Ross Thedens

Section Q

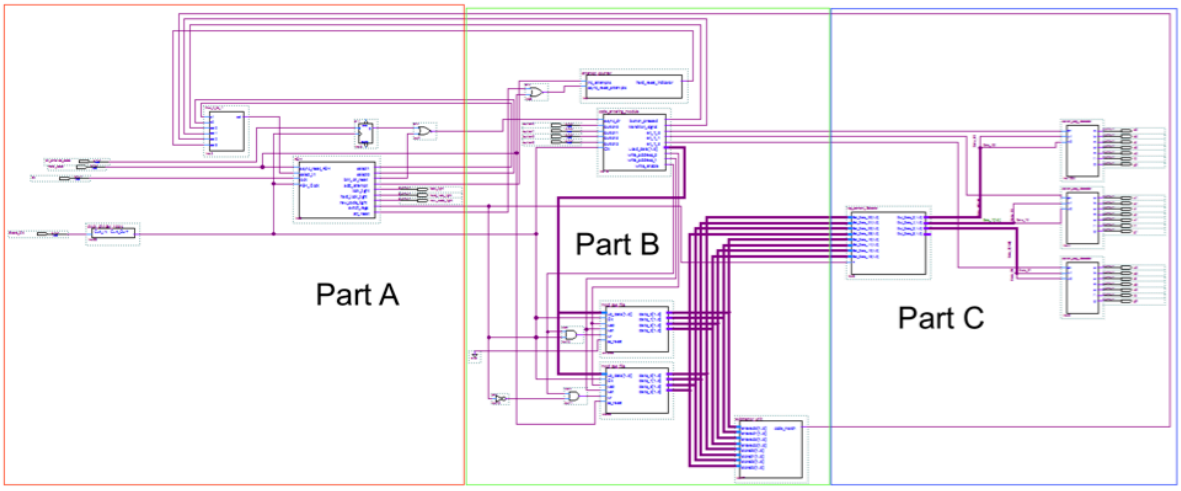
CPRE 281

Student ID: [redacted]

Final Project Report

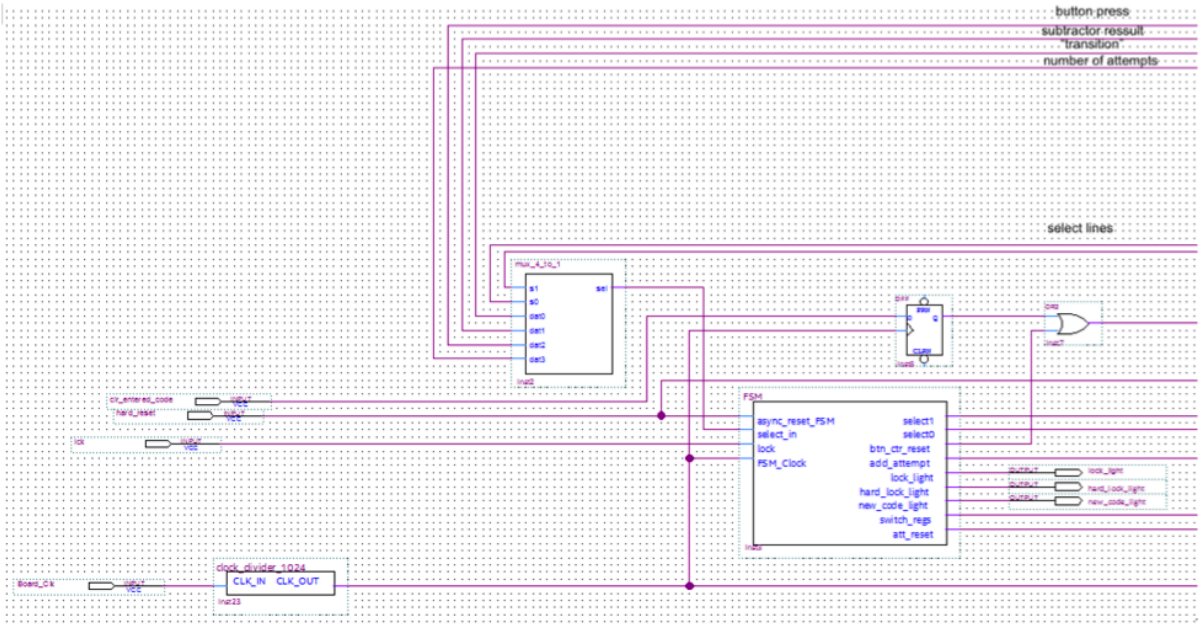
I completed project option number 3 (simple door lock). This report details all the workings and derivations required to build the circuit and Verilog modules contained in the project. It contains a general overview of every top-level module, followed by a multilevel description of the inner workings of each module that explains every submodule involved.

Top Level Diagram



This is the overall top-level diagram. Since it is difficult to view at this size, I have split it into three different sections: Part A, Part B, and Part C. These are explained below.

Part A



In this part, the inputs are the lock signal *lck*, the clear switch signal *clr_entered_code*, the reset input signal *hard_reset*, and the clock signal *Board_Clk*. The clock frequency is divided by 1024 via the **clock_divider_1024** module to give more stability to the module related to button inputs (not shown); the clock signal services all the synchronized blocks in the project. The **mux_4_to_1** block is a 4 to 1 multiplexer; it determines the source of the input signal *select_in* for the **FSM** block. Note that the **FSM** select line outputs wrap around directly to the mux outside of the screenshot. The **FSM** block itself encompasses the entirety of the state machine in the project. It contains two inputs *select_in* and *lock*, a reset input *async_reset_FSM*, and a clock signal input *FSM_clock*. The outputs are *select1, select0, btn_ctr_reset, add_attempt, lock_light, hard_lock_light, new_code_light,* and *switch_regs*. *Select1* and *select0* are the select

lines to the **mux_4_to_1** block; this helps to overcome the problem of requiring inputs from many different sources (comparator circuit, button presses, number of attempts) when at different states; the select lines will pipe in the correct input corresponding with each state (explained in detail later). The *btn_ctr_reset* input is used to reset the counter within the button input module (not shown); it is either triggered by the clear switch or the state machine's state (explained later). The *add_attempt* output is used to add an incorrect attempt to the module that counts incorrect attempts (not shown) when an incorrect code is entered to unlock the lock. *Lock_light, hard_lock_light,* and *new_code_light* work as follows: *lock_light* is mapped to LEDG0–it is lit only when the lock is unlocked, *hard_lock_light* indicates that the maximum number of incorrect attempts have been entered and that the lock must be hard reset using the hard reset switch. *Switch_regs* determines which register (stored code or entry) is written to at a particular point in time. Finally, *att_reset* resets the attempt counter, which asserts when the lock is unlocked. The *clr_entered_code* input is connected to a D-Flip-Flop that synchronizes its input with the rest of the circuit to ensure predictable behavior; it is run through an OR gate with the *btn_ctr_reset* output from the **FSM** block; this allows either input to transmit a clearing signal.

mux_4_to_1

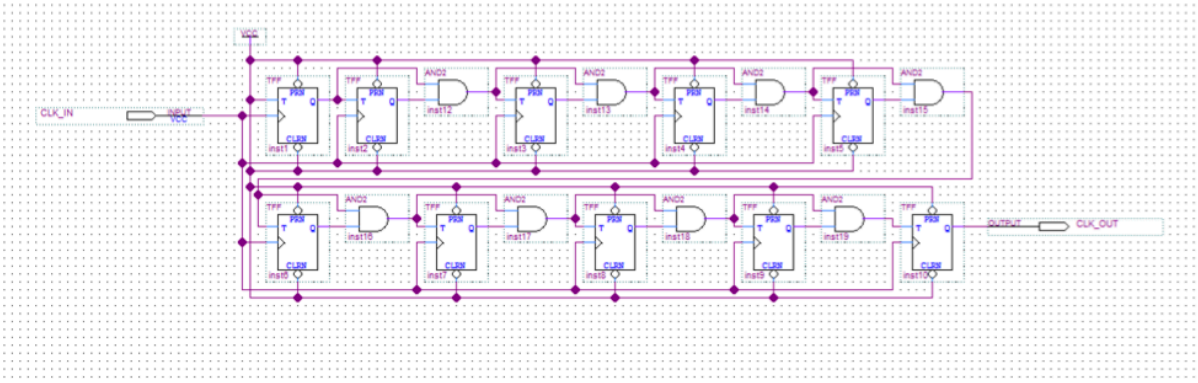
mux_4_to_1 is a 4-1 multiplexer. It is simply designed to output the value of *dat0* when both select lines are 0, *dat1* when *s1* is zero and *s0* is one, *dat2* when *s1* is 1 and *s0* is 0, and *dat3*

when both select lines are one. This module does not need further explanation.

```
module mux_4_to_1(s1, s0, sel, dat0, dat1, dat2, dat3);

    input s1, s0, dat0, dat1, dat2, dat3;
    output sel;

    assign sel = ~s1 & ~s0 & dat0 | ~s1 & s0 & dat1 | s1 & ~s0 & dat2 | s1 & s0 & dat3;

endmodule
```

