Mason Walls

Section U

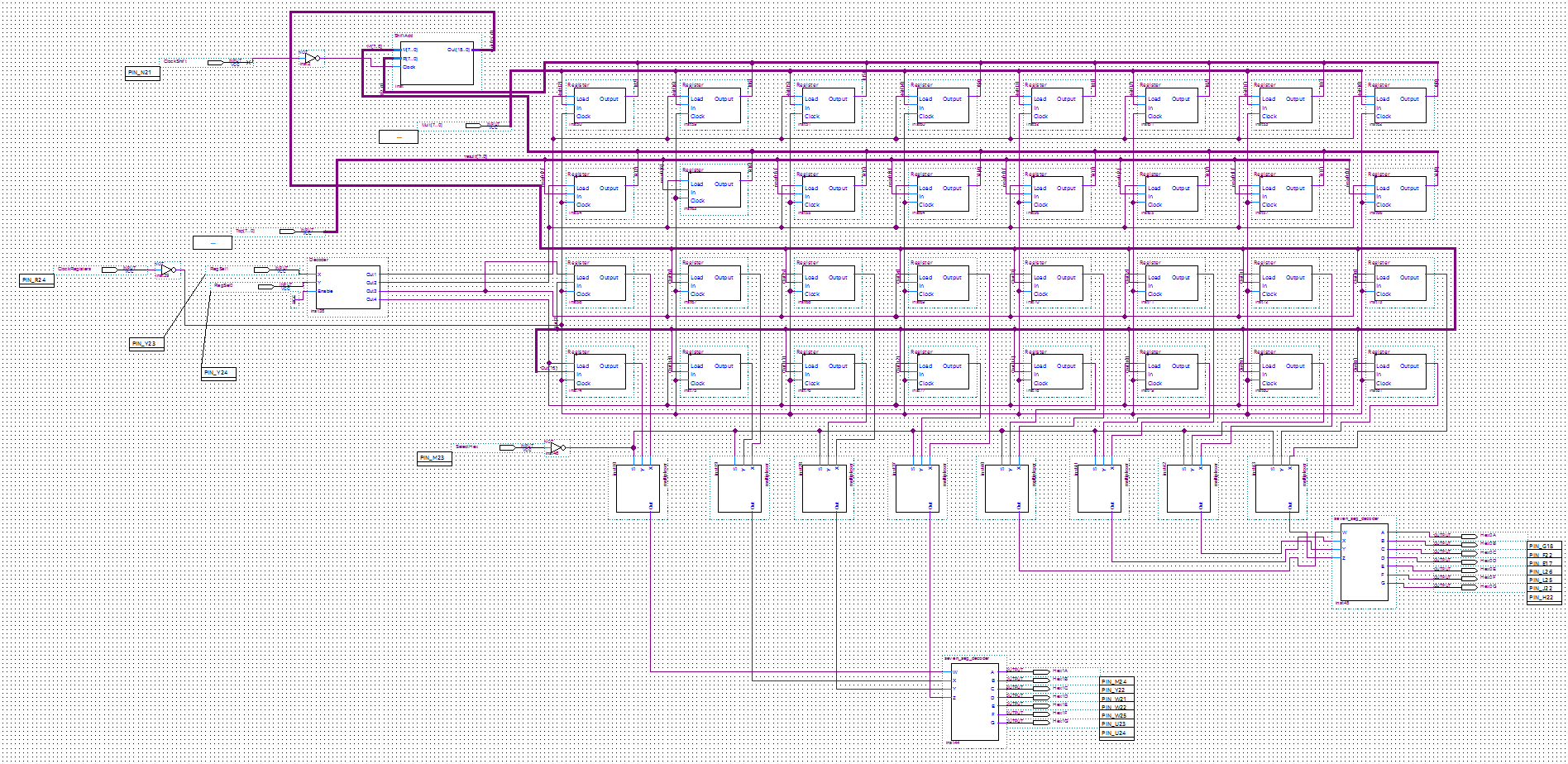
CPRE 281

Student ID: 910333448

Final Project

I completed project option 2, the booths multiplier. Details of all block diagram and verilog diagrams are shown below. It contains an overview of the final module, and each module contained in the final module.

Top Level Diagram:



The top level diagram contains 2 8-bit number inputs (Top[7..0] and Mult[7..0]), 2 1-bit decoder inputs(RegSel1, RegSel0), 2 clock inputs(ClockRegisters, ClockShift), and 1 select input(SelectHex). The state machine for my circuit was having difficulties and is not shown in the top level diagram. The different modules involved in the circuit are: 32 1-bit registers, 8 2-1 multiplexers, 2 seven segment decoders, 1 2-4 decoder, and one “ShiftAdd” module. The circuit starts by loading the first number (Pins 0-7) into the top register, with RegSel1=0 and RegSel0=0. Then the second number is loaded with RegSel1=0 and RegSel0 = 1. ClockRegisters is pulsed once for each load into the registers. Then, ClockShift is pulsed, the numbers are multiplied, and then the 16-bit result is outputted by the module. The numbers then are stored into the registers bottom two registers with the selects being 10 and 11. Once loaded, the results are displayed on hex display 1 and 0. The display will show the right half of the 16-bit number. When SelectHex is held, the display will change to the left half of the result.

