Varun Advani
Lab Section: 1
ID: 137730642

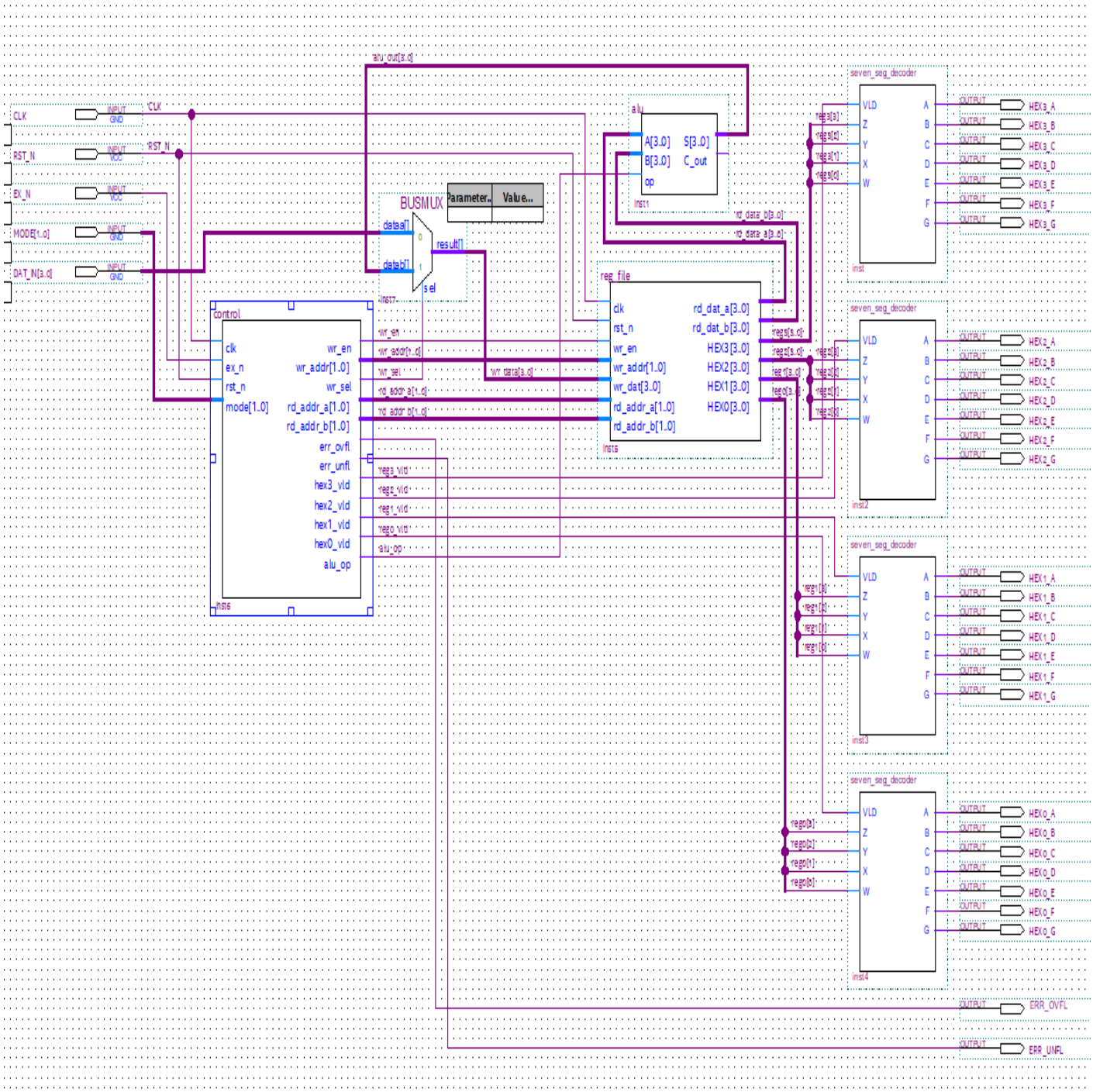
Final Project Report

The final project that I chose was the project on Stack Arithmetic (Project #2). The project report contains all the components that I used to put together this project, which includes, Verilog code for the Finite State Machine logic and the seven-segment display, schematic files for the ALU and Register file, and all the components used to put these top-level components together.

Basic Idea:

The basic idea for the implementation of this project was to use a Control Unit that would act like a Finite State Machine to evaluate the next state of the stack at any given instance, but also to evaluate other components essential to the stack operations such as the error overflow, the error underflow, and the validity of the 4 HEX panels used for the seven-segment display. With respect to loading and manipulating values on the stack, the idea was to use a 4-bit register file with 2 read ports and 1 write port that would not only load values inputted to it, but also send the read data to the Arithmetic Logic Unit (ALU), and the ALU carries out either the addition or subtraction operation, as selected by the user. Furthermore, a BUS MUX would be used to load values into the register either from the ALU or the user input, that is activated by a write select line coded in the control logic and would be sent to the write port of the register file. The register file then sends the data to the seven-segment display that displays the Stack onto the Altera Board.

Top Level Diagram:



Control Unit:

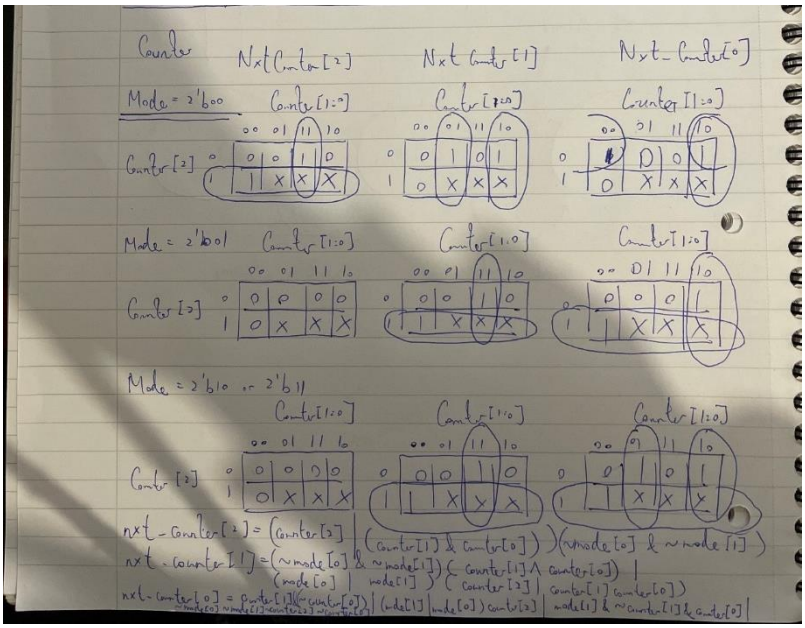
Input			State	Output												
Current Stack Counter	Action	Description	Next Stack Counter	Write Enable	Write Address	Write Select	Read Address A	Read Address B	ALU Op	Error Overflow	Error Underflow	HEX3 VALID	HEX2 VALID	HEX1 VALID	HEX0 VALID
000	00	Push	001	1	00	0	XX	XX	X	0	0	0	0	0	0
001	00	Push	010	1	01	0	XX	XX	X	0	0	1	0	0	0
010	00	Push	011	1	10	0	XX	XX	X	0	0	1	1	0	0
011	00	Push	100	1	11	0	XX	XX	X	0	0	1	1	1	0
100	00	Push	100	0	XX	X	XX	XX	X	1	0	1	1	1	1
000	01	Pop	000	0	XX	X	XX	XX	X	0	1	0	0	0	0
001	01	Pop	000	0	XX	X	XX	XX	X	0	0	1	0	0	0
010	01	Pop	001	0	XX	X	XX	XX	X	0	0	1	1	0	0
011	01	Pop	010	0	XX	X	XX	XX	X	0	0	1	1	1	0
100	01	Pop	011	0	XX	X	XX	XX	X	0	0	1	1	1	1
000	10	Pop with Add	000	0	XX	X	XX	XX	X	0	1	0	0	0	0
001	10	Pop with Add	001	0	XX	X	XX	XX	X	0	1	1	0	0	0
010	10	Pop with Add	001	1	00	1	01	00	0	0	0	1	1	0	0
011	10	Pop with Add	010	1	01	1	10	01	0	0	0	1	1	1	0
100	10	Pop with Add	011	1	10	1	11	10	0	0	0	1	1	1	1
000	11	Pop with Subtract	000	0	XX	X	XX	XX	X	0	1	0	0	0	0
001	11	Pop with Subtract	001	0	XX	X	XX	XX	X	0	1	1	0	0	0
010	11	Pop with Subtract	001	1	00	1	01	00	1	0	0	1	1	0	0
011	11	Pop with Subtract	010	1	01	1	10	01	1	0	0	1	1	1	0
100	11	Pop with Subtract	011	1	10	1	11	10	1	0	0	1	1	1	1

The control unit takes care of the state of stack at any given instance, based on the 4 modes available to the user:

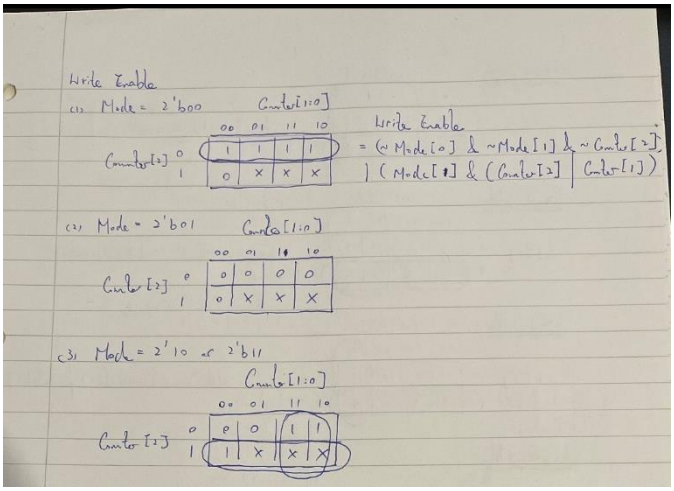
1. Push: Mode 00
2. Pop: Mode 01
3. Pop with Add: Mode 10
4. Poop with Subtract: Mode 11

Next State Counter: Next State Counter derives the next state expression for the stack (positional only) based on the mode of operation selected by the user. There are 5 different possibilities for the number of values on the stack, based on the stack depth of 4 provided in the project description.

K MAP:

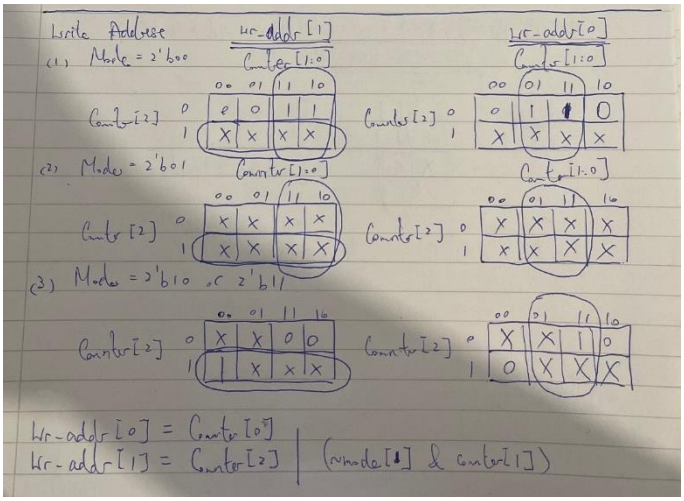


Write Enable: The write enable activates the write port of the register file, and is only active if we are in Mode 00, 10, 11, and there is no error (overflow or underflow).
K MAP:



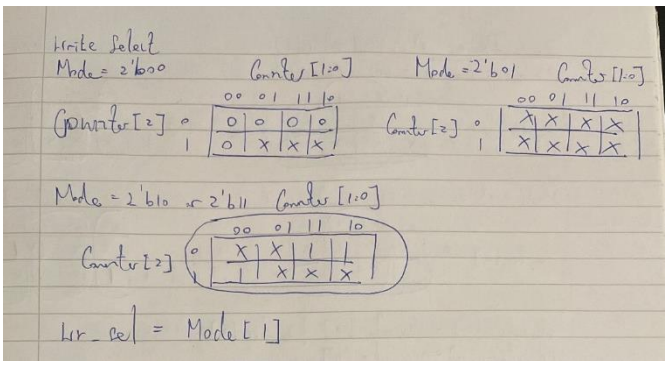
Write Address: The write address complements the write enable, and is the 2 bit-address that writes to one of the 4 registers in the register file.

K MAP:



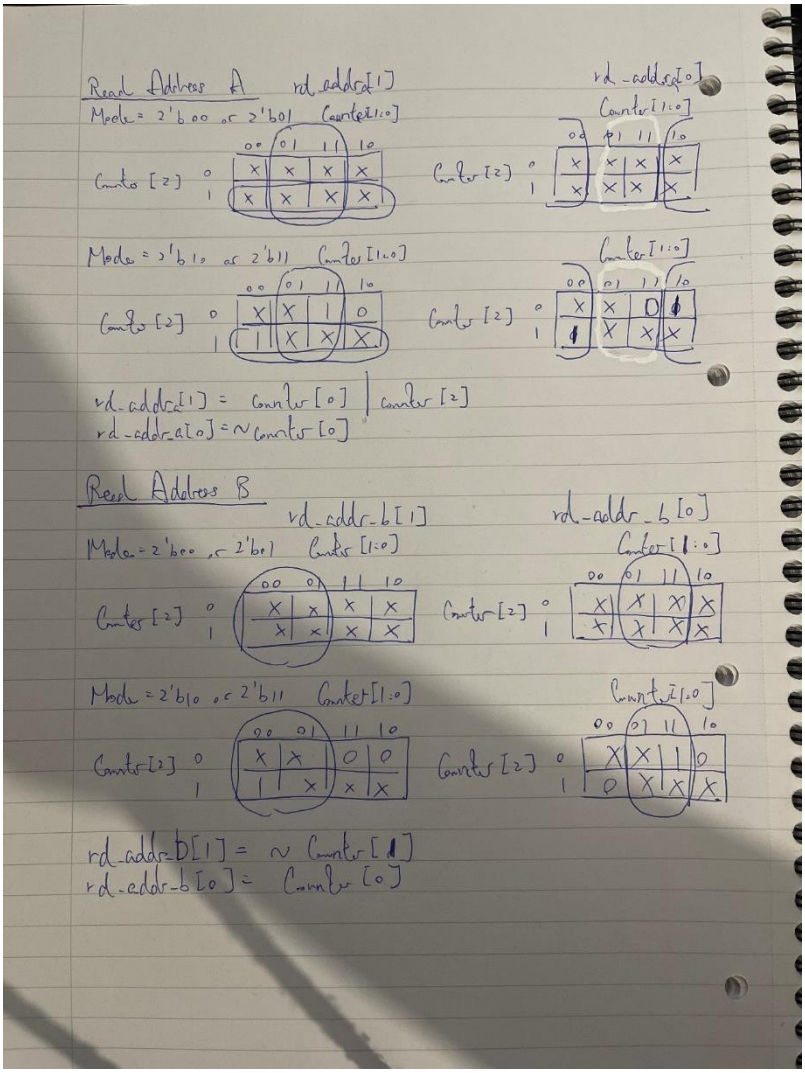
Write Select: The write select is the select line for the BUS MUX that activates the ALU (in Mode 10 and 11) to carry out some arithmetic operations, and read and write the data back into the register file.

K MAP:



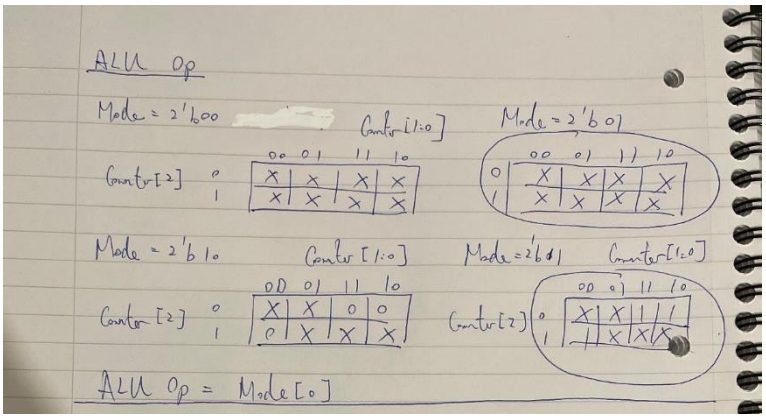
Read Address A and B: The read address is a 2-bit address that reads the data to one of the 4 registers in the register file. There are 2 read ports in the register file, that read data from the ALU, and the read address then feeds this data to the respective register in the register files and writes it back into the register file to load the data.

K MAP:



ALU Op: The ALU or the Arithmetic Logic Unit, carries out some arithmetic operations (add/subtract in this case), and feeds it back into the register file with the help of the BUS MUX. In case of subtraction, if the first popped value is less than the second popped value, the 2's compliment of the operation is pushed onto the stack. The ALU op in the control unit provides the mode to the ALU to decide whether it must add/subtract.

