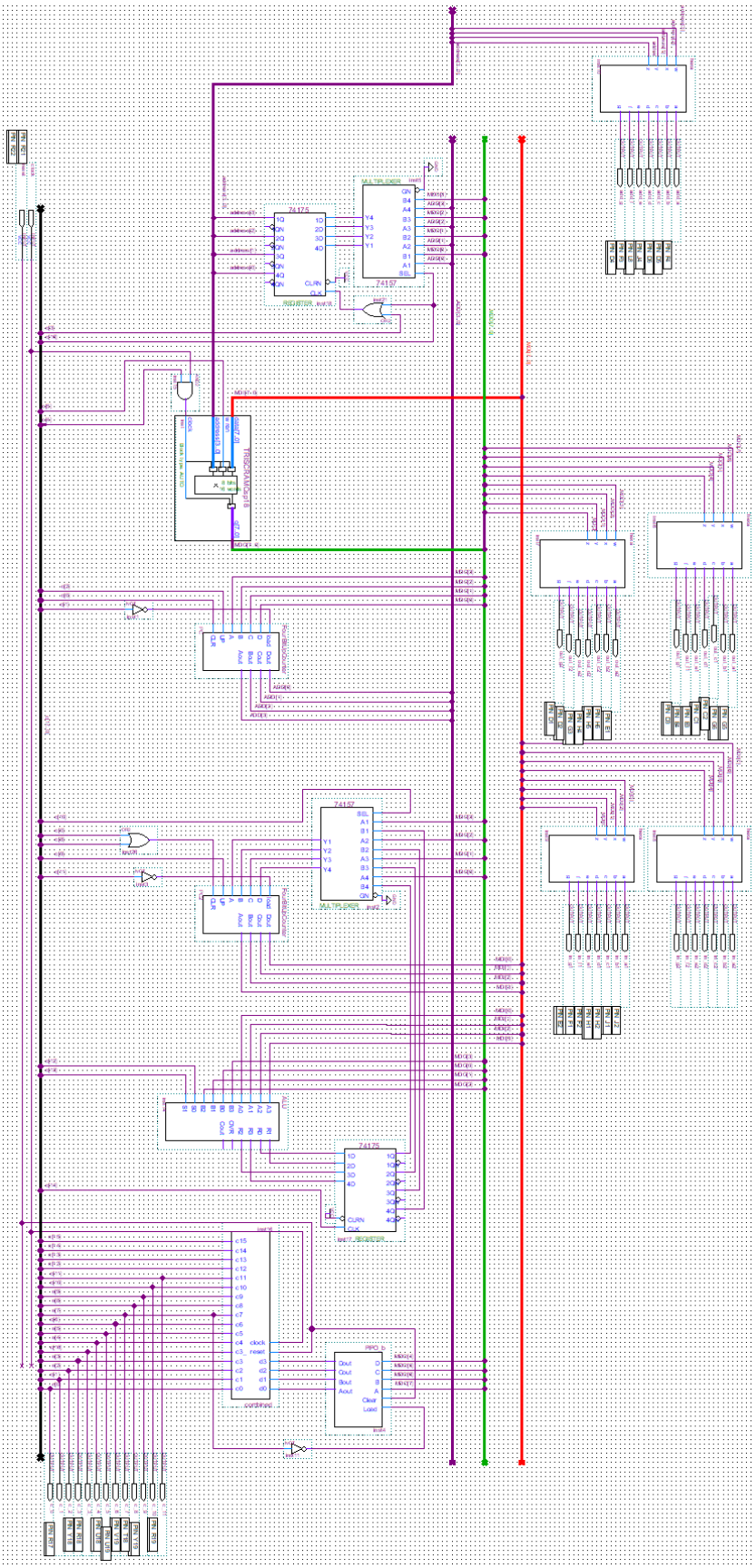


Figure 3 Part C Design Diagram

Figure 4 Figure 5 QUARTUS Design File for TRISC part C



Subsystem Descriptions

Let's talk about various subsystems and components that we used to build up the main system.

Memory (RAM)

The RAM is preloaded with program instructions from instructors which are supposed to be implemented by the system. The inputs of the RAM are 8 bits input data bus (MDI), wren, 4 bits address bus (ADD or ADR) and 8 bits output data bus (MDO). On positive edge of clock, RAM loads the data at address bus as address and on next positive edge of clock, if wren is disabled (input is 0), it reads the 8 bits of data from memory at that address and puts the read data into the output data bus. If wren is enabled (input is 1) then the RAM writes the 8 bits data at input data bus to the memory at address read before from address bus.