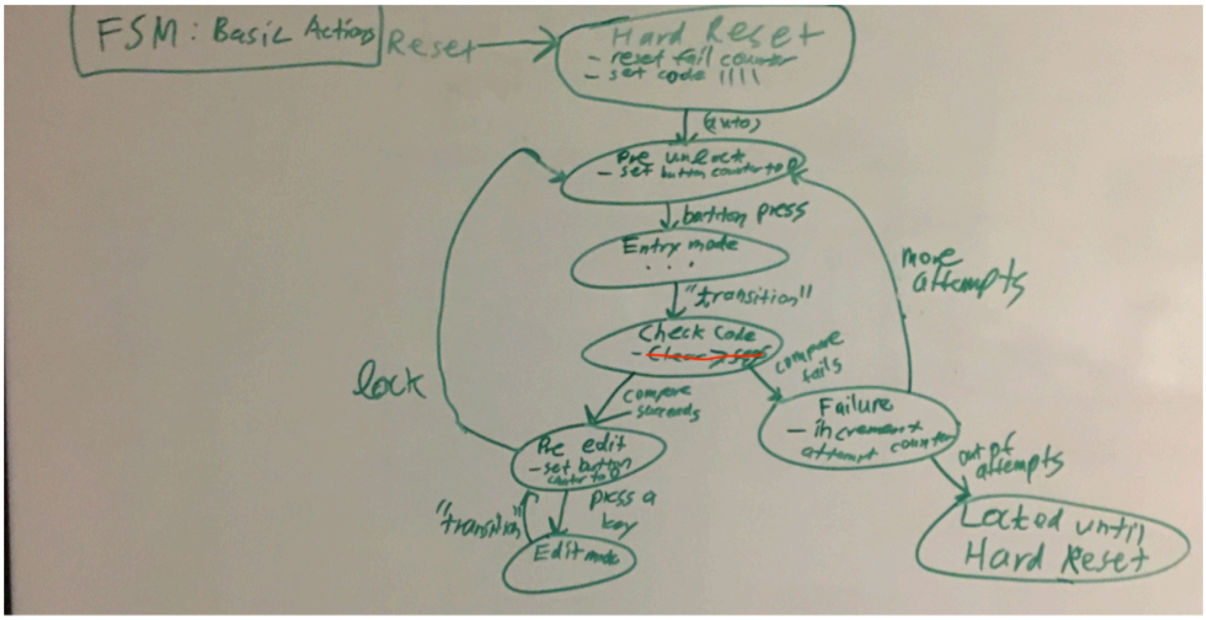
clock_divider_1024

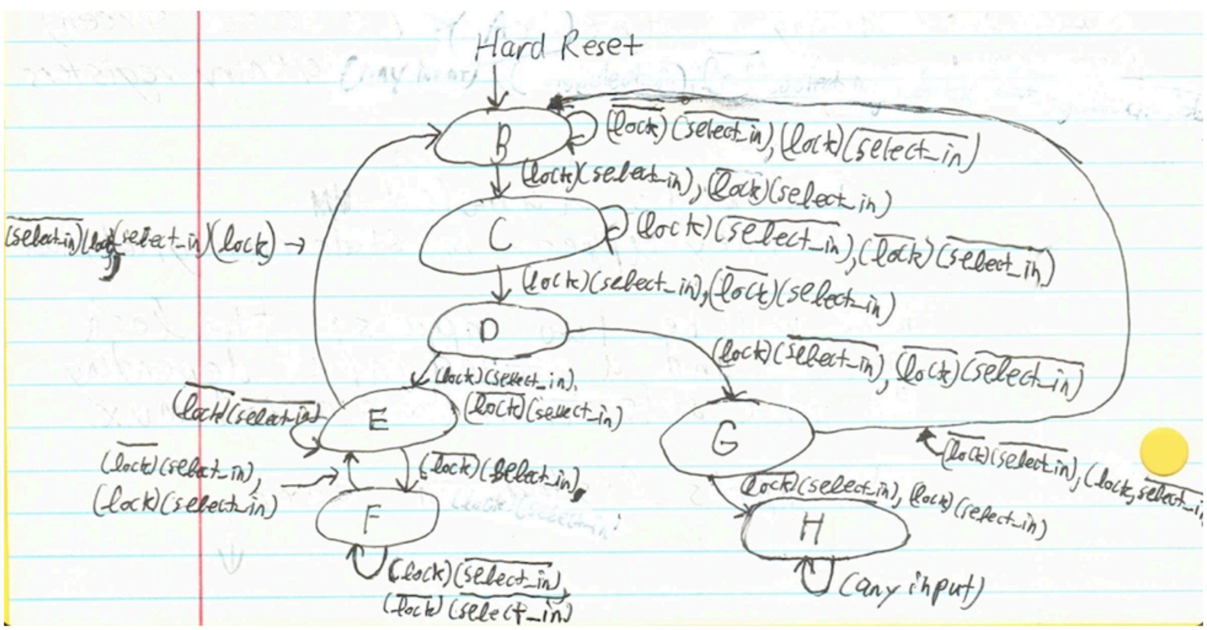
clock_divider_1024 is a clock dividing module. It is a series of T-Filp-Flops that each divide the frequency by 2; it essentially works as a ten-bit counter where the tenth bit changing between 1 and 0 is the new clock signal. The first flip flop output changes at the rate of the clock, while the second changes at half the rate of the clock; the AND gates make sure that the next flip flop does not change until the outputs of the previous two are equal to 1; this means the flip flop is changing at half the speed of the preceding one and a fourth the speed of the one before that. Since there are ten T-Flip-Flops, the frequency is divided by 1024. As stated previously, this makes the button inputs more reliable.



FSM

The bulk of Part A is manifested in the **FSM** block. This represents the state machine for the project, composed of the next state logic, flip flops, and output logic. First, we can focus on the state diagram:

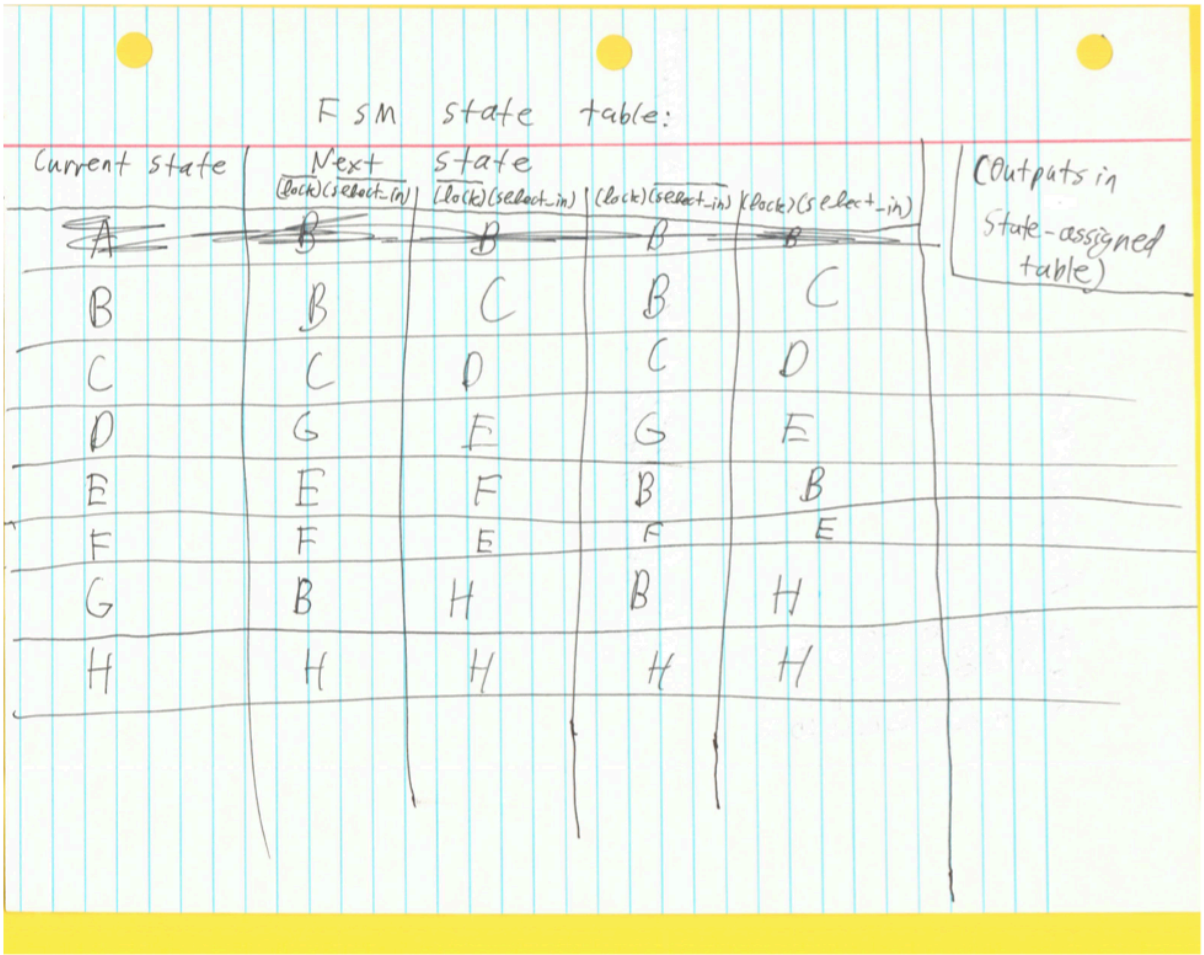




Note that there is no state A; this state was removed in the development of the machine. The whiteboard photo provides some context for the FSM diagram below it. Hard Reset is the primary reset for the machine; besides setting the machine to state B, it also sets the lock's code to 1111 and resets the attempt counter according to the specification. State B is where the machine resides before the first button is pressed. At this point, the button counter (part of the code entry module) is reset to zero, the unlock and new code entry lights are turned off, the hard reset lock is off, and the register to write to is the entry register (for unlock attempts, this corresponds to an output of 1). The select lines are *s1s0* = 00, which selects the button press input; the machine advances if a button is pressed. The next state, C, is the state while the code is entered; it is the same as before but the button counter reset output is no longer asserted and the input via the multiplexer (*s1s0* = 01) is a signal from the code entry module referred to as "transition;" at this point it is important to note that the transition signal asserts when the button press counter detects that four buttons have been pressed; this signal will move the machine to state D. D is the state where the lock checks the entered code against the actual unlock code; the main output change is the select lines switching to *s1s0* = 10 for the subtractor circuit that compares the codes. Here, the possibilities split into two. In state E, the correct code has been entered (the subtractor/comparator circuit output is 1). At E, the unlock light turns on. The select input is switched back to the button press detector, the register to write to is changed to 0 for the code register and the button counter is reset in preparation for the code to be changed. If a button is pressed, the state transitions to F; if the lock switch is flipped, the state switches back to B. State F is for setting a new code for the lock; LEDG1 is turned on to indicate this. The select lines are changed to select the "transition" output ones again. Whaen

the “transition” signal is received, the lock transitions back to E, where it can be locked or a new code can be entered again. If the correct code is received at state D, the state transitions to G, where the add attempt output is set to one to increment the attempt counter. If the maximum number of incorrect attempts is reached (five), the state transitions to H; otherwise, it goes back to B. H is the locked until hard reset state; LEDR0 is turned on to indicate this state, and there is no way to remove the machine from this state except by hard resetting it.

Next are the state and state assigned tables. Note that the outputs are only listed in the state assigned table, and that don’t cares are used where a particular output is not relevant.



State Assignments:

$Y_2 Y_1 Y_0$

A = 000
B = 001
C = 010
D = 011
E = 100
F = 101
G = 110
H = 111

State Assignment table (with outputs) on next page.

