Finite State Machine Design

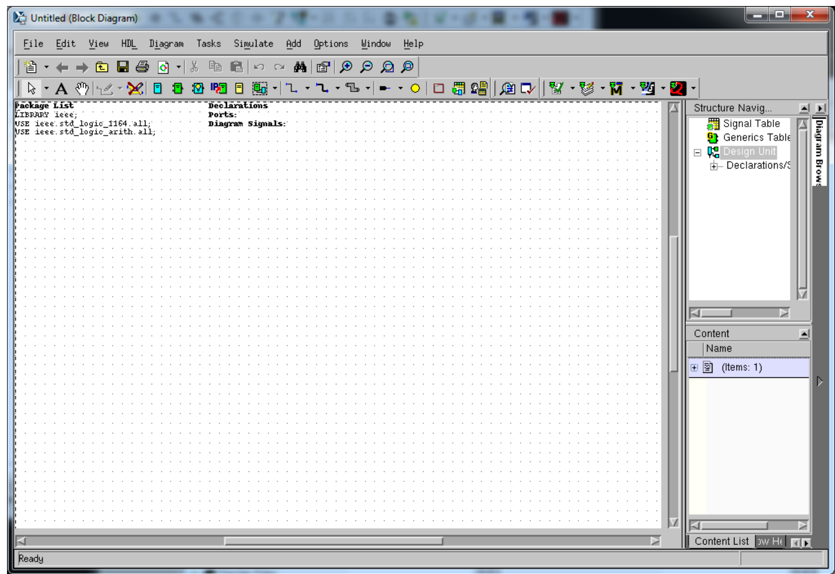
In this tutorial you will learn how to use the Mentor finite state machine editor, as well as how to interface to peripherals on the FPGA board.

More specifically, you will design a programmable combination lock and implement it on the DE2 board. The combination lock can be programmed to recognize a sequence of button presses of length 4.

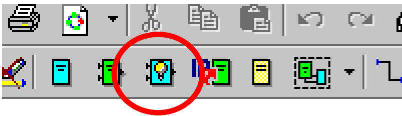
Our design will consist of the following top-level interface:

Name	Type	IN/OUT	Description
clk	std_logic	IN	clock
rst	std_logic	IN	asynchronous input
button	std_logic_vector(3 downto 0)	IN	four buttons, corresponding to the four blue buttons on the DE2 board
mode	std_logic	IN	operation mode: 0: lock mode, 1: program mode
locked	std_logic	OUT	locked indicator 0: unlocked, 1: locked
unlocked	std_logic	OUT	unlocked indicator 0: locked, 1: unlocked
sequence	std_logic_vector(3 downto 0)	OUT	During programming mode, indicates to the user which button in the sequence is being set (i.e. first button press, second button press, third button press, fourth button press)

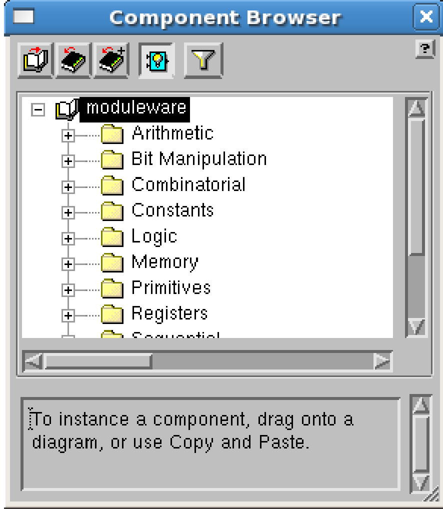
To get started, select **File | New | Design Content | Graphical View | Block Diagram** and click Finish.



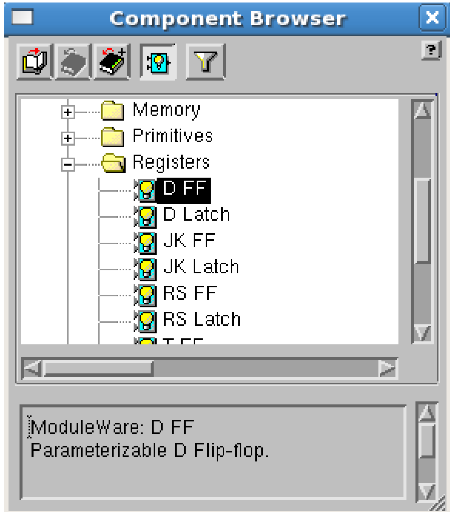
The first thing we need to do is add five registers to store the sequence. To do this, we'll use the "ModuleWare" feature of HDL Designer. Click the ModuleWare button:



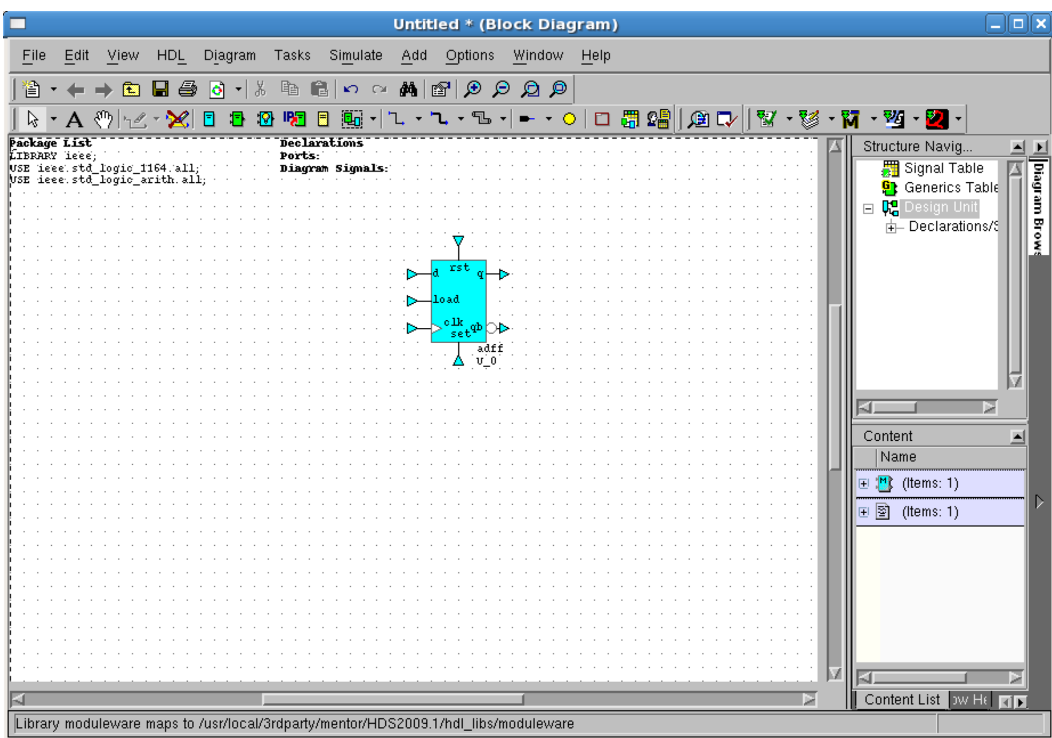
This will bring up the ModuleWare selection window:



Click the [+] next to "Registers":

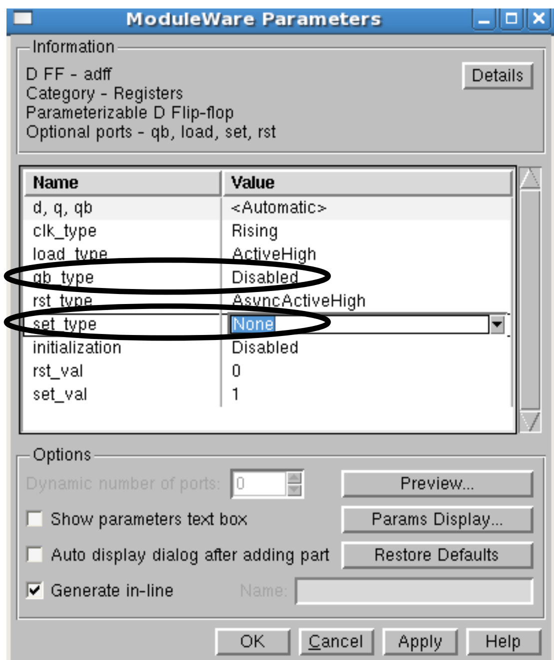


Click the D FF and drag it into your design:

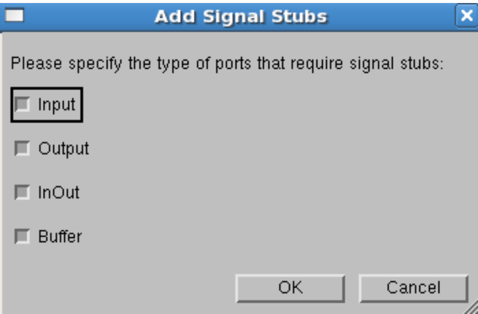


This is a generic register and we can customize it however we want. Our plan is to use 5 of these registers to hold the combination. For this, we don't need the ability to "set" the register (only reset), so we want to delete that input. Also, we don't need the inverted form of the output (the "qb" output), so we want to delete that output.

To do this, double-click the register and make those changes:



Click OK on the parameters window, then right-click the register and select "Add Signal Stubs". Click OK on the following window:

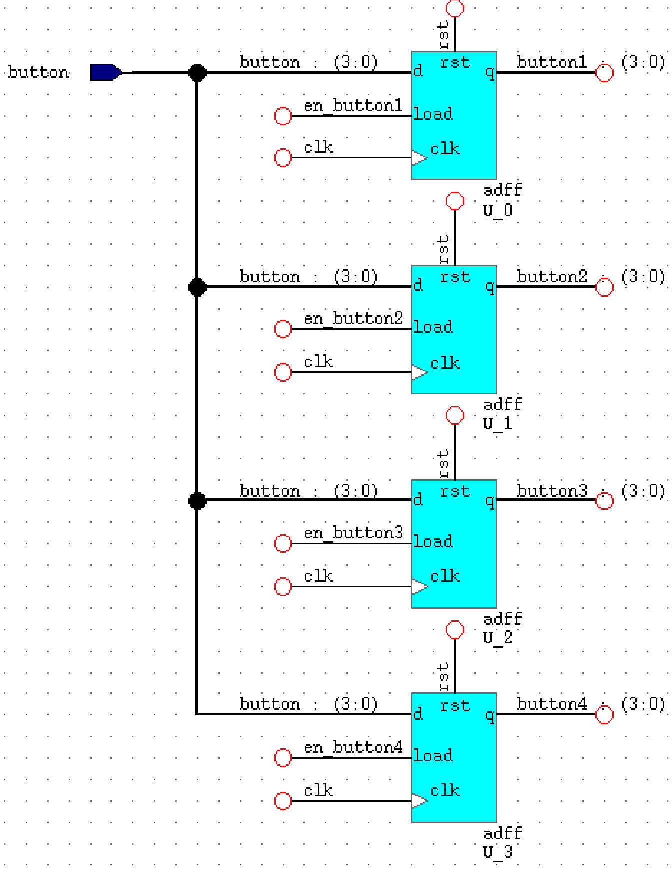


This adds wires to input and output ports. Now let's change the names and properties of the I/O signals. ModuleWare components are different than regular components in that they can automatically adjust themselves to match the signal type and widths for certain ports. In this case, the width of the register is determined by the number of bits in the input and output signals that are attached to the register.