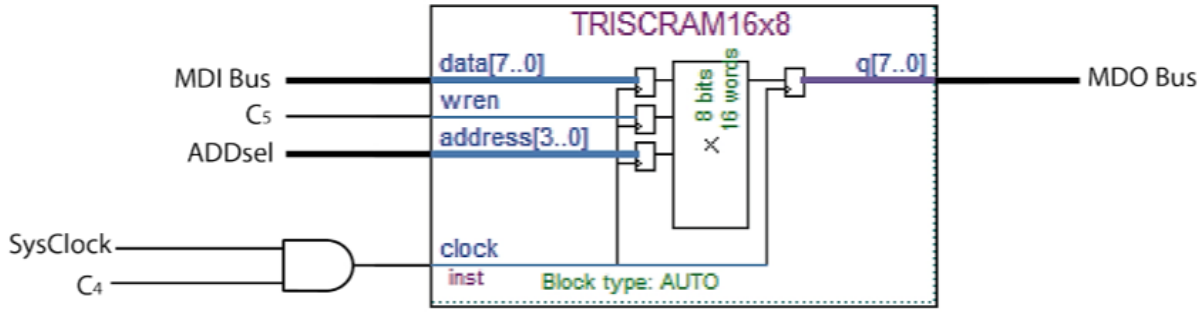


Figure 6 RAM used in TRISC part C

Program Counter (PC)

Its a four-bit up counter with capability to load last four (least significant) bits from MDO bus, count it up and clear to all zeros as determined by control signals c1, c2 and c0. The counter is made from chip 74193.

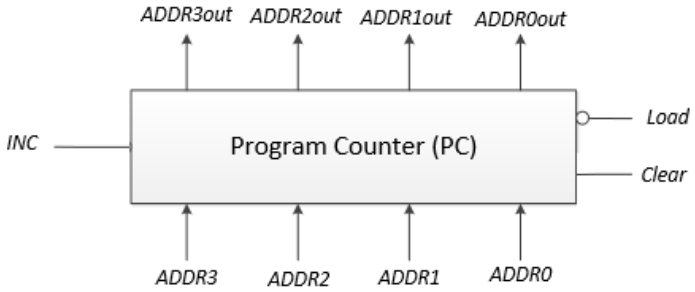


Figure 7 Program Counter

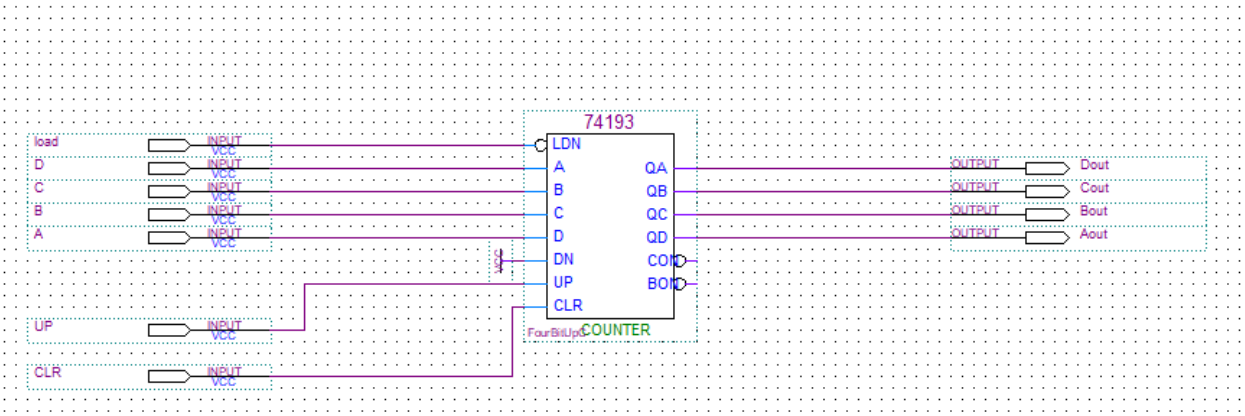


Figure 8 Program Counter Schematic Diagram, input pin A is MSB and D is LSB

Accumulator (ACC)

The core of the accumulator is the same 4-bit counter as we used in PC before. In part A and B, I only used the counter as accumulator but in part C, I added multiplexer 74157. The inputs A1, A2, A3, A4 of multiplexer is connected with the least significant four bits of MDO bus (MDO [3..0]) and the inputs B1, B2, B3, B4 of is connected with the BR. The output of multiplexer is connected to the 4bit counter whose outputs are put in MDI[3..0] bus as AC3out, AC2out, AC1out, AC0out respectively.