FSM state Assignment Table

Current State $Y_2 Y_1 Y_0$	Next State ($\overline{lock}$)($\overline{select\_in}$) $Y_2 Y_1 Y_0$	($\overline{lock}$)(select_in) $Y_2 Y_1 Y_0$	(lock)($\overline{select\_in}$) $Y_2 Y_1 Y_0$	(lock)(select_in) $Y_2 Y_1 Y_0$	Outputs (select lines) S_1	S_0	att_reset	(1: to btn_ctr
~~0 0 0 (A)~~	~~0 0 1~~	~~0 0 1~~	~~0 0 1~~	~~0 0 1~~	d	d	1	0
0 0 1 (B)	0 0 1	0 1 0	0 0 1	0 1 0	0	0	0	1
0 1 0 (C)	0 1 0	0 1 1	0 1 0	0 1 1	0	1	0	0
0 1 1 (D)	1 1 0	1 0 0	1 1 0	1 0 0	1	0	0	d
1 0 0 (E)	1 0 0	1 0 1	0 0 1	0 0 1	0	0	0	1
1 0 1 (F)	1 0 1	1 0 0	1 0 1	1 0 0	0	1	0	0
1 1 0 (G)	0 0 1	1 1 1	0 0 1	1 1 1	1	1	0	d
1 1 1 (H)	1 1 1	1 1 1	1 1 1	1 1 1	d	d	0	1

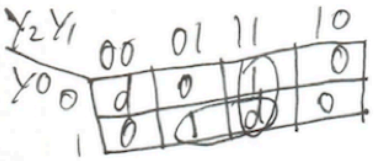
→ All top row entries are d (don't care) (1: reset)

...gle reset	(1: add, toggle) add_attempt	(1: on) lock_light	(1: on) hard_lock_light	(1: on) new_code_light	switch_regs	att_reset (1: entry, 0: lock code)
	0	0	0	0	d	d
	0	0	0	0	1	0
	0	0	0	0	1	0
	0	0	0	0	d	0
	0	1	0	0	0	1
	0	1	0	1	0	0
	1	0	0	0	d	0
	0	0	1	0	d	0

Note: the split column between the two scans is for btn_ctr_reset; it says that 1 toggles the reset.

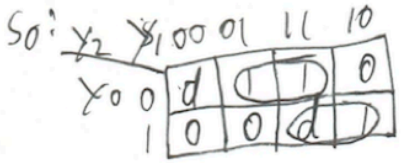
At this point, we can derive the expressions used for the outputs and next state signals.

S_1:



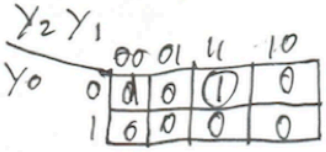
$S_1 = Y_1 Y_0 + Y_2 Y_1$

S_0:



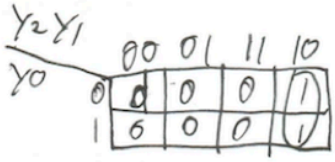
$S_0 = Y_1 \overline{Y_0} + Y_2 Y_0$

add_attempt:



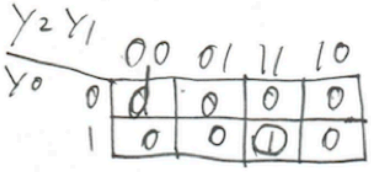
add_attempt = $Y_2 Y_1 \overline{Y_0}$

lock_light:



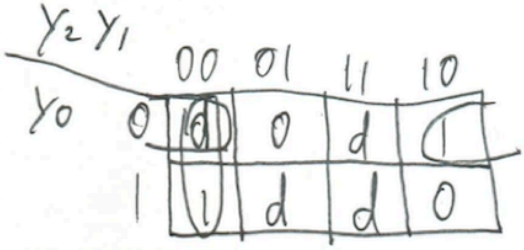
lock_light = $Y_2 \overline{Y_1}$

hard_lock_light



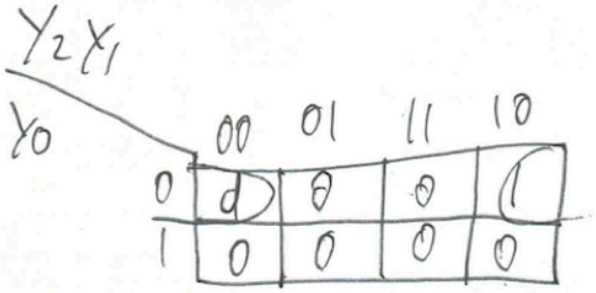
hard_lock_light = $Y_2 Y_1 Y_0$

btn_ctr_reset:



$$btn\_ctr\_reset = \overline{Y_2}\,\overline{Y_1} + \overline{Y_1}\,\overline{Y_0}$$

att_reset:



$$att\_reset = \overline{Y_1}\,\overline{Y_0}$$

Y_1:

$\overline{lock}$

select_in Y_2 / Y_1 Y_0	00	01	11	10
00	0	0	0	d
01	0	0	0	1
11	1	1	1	0
10	1	0	1	1

lock

select_in Y_2 / Y_1 Y_0	00	01	11	10
00	0	0	0	d
01	0	0	0	1
11	1	1	1	0
10	1	0	1	1

$Y_1 = select\_in\ \overline{Y_2}\ \overline{Y_1} + \overline{select\_in}\ \overline{Y_2}\ Y_1 + Y_2 Y_1 Y_0 + select\_in\ Y_1\ \overline{Y_0}$

Y_0:

$\overline{lock}$

select_in Y_2 / Y_1 Y_0	00	01	11	10
00	d	0	1	d
01	1	1	0	0
11	0	1	1	0
10	0	1	1	1

lock

select_in Y_2 / Y_1 Y_0	00	01	11	10
00	d	1	1	d
01	1	1	0	0
11	0	1	1	0
10	0	1	1	1

$Y_0 = Y_2 Y_1 + select\_in\ \overline{Y_0} + lock\ \overline{select\_in}\ \overline{Y_1} + \overline{select\_in}\ \overline{Y_1}\ Y_0$

new_code_light:

Y_2Y_1 / Y_0	00	01	11	10
0	d	0	0	0
1	0	0	0	1

$new\_code\_light = Y_2\overline{Y_1}Y_0$

switch_regs:

Y_2Y_1 / Y_0	00	01	11	10
0	d	1	d	0
1	1	d	d	0

$switch\_regs = \overline{Y_2} + Y_1$

Next state expressions:

Y_2:

$\overline{lock}$

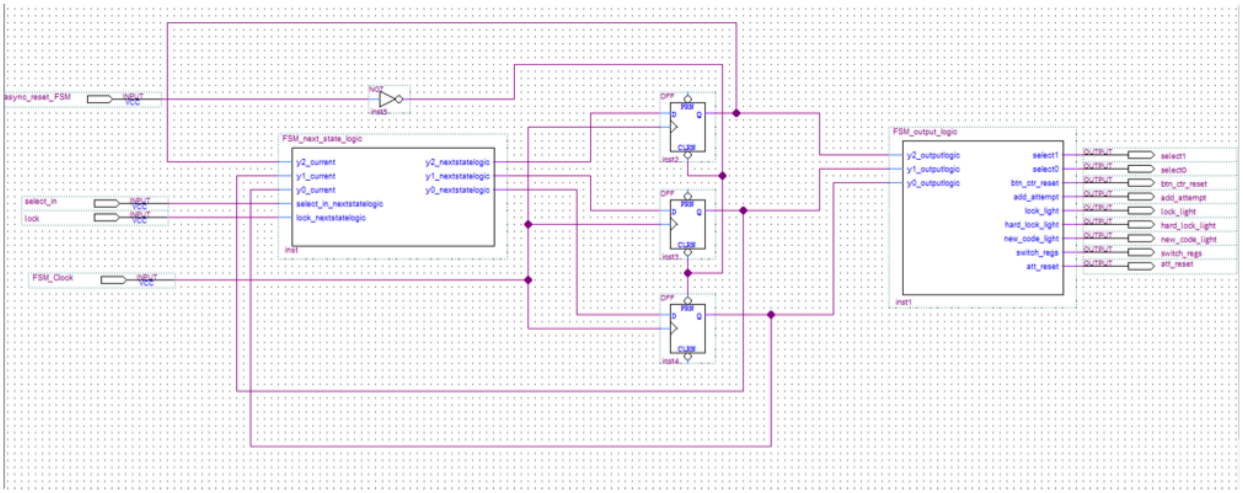
select_in Y_2 / Y_1Y_0	00	01	11	10
00	d	1	1	d
01	0	1	1	0
11	1	1	1	1
10	0	0	1	0

lock

select_in Y_2 / Y_1Y_0	00	01	11	10
00	d	0	0	d
01	0	1	1	0
11	1	1	1	1
10	0	0	1	0

$Y_2 = Y_2Y_0 + Y_1Y_0 + select\_in\ Y_2Y_1 + \overline{lock}\,\overline{Y_1}\,\overline{Y_0}$

Here is the **FSM** block as it appears in Quartus Prime:



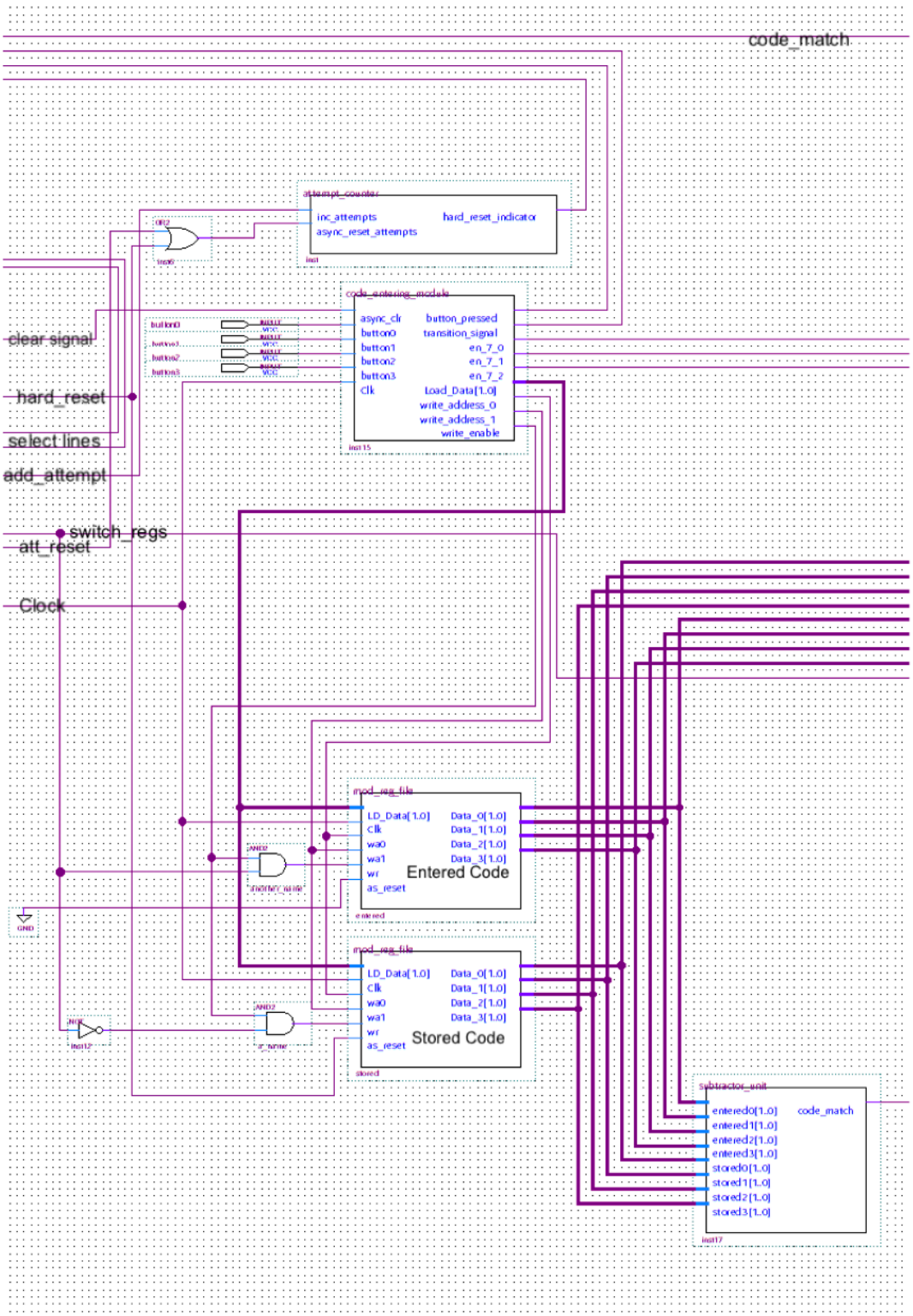
Here is the **FSM_next_state_logic** Verilog file:

```
module FSM_next_state_logic(y2_nextstatelogic, y1_nextstatelogic, y0_nextstatelogic, y2_current, y1_current, y0_current, select_in_nextstatelogic, lock_nextstatelogic);

   input y2_current, y1_current, y0_current, select_in_nextstatelogic, lock_nextstatelogic;

   output y2_nextstatelogic, y1_nextstatelogic, y0_nextstatelogic;

   assign y2_nextstatelogic = y2_current & y0_current | y1_current & y0_current | select_in_nextstatelogic & y2_current & y1_current | ~lock_nextstatelogic & ~y1_current & ~y0_current;
   assign y1_nextstatelogic = select_in_nextstatelogic & ~y2_current & ~y1_current | ~select_in_nextstatelogic & ~y2_current & y1_current | y2_current & y1_current & y0_current | select_in_nextstatelogic & y1_current & ~y0_current;
   assign y0_nextstatelogic = y2_current & y1_current | select_in_nextstatelogic & ~y0_current | lock_nextstatelogic & ~select_in_nextstatelogic & ~y1_current | ~select_in_nextstatelogic & ~y1_current & y0_current;

endmodule
```

Here is the **FSM_output_logic** Verilog file:

```
module FSM_output_logic(y2_outputlogic, y1_outputlogic, y0_outputlogic, select1, select0, ↵
btn_ctr_reset, add_attempt, lock_light, hard_lock_light, new_code_light, switch_regs, att_reset);

   input y2_outputlogic, y1_outputlogic, y0_outputlogic;

   output select1, select0, btn_ctr_reset, add_attempt, lock_light, hard_lock_light, new_code_light↵
, switch_regs, att_reset;

   assign select1 = y1_outputlogic & y0_outputlogic | y2_outputlogic & y1_outputlogic;
   assign select0 = y1_outputlogic & ~y0_outputlogic | y2_outputlogic & y0_outputlogic;
   assign btn_ctr_reset = ~y2_outputlogic & ~y1_outputlogic | ~y1_outputlogic & ~y0_outputlogic;
   assign add_attempt = y2_outputlogic & y1_outputlogic & ~y0_outputlogic;
   assign lock_light = y2_outputlogic & ~y1_outputlogic;
   assign hard_lock_light = y2_outputlogic & y1_outputlogic & y0_outputlogic;
   assign new_code_light = y2_outputlogic & ~y1_outputlogic & y0_outputlogic;
   assign switch_regs = ~y2_outputlogic | y1_outputlogic;
   assign att_reset = ~y1_outputlogic & ~y0_outputlogic;

endmodule
```

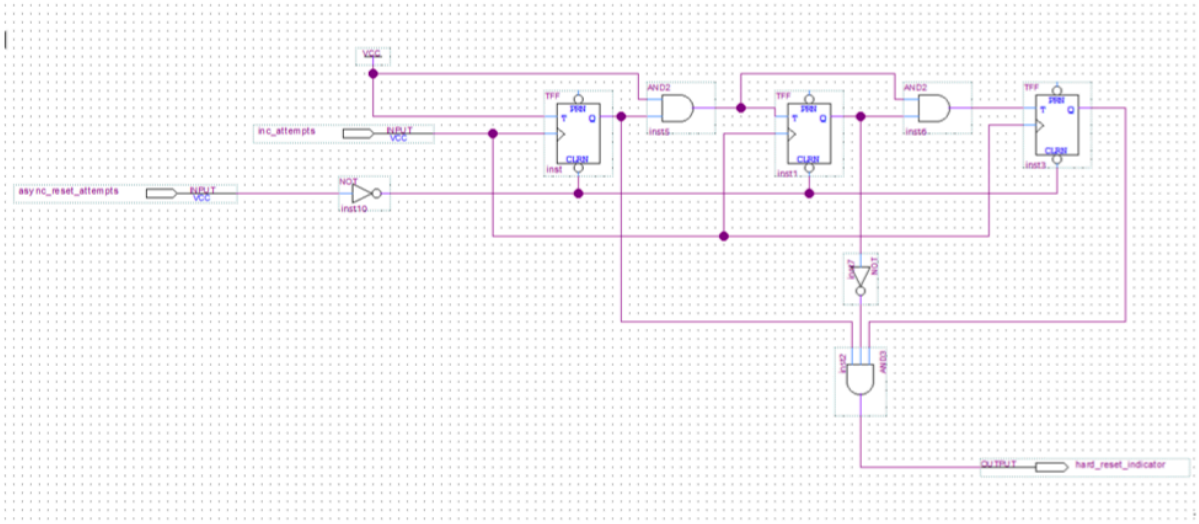
Part B


code_match
attempt_counter
inc_attempts
async_reset_attempts
hard_reset_indicator
inst
code_entering_module
async_clr
button0
button1
button2
button3
Clk
button_pressed
transition_signal
en_7_0
en_7_1
en_7_2
Load_Data[1.0]
write_address_0
write_address_1
write_enable
inst15
clear signal
hard_reset
select lines
add_attempt
switch_regs
att_reset
Clock
mod_reg_file
LD_Data[1.0]
clk
wa0
wa1
wr
as_reset
Data_0[1.0]
Data_1[1.0]
Data_2[1.0]
Data_3[1.0]
Entered Code
entered
Stored Code
stored
GND
subtractor_unit
entered0[1.0]
entered1[1.0]
entered2[1.0]
entered3[1.0]
stored0[1.0]
stored1[1.0]
stored2[1.0]
stored3[1.0]
inst17

Part B primarily involves the register files, the attempt counter, the button module, and the subtractor. Essentially, the attempt counter takes signals from the **FSM** module and the hard reset switch. It triggers the hard reset when a fifth incorrect attempt is entered. It is reset when a hard reset is performed or the lock is unlocked successfully, hence the OR gate. The **code_entering_module** takes in the four buttons inputs, outputting *button_pressed* when a button is initially pressed (for the duration of one clock cycle), *transition_signal* when the counter reaches four (all code numbers have been input), and *en7_0, en7_1,* and *en7_2* immediately following the entry of each consecutive digit to enable seven segment displays for each digit (note that since the code is cleared immediately following the entry of the fourth digit, I have omitted a fourth seven segment display). *Load_Data[1..0]* is a 2-bit bus outputting the value of the button pressed, *write_address_0* and *write_address_1* output the address to write to, and *write_enable* enables the writing capability on the register selected by *switch_regs*. Note that the AND and NOT gate associated with the signal on the **mod_reg_file** modules mean that the *write_enable* signal from the **code_entering_module** enables the Entered Code register file when the *switch_regs* output is 1 and the Stored Code register file when the *switch_regs* output is 0. Also note the Stored Code register file has an asynchronous reset input while the one on Entered Code is grounded; this is triggered when a hard reset is initiated, setting the stored code to 1111. Entered Code is not cleared, as each time a new code is entered it is completely overwritten.