K MAP:



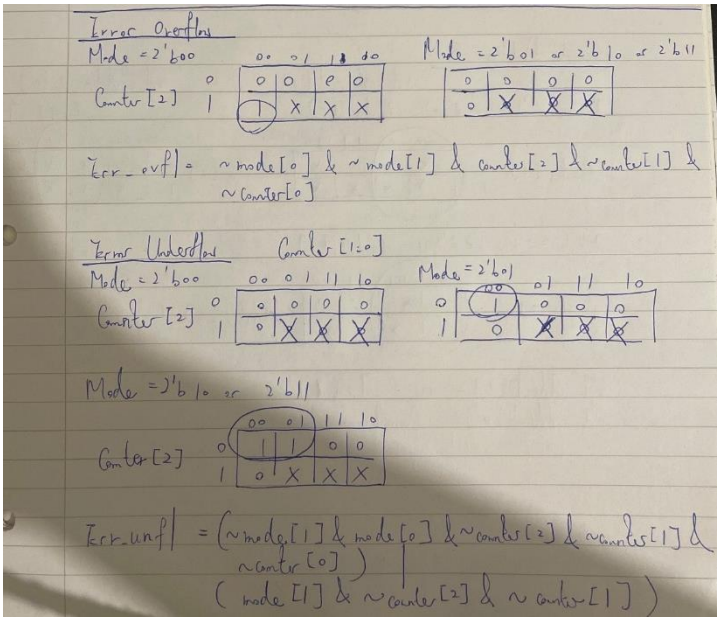
Error (Overflow and Underflow): The error overflow is the state in which the stack is full, and the user tries to push a value onto the stack, which results in an error displayed (LED illuminates).

The error underflow is the state in which:

1. The stack is empty and user tries to pop
2. The stack has less than 2 values, and the user tries to pop with add/subtract

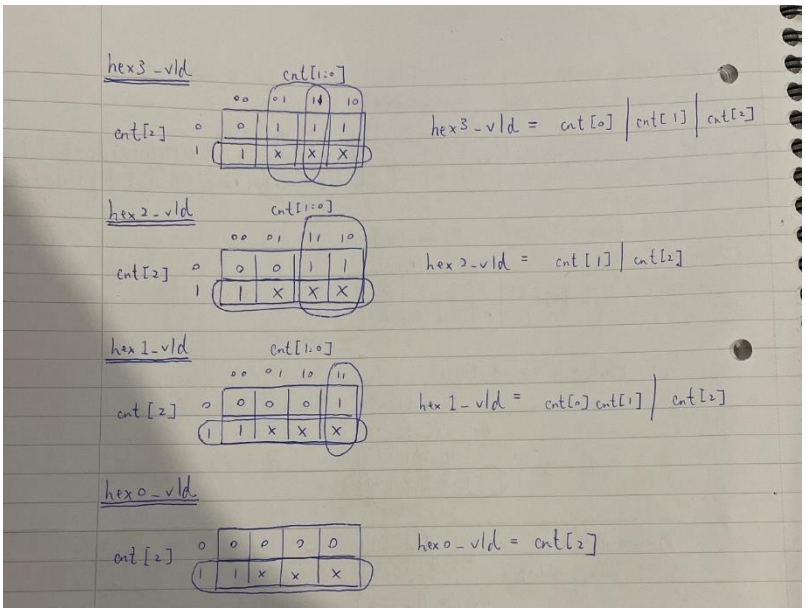
This results in an error displayed (LED illuminates).

K MAP:



Display HEX (HEX3, HEX2, HEX1, HEX0): The control unit calculates the validity of the each of the 3 HEX displays at any state of the stack. Based on all the possible operations that the user could carry out, the HEX display that was on was tabulated and the next state logic was derived. The validity of each HEX is inputted to the seven-segment display.

K MAP:



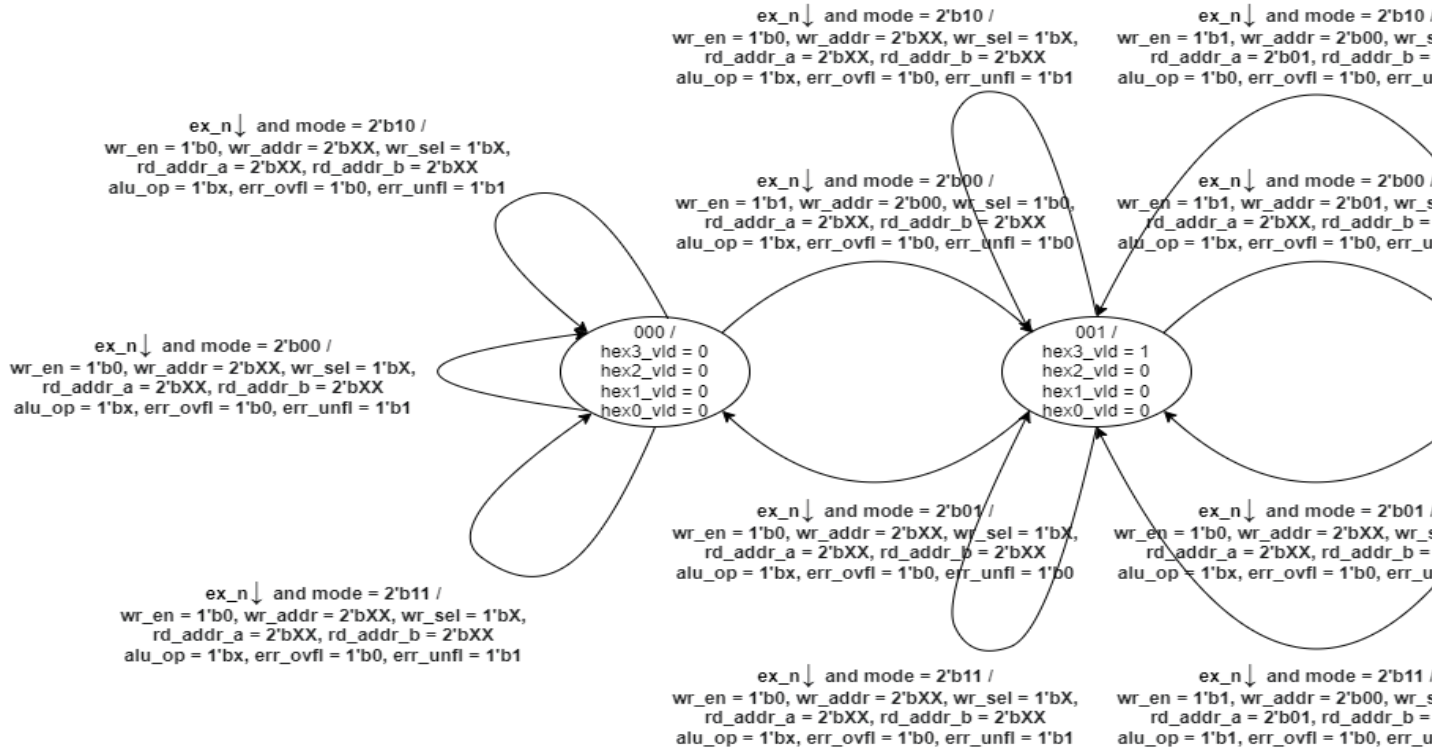
Control Unit Verilog Code:

```
module control(
    clk,
    rst_n,
    ex_n,
    mode,
    wr_en,
    err_ovfl,
    err_unfl,
    hex3_vld,
    hex2_vld,
    hex1_vld,
    hex0_vld,
    alu_op,
    wr_sel,
    rd_addr_a,
    rd_addr_b,
    wr_addr
);

input wire  clk;
input wire  rst_n;
input wire  ex_n;
input wire  [1:0] mode;
output wire wr_en;
output wire err_ovfl;
output wire err_unfl;
output wire hex3_vld;
output wire hex2_vld;
output wire hex1_vld;
output wire hex0_vld;
output wire alu_op;
output wire wr_sel;
output wire [1:0] rd_addr_a;
output wire [1:0] rd_addr_b;
output wire [1:0] wr_addr;

wire  cnt_en;
reg   [2:0] cnt;
wire[2:0] nxt_cnt;
reg   ex_dly;

assign   alu_op = mode[0];

assign   hex3_vld = cnt[0] | cnt[1] | cnt[2];
assign   hex2_vld = cnt[1] | cnt[2];
assign   hex1_vld = cnt[0] & cnt[1] | cnt[2];
assign   hex0_vld = cnt[2];

