BEE 271 labs

Nicole Hamilton
https://faculty.washington.edu/kd1uj

Background

June 2015

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	**22**	23	24	25	26	27
28	29	30				

July 2015

S	M	T	W	T	F	S
			1	2	3	4
5	**6**	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Jun 16: Spring grades in to the registrar. BEE 271 had only 10 students and was expected to be cancelled.

Jun 18: Learned the class was on and that the old version of Quartus needed to use our old Terasic FPGA boards with lab 1 would no longer run on our PCs.

Jun 19-21: Hurriedly created a lab 1 out of whatever TTL parts I could buy enough of off the racks at Fry's.

Jun 22: First day of instruction.

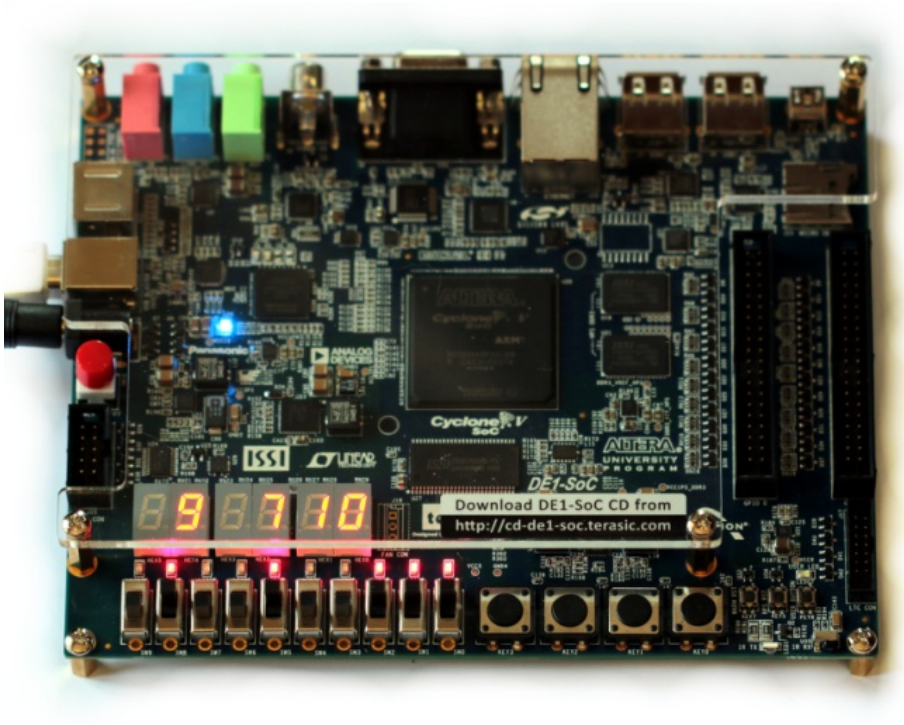
Jun 29: The best replacement was clearly the new but not yet shipping DE1-SoC. When Terasic promised they could ship 14 boards immediately, we ordered the same day.

Jul 1: Boards arrive.

Jul 3-5: Created lab 2, the adding machine assignment. The next two weekends, I wrote lab 3, which I've since split into two assignments.

Jul 6: The class would need a lab 2 assignment.

Terasic DE1-SoC



FPGA

Altera Cyclone V SoC

85K programmable logic elements

64 MB SDRAM

Hard processor system (HPS)