attempt_counter

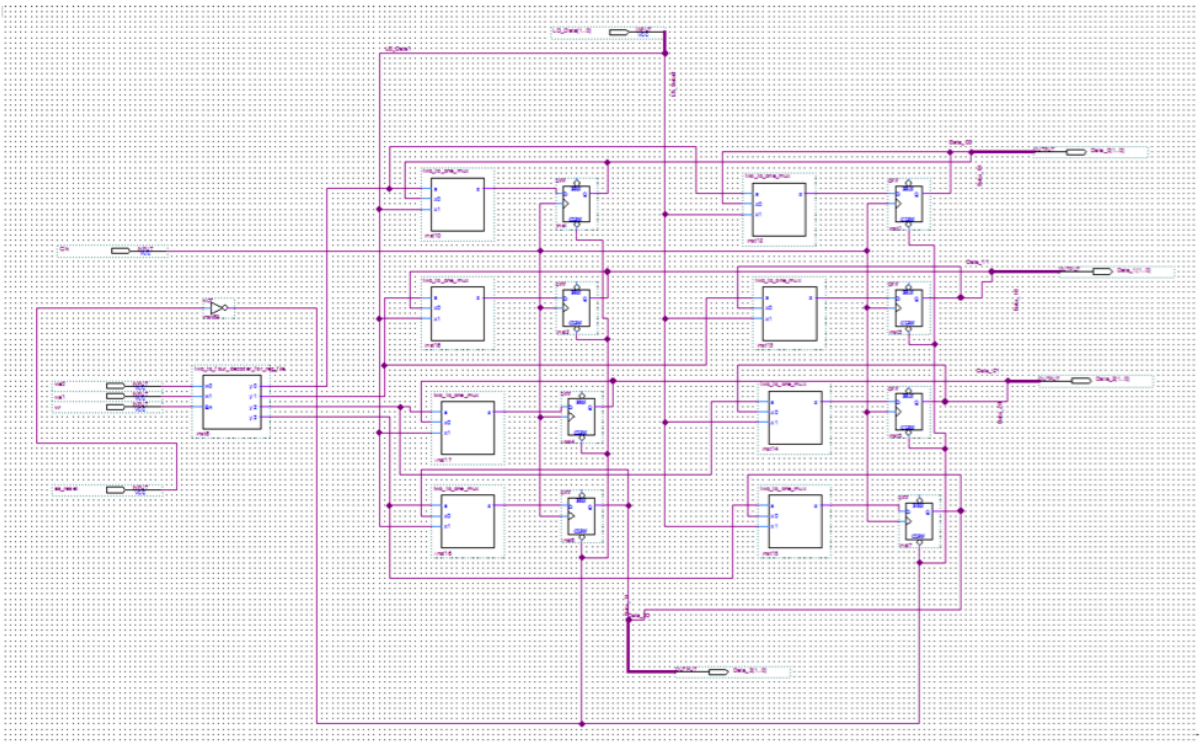
This is a basic three-bit synchronous up counter with an asynchronous reset, constructed in the same way as in the lab/textbook. The reset input is inverted such that an input of 1 resets the counter. The *hard_reset_indicator* output asserts when the counter reaches 5 (101); this sends a signal to the FSM to enter the locked until hard reset state.



mod_reg_file

This register file works exactly as the examples discussed in lecture and in the textbook, except that it has bus outputs for each 2-bit register in the file and it contains an asynchronous clear that sets the value of every flip flop to 0. When the lock is hard reset, this corresponds to a code of 1111 due to the mapping of the 2-bit values to the values shown on the 7-segment displays. The reset signal is inverted so that an input of 1 resets the register file. The 2-4 decoder and 2-1 multiplexer are described below. The 2-4 decoder determines which register in the file is enabled for writing based on the address and whether writing is enabled. The output

is sent to the 2-1 multiplexers associated with that register; this switches the register from holding the current value to accepting the input from the *LD_Data* bus. The *Data_0[1..0]* through *Data_3[1..0]* buses output the 2-bit values stored in each of the four registers.



two_to_four_decoder_for_reg

This is a basic 2-4 decoder with enable. It maps the inputs (*w0w1*) 00 01 10 11 to the outputs *y0 y1 y2 y3* respectively, outputting nothing if the decoder is not enabled. No further explanation should be necessary.

```
module two_to_four_decoder_for_reg_file(w0, w1, En, y0, y1, y2, y3);
   input w0, w1, En;
   output y0, y1, y2, y3;
   assign y0 = ~w1&~w0&En;
   assign y1 = ~w1&w0&En;
   assign y2 = w1&~w0&En;
   assign y3 = w1&w0&En;
endmodule
```

two_to_one_mux

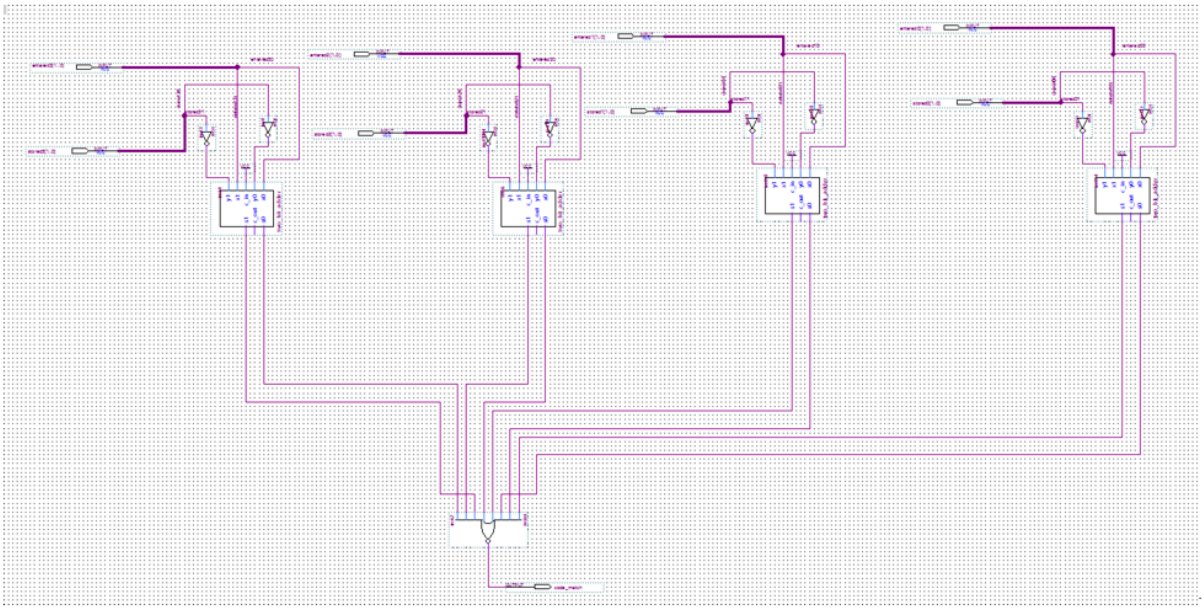
This is a basic 2-1 multiplexer. It outputs x0 when the select line is zero and x1 when the select line is one. The expressions should be obvious from this stated purpose. No further explanation should be necessary.

```
module two_to_one_mux(s, x0, x1, z);
   input s, x0, x1;
   output z;
   assign z = ~s&x0 | s&x1;
endmodule
```

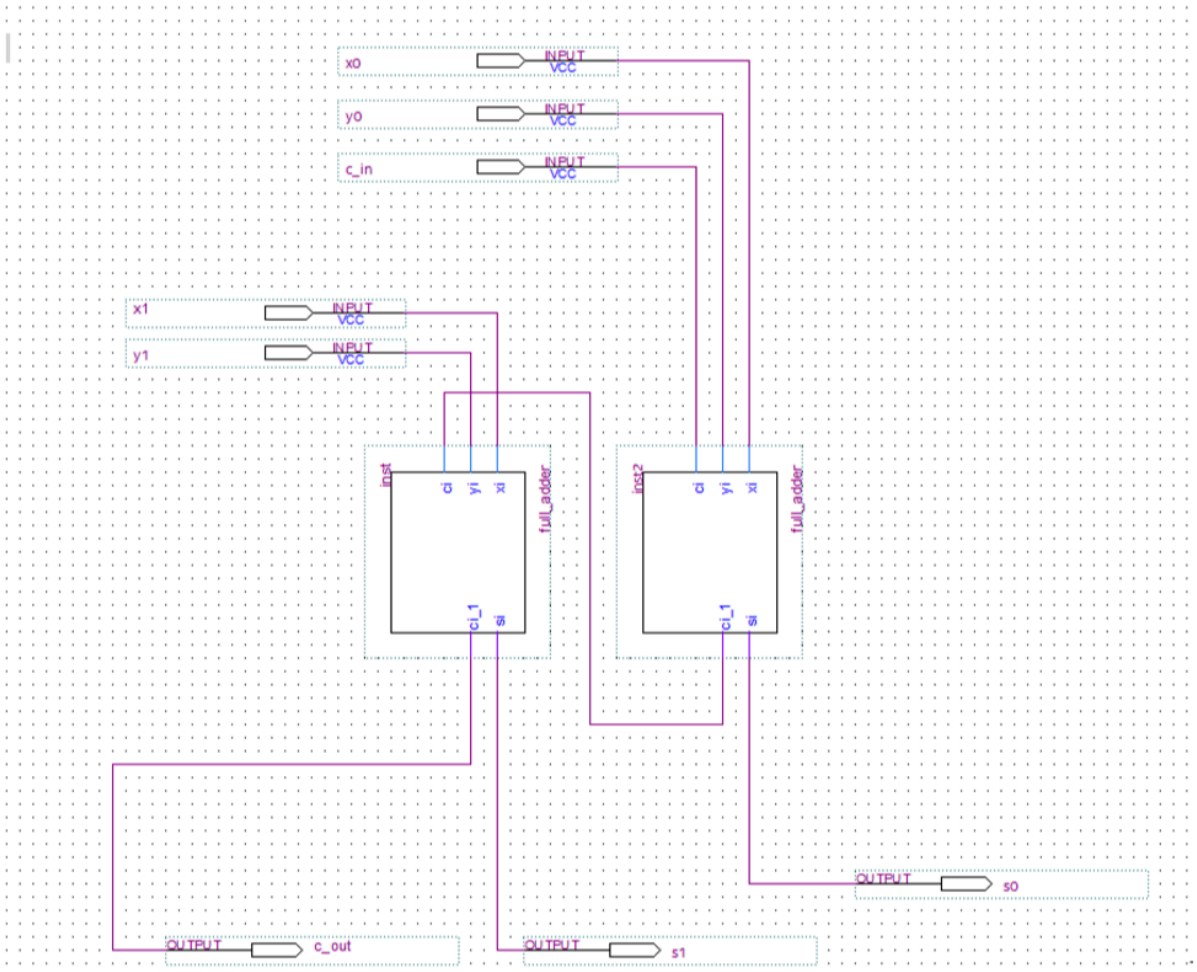
subtractor_unit

This module takes an input of two four digit numbers (the stored code and entered code) in the form of four 2-bit buses (values of 1, 2, 3, 4). It outputs a 1 if the values are exactly the same, and a 0 otherwise. The module consists of four instances of the **two_bit_adder** module, which adds two 2-bit values together. These values are the corresponding digits of the entered code and the stored code. In this case, they are subtracted in two's complement fashion. To

accomplish this, one of the inputs (the y1y0 input) is negated, and a carry-in of 1 is used, making it negative. The carry out bit is ignored, as is done in two's complement addition/subtraction. If all the digits are the same, each subtractor will output 00; the NOR gate outputs a 1 if all the inputs are 0.