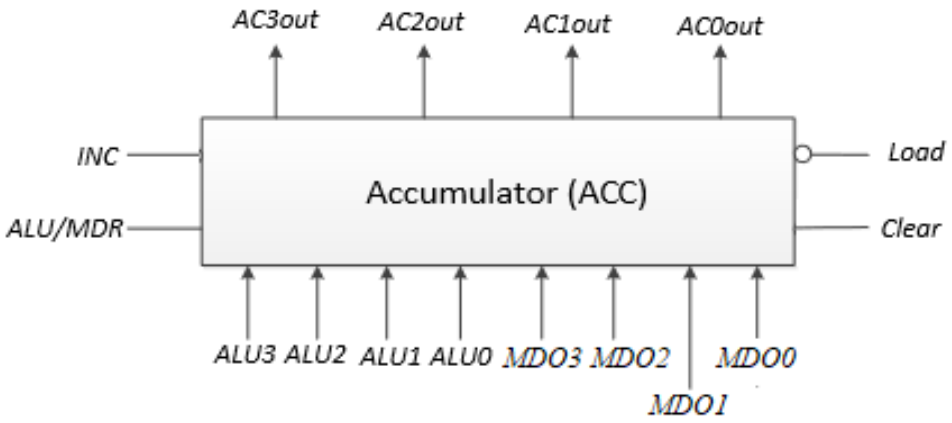
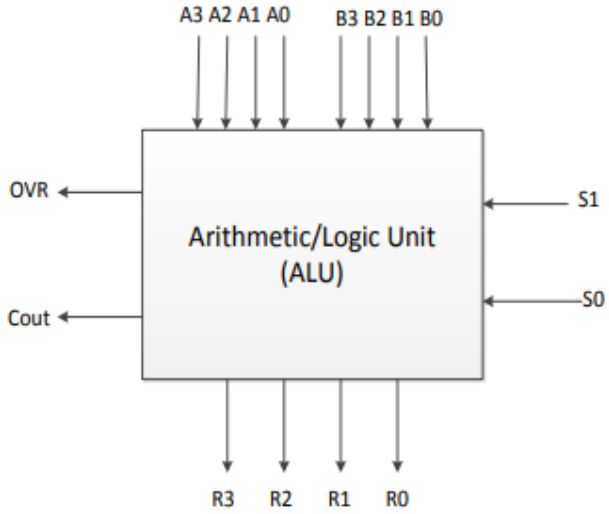


Figure 9 Accumulator

ALU

The ALU that we used in this project was designed earlier in Lab (#5). It is a 4-bits processor that has capability to do addition, subtraction, XOR and AND. It takes in 2 4-bit numbers: the first from Accumulator and second from MDO [3..0]. It also takes two bits input, s1 and s0, that signals the type of operations. The block diagram of ALU and the operation table is given below.



S1	S0	Operation
0	0	Add (A + B)
0	1	Subtract (A - B)
1	0	Logical AND (A·B)
1	1	Logical XOR (A ⊕ B)

Figure 10 ALU and the table of operation base on signals S1 and S0

Break register (BR) is added so that the output of ALU is stored here first which is then read by accumulator. BR is a PIPO made from 74175 with input values connected to the output of ALU (R3, R2, R1, R0), output connected to the multiplexer of accumulator, and control signal c14 determining when to change its state.

Hierarchical Design Structure

The final system depended on the work designed earlier which also depended on work done before that and so on. For instance, the system's ALU was made in lab 5 which was based upon labs before that. Also, the control unit was made in lab 9. Similarly, the PIPO registers and counters were designed in lab 6. Thus, the system was designed on hierarchical structure. In a way, all the knowledge that I gained in the lab were necessary to get this system working.

The project itself was divided into three parts: Part A, B and C. Part C was an extension to part B which was an extension to part C. In part A, I put together the main components that I built earlier such as control unit, RAM, Program Counter and Accumulator. The connected system was able to carry out CLR and INC instructions. In part B, I added multiplexer to address bus and made use of more control signals to carry out LDA and STA instructions. In part C, I put together what I had in part B with ALU so that ADD operations can be carried out along with JMP instruction.

Operating Procedure

Everything is done by changing clock cycles. There are instructions (opcode and data) preloaded into the RAM. On changing few clock cycles, RAM reads up data from memory at address which is zero currently and puts that in MDO. The Instruction Register (PIPO) filters and stores, on control signal c7, the data at MDO to load it with first four significant bits as OPCODE which is processed by decoder and then by controller to give respective control signals. Also, the value of program counter gets increased by 1, on control signal c2. Now based on the opcode, the control signals produced will determine what to do and when to do which is illustrated clearly in the state diagram later in figure 13.