Dual-core ARM

2 hard memory controllers

Runs Linux and “bare metal” apps

Board

6 seven segment displays

10 switches

10 LEDs

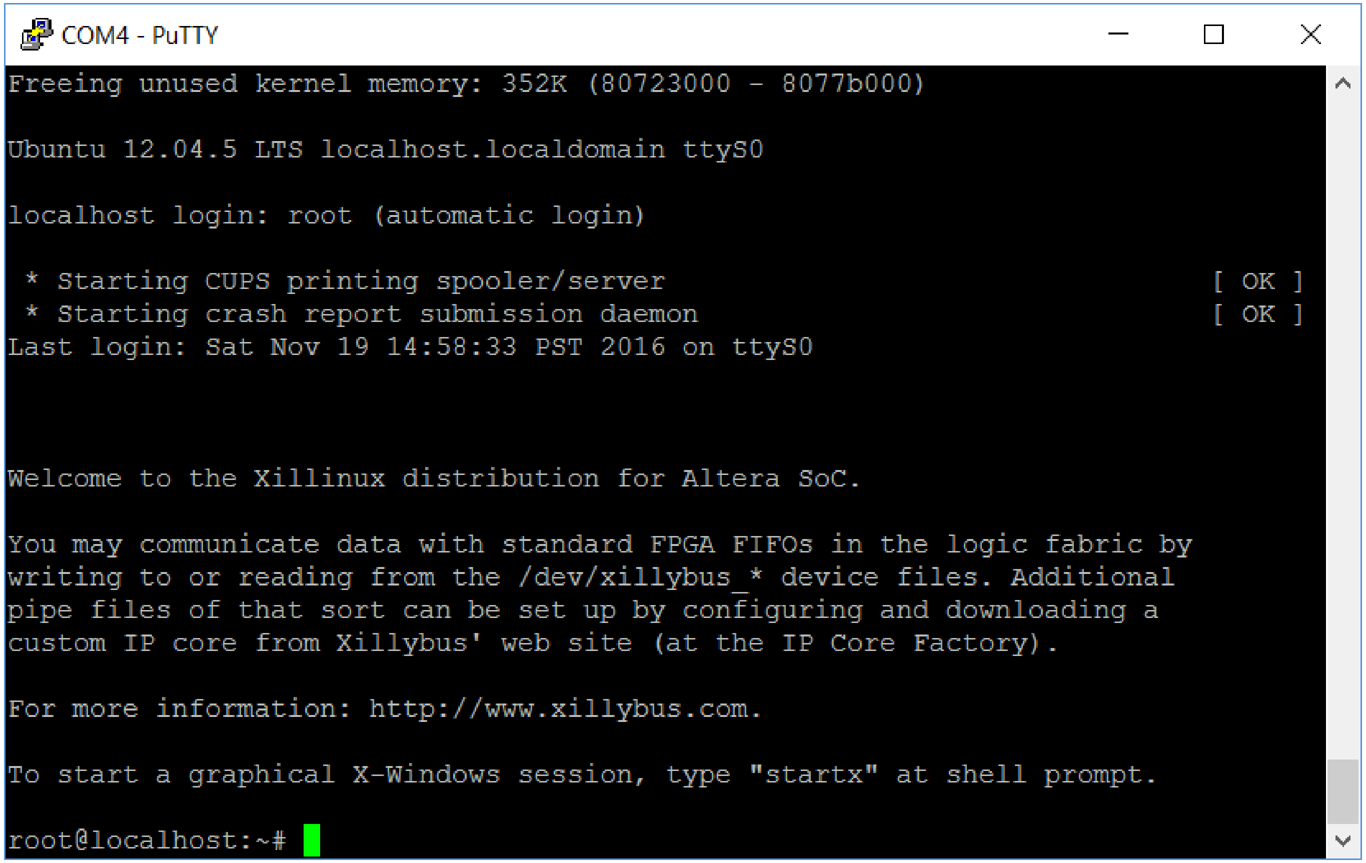
4 pushbuttons

2 40-pin GPIO headers

1 GB DDR3 SDRAM

+ Micro SD, USB, Ethernet, VGA, ADC, keyboard, mouse, audio, video

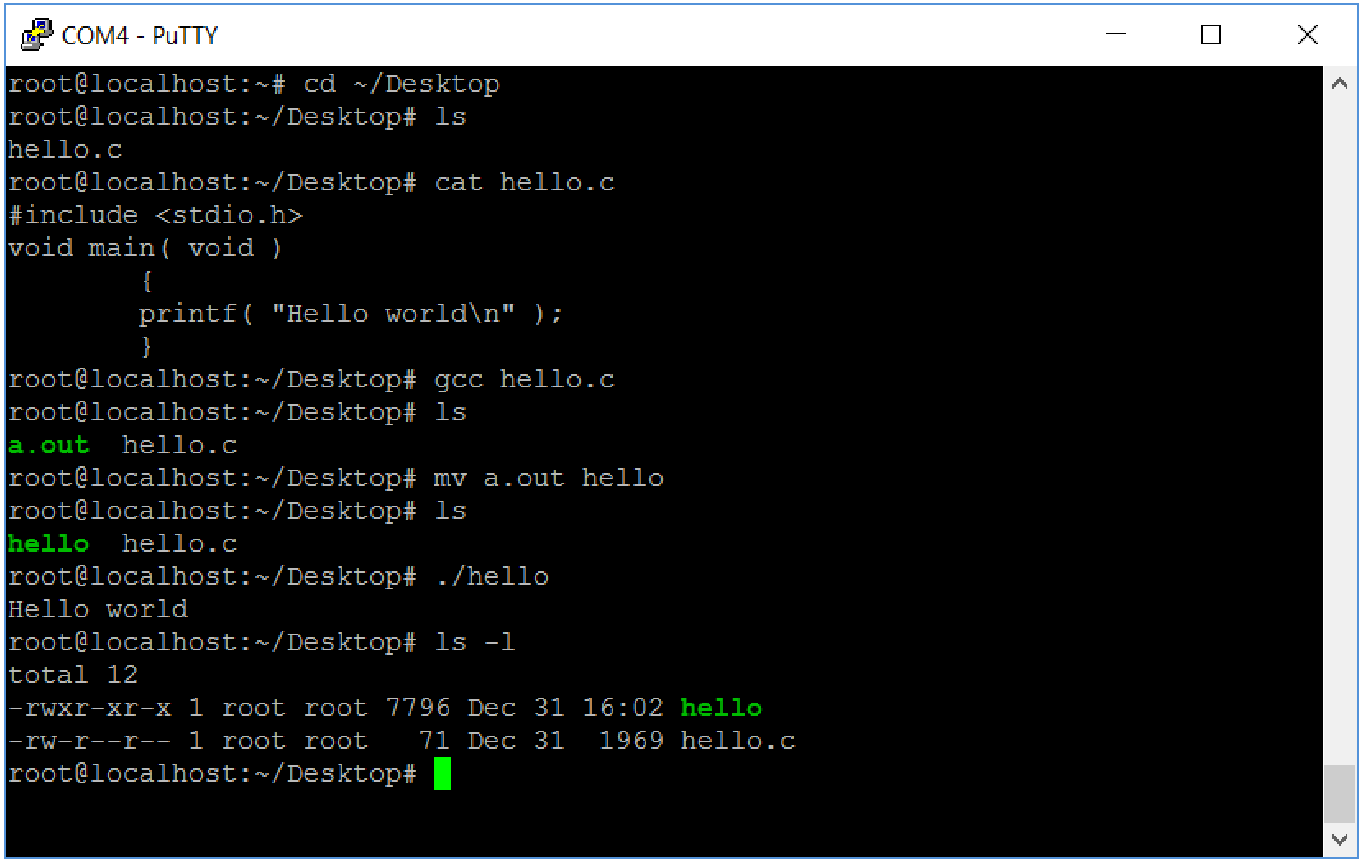
Boot Ubuntu from an Micro SD card as a command window via PuTTY.



```
Freeing unused kernel memory: 352K (80723000 - 8077b000)

Ubuntu 12.04.5 LTS localhost.localdomain ttyS0

localhost login: root (automatic login)

 * Starting CUPS printing spooler/server                                [ OK ]
 * Starting crash report submission daemon                              [ OK ]
Last login: Sat Nov 19 14:58:33 PST 2016 on ttyS0



Welcome to the Xillinux distribution for Altera SoC.

You may communicate data with standard FPGA FIFOs in the logic fabric by
writing to or reading from the /dev/xillybus_* device files. Additional
pipe files of that sort can be set up by configuring and downloading a
custom IP core from Xillybus' web site (at the IP Core Factory).

For more information: http://www.xillybus.com.

To start a graphical X-Windows session, type "startx" at shell prompt.

root@localhost:~#
```

With a keyboard, mouse and VGA display, it runs the Ubuntu desktop.

Download and run the full GCC toolchain, Firefox browser and anything else you like.



```
root@localhost:~# cd ~/Desktop
root@localhost:~/Desktop# ls
hello.c
root@localhost:~/Desktop# cat hello.c
#include <stdio.h>
void main( void )
        {
        printf( "Hello world\n" );
        }
root@localhost:~/Desktop# gcc hello.c
root@localhost:~/Desktop# ls
a.out  hello.c
root@localhost:~/Desktop# mv a.out hello
root@localhost:~/Desktop# ls
hello  hello.c
root@localhost:~/Desktop# ./hello
Hello world
root@localhost:~/Desktop# ls -l
total 12
-rwxr-xr-x 1 root root 7796 Dec 31 16:02 hello
-rw-r--r-- 1 root root   71 Dec 31  1969 hello.c
root@localhost:~/Desktop#
```

Possible very cool microprocessor course project

Build a memory-mapped device and the device driver to go with it.

1. A bare metal app in C built with DS-5 and running in the ARM.
2. A memory-mapped device in Verilog compiled with Quartus and programmed into the FPGA.
3. Simplest form: memory-mapped control and data registers only.
4. Bonus: add interrupts and interrupt service routines.

Biggest problem: I first have to figure out how to do it myself and the documentation is horrible!

Software environment

Quartus Prime

The Intel/Altera IDE for compiling Verilog to the FPGA, including SystemBuilder to create a new DE1-SoC project.

SignalTap II

A remote logic analyzer that can be compiled onto the FPGA along with your own code.

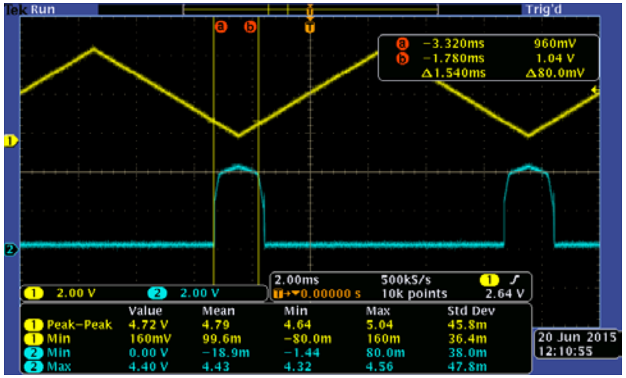
ModelSim

The Verilog simulator.

DS-5

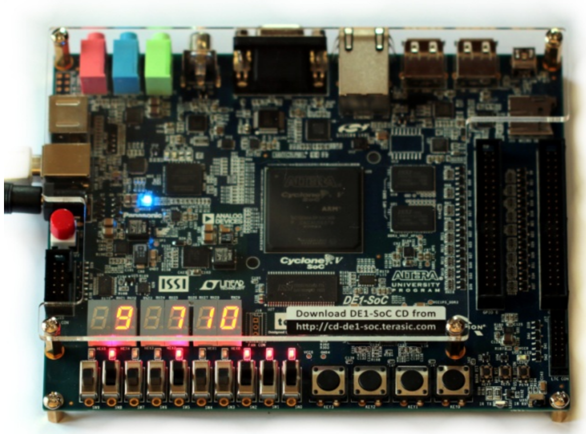
Not used in 271, this is the Eclipse-based ARM IDE for developing and debugging “bare metal” apps. (We have 100 floating licenses that can be used even from home with Husky OnNet.)

Four labs

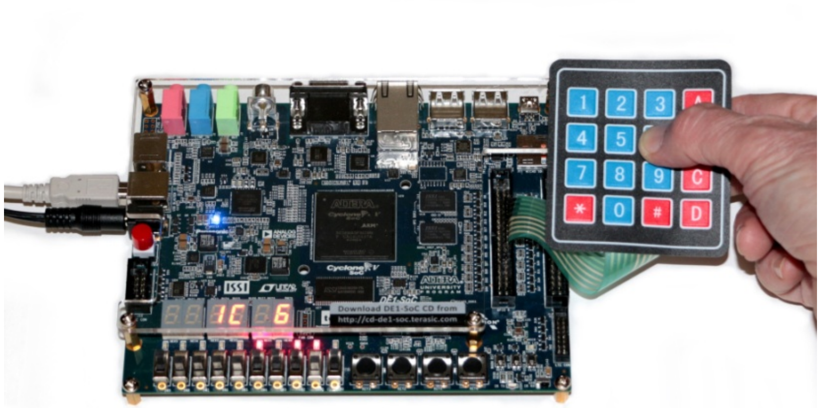


```
a  −3.320ms     960mV
b  −1.780ms     1.04 V
   Δ1.540ms     Δ80.0mV

1  2.00 V      2  2.00 V      2.00ms        500kS/s     1 ⨍
                              T→▼0.00000 s  10k points  2.64 V

               Value    Mean     Min      Max     Std Dev
1 Peak−Peak    4.72 V   4.79     4.64     5.04    45.8m
1 Min          160mV    99.6m    −80.0m   160m    36.4m
2 Min          0.00 V   −18.9m   −1.44    80.0m   38.0m
2 Max          4.40 V   4.43     4.32     4.56    47.8m

20 Jun 2015
12:10:55
```

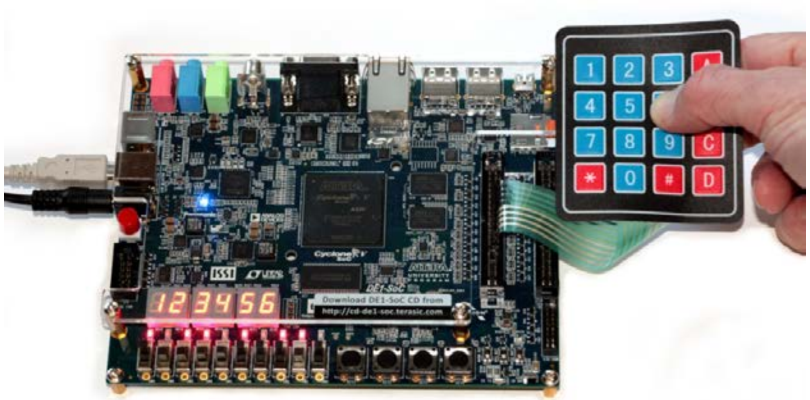
1. Digital logic devices.