How many bits we do need? This register will hold one of the "digits" of the combination. There are four buttons, so let's make the register 4 bits. Change the I/O signals according to the table below:

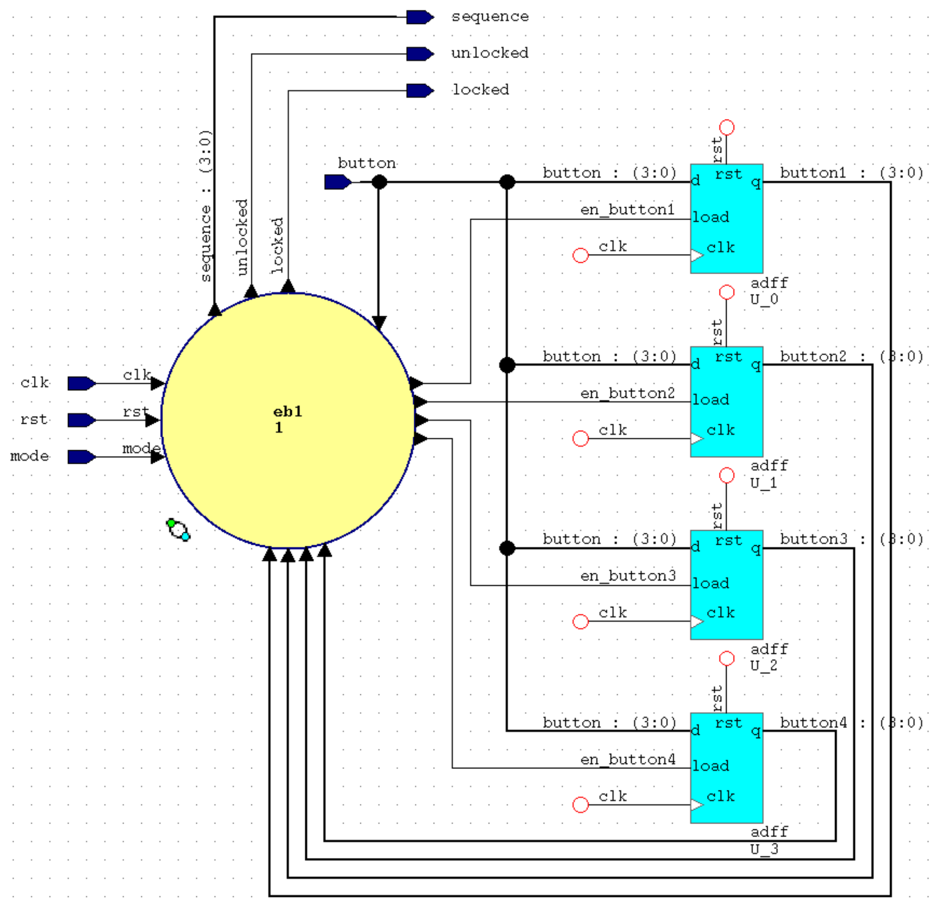
Old name	Old type	New name	New type
d	std_logic	button	std_logic_vector(3 downto 0)
load	std_logic	en_button1	std_logic
q	std_logic	button1	std_logic_vector(3 downto 0)

After this, copy and paste the register and its signals three times, add an input port, and update the signals connected to the new registers according to the diagram below:



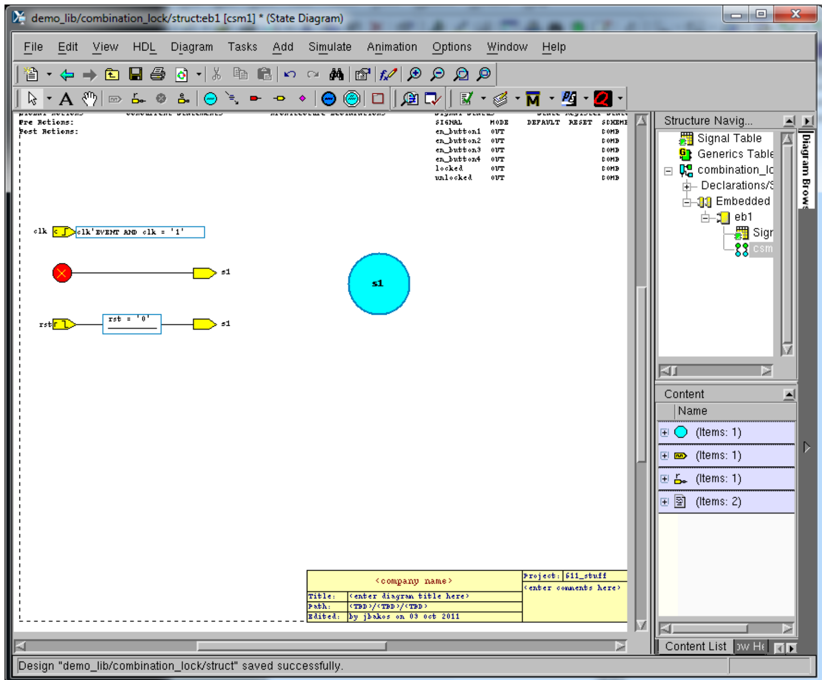
Now let's add the controller. Add an embedded block and make the following changes:

- change the controller shape to a circle (the conventional symbol for a controller),
- add clk and rst inputs to the controller along with input ports,
- connect the four **en_button** signals to the controller as outputs,
- connect the four register outputs to the controller as inputs, and
- add locked and unlocked signals as outputs of the controller with ports,
- add an output to the controller called "sequence" and make it 4 bits:



Now let's design the controller. Double-click the controller and select "State Diagram".

You'll see the following editor window:



In the sidebar on the right side, click “Signals Table” just above “csm”.

“clk” and “rst” will appear in the table, along with the other inputs and outputs (actually since the controller is an embedded block, you’ll see all I/O and local signals from the parent diagram).