2-4 Decoder:

module Decoder(X,Y,Out1,Out2,Out3,Out4);

input X,Y;

output Out1,Out2,Out3,Out4;

assign Out1 = ~X&~Y;

assign Out2 = ~X&Y;

assign Out3 = X&~Y;

assign Out4 = X&Y;

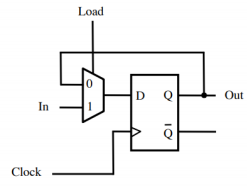
endmodule

This module takes 2 different bit values and assigns them to one of the for values. This is used to select which register we write to. The truth table is shown below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | Y | Out4 | Out3 | Out2 | Out1 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Register:

The register is what stores the data that we are using. There are 32 of them in total to make a 4x8 register file. The block diagram looks as follows:



The load is taken from the 2-4 decoder. In is taken from the input values Top and Mult. Clock is from ClockRegisters, and Out is what we give the ShiftAdd module. The register will keep its current value forever as long as we put load as 0. Once we want to load a new value, we set load to 1, and the input will be the new value.

ShiftAdd:

ShiftAdd does exactly what the name says. It shifts a number, then adds two together. The cpde is as follows:

module ShiftAdd(M, R, Clock, Out);

input Clock;

input signed [7:0]M, R;

output reg [15:0] Out;

wire [15:0] T = {8'b0,R};

reg signed [16:0] P;

wire [16:0] MP = {M,9'b0};

wire [16:0] MM = {~M+1'b1,9'b0};

integer i;

always @(posedge Clock)

begin

P = {T, 1'b0};

for(i = 0; i < 8; i = i+1)

begin

if(P[1] == 1 & P[0] == 0)

begin

P = P+MM;

end

if(P[0] == 1 & P[1] == 0)

begin

P = P+MP;

end

P = P>>>1;

end

Out = P[16:1];

end

endmodule

We have the inputs from ClockShift, and the top two register files in bus form. The inputs are stored as signed values because we need to do sign extensions later. First we make the input ‘R’ a 16-bit number by adding 8 0’s to the front of it. We then create MP by taking M and adding 9 0’s to the back of it for easy access to the receptacle of the algorithm. We do the same for MM, but M is in two’s compliment form. P is the number that we are going to be shifting, and we will add MM and MP to it when necessary. This value is 17 bits because we need to know that P-1 value as well as P15-0. The code checks if P[1][0] = 10, and if it is, we add MM to P. otherwise, if P[1][0] = 01, we add MP to P. Once this is all done we use the >>> operator to perform an arithmetic right shift of one bit onto P. This operator does a sign extension for us if P is ‘signed’. In this case, it is. The adding and shifting is in a for loop which will execute that code 8 times. Once its done, we return bits 16-0, because we don’t want P-1;

Seven Segment Decoder:

module seven\_seg\_decoder(A, B, C, D, E ,F ,G, W, X, Y, Z);

input W, X, Y, Z;

output A, B, C, D, E, F, G;

reg A, B ,C ,D , E, F, G;

always@(W or X or Y or Z)

begin

case({W, X, Y, Z})

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;