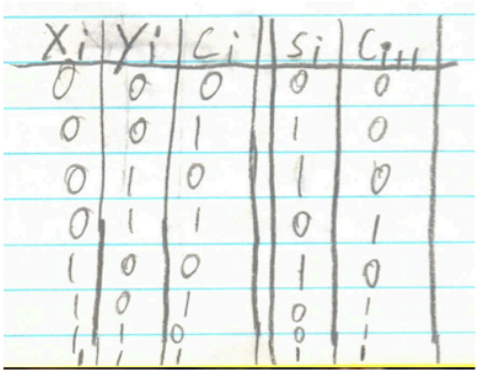
two_bit_adder

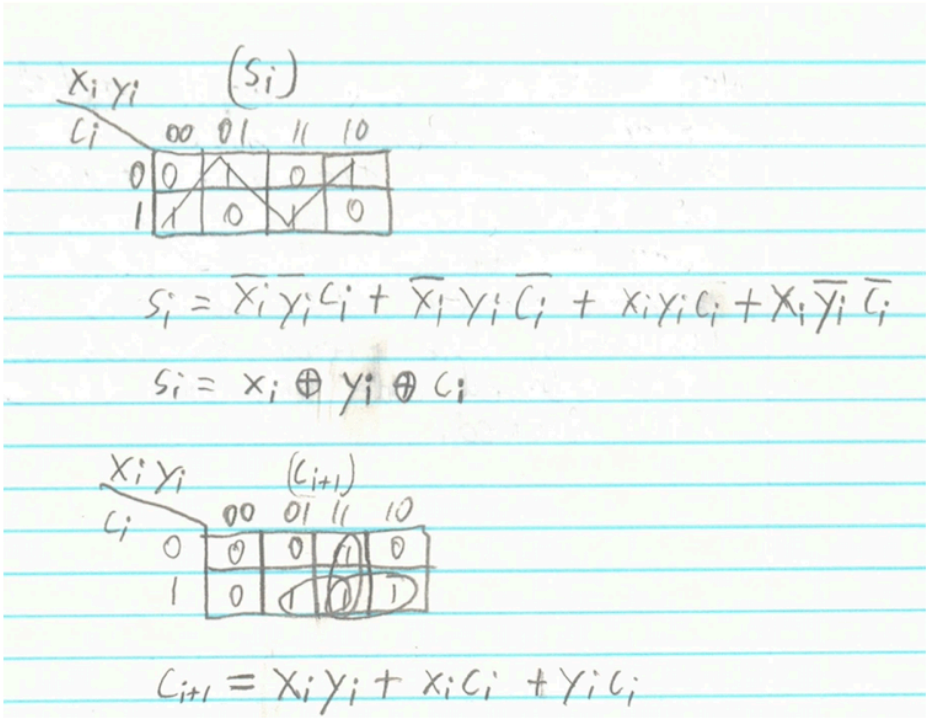


This module uses two full adders to create a ripple carry two-bit adder as seen in the textbook/lecture. First, x0 and y0 are added (along with the carry in), s0 is output, and the carry out is carried in to the next adder, which adds x1 and y1. This outputs s1 and the carry out signal.

full_adder

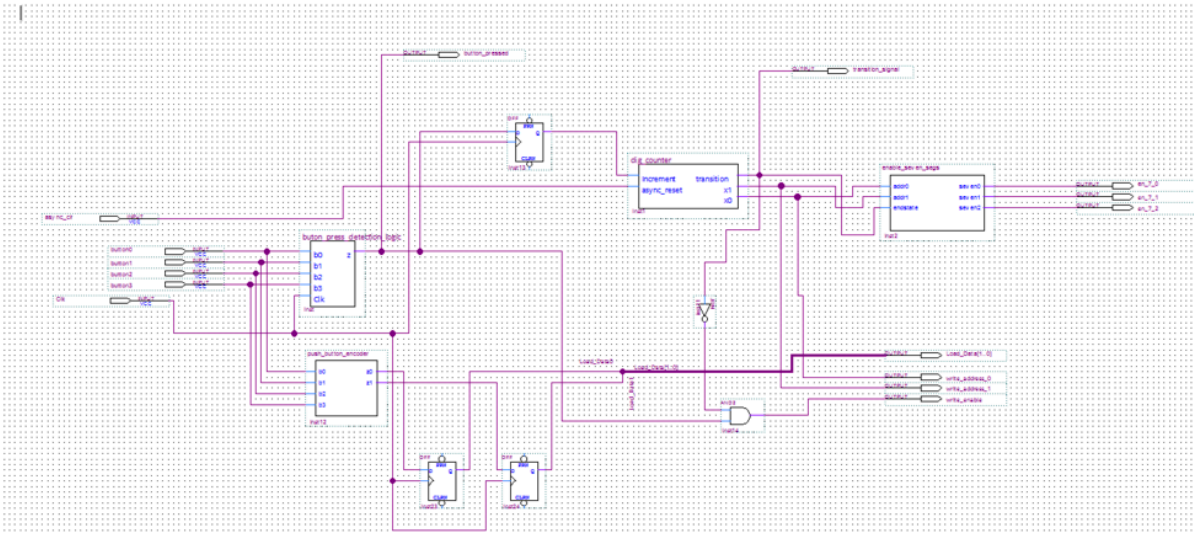
This is a basic implementation of a full adder. The derivation is shown below. Note that xi yi are the addends, and ci and ci+1 represent the carry in and carry out, respectively.





```
module full_adder(xi, yi, ci, si, ci_1);

    input xi, yi, ci;
    output si, ci_1;

    assign si = xi ^ yi ^ ci;
    assign ci_1 = xi&yi | xi&ci | yi&ci;

endmodule
```

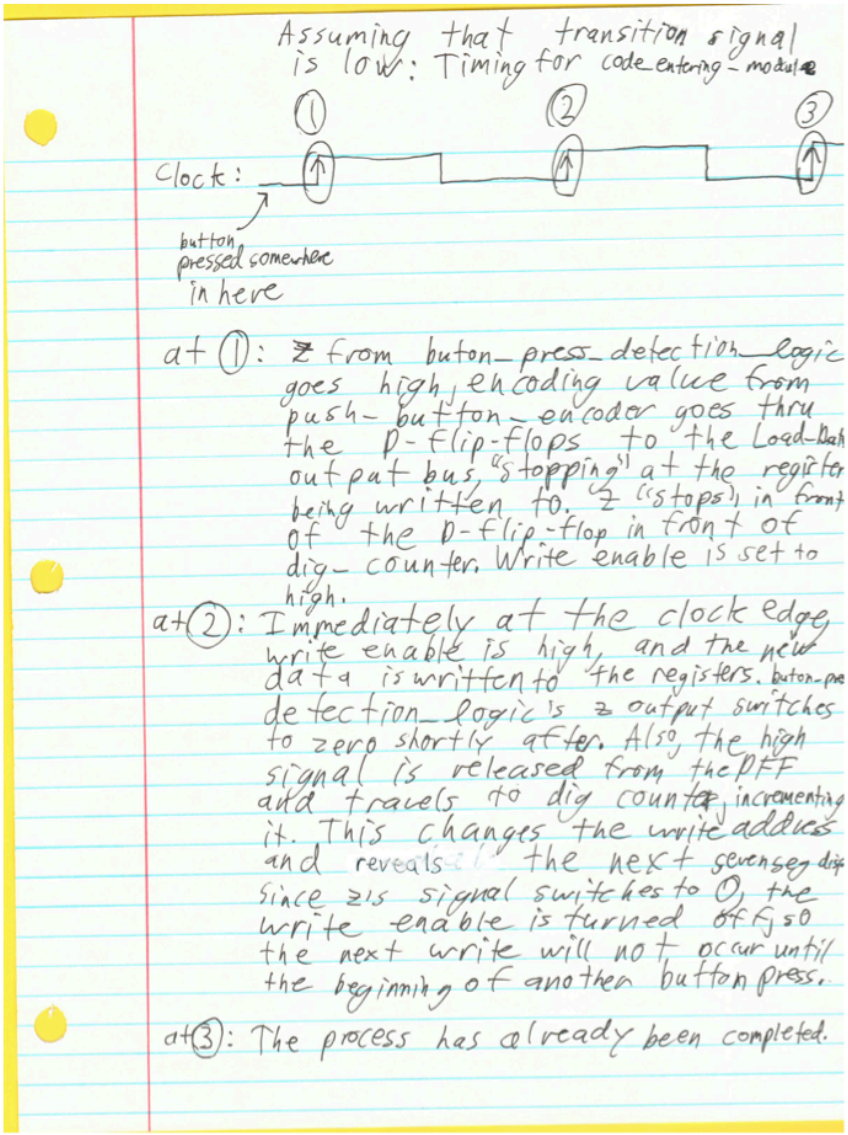
code_entering_module



This is the module that reads and synchronizes all button inputs with the rest of the project. The (intentionally) misspelled **buton_press_detection_logic** block outputs a 1 if a button is pressed and no button was pressed on the previous clock cycle. In parallel is the **push_button_encoder**, which encodes the signals (described under that block's main heading). The **dig_counter** block is incremented by the **buton_press_detection_logic** output *z*. The *async_reset* input to **dig_counter** resets the counter. The *x1* and *x0* outputs from **dig_counter** serve as the write addresses for the next button entry and also serve as inputs to **enable_seven_segs**, the block that determines which seven segment displays are turned on. The *transition* signal from **dig_counter** provides the signal to the FSM that the button entry is finished; it asserts when the counter reaches 4 (100, more on this later). This also feeds the *endstate* input into **enable_seven_segs**, which disables all the seven segment displays; this means that the seven segment displays are turned off while the counter is "full," i.e. all digits have been entered. This means that the code clears soon after entry of the fourth digit. The

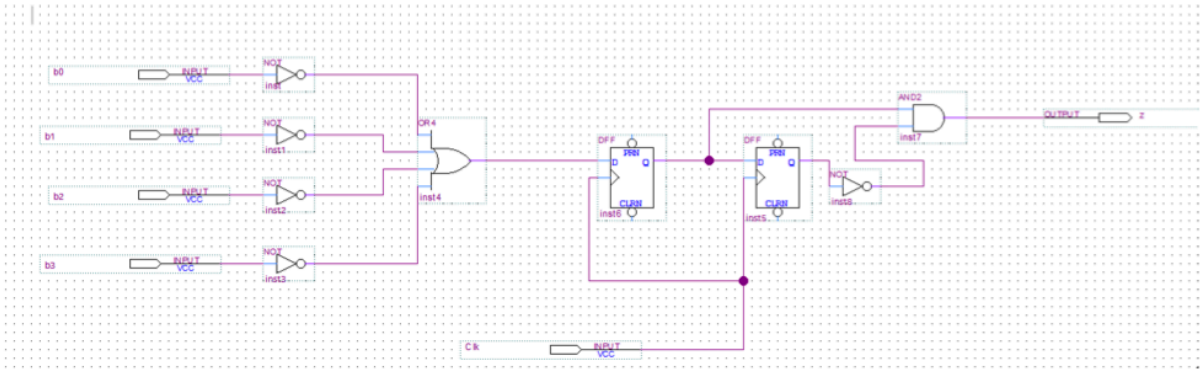
Load_Data[1..0] output bus outputs the binary value supplied by the button presses. The *write_enable* output, which allows writing to the registers, is activiated when the buttons are pressed (*z* from **buton_press_detection_logic**) AND the counter is NOT full (*transition* is not equal to 1), hence the AND and NOT gates.