2. Hex adding machine.



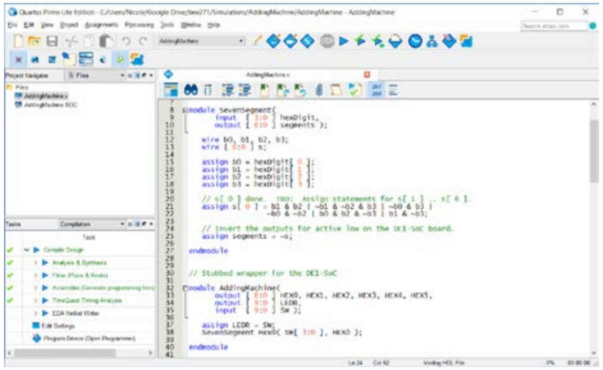
3. Keypad scanner.



4. Keypad debouncer.

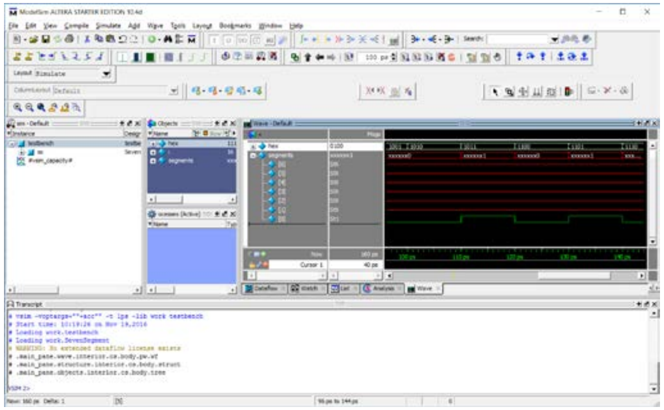
Four group exercises

1. Using the lab instruments.



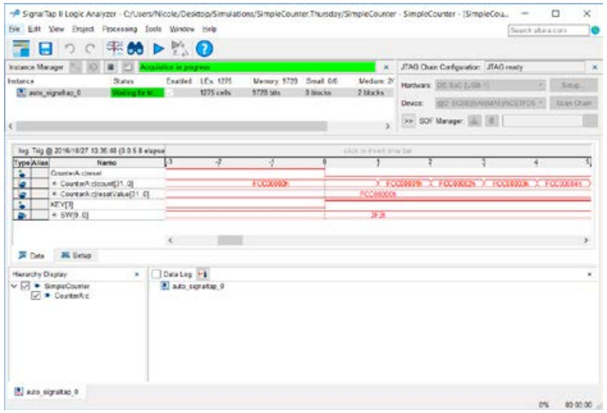
```
module SevenSegment(
        input  [ 3:0 ] hexDigit,
        output [ 6:0 ] segments );

   wire b0, b1, b2, b3;
   wire [ 6:0 ] s;

   assign b0 = hexDigit[ 0 ];
   assign b1 = hexDigit[ 1 ];
   assign b2 = hexDigit[ 2 ];
   assign b3 = hexDigit[ 3 ];

   // s[ 0 ] done.  TBD:  Assign statements for s[ 1 ] .. s[ 6 ].
   assign s[ 0 ] = b1 & b2 | ~b1 & ~b2 & b3 | ~b0 & b3 |
                     ~b0 & ~b2 | b0 & b2 & ~b3 | b1 & ~b3;

   // Invert the outputs for active low on the DE1-SoC board.
   assign segments = ~s;

endmodule


// Stubbed wrapper for the DE1-SoC

module AddingMachine(
        output [ 6:0 ] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
        output [ 9:0 ] LEDR,
        input  [ 9:0 ] SW );

   assign LEDR = SW;
   SevenSegment Hex0( SW[ 3:0 ], HEX0 );

endmodule
```

2. Creating and running a new Verilog project using Quartus.



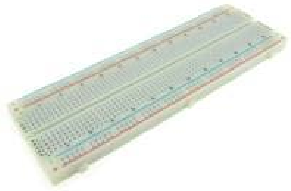
```
# vsim -voptargs=""+acc"" -t 1ps -lib work testbench
# Start time: 10:19:26 on Nov 19,2016
# Loading work.testbench
# Loading work.SevenSegment
# WARNING: No extended dataflow license exists
# .main_pane.wave.interior.cs.body.pw.wf
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree

VSIM 2>
```

3. Using the ModelSim simulator.



4. Using the SignalTap II logic analyzer.

BEE 271 lab kits



830-point (full-size) breadboard

Precut and preformed breadboard jumper wires

16-key numerical keypad

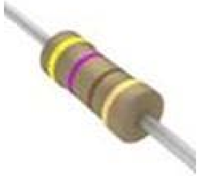
3-piece 20 cm multicolored 40-pin jumper wire “Dupont” ribbon cable set

Texas Instruments SN7400N Quad NAND or equivalent.

Texas Instruments SN7402N Quad NOR or equivalent.

Texas Instruments SN7404N Hex Inverter or equivalent.

Texas Instruments SN74LS86AN Quad XOR or equivalent.

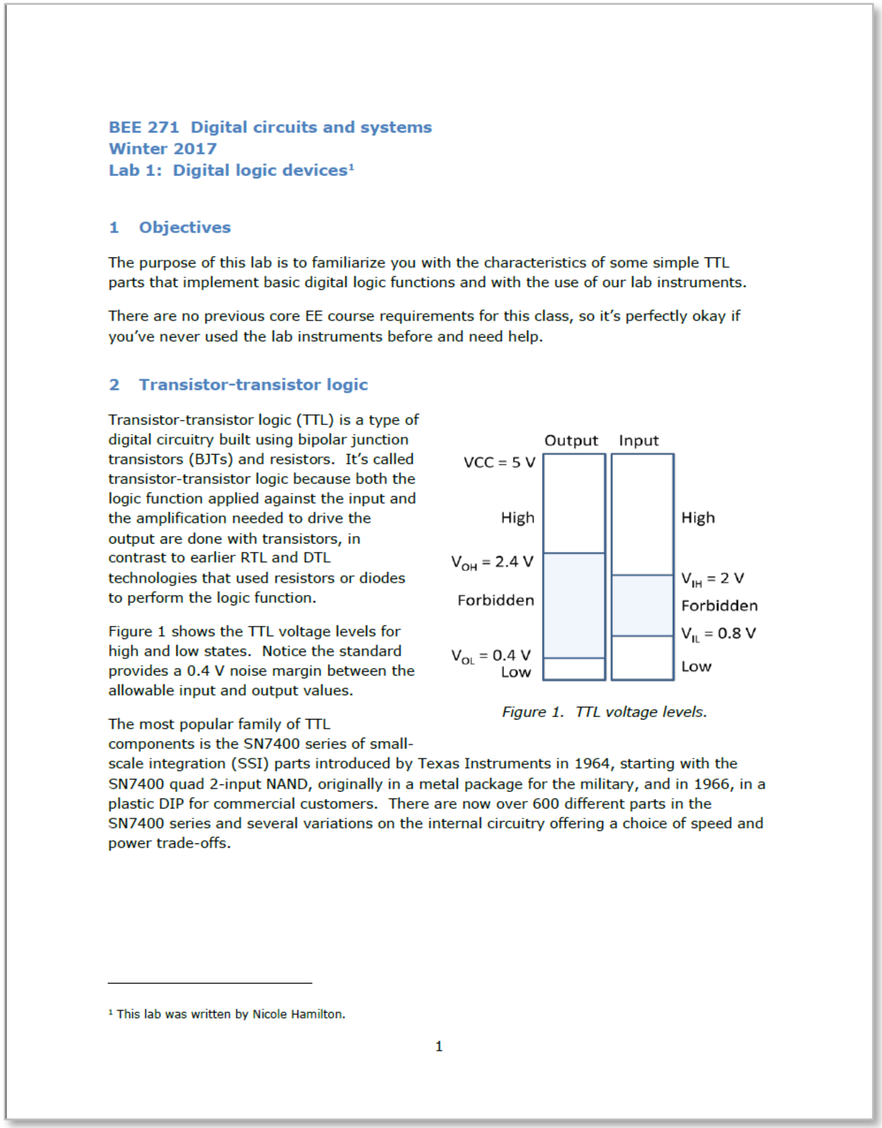


3 Generic 470 ohm, 1/4 watt, 5% resistors

4 Generic 10K ohm, 1/4 watt, 5% resistors

3 Generic red LEDs

Lab 1 Digital logic devices



BEE 271 Digital circuits and systems
Winter 2017
Lab 1: Digital logic devices[1]

1 Objectives

The purpose of this lab is to familiarize you with the characteristics of some simple TTL parts that implement basic digital logic functions and with the use of our lab instruments.

There are no previous core EE course requirements for this class, so it's perfectly okay if you've never used the lab instruments before and need help.

2 Transistor-transistor logic

Transistor-transistor logic (TTL) is a type of digital circuitry built using bipolar junction transistors (BJTs) and resistors. It's called transistor-transistor logic because both the logic function applied against the input and the amplification needed to drive the output are done with transistors, in contrast to earlier RTL and DTL technologies that used resistors or diodes to perform the logic function.

Figure 1 shows the TTL voltage levels for high and low states. Notice the standard provides a 0.4 V noise margin between the allowable input and output values.

Figure 1. TTL voltage levels.

The most popular family of TTL components is the SN7400 series of small-scale integration (SSI) parts introduced by Texas Instruments in 1964, starting with the SN7400 quad 2-input NAND, originally in a metal package for the military, and in 1966, in a plastic DIP for commercial customers. There are now over 600 different parts in the SN7400 series and several variations on the internal circuitry offering a choice of speed and power trade-offs.