Controller Design Details

Functional description and diagram showing I/O

The main control unit is the combination of decoder, that decodes 4 bits instructions into one of the eleven of the instruction signal, and controller, that performs out necessary state changes giving out necessary control signals from c0 to c15. The block diagram of controller is given below:

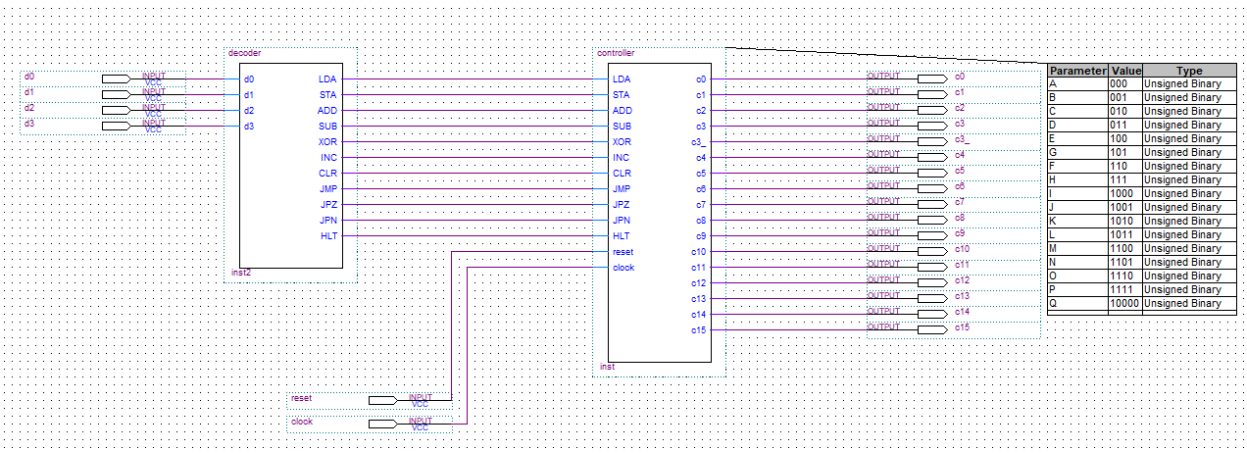


Figure 11 Controller Block Diagram

First, the decoder takes inputs 4bits input d3..d0 and based on it activates only one of the LDA, STA, ADD, etc. signals based on the opcode. Then when one (and only one) of them is activated, the controller performs necessary operations is series of steps as given in state diagram. The opcode table is given below:

Instruction	Function	Register Transfer	Op Code
LDA	Load ACC	ACC ← (MDR)	0000
STA	Store ACC	MDR ← (ACC)	0001
ADD	Add ACC	ACC ← (ACC) + (MDR)	0010
SUB	Subtract ACC	ACC ← (ACC) – (MDR)	0011
XOR	XOR ACC	ACC ← (ACC) ⊕ (MDR)	0100
INC	Increment ACC	ACC ← (ACC) + 1	0110
CLR	Clear ACC	ACC ← 0	0111
JMP	Jump	PC ← (MDR)	1000
JPZ	Jump if 0	PC ← (MDR) if Z = 1	1100
JPN	Jump if < 0	PC ← (MDR) if N = 1	1001
HLT	Halt	PC ← 0	1111

Figure 12 Instruction and Op Code Table

The yellow labelled instructions are the instructions that my system implementation currently supports.