Change the rst signal to be an active high signal (the default is active low). To do this, under the “Category” column, change the cell for rst to “Reset (Async High)”:

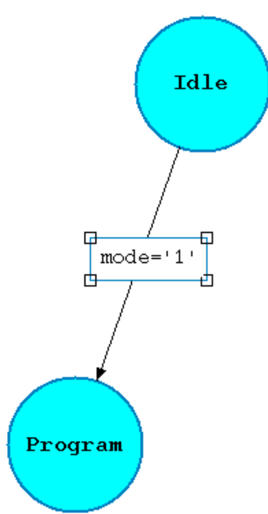
	A	B	C	D	E	F	G	H	I	J	K	L	M
(A)	Group	Name	Mode	Type	Bounds	Initial	Category	Assign In	Expression	Scheme	Default	Reset	Comment
1		button	IN	:d_logic_vecto	(3:0)		Data						
2		clk	IN	std_logic			Clock (Rising)		lk'EVENT AND clk = '1				
3		rst	IN	std_logic			eset (Async High)		rst = '1'				
4		locked	OUT	std_logic			Data	<auto>		Comb			
5		unlocked	OUT	std_logic			Data	<auto>		Comb			
6		mode	IN	std_logic			Data						
7		sequence	OUT	:d_logic_vecto	(3:0)		Data	<auto>		Comb			
8		button1	IN	:d_logic_vecto	(3:0)		Data						
9		button2	IN	:d_logic_vecto	(3:0)		Data						
10		button3	IN	:d_logic_vecto	(3:0)		Data						
11		button4	IN	:d_logic_vecto	(3:0)		Data						
12		en_button1	OUT	std_logic			Data	<auto>		Comb			
13		en_button2	OUT	std_logic			Data	<auto>		Comb			
14		en_button3	OUT	std_logic			Data	<auto>		Comb			
15		en_button4	OUT	std_logic			Data	<auto>		Comb			
16													

Switch back to “csm” in the sidebar.

When designing the controller, an important thing to keep in mind is that our clock signal will be sampling the input values at a much faster rate than the user will be pressing the buttons, so in most cases the state machine will be sampling “0000” as the button values (i.e. no button

pressed). On the other hand, when the user presses a button, the state machine will sample the same button press multiple times.

First let's implement program mode (setting the combination lock). Rename the first state "Idle", add a new state using the button and name it "Program". Connect the Idle state with the Program state with a transition using the button. After this, double-click the transition and in the "IF Condition" field type "mode='1'":



Notice that there is no semicolon in the text you typed into the transition because this is snippet of code will be used as a condition in an IF-statement in the generated HDL.

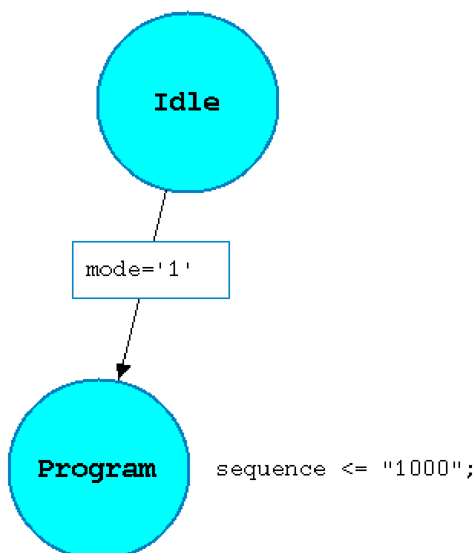
In "Program" mode, we want to indicate to the user that the system is ready to read the first input, so we'll illuminate the first LED light using the "sequence" output.

In addition, we want to enable the first register so that we can capture the first button press from the user.

To do this, double-click the "Program" state and enter the following in to the "State Actions" field:

sequence <= "1000";

You should now have the following:



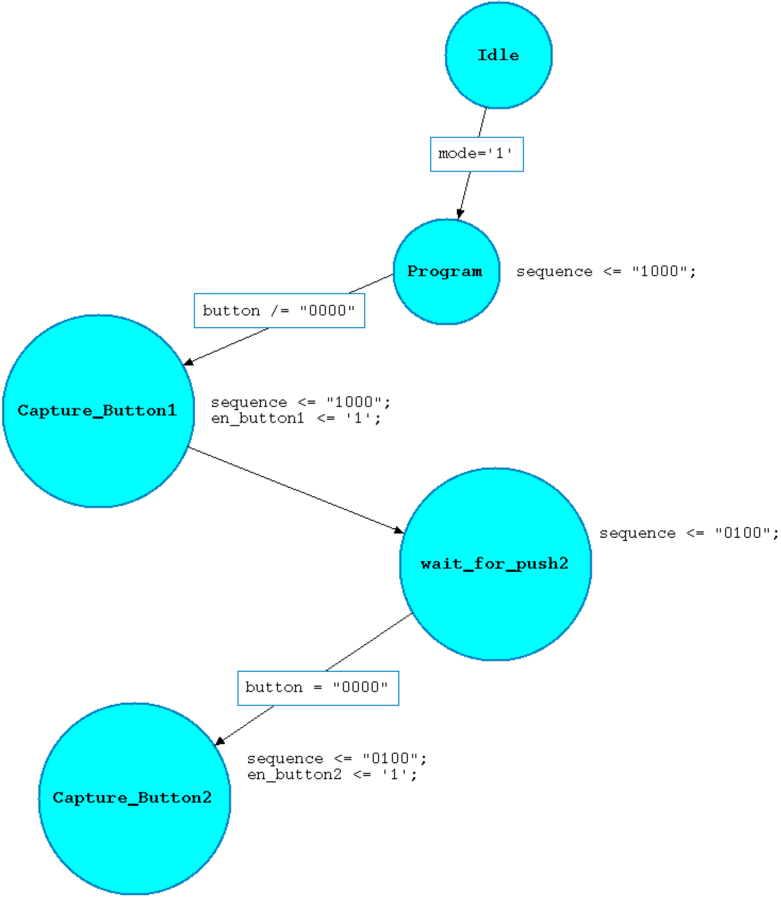
Notice that we didn’t specify the value for **sequence** or in the “Idle” state. This means we assumed they would be assigned to some “default” values whenever we don’t explicitly specify them. However, we must still provide these default values to the editor. Go back to the Signal Table and enter default values for all the controller outputs:

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	Group	Name	Mode	Type	Bounds	Initial	Category	Assign In	Expression	Scheme	Default	Reset	Comment
1		button	IN	:d_logic_vecto	(3:0)		Data						
2		clk	IN	std_logic			Clock (Rising)		lk'EVENT AND clk = '1				
3		rst	IN	std_logic			.eset (Async High)		rst = '1'				
4		locked	OUT	std_logic			Data	<auto>		Comb	'1'		
5		unlocked	OUT	std_logic			Data	<auto>		Comb	'0'		
6		mode	IN	std_logic			Data						
7		sequence	OUT	:d_logic_vecto	(3:0)		Data	<auto>		Comb	hers => '		
8		button1	IN	:d_logic_vecto	(3:0)		Data						
9		button2	IN	:d_logic_vecto	(3:0)		Data						
10		button3	IN	:d_logic_vecto	(3:0)		Data						
11		button4	IN	:d_logic_vecto	(3:0)		Data						
12		en_button1	OUT	std_logic			Data	<auto>		Comb	'0'		
13		en_button2	OUT	std_logic			Data	<auto>		Comb	'0'		
14		en_button3	OUT	std_logic			Data	<auto>		Comb	'0'		
15		en_button4	OUT	std_logic			Data	<auto>		Comb	'0'		
16													

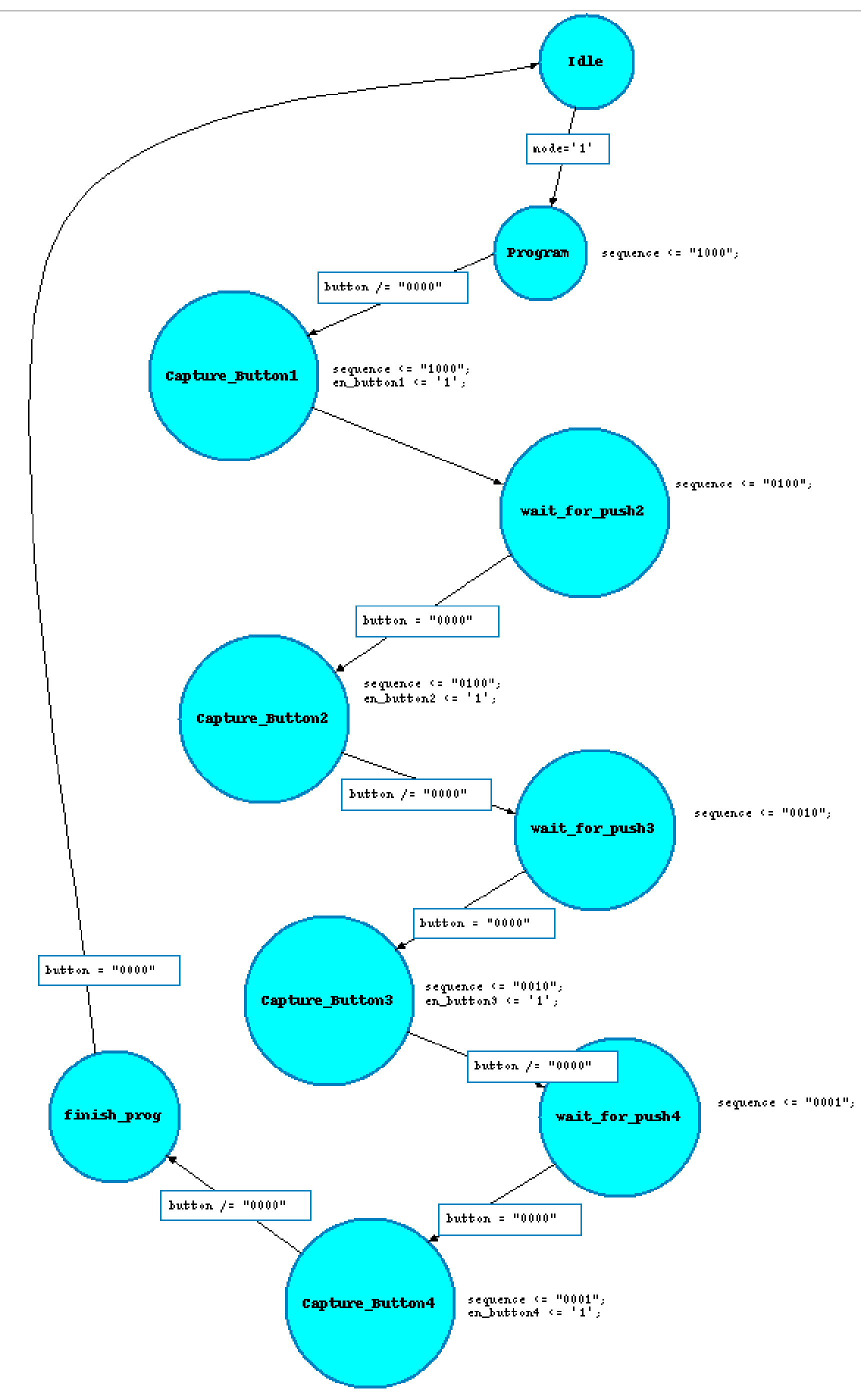
(others => ‘0’)

Note that you may need to scroll the Signals Table window to the right using the scroll bar at the bottom of the window. Return to the “csm” window.