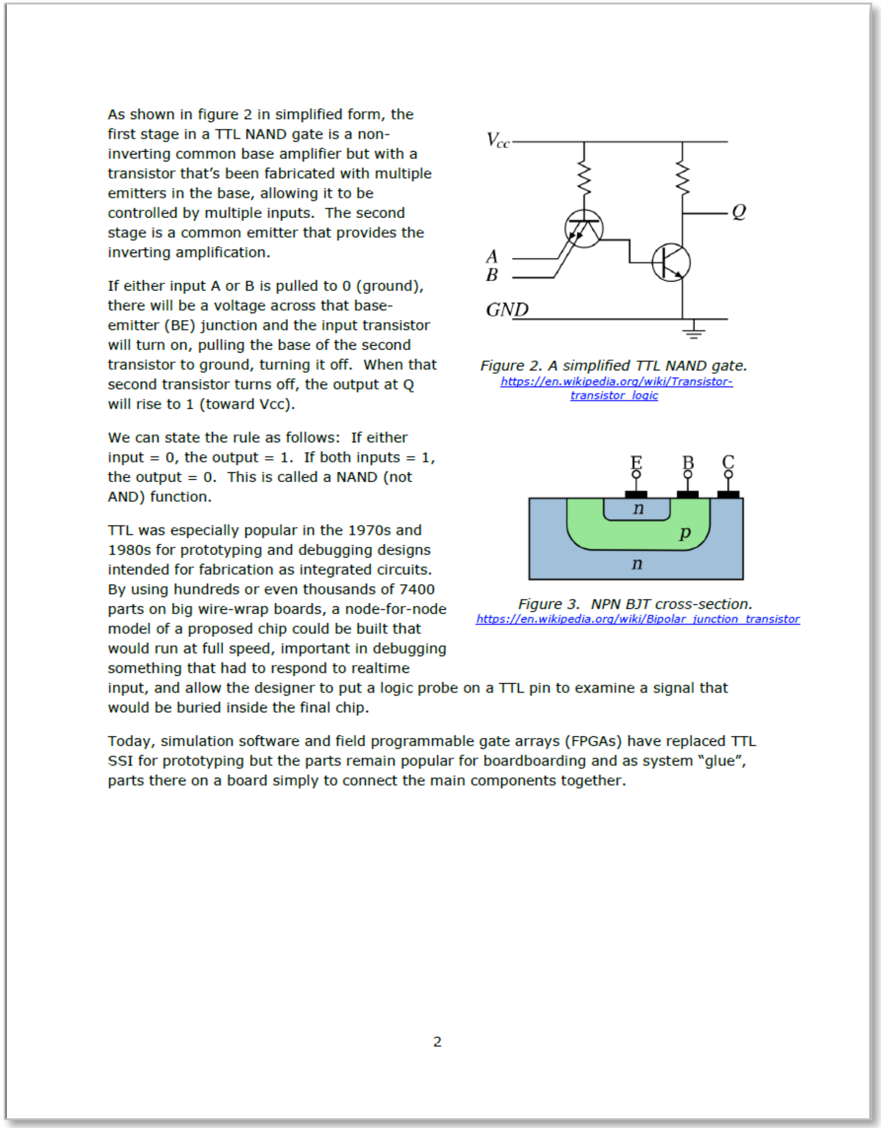
[1] This lab was written by Nicole Hamilton.

For many students, 271 is their first EE class and their first time in the lab. They're fascinated by all our wonderful and very expensive instruments and they want to know how electronic stuff works.

Learning objectives

1. Learn how to use our lab instruments.
2. Make a connection between Boolean algebra and the circuits we use to build things.
3. Learn how ones and zeros are represented as voltages in the TTL standard.

4. Brief explanation of how TTL circuits work.



As shown in figure 2 in simplified form, the first stage in a TTL NAND gate is a non-inverting common base amplifier but with a transistor that's been fabricated with multiple emitters in the base, allowing it to be controlled by multiple inputs. The second stage is a common emitter that provides the inverting amplification.

If either input A or B is pulled to 0 (ground), there will be a voltage across that base-emitter (BE) junction and the input transistor will turn on, pulling the base of the second transistor to ground, turning it off. When that second transistor turns off, the output at Q will rise to 1 (toward Vcc).

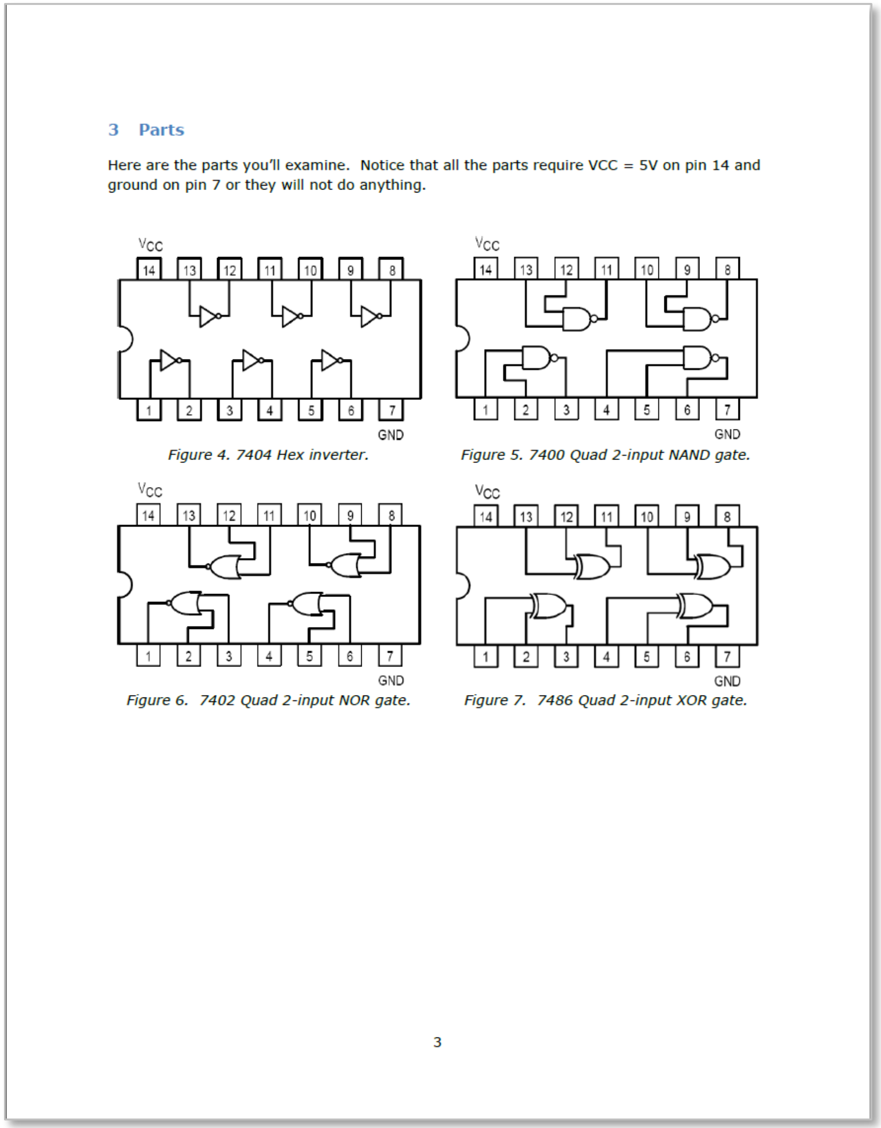
We can state the rule as follows: If either input = 0, the output = 1. If both inputs = 1, the output = 0. This is called a NAND (not AND) function.

Figure 2. A simplified TTL NAND gate.
https://en.wikipedia.org/wiki/Transistor-transistor_logic

Figure 3. NPN BJT cross-section.
https://en.wikipedia.org/wiki/Bipolar_junction_transistor

TTL was especially popular in the 1970s and 1980s for prototyping and debugging designs intended for fabrication as integrated circuits. By using hundreds or even thousands of 7400 parts on big wire-wrap boards, a node-for-node model of a proposed chip could be built that would run at full speed, important in debugging something that had to respond to realtime input, and allow the designer to put a logic probe on a TTL pin to examine a signal that would be buried inside the final chip.

Today, simulation software and field programmable gate arrays (FPGAs) have replaced TTL SSI for prototyping but the parts remain popular for boardboarding and as system "glue", parts there on a board simply to connect the main components together.



3 Parts

Here are the parts you'll examine. Notice that all the parts require VCC = 5V on pin 14 and ground on pin 7 or they will not do anything.

Figure 4. 7404 Hex inverter.