```

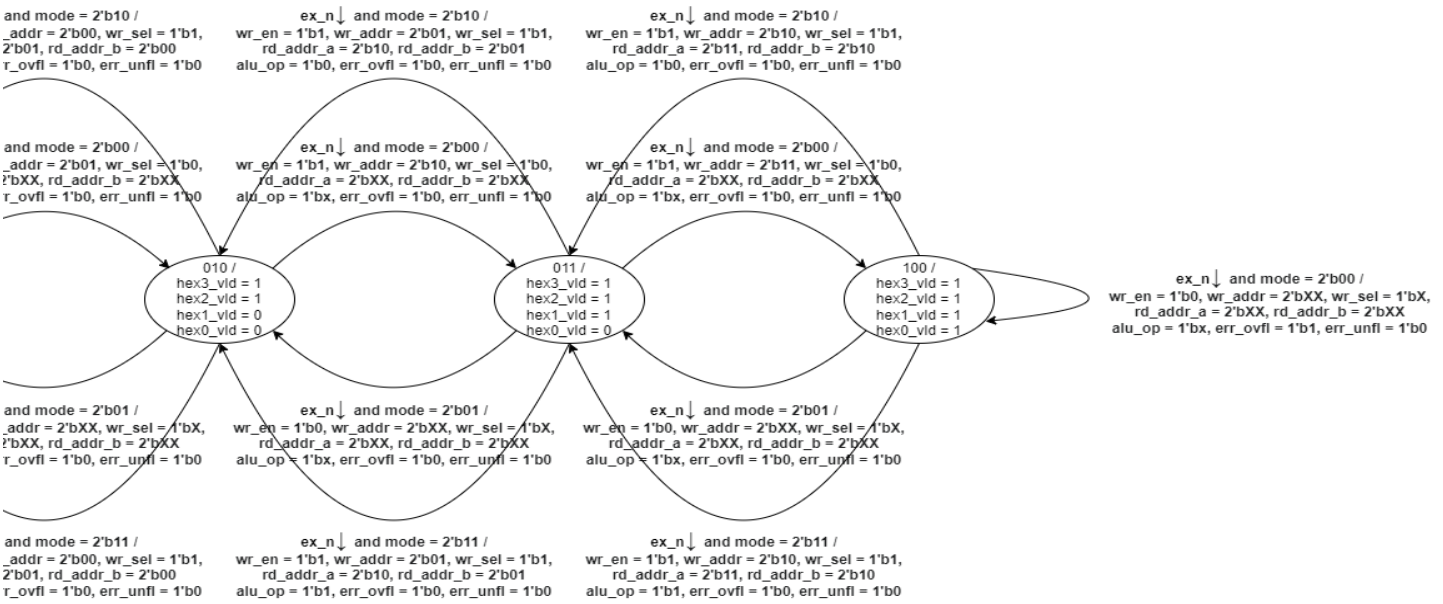
```
assign   wr_en = cnt_en & (
                  (~mode[0] & ~mode[1] & ~cnt[2]) |
                  (mode[1] & (cnt[1] | cnt[2]))
                );

assign  wr_sel = mode[1];

assign  wr_addr[1] = cnt[2] | (~mode[1] & cnt[1]);
assign  wr_addr[0] = cnt[0];

assign  rd_addr_a[1] = cnt[0] | cnt[2];
assign  rd_addr_a[0] = ~cnt[0];

assign  rd_addr_b[1] = ~cnt[1];
assign  rd_addr_b[0] = cnt[0];

assign  err_ovfl = ~mode[0] & ~mode[1] & cnt[2];
assign  err_unfl = (mode[0] & ~mode[1] & ~cnt[2] & ~cnt[1] & ~cnt[0]) |
                   (mode[1] & ~cnt[2] & ~cnt[1]);

//  Falling Edge Eetector
always @(posedge clk or negedge rst_n) begin
  if (rst_n == 1'b0) begin
    ex_dly <= 1'b1;
  end else begin
    ex_dly <= ex_n;
  end
end
assign cnt_en = ex_dly & ~ex_n;

assign  nxt_cnt[2] = (~mode[0] & ~mode[1] & (cnt[2] | (cnt[1] & cnt[0])));
assign  nxt_cnt[1] = (~mode[0] & ~mode[1] & (cnt[0] ^ cnt[1])) |
                     ((mode[0] | mode[1]) & (cnt[2] | (cnt[1] & cnt[0])));
assign  nxt_cnt[0] = (~mode[0] & ~mode[1] & ~cnt[2] & ~cnt[0]) |
                     ((mode[0] | mode[1]) & cnt[2]) |
                     (mode[1] & ~cnt[1] & cnt[0]) |
                     (cnt[1] & ~cnt[0]);

always @(posedge clk or negedge rst_n) begin
  if ( rst_n == 1'b0 ) begin
    cnt <= 3'b000;
  end else if (cnt_en) begin
    cnt <= nxt_cnt;
  end
end

endmodule
```

Finite State Machine Diagram:

First Half:



Second Half:

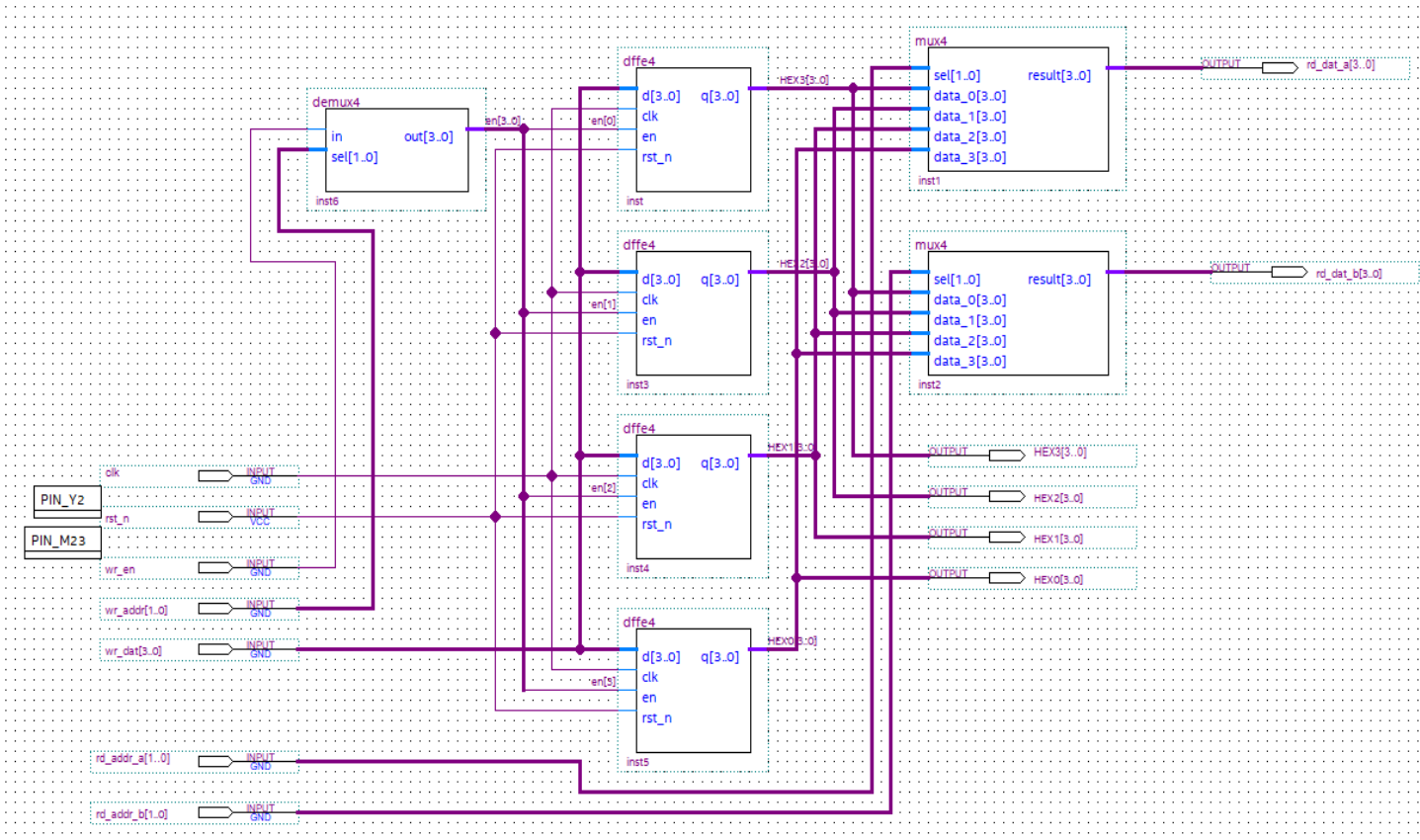


Register File:

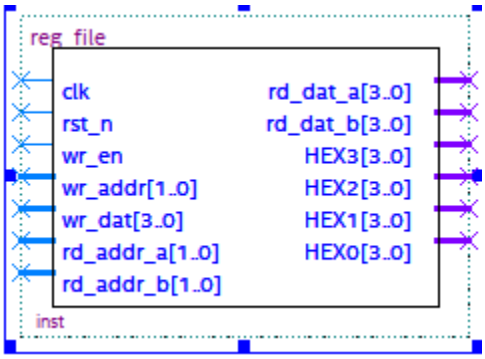
The register file is a 4-bit 4 register, register file with 2 read ports and 1 write port. The register file consists of a write enable that enables one of the 4 registers based on their 2-bit write address. The register file has a reset signal and like the control logic, operates based on the positive edge of the system clock (50 MHz) or the negative edge of the reset signal.

Data is loaded onto the register file based on a 4-bit user input, or the arithmetic operations conducted by the ALU, with the help of a BUS MUX of width 4. The add/subtract value from the ALU is sent to the BUS MUX, and if the user selected a mode of operation that would activate the write select (mode 10 or 11), the write select may send the output of the ALU to the write port of the register file.