Since the timing of this module is somewhat complex with the flip flops and synchronization of elements, here is timing explanation for the module's operation.



buton_press_detection_logic

This module first inverts every button signal so that the inputs to the OR gate are high when the buttons are pressed, not the other way around. The OR gate detects if any of the inputs are high; it is assumed that the user presses a single button at a time. The D-Flip-Flops (abbreviated DFF from this point onward) allow the module to hold the state of the buttons at the previous clock edge and the clock edge before that. This means that the module can detect when button state changes from none pressed to button(s) pressed; it functions as a simple 2-bit shift register. When the earlier state is not pressed (indicated by the not gate on the left DFF) AND the later state is pressed (the right DFF outputs 1), the output *z* asserts. This means that *z* only asserts when the button is initially pressed.



push_button_encoder

This module takes the button inputs *b0, b1, b2,* and *b3*, and, assuming they are one-hot inputs, encodes them to the binary values 00, 01, 10, 11, respectively, on the outputs *z1* and *z0*. Note that the buttons output zero when pressed; the inputs are not inverted going into the module.

b_3	b_2	b_1	b_0	z_1	z_0
0	0	0	0	d	d
0	0	0	1	d	d
0	0	1	0	d	d
0	0	1	1	d	d
0	1	0	0	d	d
0	1	0	1	d	d
0	1	1	0	d	d
0	1	1	1	1	1
1	0	0	0	d	d
1	0	0	1	d	d
1	0	1	0	d	d
1	0	1	1	1	0
1	1	0	0	d	d
1	1	0	1	0	1
1	1	1	0	0	0
1	1	1	1	d	d

z_1:

b_1b_0 \ b_3b_2	00	01	11	10
00	d	d	d	d
01	d	d	0	d
11	d	1	d	1
10	d	d	0	d

$z_1 = \overline{b_3} + \overline{b_2}$

z_0:

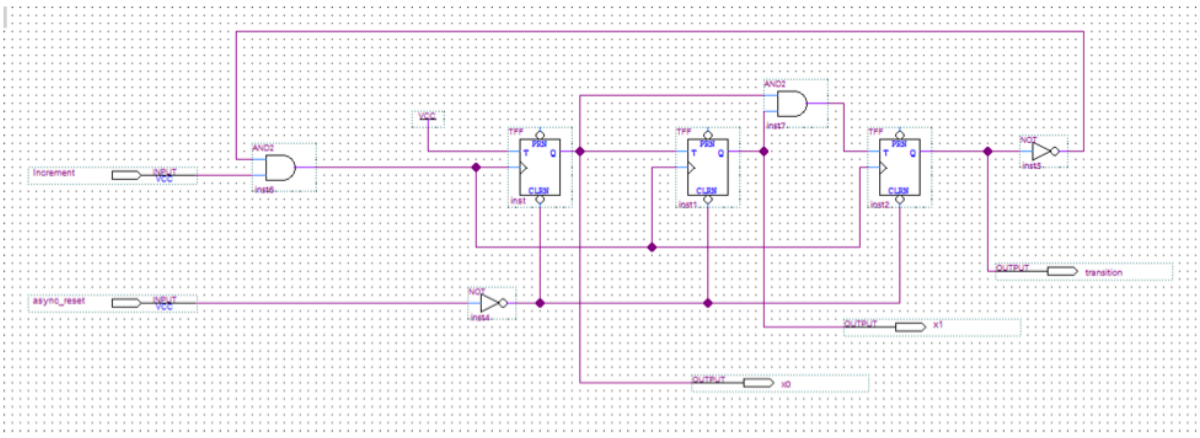
b_1b_0 \ b_3b_2	00	01	11	10
00	d	d	d	d
01	d	d	1	d
11	d	1	d	0
10	d	d	0	d

$z_0 = \overline{b_3} + \overline{b_1}$

```
module push_button_encoder(b0, b1, b2, b3, z0, z1);

    input b0, b1, b2, b3;
    output z0, z1;

    assign z0 = ~b1 | ~b3;
    assign z1 = ~b2 | ~b3;

endmodule
```

dig_counter

This module stores a number associated with the number of digits entered, the current address to write to, and which seven segment displays to show. It is a three-bit up counter that holds numbers from 0 to 4, depending on the situation. The asynchronous reset input sets the value stored to zero. This is the state before buttons have been pressed in the code entry process; the write address is 00, none of the seven segment displays are shown, and no digits have been entered. The increment input, which comes in from the **buton_press_detection_logic** module, is connected to the clock of the TFFs in the module, causing the counter to increment by 1. When the counter hits 4, the AND gate the *increment* input passes through normally is closed. This “locks up” the counter until a reset signal comes through. Otherwise, this is essentially a regular 3-bit up counter like those seen in lecture and lab.



enable_seven_segs

This block enables the seven segment displays as digits are entered based on the output of the **dig_counter** module. If each seven-segment display is indexed from 0 to 2 (excluding 3 for reasons described above), the enabled displays should be those with lower indices than the stored value of **dig_counter**. When **dig_counter** is initially at zero, there should be no active displays, and so on. When the counter reaches four, all the displays should be shut off; this is handled by the input *endstate*. *seven0, seven1,* and *seven2* are the outputs to enable the seven segment displays. *addr0* and *addr1* are the values from **dig_counter**. Below is the derivation and Verilog module.

endstate	addr1	addr0	seven0	seven1	seven2	seven3
0	0	0	0	0	0	
0	0	1	1	0	0	
0	1	0	1	1	0	
0	1	1	1	1	1	
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0

seven 0:

endstate addr1 / addr0	00	01	11	10
0	0	1	0	0
1	1	1	0	0

seven 2:

endstate addr1 / addr0	00	01	11	10
0	0	0	0	0
1	0	1	0	0

seven 0 = $\overline{\text{endstate}}\ \text{addr1} + \overline{\text{endstate}}\ \text{addr0}$

seven 1:

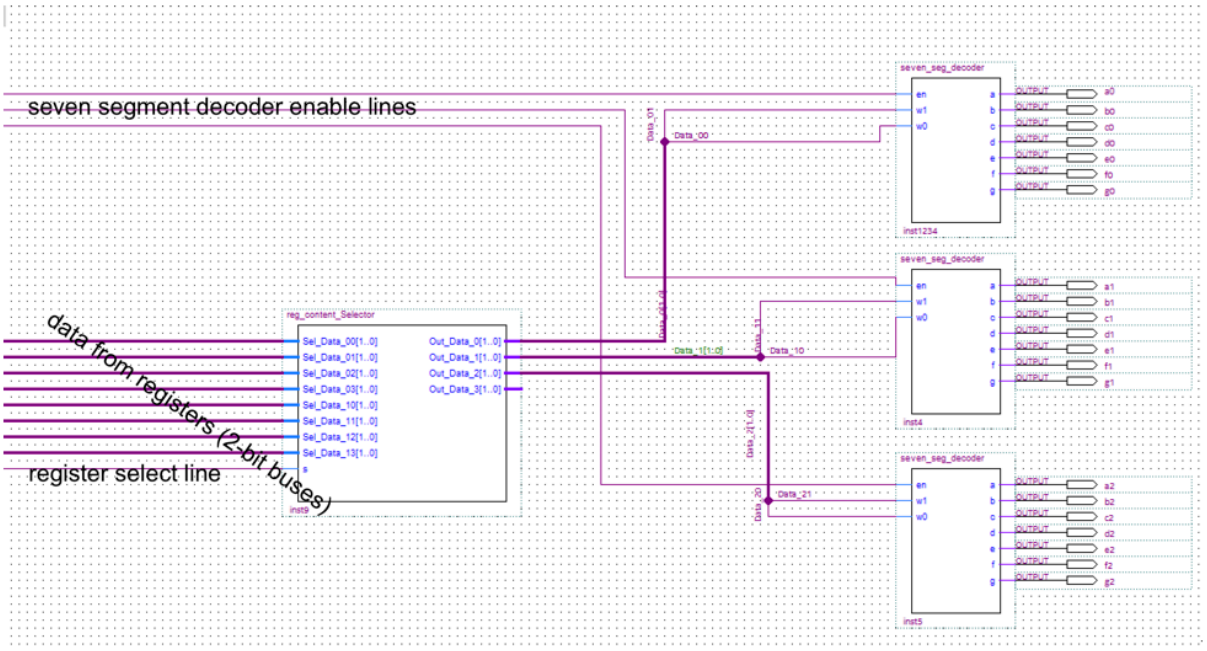
endstate addr1 / addr0	00	01	11	10
0	0	1	0	0
1	0	1	0	0

seven 0 = $\overline{\text{endstate}}\ \text{addr1}\ \text{addr0}$

seven 1 = $\overline{\text{endstate}}\ \text{addr1}$

```
module enable_seven_segs(addr0, addr1, endstate, seven0, seven1, seven2);
    input addr0, addr1, endstate;
    output seven0, seven1, seven2;
    assign seven0 = addr0 & ~endstate | addr1 & ~endstate;
    assign seven1 = addr1 & ~endstate;
    assign seven2 = addr1 & addr0 & ~endstate;
endmodule
```

Part C



Part C is devoted exclusively to displaying the entered code to seven segment displays. The **reg_content_Selector** module takes the four two-bit buses from each of the registers (eight buses total) and uses a select line to choose which register's data is passed through. The Entry

Code register's buses go into the inputs *Set_Data00, Set_Data01, Set_Data02,* and *Set_Data03*, while the Stored Code register's buses go into the inputs *Set_Data10, Set_Data11, Set_Data12,* and *Set_Data13.* The select line is directly connected to the *switch_regs* output of the **FSM** block. As described before, only three seven segment displays are used because the fourth digit would be cleared immediately with the rest of the code when the fourth key is pressed. The seven segment decoders output to the seven segment displays indexed HEX0, HEX1, and HEX2 on the Altera board.