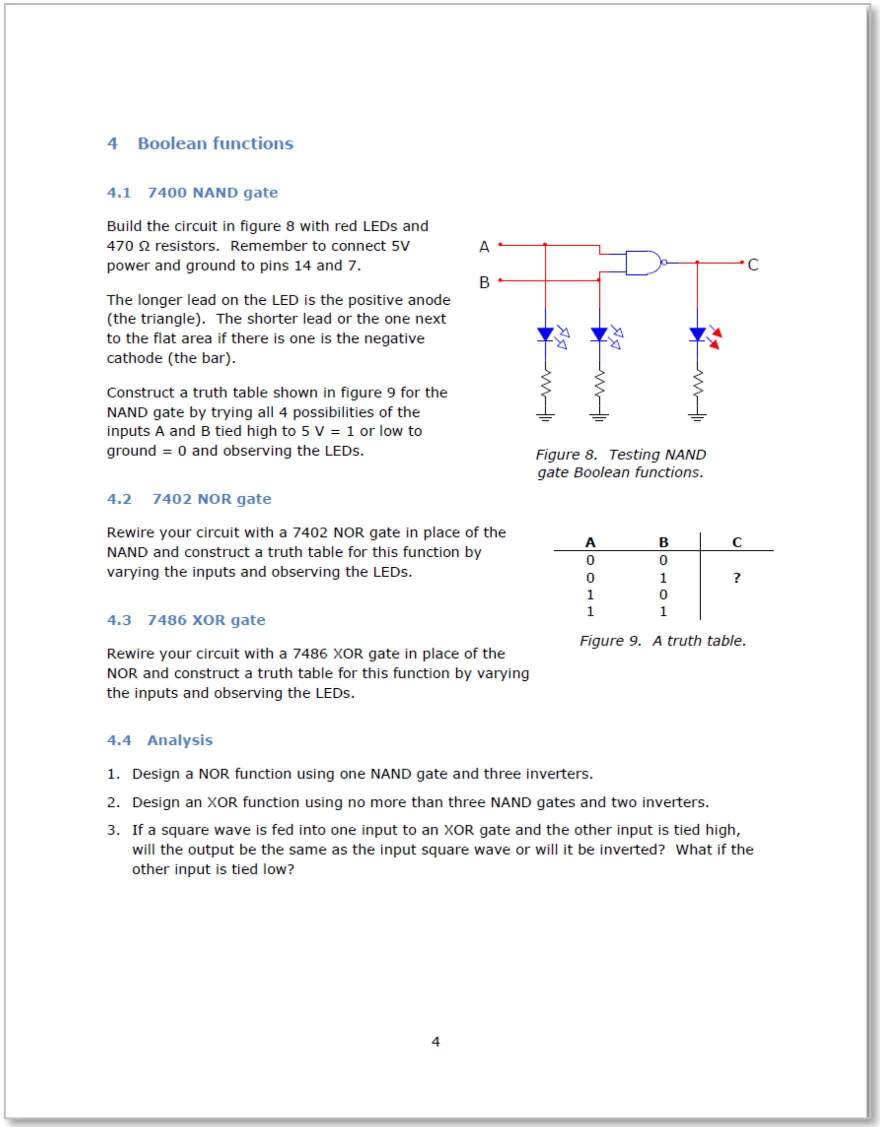
Figure 5. 7400 Quad 2-input NAND gate.

Figure 6. 7402 Quad 2-input NOR gate.

Figure 7. 7486 Quad 2-input XOR gate.

3

5. Breadboard and successfully debug some simple TTL circuits using inverters, NANDs, NORs and XORs.



4 Boolean functions

4.1 7400 NAND gate

Build the circuit in figure 8 with red LEDs and 470 Ω resistors. Remember to connect 5V power and ground to pins 14 and 7.

The longer lead on the LED is the positive anode (the triangle). The shorter lead or the one next to the flat area if there is one is the negative cathode (the bar).

Construct a truth table shown in figure 9 for the NAND gate by trying all 4 possibilities of the inputs A and B tied high to 5 V = 1 or low to ground = 0 and observing the LEDs.

Figure 8. Testing NAND gate Boolean functions.

4.2 7402 NOR gate

Rewire your circuit with a 7402 NOR gate in place of the NAND and construct a truth table for this function by varying the inputs and observing the LEDs.

A	B	C
0	0	
0	1	?
1	0	
1	1	

Figure 9. A truth table.

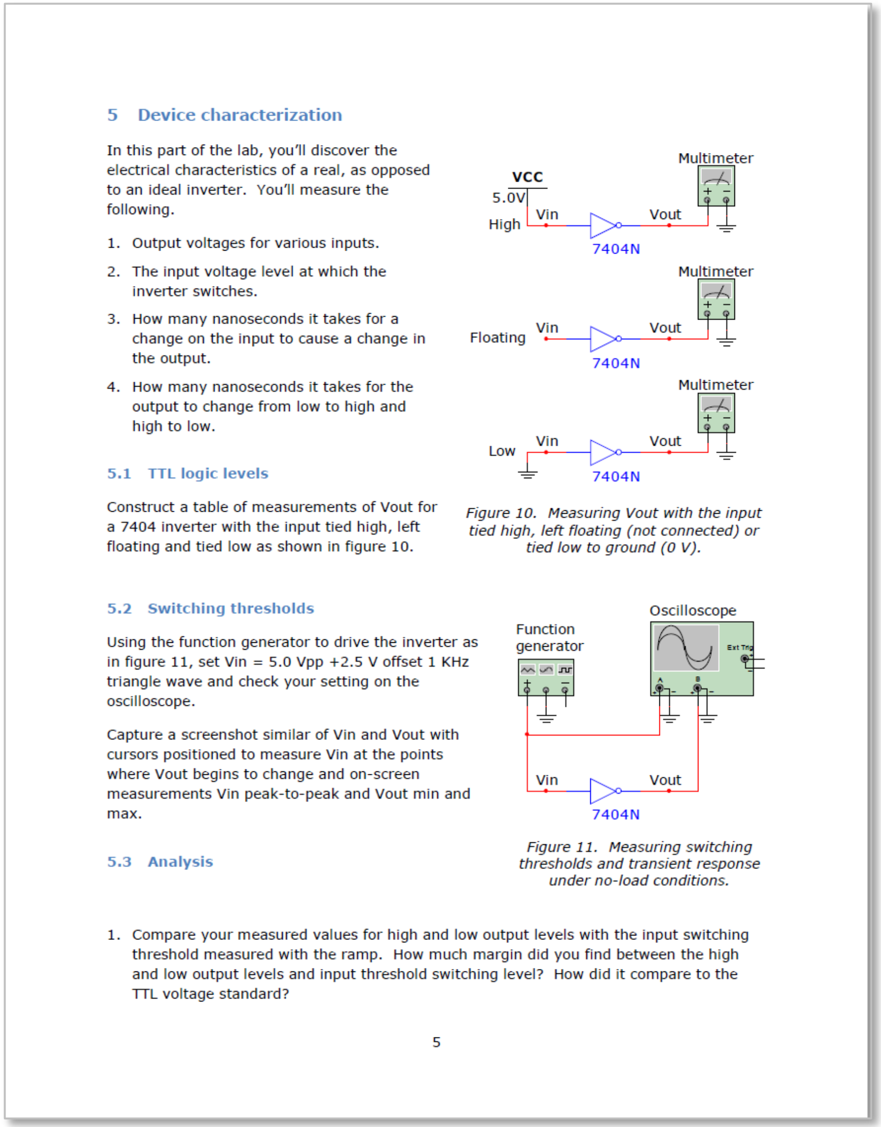
4.3 7486 XOR gate

Rewire your circuit with a 7486 XOR gate in place of the NOR and construct a truth table for this function by varying the inputs and observing the LEDs.

4.4 Analysis

1. Design a NOR function using one NAND gate and three inverters.
2. Design an XOR function using no more than three NAND gates and two inverters.
3. If a square wave is fed into one input to an XOR gate and the other input is tied high, will the output be the same as the input square wave or will it be inverted? What if the other input is tied low?

8. Be able to construct truth tables for NAND, NOR and XOR functions.



5 Device characterization

In this part of the lab, you'll discover the electrical characteristics of a real, as opposed to an ideal inverter. You'll measure the following.

1. Output voltages for various inputs.
2. The input voltage level at which the inverter switches.
3. How many nanoseconds it takes for a change on the input to cause a change in the output.
4. How many nanoseconds it takes for the output to change from low to high and high to low.

5.1 TTL logic levels

Construct a table of measurements of Vout for a 7404 inverter with the input tied high, left floating and tied low as shown in figure 10.

Figure 10. Measuring Vout with the input tied high, left floating (not connected) or tied low to ground (0 V).

5.2 Switching thresholds

Using the function generator to drive the inverter as in figure 11, set Vin = 5.0 Vpp +2.5 V offset 1 KHz triangle wave and check your setting on the oscilloscope.

Capture a screenshot similar of Vin and Vout with cursors positioned to measure Vin at the points where Vout begins to change and on-screen measurements Vin peak-to-peak and Vout min and max.

Figure 11. Measuring switching thresholds and transient response under no-load conditions.

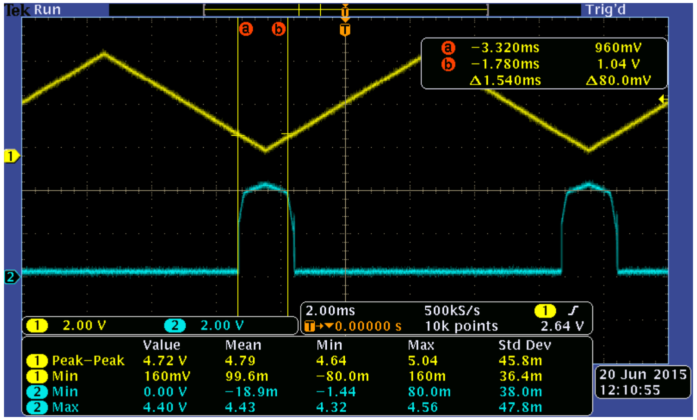
5.3 Analysis

1. Compare your measured values for high and low output levels with the input switching threshold measured with the ramp. How much margin did you find between the high and low output levels and input threshold switching level? How did it compare to the TTL voltage standard?