Block Diagram:



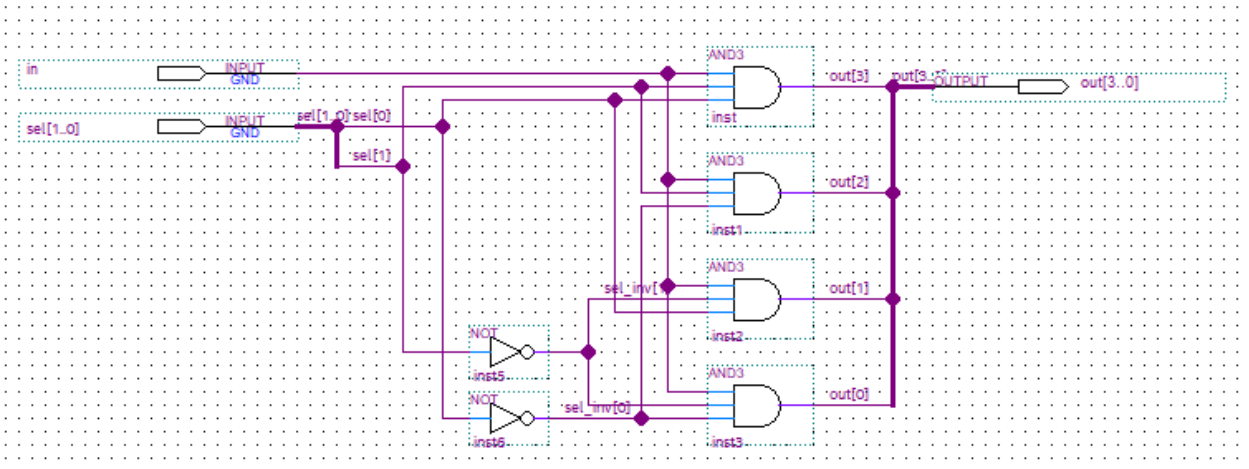
Symbol File:



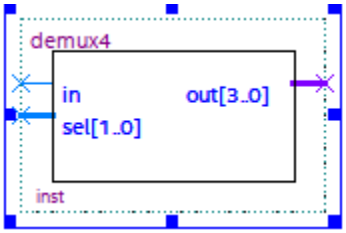
1 to 4 De-Multiplexer:

A 1 to 4 De-Multiplexer was used which takes the write enable as an input, and the 2-bit write address act as select lines s0 and s1. The output of the de-multiplexer act as the load for the 4-bit D-flip flops, each representing a single register.

Block Diagram:



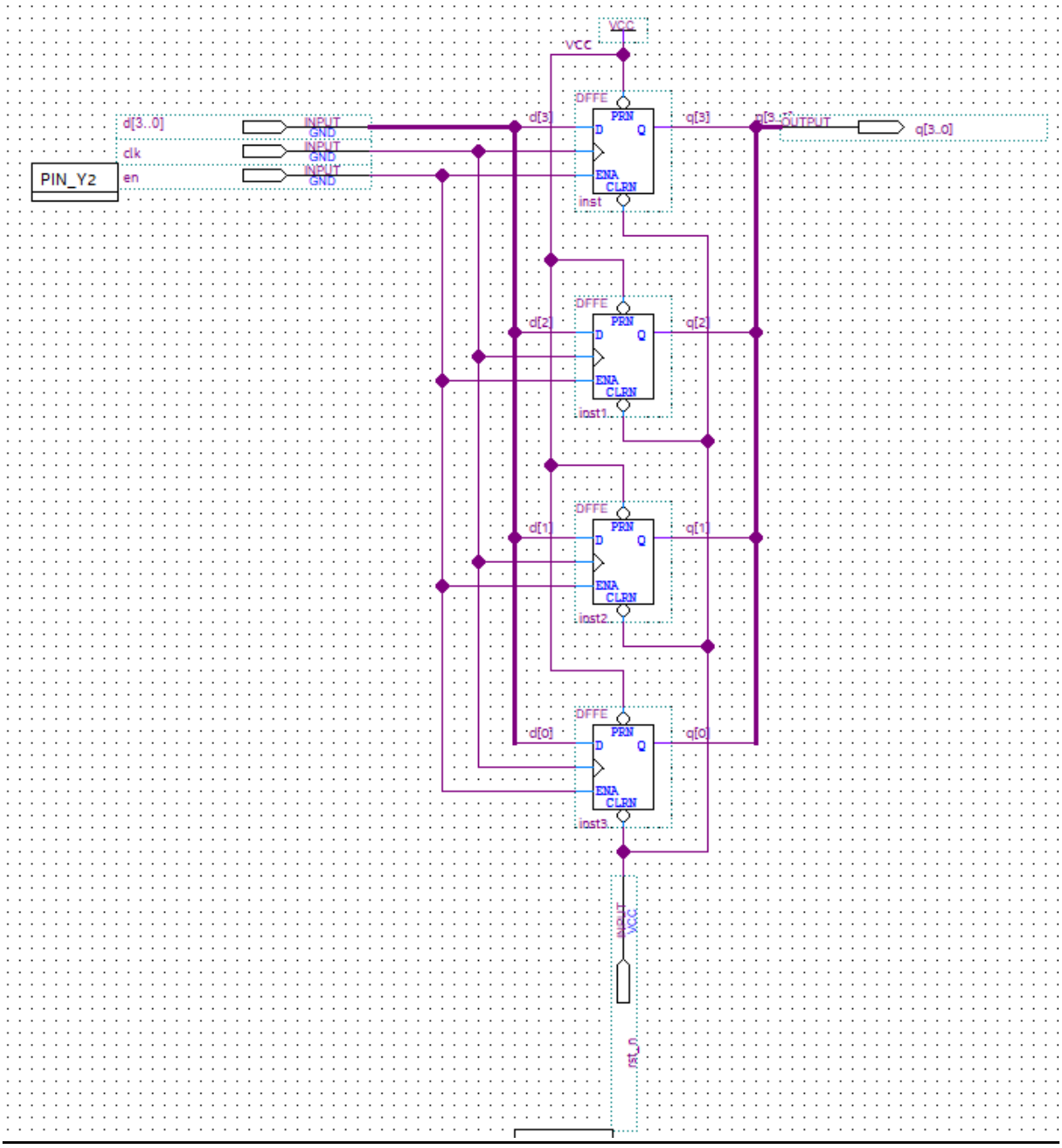
Symbol File:



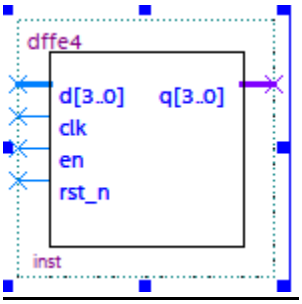
4-bit D-flip flop:

The 4-bit D-flip flop is an arrangement of 4 D-flip flops, each of which represents a single register. The D flip flops act as a memory limit for the register, and load the values onto the register based on the write address, at the positive edge or negative reset of the system clock (50 MHz). The D-flip flop also has a reset signal to clear the values on the registers.

Block Diagram:



Symbol File:



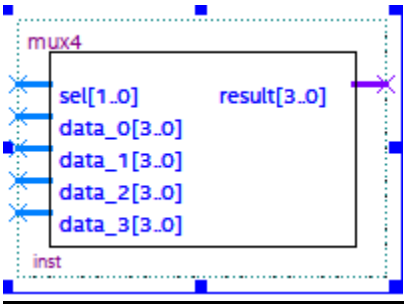
4 to 1 Multiplexer:

The two 4 to 1 Multiplexer's each represent a read port for the register file that has a 2-bit read address as the select lines, which reads the data to one of the 4 registers on the register file. The data from 4 registers is sent to the 4 to 1 MUX, which then outputs the read data based on the read address of the registers in the register file.

Verilog:

```
module mux4(
    data_0,
    data_1,
    data_2,
    data_3,
    sel,
    result
);

input wire  [3:0] data_0;
input wire  [3:0] data_1;
input wire  [3:0] data_2;
input wire  [3:0] data_3;
input wire  [1:0] sel;
output wire [3:0] result;

assign result[0] = ( ~sel[1] & ~sel[0] & data_0[0] )
                 | ( ~sel[1] &  sel[0] & data_1[0] )
                 | (  sel[1] & ~sel[0] & data_2[0] )
                 | (  sel[1] &  sel[0] & data_3[0] );

assign result[1] = ( ~sel[1] & ~sel[0] & data_0[1] )
                 | ( ~sel[1] &  sel[0] & data_1[1] )
                 | (  sel[1] & ~sel[0] & data_2[1] )
                 | (  sel[1] &  sel[0] & data_3[1] );

assign result[2] = ( ~sel[1] & ~sel[0] & data_0[2] )
                 | ( ~sel[1] &  sel[0] & data_1[2] )
                 | (  sel[1] & ~sel[0] & data_2[2] )
                 | (  sel[1] &  sel[0] & data_3[2] );

assign result[3] = ( ~sel[1] & ~sel[0] & data_0[3] )
                 | ( ~sel[1] &  sel[0] & data_1[3] )
                 | (  sel[1] & ~sel[0] & data_2[3] )
                 | (  sel[1] &  sel[0] & data_3[3] );

endmodule
```