State diagrams

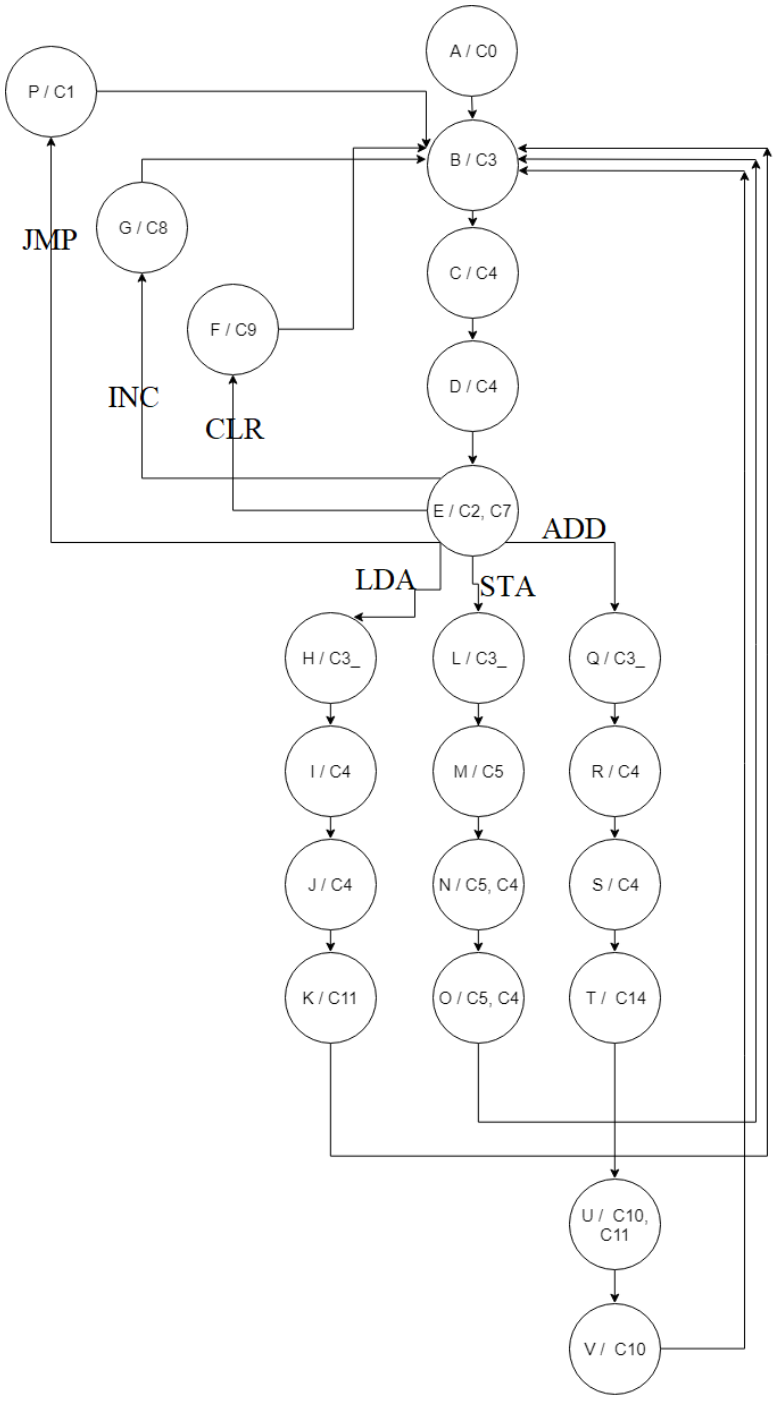


Figure 13 State Diagrams of CU

Control Signals Explained

C0 PC <- 0

C1 PC <- (MDO[3..0])

C2 PC <- (PC) + 1

C3 MAR <- (PC)

C3*/C3_ MAR <- (MDO[3..0)]

C4 Read Memory (Load Address)

C4 Read Memory (Transfer Data to MDO)

C5 Write Memory

C7 IR <- (MDO[7..4])

C8 ACC <- 0

C9 ACC <- (ACC) + 1

C10 ACC <- (BR), needs C11 to load though

C11 Loads the ACC with current input

C12, C13 Signals the operation type for ALU, for ADD they are 0 and 0

C14 BR <- (ALU)

Verilog code

The Verilog Code for decoder and controller is:

```
module decoder(
      input d0, d1, d2, d3,
      output reg LDA, STA, ADD ,SUB, XOR, INC, CLR, JMP, JPZ, JPN, HLT);
      always @ (d0, d1, d2, d3) begin
            {LDA, STA, ADD ,SUB, XOR, INC, CLR, JMP, JPZ, JPN, HLT} = 11'b00000000000;
            case ({d0, d1, d2, d3})
                  4'b0000: LDA = 1;
                  4'b0001: STA = 1;
                  4'b0010: ADD = 1;
                  4'b0011: SUB = 1;
                  4'b0100: XOR = 1;
                  4'b0110: INC = 1;
                  4'b0111: CLR = 1;
                  4'b1000: JMP = 1;
                  4'b1100: JPZ = 1;
                  4'b1001: JPN = 1;
                  4'b1111: HLT = 1;
            endcase
      end
endmodule
```

```
module controller(
   input LDA, STA, ADD ,SUB, XOR, INC, CLR, JMP, JPZ, JPN, HLT, reset, clock,
   output reg c0, c1, c2, c3, c3_, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15);

   reg [5:0] state, nextstate;
   parameter A = 5'b00000, B = 5'b00001, C = 5'b00010, D = 5'b00011, E = 5'b00100, G = 5'b00101, F = 5'b00110, H = 5'b00111;
   parameter I = 5'b01000, J = 5'b01001, K = 5'b01010, L = 5'b01011, M = 5'b01100, N = 5'b01101, O = 5'b01110, P = 5'b01111;
   parameter Q = 5'b10000, R = 5'b10001, S = 5'b10010, T = 5'b10011, U = 5'b10100, V = 5'b10101;
   always @ (posedge clock, negedge reset)
      if (reset == 0) state <= A;
      else state <= nextstate;
   always @ (state) begin
      {c0, c1, c2, c3, c3_, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15} = 17'b00000000000000000;
      case (state)
         A: begin c0 = 1; nextstate = B; end
         B: begin c3 = 1; nextstate = C; end
         C: begin c4 = 1; nextstate = D; end
         D: begin c4 = 1; nextstate = E; end
         E: begin
               c2 = 1; c7=1;
               if (INC) nextstate = F; if (CLR) nextstate = G; if (LDA) nextstate = H;
               if (STA) nextstate = L; if (JMP) nextstate = P; if (ADD) nextstate = Q;
            end
         // for INC
         F: begin c9 = 1; nextstate = B; end
         // for CLR
         G: begin c8 = 1; nextstate = B; end
         // for LDA
         H: begin c3_ = 1; nextstate = I; end
         I: begin c4 = 1; nextstate = J; end
         J: begin c4 = 1; nextstate = K; end
         K: begin c11 = 1; nextstate = B; end
         // for STA
         L: begin c3_ = 1; nextstate = N; end
         M: begin c5 = 1; nextstate = N; end
         N: begin c5 = 1; c4 = 1; nextstate = O; end
         O: begin c5 = 1; c4 = 1; nextstate = B; end
         // for JMP
         P: begin c1 = 1; nextstate = B; end

         // for ADD
         Q: begin c3_ = 1; nextstate = R; end
         R: begin c4 = 1; nextstate = S; end
         S: begin c4 = 1; nextstate = T; end
         T: begin c14 = 1; nextstate = U; end
         U: begin c10 = 1; c11 = 1; nextstate = V; end
         V: begin c10 = 1; nextstate = B; end

//       Q: begin c10 = 1; c11 = 1; nextstate = B; end
      endcase
   end
endmodule
```