9. Measure the output levels associated with 1s and 0s and determine what a float looks like.

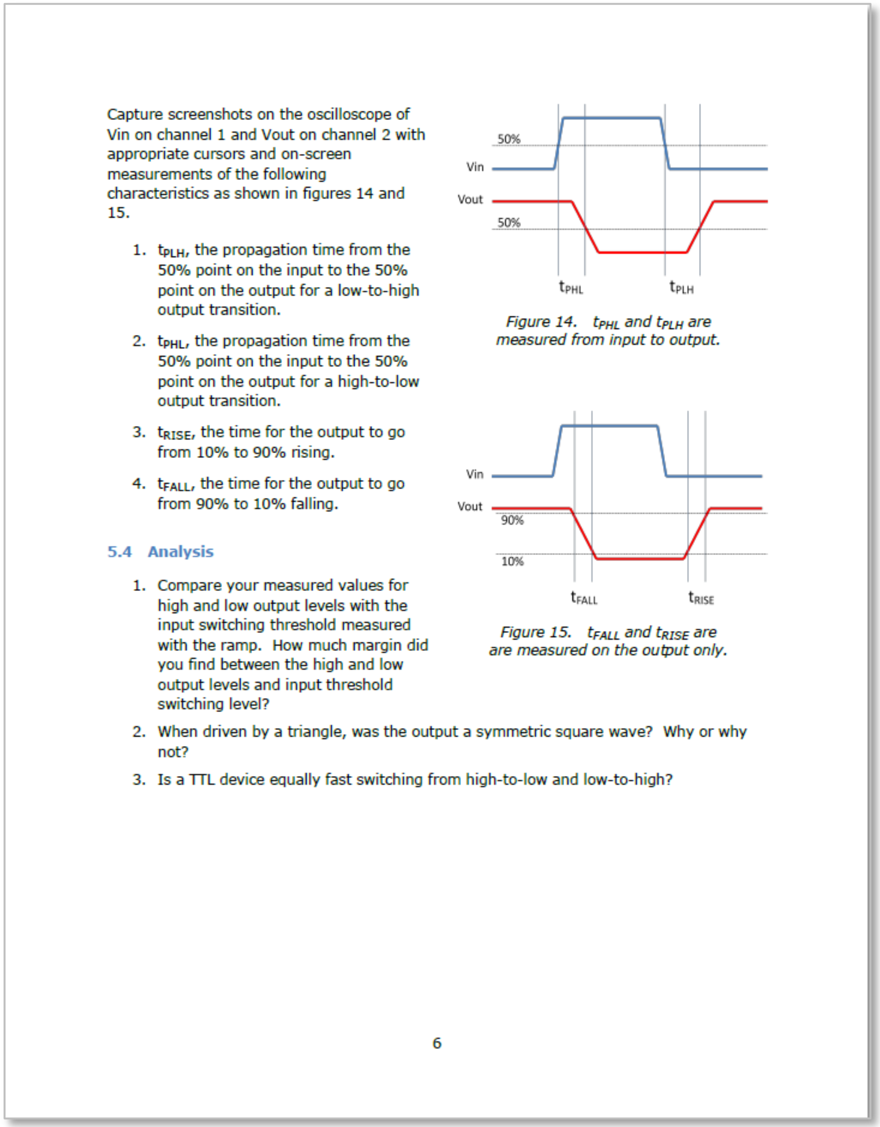
10. Measure the input switching thresholds rising and falling.

(We do this as a group.)



```
a  −3.320ms      960mV
b  −1.780ms      1.04 V
   Δ1.540ms      Δ80.0mV

1  2.00 V      2  2.00 V      2.00ms        500kS/s      1 ⨍
                              T→▼0.00000 s  10k points   2.64 V

                Value    Mean     Min      Max     Std Dev
1 Peak−Peak     4.72 V   4.79     4.64     5.04    45.8m
1 Min           160mV    99.6m    −80.0m   160m    36.4m
2 Min           0.00 V   −18.9m   −1.44    80.0m   38.0m
2 Max           4.40 V   4.43     4.32     4.56    47.8m

20 Jun 2015
12:10:55
```

13. Understand propagation delays and how to measure t_{PHL}, t_{PLH}, t_{RISE}, t_{FALL}.



Capture screenshots on the oscilloscope of Vin on channel 1 and Vout on channel 2 with appropriate cursors and on-screen measurements of the following characteristics as shown in figures 14 and 15.

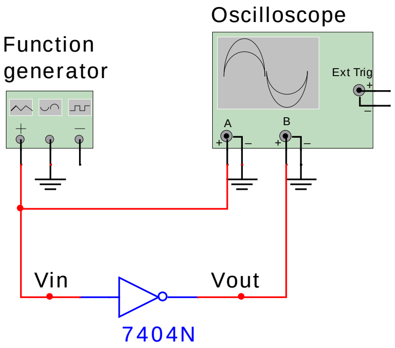
1. t_{PLH}, the propagation time from the 50% point on the input to the 50% point on the output for a low-to-high output transition.
2. t_{PHL}, the propagation time from the 50% point on the input to the 50% point on the output for a high-to-low output transition.
3. t_{RISE}, the time for the output to go from 10% to 90% rising.
4. t_{FALL}, the time for the output to go from 90% to 10% falling.

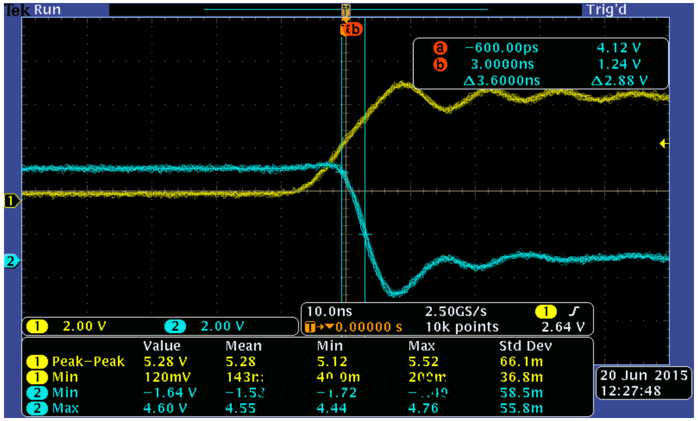
Figure 14. t_{PHL} and t_{PLH} are measured from input to output.

Figure 15. t_{FALL} and t_{RISE} are are measured on the output only.

5.4 Analysis

1. Compare your measured values for high and low output levels with the input switching threshold measured with the ramp. How much margin did you find between the high and low output levels and input threshold switching level?
2. When driven by a triangle, was the output a symmetric square wave? Why or why not?
3. Is a TTL device equally fast switching from high-to-low and low-to-high?





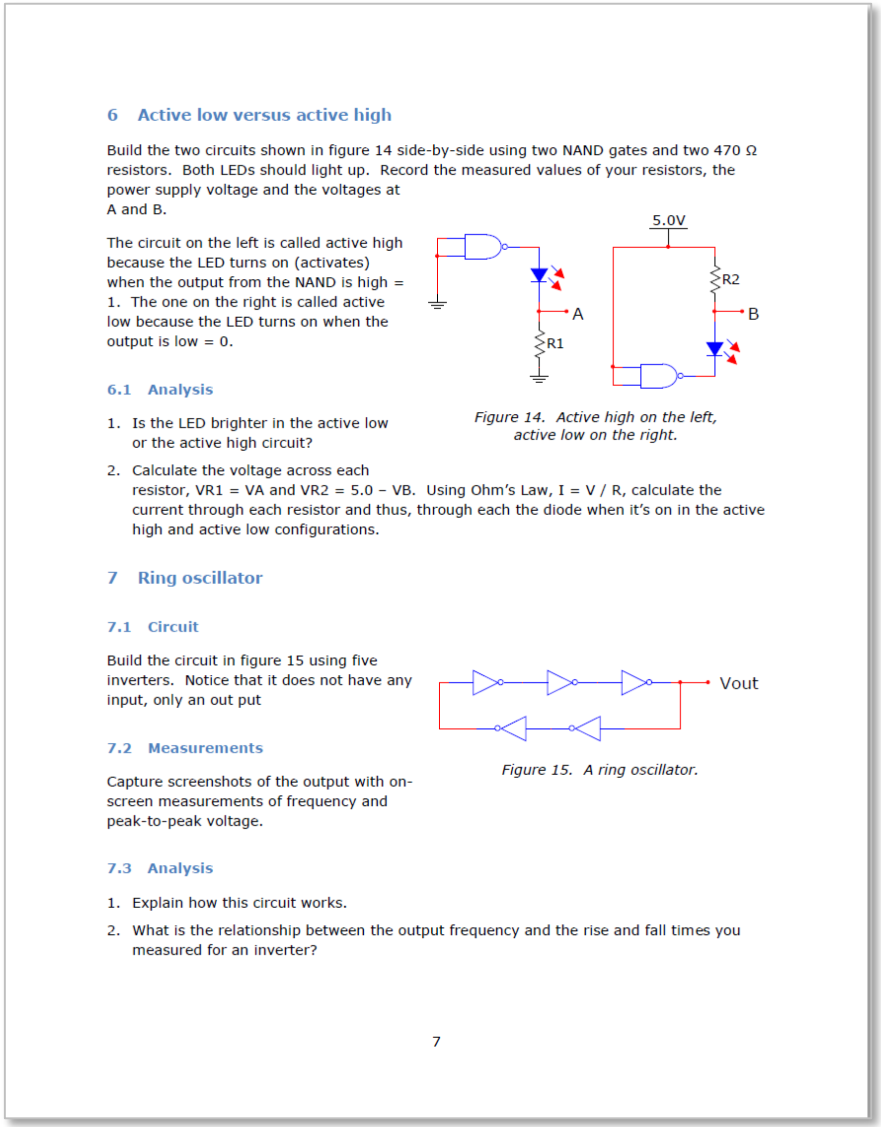
a −600.00ps 4.12 V
b 3.0000ns 1.24 V
Δ3.6000ns Δ2.88 V