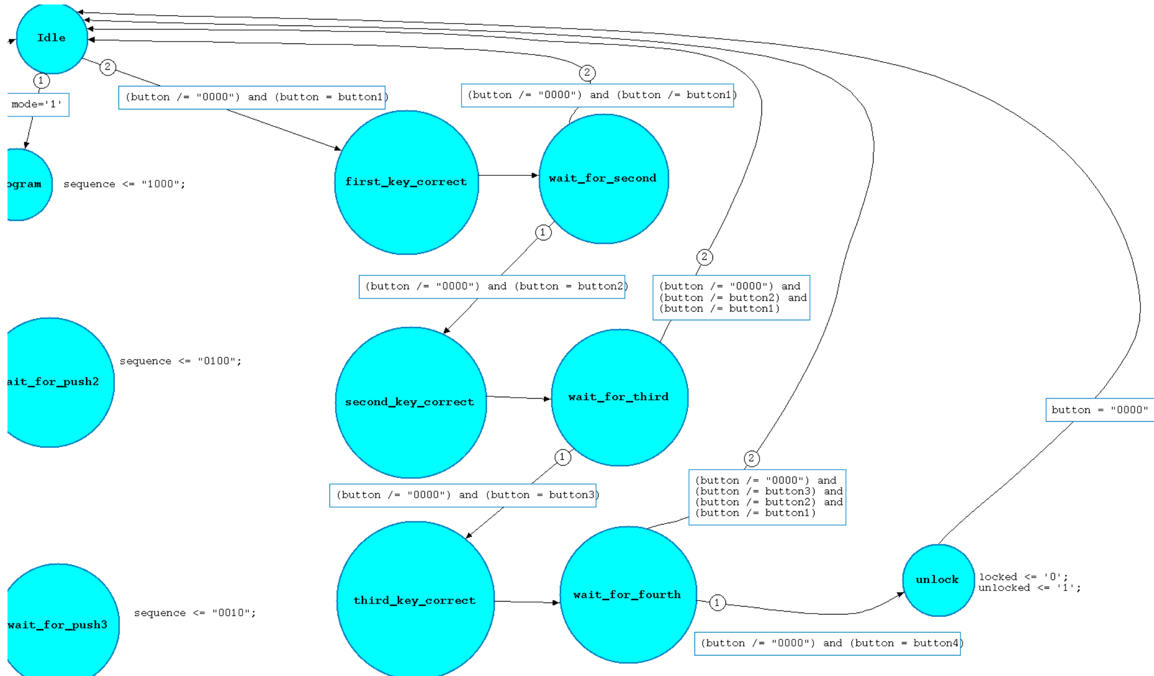
We now have one program state, allowing us to program one of the four values in the sequence. In order to advance to the next state, we must wait until the user presses one of the buttons. To do this, add two new states as shown below:



This way, the user must press a button to progress to the "Capture_Button1" state where the button push is saved in the register. In the next clock cycle, the controller will proceed to the wait_for_push2 state, where it will wait for the next button push and repeat the process for buttons 2-4:



If the controller is not in program mode, it must be able to recognize a correctly-entered sequence. To do this, add the following states and transitions:



This looks complicated, but this is just a method to check that each button is pressed in order. In general, any error will cause the FSM to return to the start state unless there's a repeated button in the combination. If the entire sequence is entered correctly, the controller will signal that the lock is now unlocked and wait for a button press to return to being locked (and return back to the Idle state).

Now let's simulate your combination lock design. Plot the following signals in the wave window:

- clk
- rst
- button, mode
- button1, button2, button3, button4
- sequence
- unlocked, locked
- csm1_current_state

You can use the following Modelsim script to setup the signals and provide sample stimulus for your design:

```
# add the waves
add wave -divider globals
add wave clk
add wave rst
add wave -divider inputs
add wave button
add wave mode
add wave -divider outputs
add wave locked
```

```
add wave unlocked
add wave sequence
add wave -divider {internal signals}
add wave button1
add wave button2
add wave button3
add wave button4
add wave csm1_current_state

# setup the clock, initialize input signals, do reset
force clk 1 0, 0 {50 ns} -r 100
force rst 1
force button "0000"
force mode 0
run
force rst 0
run

# switch to program mode
force mode 1
run 200
# enter new combination
force mode 0
force button "1000"
run 200
force button "0000"
run 200
force button "0100"
run 200
force button "0000"
run 200
force button "0010"
run 200
force button "0000"
run 200
force button "0001"
run 200
force button "0000"
run 200

# try to unlock, but get the second value in the sequence wrong
force button "1000"
run 200
force button "0000"
run 200
force button "0010"
run 200
force button "0000"
```

```
run 200

# try again, but this time get the correct combination
force button "1000"
run 200
force button "0000"
run 200
force button "0100"
run 200
force button "0000"
run 200
force button "0010"
run 200
force button "0000"
run 200
force button "0001"
run 200
force button "0000"
run 200
```

If the design passed simulation, it's now time to test it on the DE2 board.

For this design we need the following two inputs:

- **button**: For this we'll use the four blue buttons on the lower-right side of the board. In the DE2 board, these buttons are accessed using the input signal KEY(3 downto 0), so we'll need to reconcile this difference in name. In addition, the buttons on the DE2 board are active-low, meaning that they are assigned to 1 when not pressed and 0 when pressed, so we'll need to invert these values before passing them to the combination lock design

- **mode**: For this we'll use the left-most toggle switch located on the lower-left of the DE2 board. On the DE2 board, this array of switches is named SW(17 downto 0) but we'll just get need SW(17). When the switch is down the corresponding value is 0. When it's up the corresponding value is 1, so up means program mode.

We'll also need the following outputs:

- sequence: We need four LEDs to act as our programming sequence indicators. The DE2 board has 18 red LEDs and 9 green LEDs. To get the most "bang for our buck" we'll use as many of the LEDs as possible, so we'll associate each group of four LEDs—starting from the left—as the four program sequence indicators. This only requires 16 total LEDs, so we'll leave the right-most two off. The DE2 board names these signals LEDR(17 downto 0). Also, like the buttons, the LEDs are also active-low