Figure 14 Verilog Code for Decoder and Controller

DE1 pin assignments

	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓		add_a	Location	PIN_F4	Yes			
2	✓		add_b	Location	PIN_D5	Yes			
3	✓		add_c	Location	PIN_D6	Yes			
4	✓		add_d	Location	PIN_J4	Yes			
5	✓		add_e	Location	PIN_L8	Yes			
6	✓		add_f	Location	PIN_F3	Yes			
7	✓		add_g	Location	PIN_D4	Yes			
8	✓		out_a1	Location	PIN_G5	Yes			
9	✓		out_b1	Location	PIN_G6	Yes			
10	✓		out_c1	Location	PIN_C2	Yes			
11	✓		out_d1	Location	PIN_C1	Yes			
12	✓		out_e1	Location	PIN_E3	Yes			
13	✓		out_f1	Location	PIN_E4	Yes			
14	✓		out_g1	Location	PIN_D3	Yes			
15	✓		out_a2	Location	PIN_E1	Yes			
16	✓		out_b2	Location	PIN_H6	Yes			
17	✓		out_c2	Location	PIN_H5	Yes			
18	✓		out_d2	Location	PIN_H4	Yes			
19	✓		out_e2	Location	PIN_G3	Yes			
20	✓		out_f2	Location	PIN_D2	Yes			
21	✓		out_g2	Location	PIN_D1	Yes			
22	✓		clock	Location	PIN_R21	Yes			
23	✓		reset	Location	PIN_R22	Yes			
24	✓		c_0	Location	PIN_R17	Yes			
25	✓		c_3	Location	PIN_R18	Yes			
26	✓		c_4	Location	PIN_U18	Yes			
27	✓		c_2	Location	PIN_Y18	Yes			
28	✓		c_7	Location	PIN_V19	Yes			
29	✓		c_8	Location	PIN_T18	Yes			
30	✓		c_9	Location	PIN_Y19	Yes			
31	✓		in_a1	Location	PIN_J2	Yes			
32	✓		in_b1	Location	PIN_J1	Yes			
33	✓		in_c1	Location	PIN_H2	Yes			
34	✓		in_d1	Location	PIN_H1	Yes			
35	✓		in_e1	Location	PIN_F2	Yes			
36	✓		in_f1	Location	PIN_F1	Yes			
37	✓		in_g1	Location	PIN_E2	Yes			
38	✓		TESTUPCLR	Location	PIN_L2	Yes			
39	✓		c_5	Location	PIN_U19	Yes			
40	✓		c_11	Location	PIN_R19	Yes			
41	✓		TEST1	Location	PIN_U21	Yes			
42	✓		TEST2	Location	PIN_V22	Yes			
43	✓		test1_in	Location	PIN_T21	Yes			
44	✓		TEST0	Location	PIN_U22	Yes			
45	✓		TEST3	Location	PIN_V21	Yes			
46		<<new>>	<<new>>	<<new>>					

Figure 15 DES Pin Assignment

Alternative Design Considerations

To design a control unit, I had couple of ways in my mind: design by using Verilog or design by using schematic diagram. I chose the Verilog because I remembered how troublesome it was working on states while designing Digital Lock Controller in lab 7. Its a wise decision to use Verilog if there are a lot of states in the system.