1 2.00 V 2 2.00 V

10.0ns 2.50GS/s 1 ⨍
T→▼0.00000 s 10k points 2.64 V

	Value	Mean	Min	Max	Std Dev
1 Peak−Peak	5.28 V	5.28	5.12	5.52	66.1m
1 Min	120mV	143m	40.0m	200m	36.8m
2 Min	−1.64 V	−1.53	−1.72	−1.49	58.5m
2 Max	4.60 V	4.55	4.44	4.76	55.8m

20 Jun 2015
12:27:48



6 Active low versus active high

Build the two circuits shown in figure 14 side-by-side using two NAND gates and two 470 Ω resistors. Both LEDs should light up. Record the measured values of your resistors, the power supply voltage and the voltages at A and B.

The circuit on the left is called active high because the LED turns on (activates) when the output from the NAND is high = 1. The one on the right is called active low because the LED turns on when the output is low = 0.

Figure 14. Active high on the left, active low on the right.

6.1 Analysis

1. Is the LED brighter in the active low or the active high circuit?
2. Calculate the voltage across each resistor, VR1 = VA and VR2 = 5.0 – VB. Using Ohm's Law, I = V / R, calculate the current through each resistor and thus, through each the diode when it's on in the active high and active low configurations.

7 Ring oscillator

7.1 Circuit

Build the circuit in figure 15 using five inverters. Notice that it does not have any input, only an out put

Figure 15. A ring oscillator.

7.2 Measurements

Capture screenshots of the output with on-screen measurements of frequency and peak-to-peak voltage.

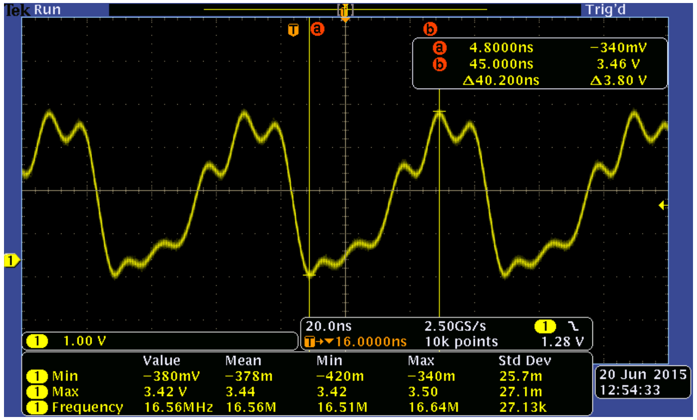
7.3 Analysis

1. Explain how this circuit works.
2. What is the relationship between the output frequency and the rise and fall times you measured for an inverter?

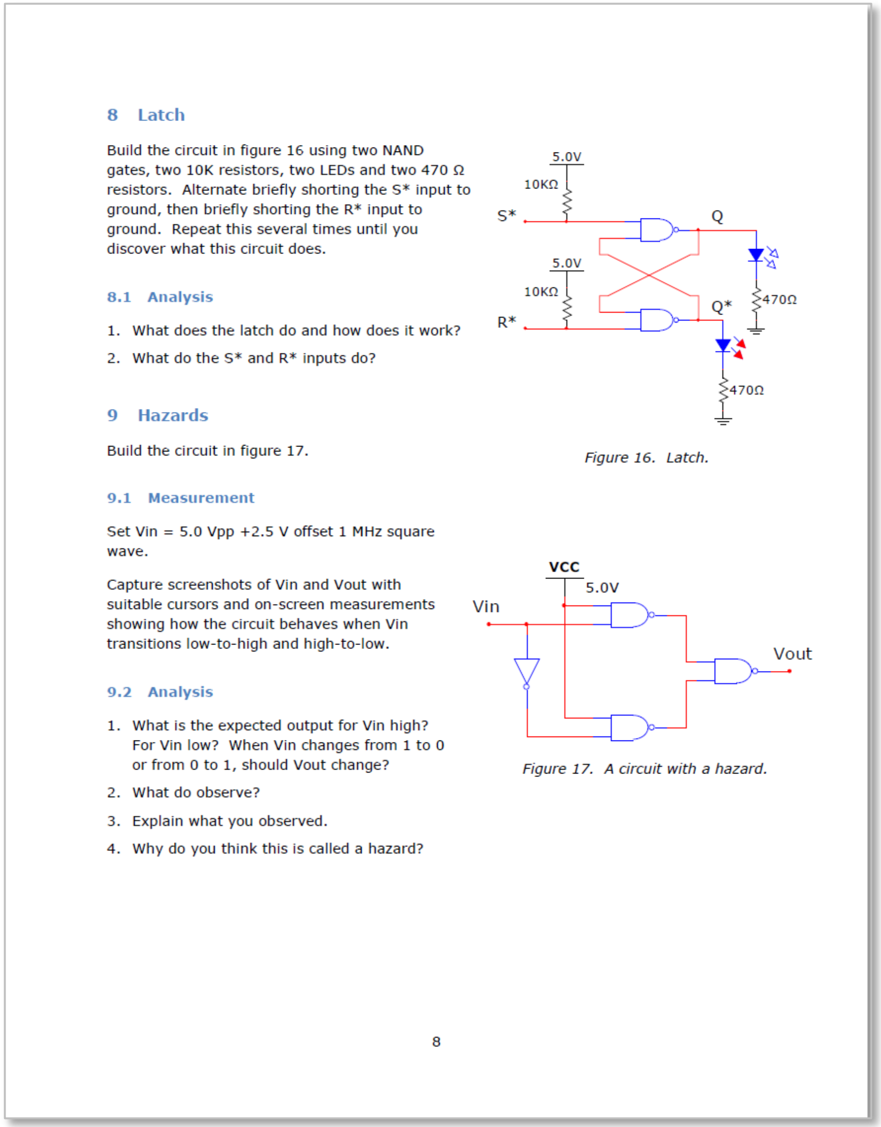
14. Learn the meaning of active low and active high.

    (The 7-segment displays they'll encounter in lab 2 are active low.)

15. Build a ring oscillator and relate the frequency to propagation delays.



```
a   4.8000ns     -340mV
b   45.000ns     3.46 V
    Δ40.200ns    Δ3.80 V

1  1.00 V        20.0ns          2.50GS/s      1 ↘
                 T→▼16.0000ns    10k points    1.28 V

               Value      Mean     Min      Max      Std Dev
1 Min          -380mV     -378m    -420m    -340m    25.7m
1 Max          3.42 V     3.44     3.42     3.50     27.1m
1 Frequency    16.56MHz   16.56M   16.51M   16.64M   27.13k

20 Jun 2015
12:54:33
```



8 Latch

Build the circuit in figure 16 using two NAND gates, two 10K resistors, two LEDs and two 470 Ω resistors. Alternate briefly shorting the S* input to ground, then briefly shorting the R* input to ground. Repeat this several times until you discover what this circuit does.

Figure 16. Latch.

8.1 Analysis

1. What does the latch do and how does it work?
2. What do the S* and R* inputs do?

9 Hazards

Build the circuit in figure 17.

9.1 Measurement

Set Vin = 5.0 Vpp +2.5 V offset 1 MHz square wave.

Capture screenshots of Vin and Vout with suitable cursors and on-screen measurements showing how the circuit behaves when Vin transitions low-to-high and high-to-low.

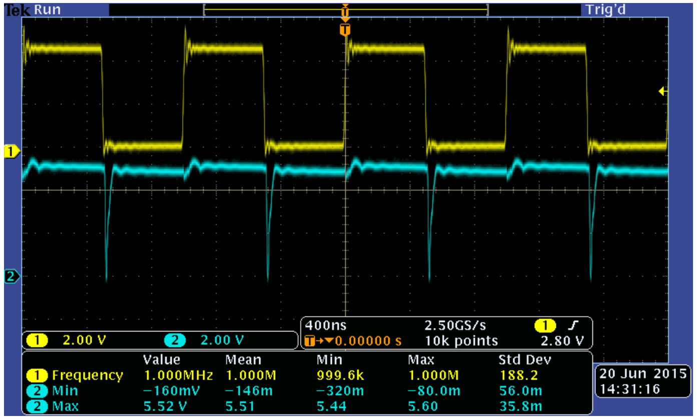
Figure 17. A circuit with a hazard.