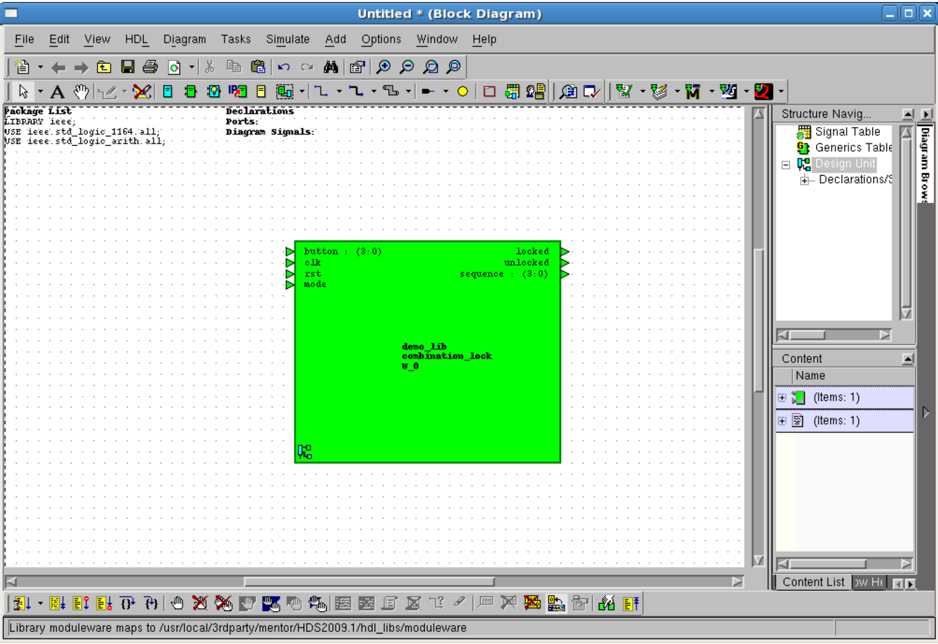
- **locked** and **unlocked**: We'll use the left-most four green LEDs to indicate locked and the right-most four green LEDs to indicate unlocked. The DE2 board names these signals LEDG(17 downto 0).

Finally, we'll need the following "global" signals:

- clk: The DE2 board provides a 50 MHz clock, which is named CLOCK_50. We'll need to reconcile the naming difference.

- rst: For reset we'll use the right-most toggle switch, SW(0). We'll need to also reconcile the naming difference for this signal.

In order to adapt our combination lock design's interface to the DE2 signals, we'll need to create a *wrapper*. Create a new block diagram and instance your combination lock design:



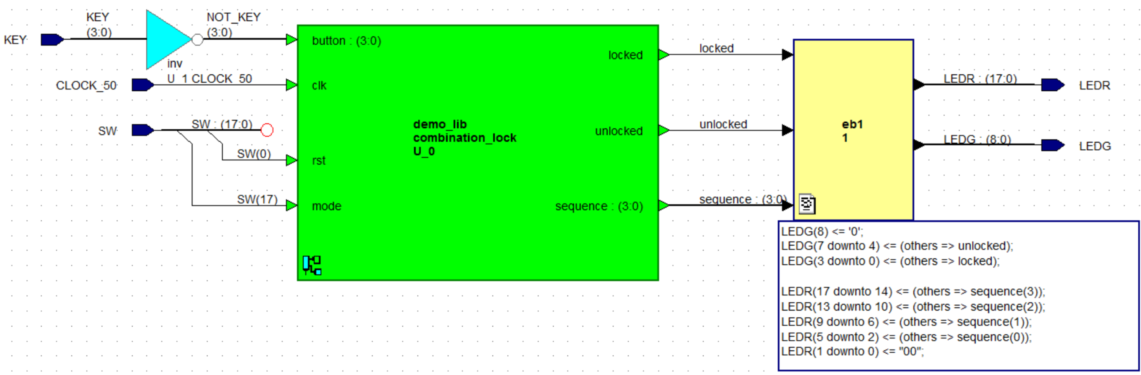
Let's the I/O pins corresponding to each of the DE2 board signals and wire them up as follows:

combination_lock interface	wrapper interface
button(3 downto 0)	KEY(3 downto 0)
clk	CLOCK_50
rst	SW(0)
mode	SW(17)
locked	all of LEDG(7 downto 4)
unlocked	all of LEDG(3 downto 0)
sequence(3)	all of LEDR(17 downto 14)
sequence(2)	all of LEDR(13 downto 10)
sequence(1)	all of LEDR(9 downto 6)
sequence(0)	all of LEDR(5 downto 2)

Also, we need to drive the remaining LED signals using constants:

LEDR(1 downto 0) <= “00”;
LEDG(8) <= ‘0’;

To do this, create a new block diagram named “combination_lock_wrapper” as shown:



Notice that I used a ModuleWare component to invert the KEY signal.

Once you designed this, generate its VHDL and then perform the Quartus II Synthesis flow on this design.

If prompted, using the following FPGA part information (if you are prompted, although you may not be prompted if you already entered this information in the previous tutorial):