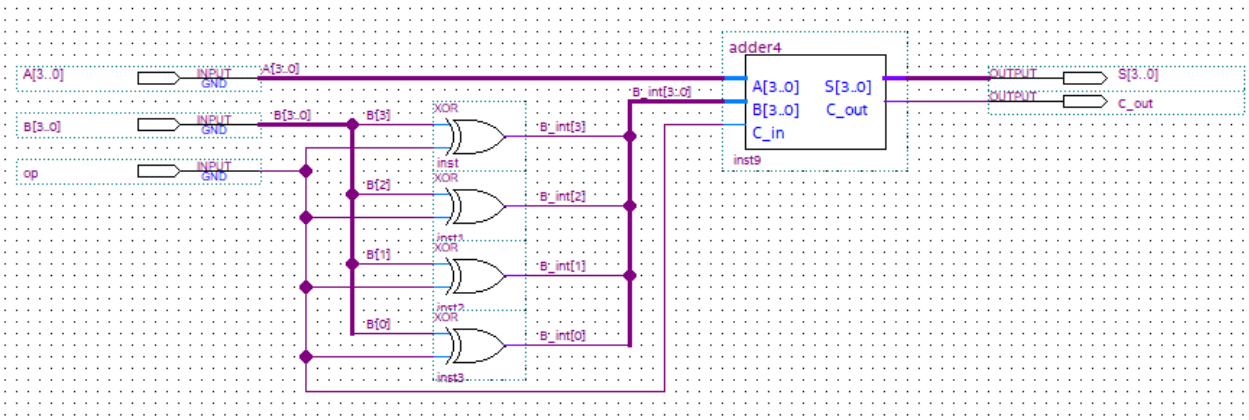
Symbol File:



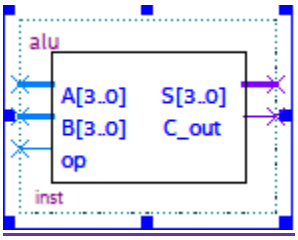
ALU (Arithmetic Logic Unit):

The ALU is used for the operation mode pop with add or pop with subtract, which adds or subtracts the top 2 popped values from the stack, and pushes it back onto the stack. The output of the ALU is sent to the BUS MUX that writes the data to the register file if the write select is active (= 1). The ALU gets the 2 popped values from the read data of the register file, that reads the data from the register corresponding to the read address, and sends it to the ALU to carry out the add/subtract operations. The ALU uses a 4-bit adder and XOR gates to carry out both the add and subtract operations.

Block Diagram File:



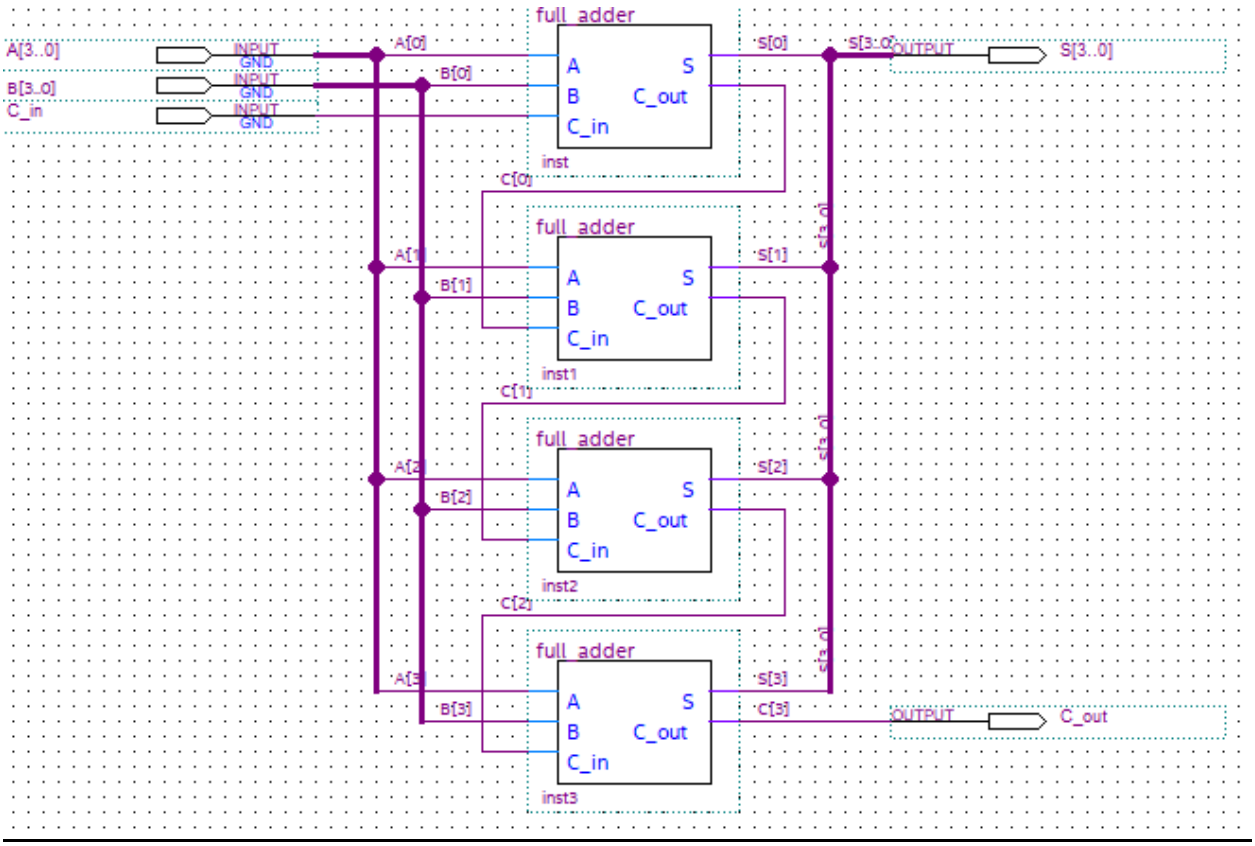
Symbol File:



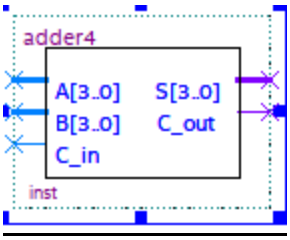
4-bit Adder/Subtracter:

The ALU implements a 4-bit adder circuit to carry out both addition and subtraction arithmetic operations based on the mode of operation (ALU_op) selected by the user. The 4-bit adder circuit implements 4 full adder circuits, each of which have two 4-bit inputs and a carry input which gets the value of ALU_op, and produce a 3-bit sum output and a carry output.

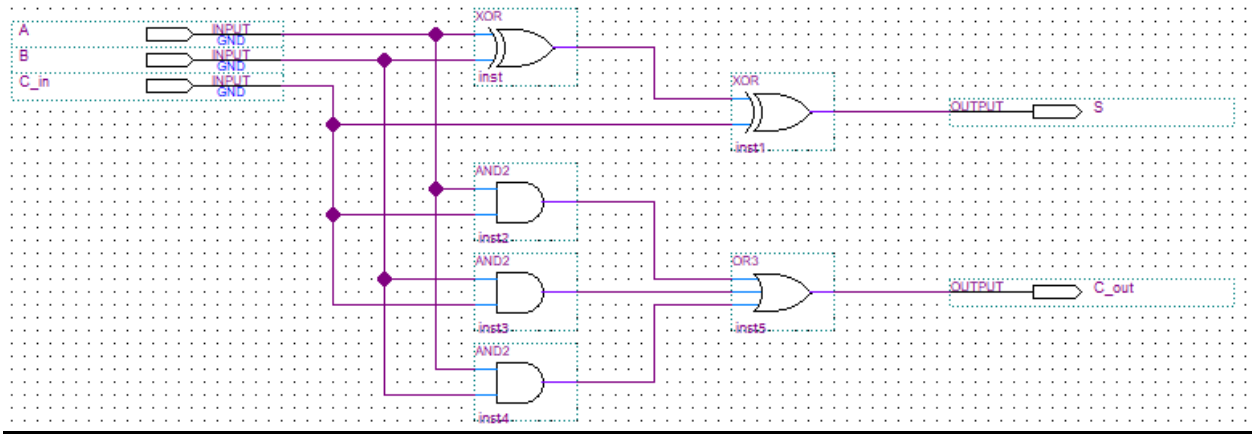
4-bit Adder/Subtracter Block Diagram:



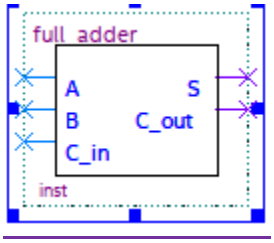
4-bit Adder/Subtracter Symbol File:



Full Adder Block Diagram:



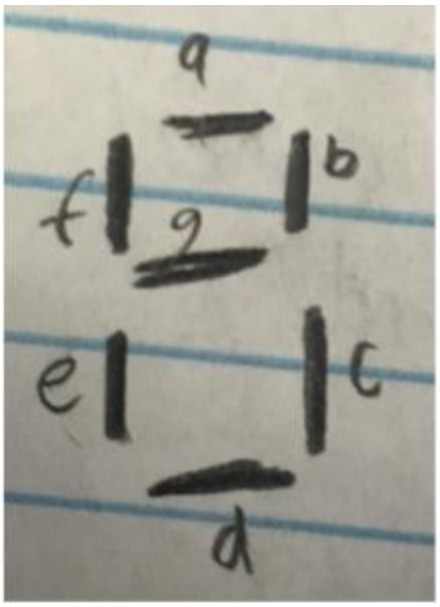
Full Adder Symbol File:



Seven-Segment Display:

The seven-segment display is a key component of the top-level diagram, as it displays the values on the stack. The seven-segment display considers seven different segments(a-g) of the HEX display and displays only the segments relevant to the output. Additionally, the design for my seven-segment display has an additional input VLD, which checks if that HEX is valid, and should be in use. The seven-segment display is activated by an enable, and takes 4 inputs Z, Y, X, W.