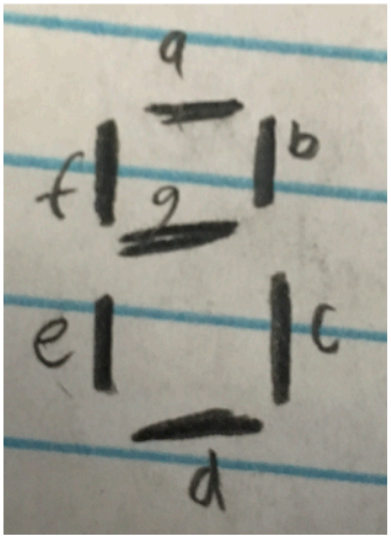
reg_content_Selector

Below is the Verilog code for this module. This is essentially a 2-1 mux, where each input/output consists of four two-bit buses. The first four assign statements assign the zeroth bit of each output bus, using the zeroth bit of the "zero" input if the select line is zero and the zeroth bit of the "one" input if the select line is 1. The next four assign statements accomplish the same thing on the first bit. The similarities to the basic 2-1 mux as described in lab and lecture should be obvious.

```
module reg_content_Selector(Sel_Data_00, Sel_Data_01, Sel_Data_02, Sel_Data_03, Sel_Data_10, Sel_Data_11, Sel_Data_12, Sel_Data_13, s, Out_Data_0, Out_Data_1, 
Out_Data_2, Out_Data_3);

   input [1:0] Sel_Data_00, Sel_Data_01, Sel_Data_02, Sel_Data_03, Sel_Data_10, Sel_Data_11, Sel_Data_12, Sel_Data_13;
   input s;
   output [1:0] Out_Data_0, Out_Data_1, Out_Data_2, Out_Data_3;

   assign Out_Data_0 [0] = ~s & Sel_Data_00 [0] | s & Sel_Data_10 [0];
   assign Out_Data_1 [0] = ~s & Sel_Data_01 [0] | s & Sel_Data_11 [0];
   assign Out_Data_2 [0] = ~s & Sel_Data_02 [0] | s & Sel_Data_12 [0];
   assign Out_Data_3 [0] = ~s & Sel_Data_03 [0] | s & Sel_Data_13 [0];

   assign Out_Data_0 [1] = ~s & Sel_Data_00 [1] | s & Sel_Data_10 [1];
   assign Out_Data_1 [1] = ~s & Sel_Data_01 [1] | s & Sel_Data_11 [1];
   assign Out_Data_2 [1] = ~s & Sel_Data_02 [1] | s & Sel_Data_12 [1];
   assign Out_Data_3 [1] = ~s & Sel_Data_03 [1] | s & Sel_Data_13 [1];

endmodule
```

seven_seg_decoder

This module is very similar to the seven-segment decoder used in multiple labs for the course, but it only accepts two bit input values, providing for values in the range of 1-4. As such, an input of 00 will result in 1, 01 is 2, 10 is 3, and 11 is 4 on the seven-segment display. *en* is an additional input that will prevent any of the segments from lighting while it is asserted. *w1* and *w0* are the most significant bit and least significant bit, respectively, that determine the value displayed. When input into this module, the two-bit buses are broken into their individual bits and labeled in the Part C screenshot as shown to distinguish them. The attached image below shows which segments the *a, b, c, d, e, f,* and *g* outputs and the truth table. Note that the Verilog module is implemented via the truth table method; no K-maps or boolean expressions are used. Note that the segments are lit when their value is 0 and unlit when their value is 1.



en	w_1	w_0	a	b	c	d	e	f	g
0	0	0	1	0	0	1	1	1	1
0	0	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	1	1	0
0	1	1	1	0	0	1	1	0	0
1	0	0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

```
module seven_seg_decoder(en, w1, w0, a, b, c, d, e, f, g);
   input en, w1, w0;
   output reg a, b, c, d, e, f, g;
   always @ (en or w1 or w0)
   begin
      case({en, w1, w0})
         3'b100:
            begin
               a = 1;
               b = 0;
               c = 0;
               d = 1;
               e = 1;
               f = 1;
               g = 1;
            end
         3'b101:
            begin
               a = 0;
               b = 0;
               c = 1;
               d = 0;
               e = 0;
               f = 1;
               g = 0;
            end
         3'b110:
            begin
               a = 0;
               b = 0;
               c = 0;
               d = 0;
               e = 1;
               f = 1;
               g = 0;
            end
         3'b111:
            begin
               a = 1;
               b = 0;
               c = 0;
               d = 1;
               e = 1;
               f = 0;
               g = 0;
            end
```

```
            3'b000:
               begin
                  a = 1;
                  b = 1;
                  c = 1;
                  d = 1;
                  e = 1;
                  f = 1;
                  g = 1;
               end

            3'b001:
               begin
                  a = 1;
                  b = 1;
                  c = 1;
                  d = 1;
                  e = 1;
                  f = 1;
                  g = 1;
               end

            3'b010:
               begin
                  a = 1;
                  b = 1;
                  c = 1;
                  d = 1;
                  e = 1;
                  f = 1;
                  g = 1;
               end

            3'b011:
               begin
                  a = 1;
                  b = 1;
                  c = 1;
                  d = 1;
                  e = 1;
                  f = 1;
                  g = 1;
               end

         endcase

      end

endmodule
```

Note that this report is only on part D in the rubric, as that part is the final, assembled product. Parts A-C primarly focus on the **code_entering_module**. Below is the document with the test cases I chose.

Cpr E 281 FINAL PROJECT
ELECTRICAL AND COMPUTER ENGINEERING
IOWA STATE UNIVERSITY

Final Project Answer Sheet

Name and Student ID: Ross Thedens [redacted]

Date: 11/25/17

PRELAB:

Q1. Design of the project. (Use sheets as needed, attach them here and leave with TA.

Q2. Test plan of the details in terms of input and output.

1) Hard reset, then enter 1111 and the unlock light should appear.

2) Hard reset, then Enter 43, notice it appears on the display, and clear the input with the clear switch. Then press another four keys to see that the key press counter was reset and the display clears after the 4th key press.

3) Hard reset, then enter 1111 to unlock, then enter 3214, noting that the green light next to the unlock light appears. Then flip the lock switch and verify that 3214 unlocks the lock.

4) Hard reset, then Enter 4444 five times total. The red "locked until hard reset" indicator should turn on, as the code is entered

Demonstration Results: _____

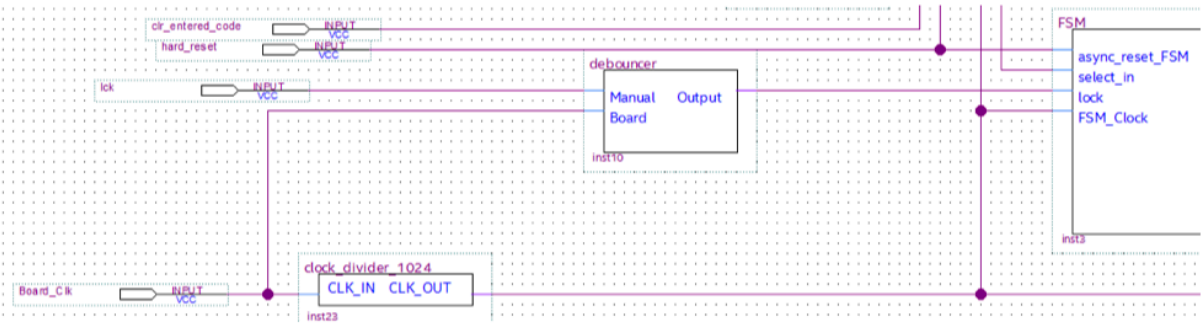
Quality of implementation: _____

Operation of machine: _____

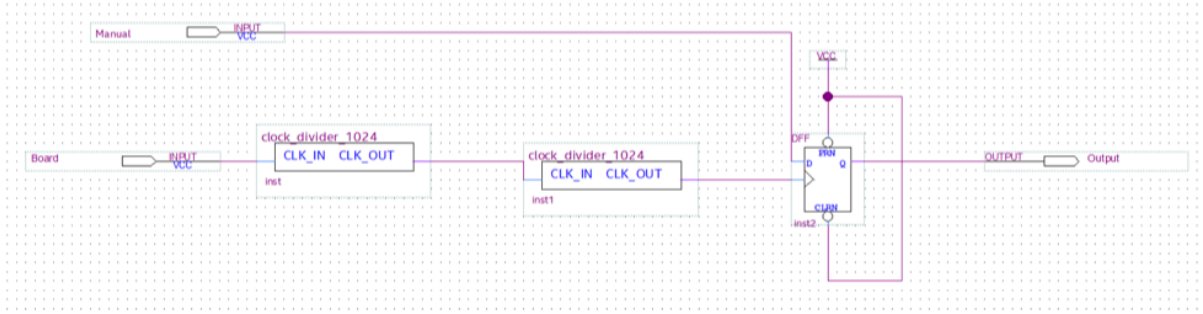
Signature of TA: _____

Addendum: Debouncing the Lock Switch

Note that the lock switch selected (pin AC28) may suffer from bouncing issues on some Altera boards. This typically results in the lock entering state 101 instead of 001 (set new code and lock and enter new unlock code, respectively). The following debounce module attached in the following manner will solve this issue if it is occurring with a particular board:



The debouncer consists of the following:



The effect of this circuit is to divide the clock frequency by 1024 two times (in this case, a 50 MHz clock) and synchronize the switch input with this. This fixes the signal for extended periods to prevent the anomalies that bouncing issues typically cause.