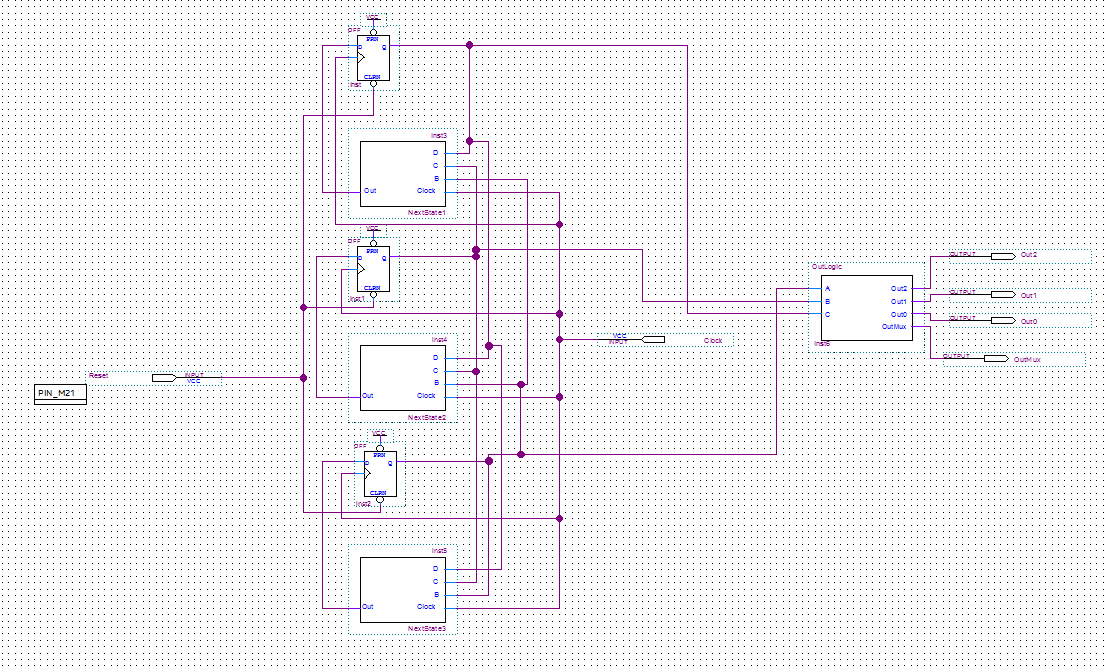
endcase

end

endmodule

This module takes a 4-bit input, and outputs 7 separate bits. These 7 bits tells what segments should be on in the hex display. The code is basically just a large truth table.

State Machine:



The state machine consists of 3 DFF’s, next state logic for each, and output logic to control the circuit. The state machine has one input, which is the clock, and 4 outputs which are for reading to registers and enabling the decoder, as well as selecting a multiplexer for one of the register inputs.

Next State 1:

module NextState1(Clock, B, C, D, Out);

input Clock, B, C, D;

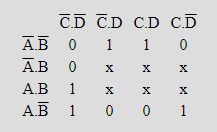
output Out;

assign Out = ((~Clock&D)|(Clock&~D&~C));

endmodule

The code for NextState1 was created from the karnaugh map of the state table. The map and sop expression are below. This is what helps the last bit of data progress from state to state. This module take all three DFF outputs and the clock as inputs, and outputs the DFF’s new input.

U:\CprE281\Final Project\Pictures\Output Next STate For C.PNG



Next State 2:

module NextState2(Clock, B, C, D, Out);

input Clock, B, C, D;

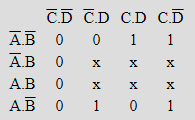
output Out;

assign Out = ((~Clock&C)|(C&~D)|(Clock&~C&D));

endmodule

The code for NextState2 was created from the karnaugh map of the state table. The map and sop expression are below. This is what helps the middle bit of data progress from state to state. This module take all three DFF outputs and the clock as inputs, and outputs the DFF’s new input.

U:\CprE281\Final Project\Pictures\Output Next State For B.PNG



Next State 3:

module NextState3(Clock, B, C, D, Out);

input Clock, B, C, D;

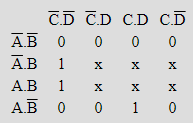
output Out;

assign Out = (B|(Clock&C&D));

endmodule

The code for NextState2 was created from the karnaugh map of the state table. The map and sop expression are below. This is what helps the first bit of data progress from state to state. This module take all three DFF outputs and the clock as inputs, and outputs the DFF’s new input.

U:\CprE281\Final Project\Pictures\Output Next State For A.PNG



Output Logic:

module OutLogic(A, B, C, Out2, Out1, Out0, OutMux);

input A, B, C;

output Out2, Out1, Out0, OutMux;

assign Out2 = (C|(~A&~B));

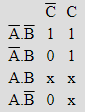
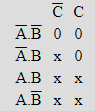
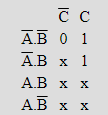
assign Out1 = 0;

assign Out0 = (~B&C);

assign OutMux = (~C&B&A);

endmodule

The output logic was created using karnaugh maps as well. Each assign statement uses the sop from the specific karnaugh map. The output logic takes all three DFF values as inputs. The outputs are the enable for the decoder, the 2 bit input of the decoder and the selector value of the multiplexer to choose which set of data goes into one of the registers.

U:\CprE281\Final Project\Pictures\Output 2 Logic.PNGU:\CprE281\Final Project\Pictures\Output 1 logic.PNGU:\CprE281\Final Project\Pictures\Output 0 logic.PNG

Test Cases:

In order to test my circuit, there are 7 steps.

Step 1: From switches 7-0, choose the first number to multiply.

Step 2: set switches 17-16 to 00.

Step 3: Pulse the clock with pushbutton[3]

Step 4: From switches 15-8, choose the second number to multiply.

Step 5: Set switches 17-16 to 01

Step 6: Pulse the clock again with pushbutton[3]

Step 7: Pulse the multiplier clock with pushbutton[2]

Step 8: Load the bottom half of the output into the register file with switches 17-16 set to 10

Step 9: Pulse the clock with pushbutton[3]

Step 10: Load the top half of the output into the register file with switches 17-16 set to 11

Step 11: Pulse the clock with pushbutton[3]

Step 12: Done! The bottom half of the answer will be displayed on the hex display. To view the top half, hold pushbutton[0]

Examples tests are as follows:

00001000 x 00000100 (8\*4)= 32 in decimal = 0020 in hex.

11111101 x 00010100 (-3\*20) = -60 in decimal = -FFC4 in hex