FPGA Vendor: Altera
Family: cyclone ii
Device: ep2c35
Package: f672c
Speed: 6

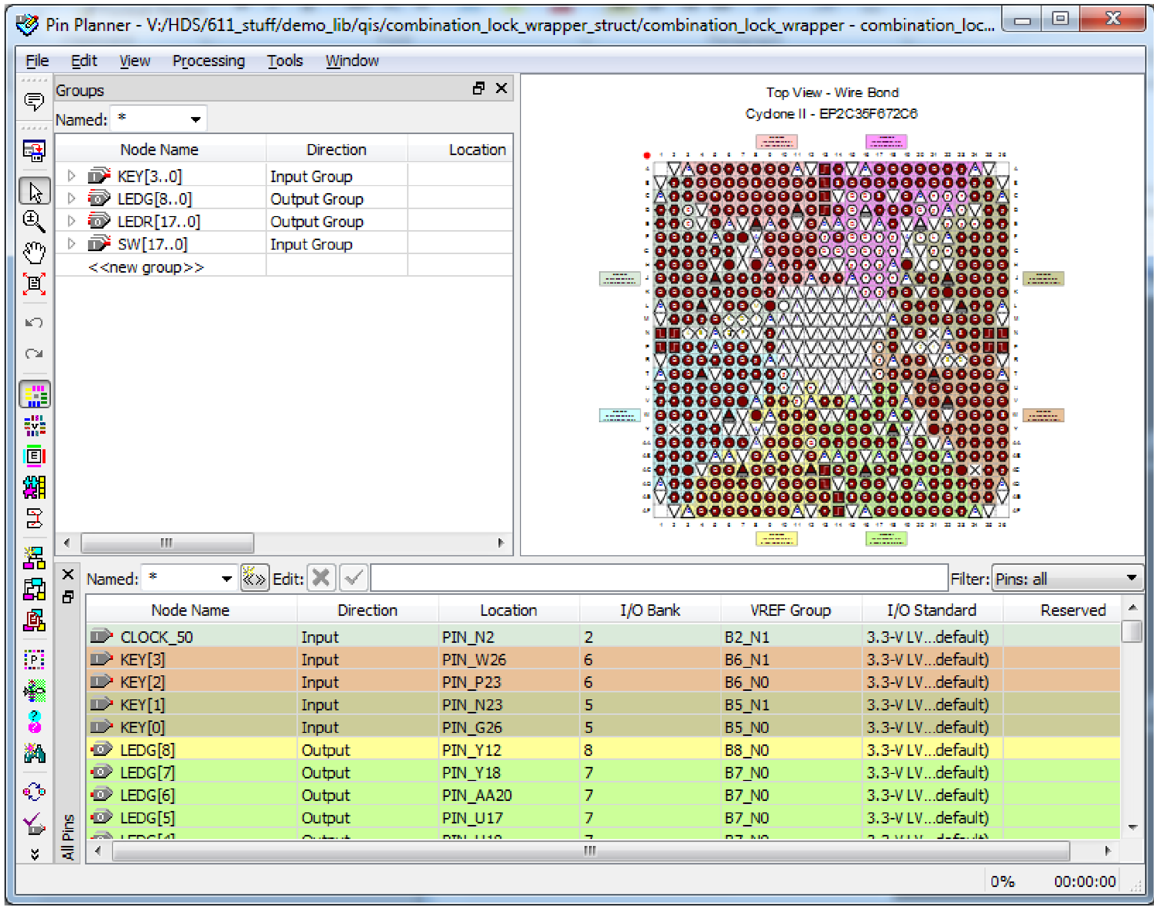
After this, specify 50 MHz as your design frequency, make sure both checkboxes are checked, and click OK.

Assuming the synthesis was successful, open Quartus II by typing “quartus” into your terminal. Open the Quartus project file that was generated by HDL Designer by selecting File | Open Project, then browsing to the HDS directory, into your project directory, into your library directory, then into the qis directory, into the “combination_lock_wrapper_struct” directory, then open the QPF file.

At this point the only thing we need to do before place-and-route is to import the FPGA pin mappings, which is a text file that associates your top-level signals, such as KEY, to actual physical pin numbers on the FPGA.

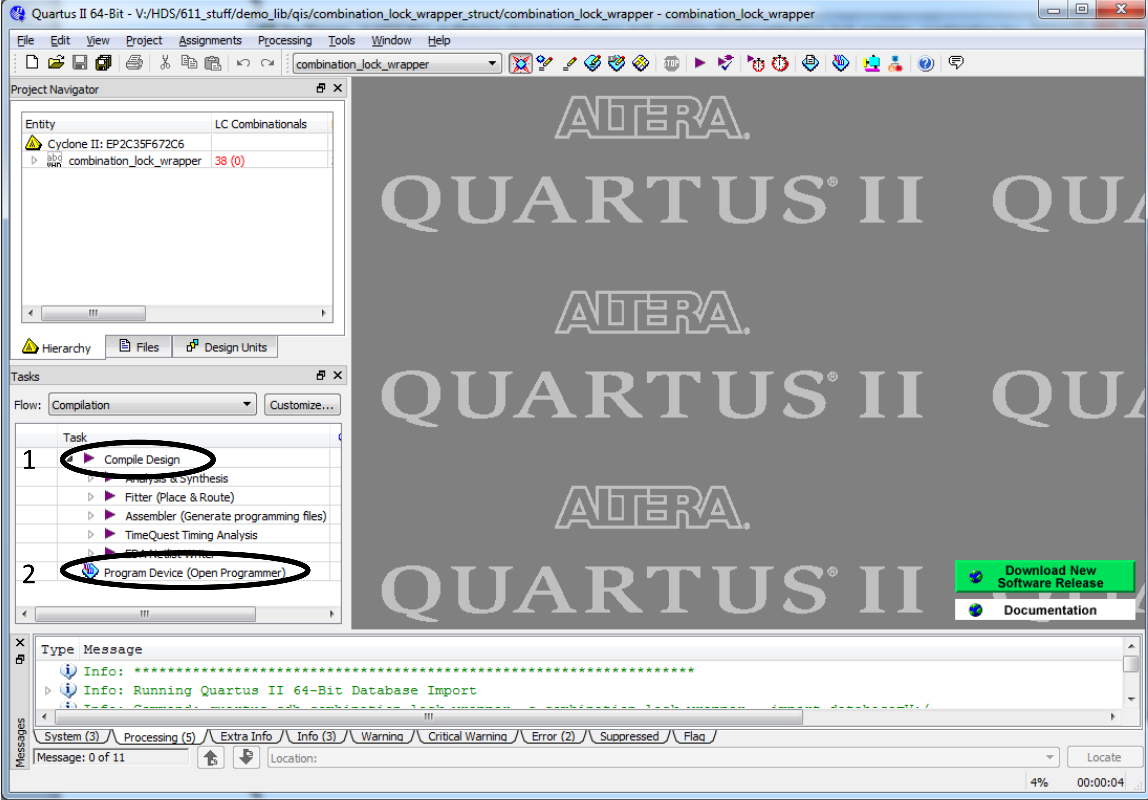
To do this, select Assignments | Import Assignments then select “/usr/local/3rdparty/csce611/CPU_support_files/DE2_pin_assignments.csv” and click OK.

After this, select Assignments | Pin Planner and you should see a window like this:



Notice how each of your top-level ports has an associated physical pin number.

Once you’ve done this, close the Pin Planner and double-click “Compile Design”:



This will place-and-route the design.

Assuming this is successful, attach your FPGA board and double-click on “Program Device”.

Test the functionality of your design on the FPGA board.