Integration and Test Plan

Integration Strategy

We started integrating a little by little instead of putting everything together at once. Even doing so, it gave a lot of headache and I can't imagine what I would do if I had to put them all together. There were bugs in every new steps. And it was terribly difficult to debug as the system gets bigger and bigger. I used test inputs and outputs in every doubtful points to make sure signals are flowing as I deemed right.

Test Strategy

To test if our system was performing necessary, I compiled it, programmed/uploaded it into my DE1 board and went along the instructions by changing control unit's clock cycle. The DE1 was made to display the outputs on the HEX on its board. The first hex was used for reading address RAM is currently working on. The second and third hex was used for showing out the values at MDO. The forth hex was used for displaying the accumulator value.

Simulation results from Quartus II

I didn't use simulation later when I integrated the components together, but I did run a simulation to make sure that my controllers were working correctly. For instance, some of the simulation diagrams are given below.

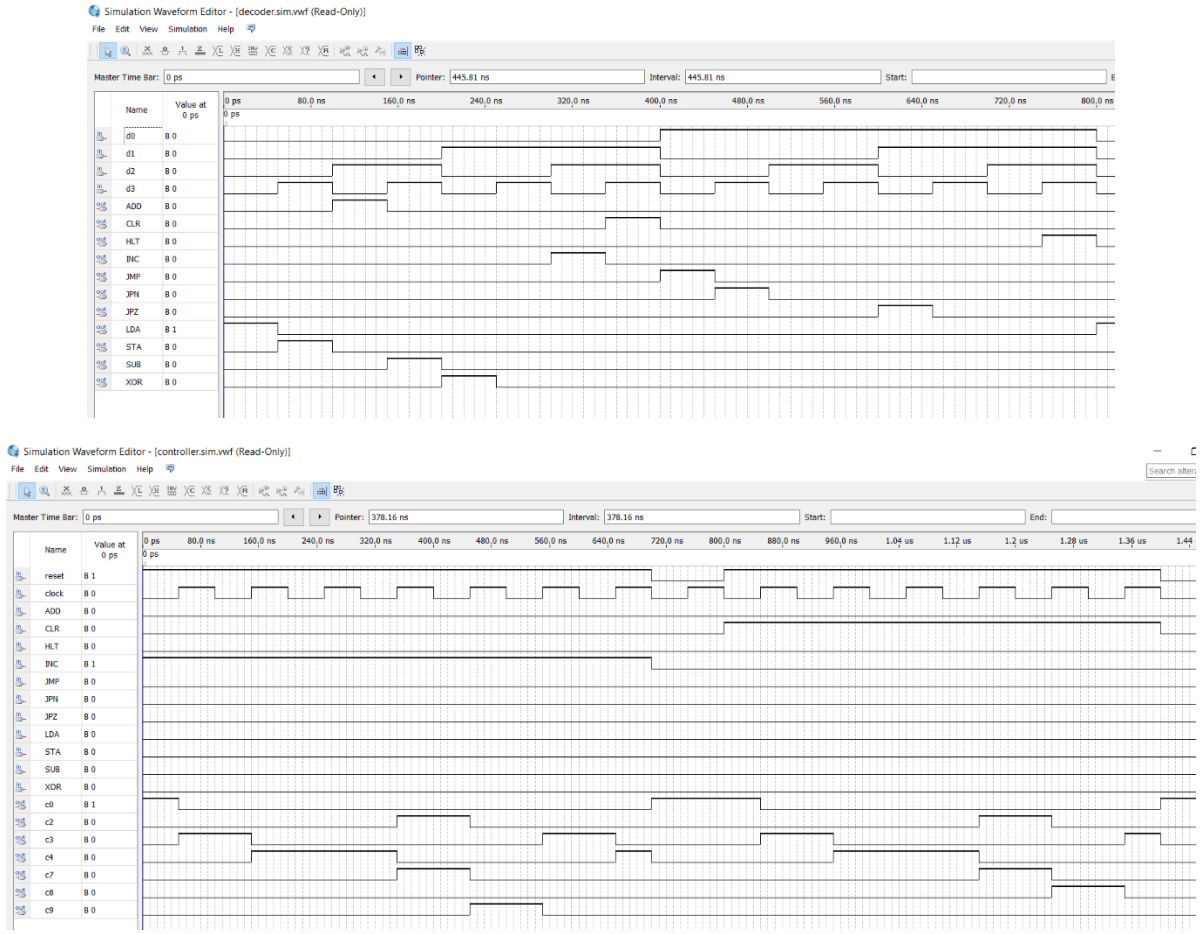


Figure 16 Simulation result for Decoder and Controller

Test results from DE1 implementation

The hex-output of DE1 on various parts are given below. The first is the memory address, the second and third are the values on MDO bus and the fourth is the value of Accumulator.