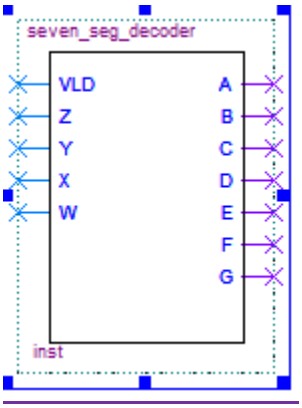
Seven-Segment Display segments:



Verilog Code:

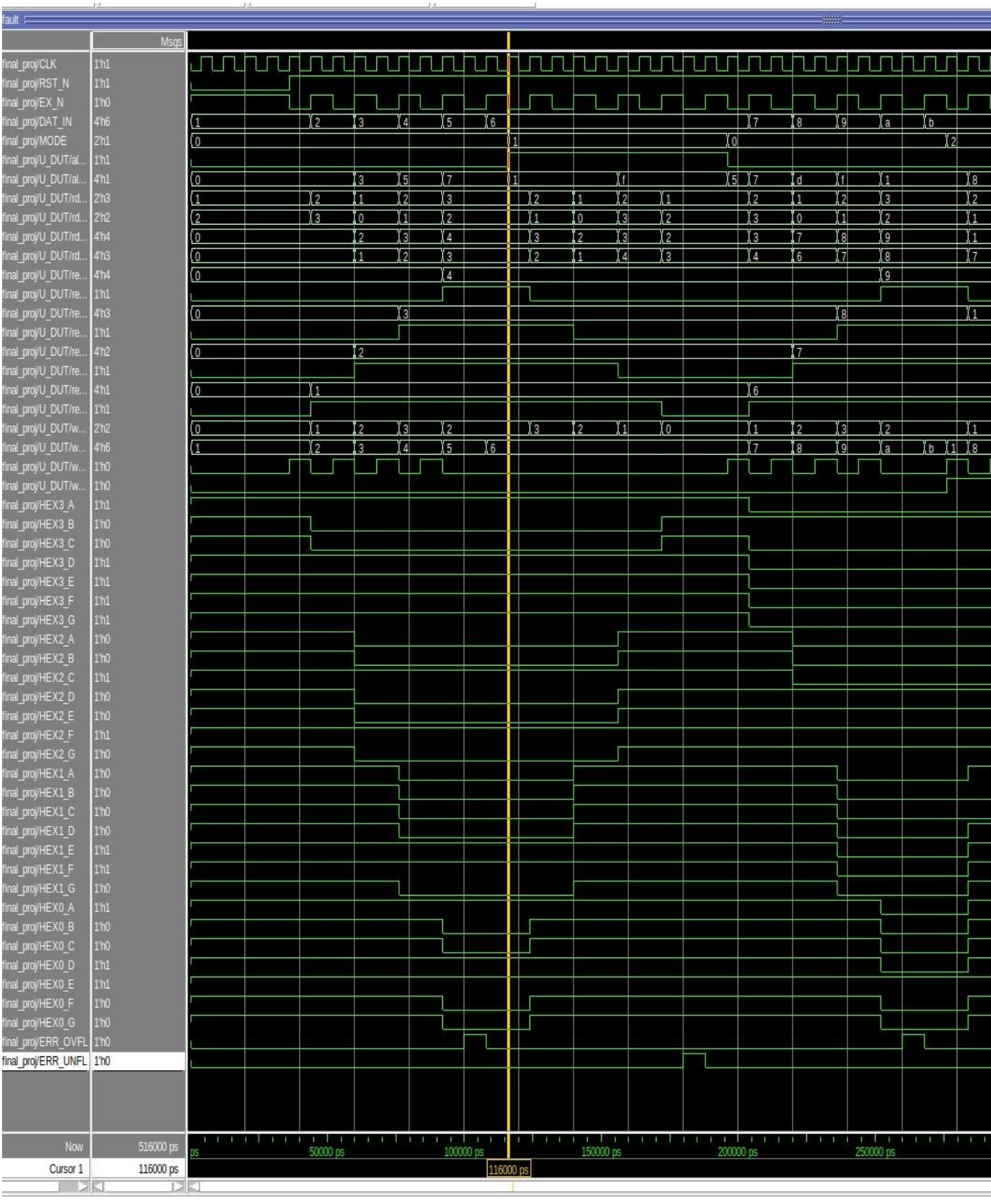
```
module seven_seg_decoder(VLD, Z, Y, X, W, A, B, C, D, E, F, G);
    input VLD, Z, Y, X, W;
    output reg A, B, C, D, E, F, G;

    always @(VLD or Z or Y or X or W)
    begin
        if (VLD == 1'b0) begin
            {A, B, C, D, E, F, G} = 7'b1111111;
        end else begin
            case({Z, Y, X, W})
                //truth tabling
                4'b0000: {A, B, C, D, E, F, G} = 7'b0000001;
                4'b0001: {A, B, C, D, E, F, G} = 7'b1001111;
                4'b0010: {A, B, C, D, E, F, G} = 7'b0010010;
                4'b0011: {A, B, C, D, E, F, G} = 7'b0000110;
                4'b0100: {A, B, C, D, E, F, G} = 7'b1001100;
                4'b0101: {A, B, C, D, E, F, G} = 7'b0100100;
                4'b0110: {A, B, C, D, E, F, G} = 7'b0100000;
                4'b0111: {A, B, C, D, E, F, G} = 7'b0001111;
                4'b1000: {A, B, C, D, E, F, G} = 7'b0000000;
                4'b1001: {A, B, C, D, E, F, G} = 7'b0000100;
                4'b1010: {A, B, C, D, E, F, G} = 7'b0001000;
                4'b1011: {A, B, C, D, E, F, G} = 7'b1100000;
                4'b1100: {A, B, C, D, E, F, G} = 7'b0110001;
                4'b1101: {A, B, C, D, E, F, G} = 7'b1000010;
                4'b1110: {A, B, C, D, E, F, G} = 7'b0110000;
                4'b1111: {A, B, C, D, E, F, G} = 7'b0111000;
            endcase
        end
    end

endmodule
```

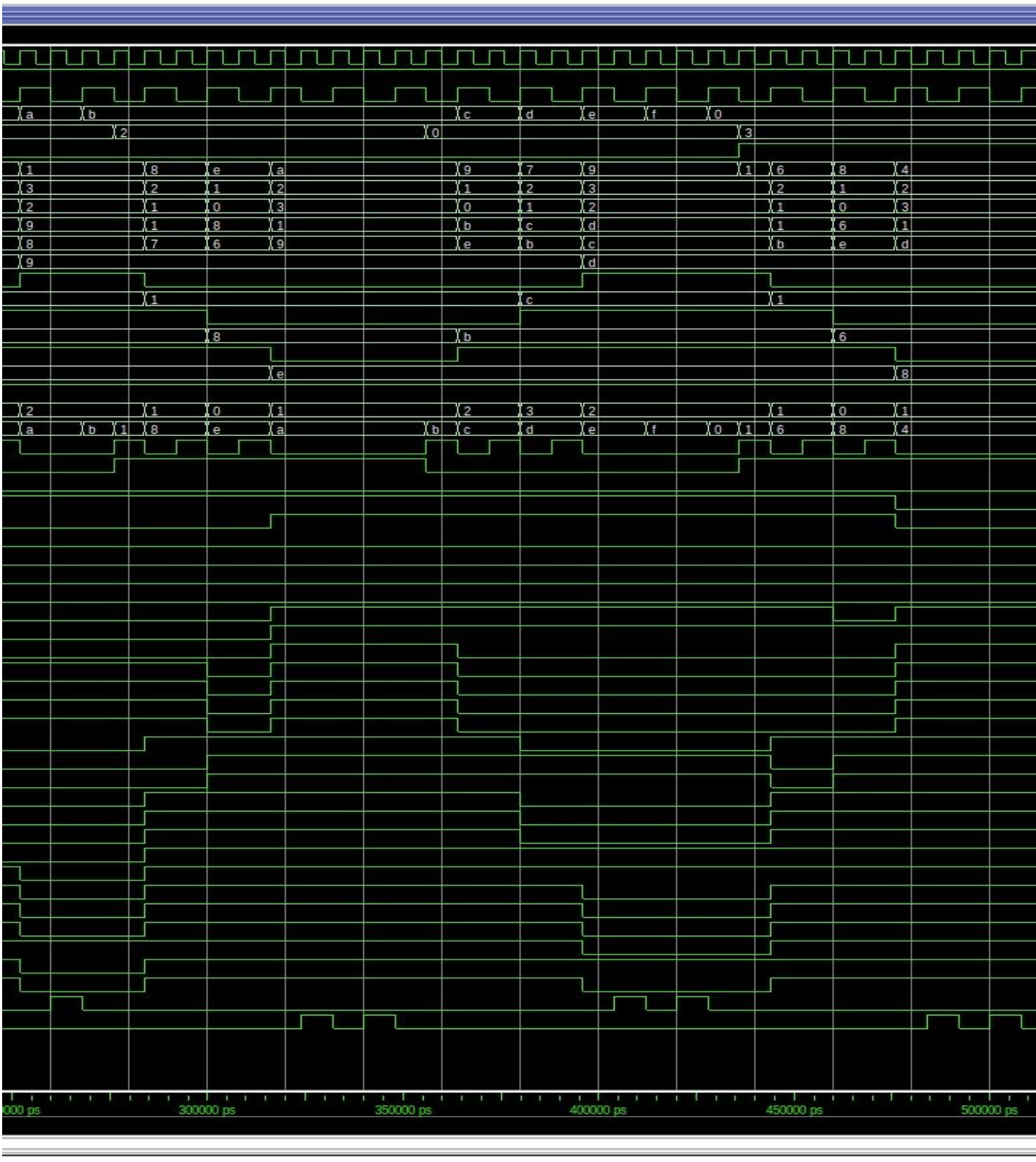
Symbol File:



ModelSim Tests:

I conducted ModelSim tests, for all the top-level components and I've attached the waveform for all the components of this final project below:


fault
Msgs
final_proj/CLK
final_proj/RST_N
final_proj/EX_N
final_proj/DAT_IN
final_proj/MODE
final_proj/U_DUT/al...
final_proj/U_DUT/al...
final_proj/U_DUT/rd...
final_proj/U_DUT/rd...
final_proj/U_DUT/rd...
final_proj/U_DUT/rd...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/re...
final_proj/U_DUT/w...
final_proj/U_DUT/w...
final_proj/U_DUT/w...
final_proj/U_DUT/w...
final_proj/HEX3_A
final_proj/HEX3_B
final_proj/HEX3_C
final_proj/HEX3_D
final_proj/HEX3_E
final_proj/HEX3_F
final_proj/HEX3_G
final_proj/HEX2_A
final_proj/HEX2_B
final_proj/HEX2_C
final_proj/HEX2_D
final_proj/HEX2_E
final_proj/HEX2_F
final_proj/HEX2_G
final_proj/HEX1_A
final_proj/HEX1_B
final_proj/HEX1_C
final_proj/HEX1_D
final_proj/HEX1_E
final_proj/HEX1_F
final_proj/HEX1_G
final_proj/HEX0_A
final_proj/HEX0_B
final_proj/HEX0_C
final_proj/HEX0_D
final_proj/HEX0_E
final_proj/HEX0_F
final_proj/HEX0_G
final_proj/ERR_OVFL
final_proj/ERR_UNFL
1'h1
1'h1
1'h0
4'h6
2'h1
1'h1
4'h1
2'h3
2'h2
4'h4
4'h3
4'h4
1'h1
4'h3
1'h1
4'h2
1'h1
4'h1
1'h1
2'h2
4'h6
1'h0
1'h0
1'h1
1'h0
1'h0
1'h1
1'h1
1'h1
1'h1
1'h0
1'h0
1'h1
1'h0
1'h0
1'h0
1'h0
1'h0
1'h1
1'h1
1'h0
1'h1
1'h0
1'h0
1'h0
1'h0
Now
516000 ps
Cursor 1
116000 ps
ps
50000 ps
100000 ps
150000 ps
200000 ps
250000 ps
116000 ps


300000 ps
350000 ps
400000 ps
450000 ps
500000 ps

Pin Assignments:

Pin Assignment	Description	I/O Direction	Board Signal Name	Pin Assignment	Voltage						
CLK	System Clock	Input	CLOCK_50	PIN_Y2	3.3V		50 MHz	Oscillator			
RST_N	Global Reset (Active Low)	Input	KEY0	PIN_M23	3.3V		Push Button				
EX_N	Execute (Active Low)	Input	KEY1	PIN_M21	3.3V		Push Button				
MODE[1]	Operation Mode	Input	SW1	PIN_AC28	Depending on JP7	2.5V	Slide Switch				
MODE[0]		Input	SW0	PIN_AB28	Depending on JP7	2.5V	Slide Switch				
DAT_IN[3]	Input Data	Input	SW17	PIN_Y23	Depending on JP7	2.5V	Slide Switch				
DAT_IN[2]		Input	SW16	PIN_Y24	Depending on JP7	2.5V	Slide Switch				
DAT_IN[1]		Input	SW15	PIN_AA22	Depending on JP7	2.5V	Slide Switch				
DAT_IN[0]		Input	SW14	PIN_AA23	Depending on JP7	2.5V	Slide Switch				
ERR_OVFL	Overflow Error Indicator	Output	LEDR[0]	PIN_G19	2.5V		LED				
ERR_UNFL	Underflow Error Indicator	Output	LEDR[1]	PIN_F19	2.5V		LED				
HEX0_A		Output	HEX0[0]	PIN_G18	2.5V		7-Segment Displays				
HEX0_B		Output	HEX0[1]	PIN_F22	Depending on JP7	2.5V	7-Segment Displays				
HEX0_C		Output	HEX0[2]	PIN_E17	Depending on JP7	2.5V	7-Segment Displays				
HEX0_D		Output	HEX0[3]	PIN_L26	Depending on JP7	2.5V	7-Segment Displays				
HEX0_E		Output	HEX0[4]	PIN_L25	Depending on JP7	2.5V	7-Segment Displays				
HEX0_F		Output	HEX0[5]	PIN_J22	Depending on JP7	2.5V	7-Segment Displays				
HEX0_G		Output	HEX0[6]	PIN_H22	Depending on JP7	2.5V	7-Segment Displays				
HEX1_A		Output	HEX1[0]	PIN_M24	Depending on JP7	2.5V	7-Segment Displays				
HEX1_B		Output	HEX1[1]	PIN_Y22	Depending on JP7	2.5V	7-Segment Displays				
HEX1_C		Output	HEX1[2]	PIN_W21	Depending on JP7	2.5V	7-Segment Displays				
HEX1_D		Output	HEX1[3]	PIN_W22	Depending on JP7	2.5V	7-Segment Displays				
HEX1_E		Output	HEX1[4]	PIN_W25	Depending on JP7	2.5V	7-Segment Displays				
HEX1_F		Output	HEX1[5]	PIN_U23	Depending on JP7	2.5V	7-Segment Displays				
HEX1_G		Output	HEX1[6]	PIN_U24	Depending on JP7	2.5V	7-Segment Displays				
HEX2_A		Output	HEX2[0]	PIN_AA25	Depending on JP7	2.5V	7-Segment Displays				
HEX2_B		Output	HEX2[1]	PIN_AA26	Depending on JP7	2.5V	7-Segment Displays				
HEX2_C		Output	HEX2[2]	PIN_Y25	Depending on JP7	2.5V	7-Segment Displays				
HEX2_D		Output	HEX2[3]	PIN_W26	Depending on JP7	2.5V	7-Segment Displays				
HEX2_E		Output	HEX2[4]	PIN_Y26	Depending on JP7	2.5V	7-Segment Displays				
HEX2_F		Output	HEX2[5]	PIN_W27	Depending on JP7	2.5V	7-Segment Displays	**Jumper**			
HEX2_G		Output	HEX2[6]	PIN_W28	Depending on JP7	2.5V	7-Segment Displays	JP7	Short Pins 5 and 6	2.5V	**Default**
HEX3_A		Output	HEX3[0]	PIN_V21	Depending on JP7	2.5V	7-Segment Displays				
HEX3_B		Output	HEX3[1]	PIN_U21	Depending on JP7	2.5V	7-Segment Displays	**Jumper**			
HEX3_C		Output	HEX3[2]	PIN_AB20	Depending on JP6	3.3V	7-Segment Displays	JP6	Short Pins 5 and 6	2.5V	
HEX3_D		Output	HEX3[3]	PIN_AA21	Depending on JP6	3.3V	7-Segment Displays		Short Pins 7 and 8	3.3V	**Default**
HEX3_E		Output	HEX3[4]	PIN_AD24	Depending on JP6	3.3V	7-Segment Displays				
HEX3_F		Output	HEX3[5]	PIN_AF23	Depending on JP6	3.3V	7-Segment Displays				
HEX3_G		Output	HEX3[6]	PIN_Y19	Depending on JP6	3.3V	7-Segment Displays				