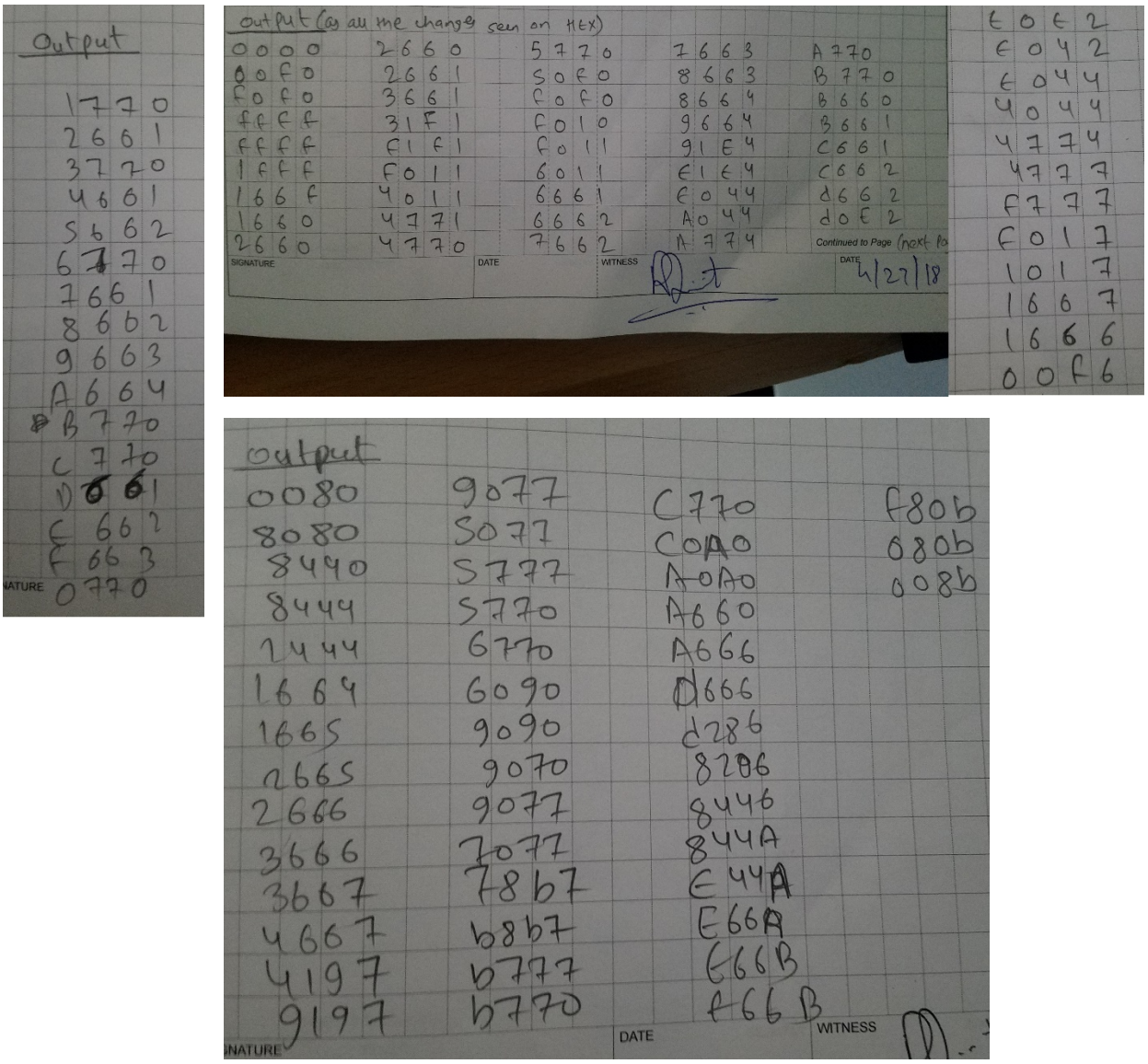


Figure 17 DES HEX Outputs on Part A, B and C respectively

Conclusion

Resolution of design and/or implementation issues

Many issues came up during the design and implementations. Sometimes STA wouldn't write the address, sometimes ADD would add to wrong values, sometimes JMP would jump but then everything goes weird, sometimes the counter would count as 1, 2, 3, C, D... Debugging them were time consuming and hectic. I used to put in test outputs and tests input with pin assignments to LEDs and Switches to figure out where I went wrong and fix them. My professor helped me with some issues too.

Lessons learned

It was one of the most interesting project I have ever done. Now I know how to make my own CPU (HOW COOL IS THAT?!). This project utilized knowledges that I learned in the class entire semester. This project utilized back to back labs in designing ALU, CU, Registers, etc. I also learned how to work on big system, how to start by working on simple components and put them together to form larger and so on. I learned how to find out and fix bugs in big buggy systems. I learned to do things "Digital-Logically".