9.2 Analysis

1. What is the expected output for Vin high? For Vin low? When Vin changes from 1 to 0 or from 0 to 1, should Vout change?
2. What do observe?
3. Explain what you observed.
4. Why do you think this is called a hazard?

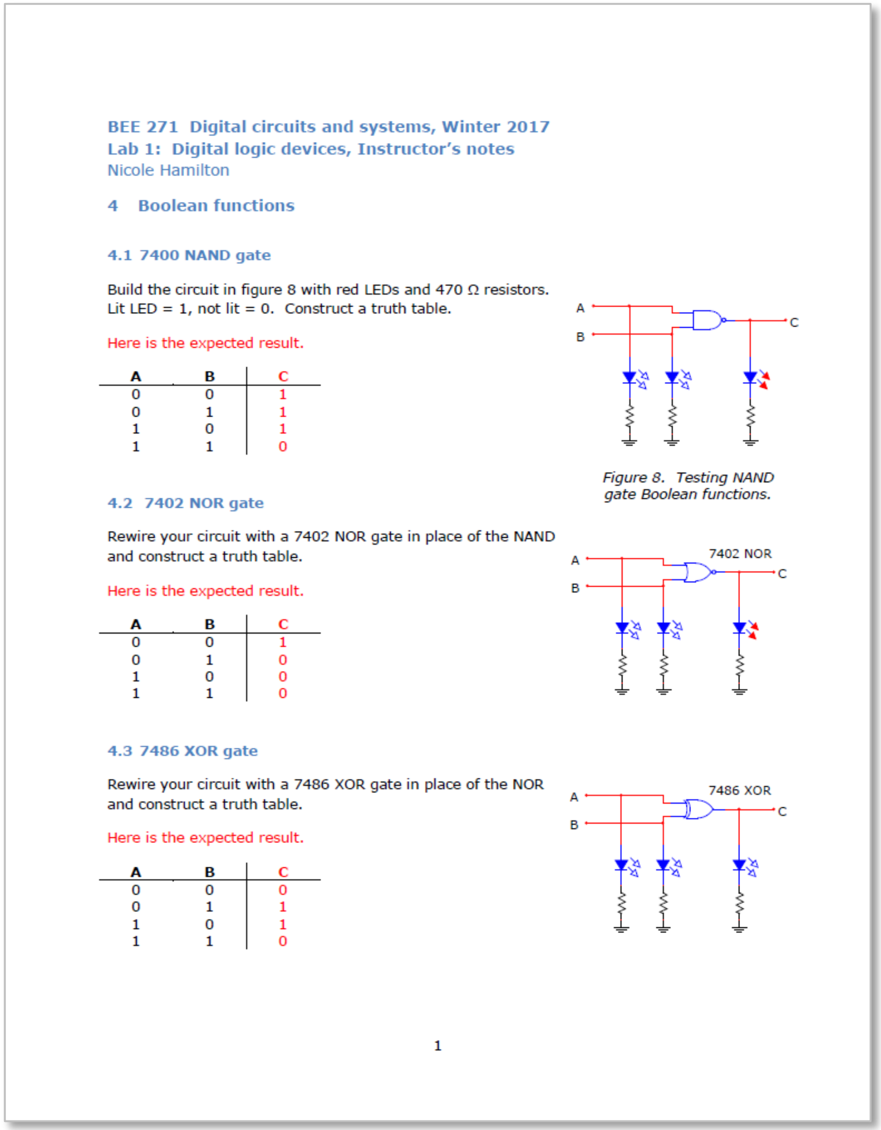
15. Learn how a latch works.

16. Challenge them to explain the behavior of a static 1 hazard.



```
1  2.00 V        2  2.00 V        400ns            2.50GS/s      1  ⨍
                                  T→▼0.00000 s     10k points    2.80 V
               Value      Mean      Min       Max       Std Dev
1 Frequency    1.000MHz   1.000M    999.6k    1.000M    188.2
2 Min          −160mV     −146m     −320m     −80.0m    56.0m
2 Max          5.52 V     5.51      5.44      5.60      35.8m
                                                                 20 Jun 2015
                                                                 14:31:16
```

Lab 1 Instructor's notes



BEE 271 Digital circuits and systems, Winter 2017
Lab 1: Digital logic devices, Instructor's notes
Nicole Hamilton

4 Boolean functions

4.1 7400 NAND gate

Build the circuit in figure 8 with red LEDs and 470 Ω resistors. Lit LED = 1, not lit = 0. Construct a truth table.

Here is the expected result.

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Figure 8. Testing NAND gate Boolean functions.

4.2 7402 NOR gate

Rewire your circuit with a 7402 NOR gate in place of the NAND and construct a truth table.

Here is the expected result.

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

4.3 7486 XOR gate

Rewire your circuit with a 7486 XOR gate in place of the NOR and construct a truth table.

Here is the expected result.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

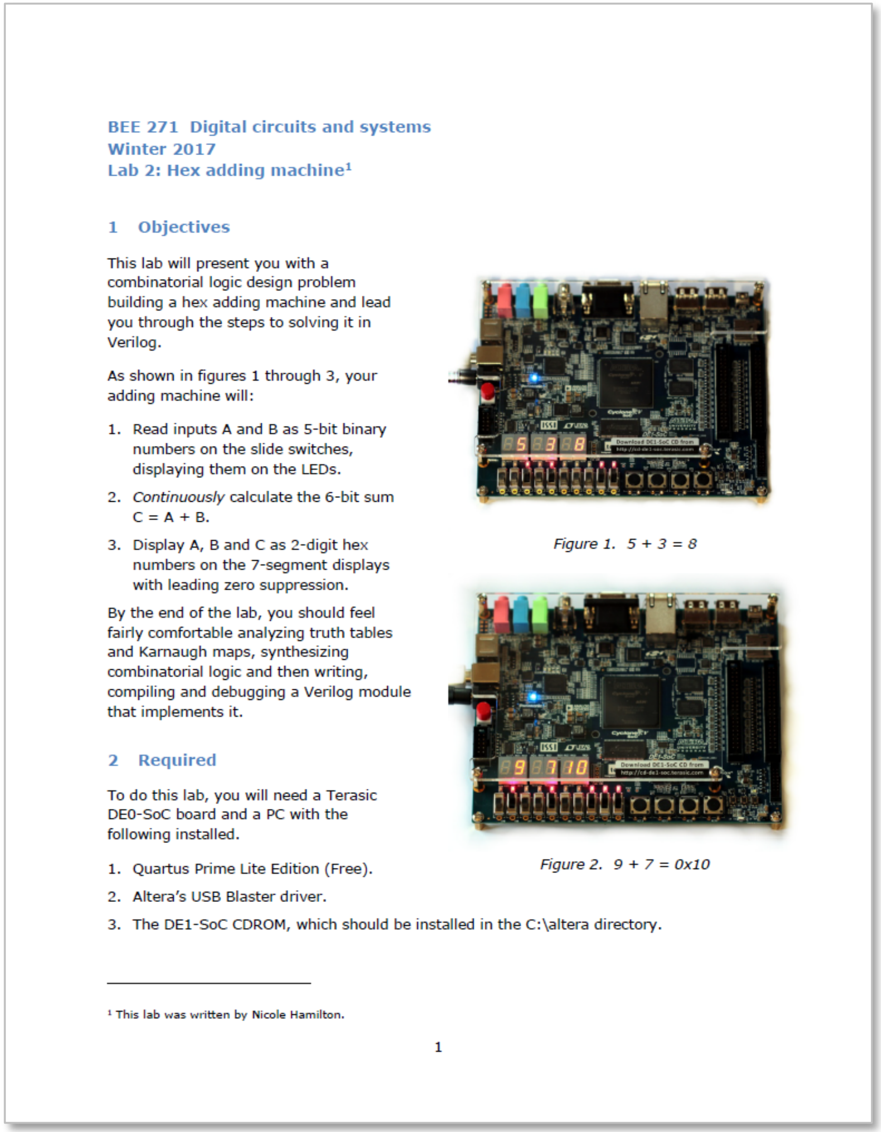
Example expected results, including screenshots and answers to the questions.

13 pages.

Lab 1 reflections

1. Students report on surveys that they like the breadboarding exercises.
2. Students are more unprepared to use the instruments, read a schematic or wire up a simple circuit on a breadboard than I'd have expected.
3. I'd have expected more to have played with electronics as a hobby or to have technician experience.

Lab 2 Hex adding machine



BEE 271 Digital circuits and systems
Winter 2017
Lab 2: Hex adding machine[1]

1 Objectives

This lab will present you with a combinatorial logic design problem building a hex adding machine and lead you through the steps to solving it in Verilog.

As shown in figures 1 through 3, your adding machine will:

1. Read inputs A and B as 5-bit binary numbers on the slide switches, displaying them on the LEDs.
2. *Continuously* calculate the 6-bit sum C = A + B.
3. Display A, B and C as 2-digit hex numbers on the 7-segment displays with leading zero suppression.

By the end of the lab, you should feel fairly comfortable analyzing truth tables and Karnaugh maps, synthesizing combinatorial logic and then writing, compiling and debugging a Verilog module that implements it.

Figure 1. 5 + 3 = 8

Figure 2. 9 + 7 = 0x10

2 Required

To do this lab, you will need a Terasic DE0-SoC board and a PC with the following installed.

1. Quartus Prime Lite Edition (Free).
2. Altera's USB Blaster driver.
3. The DE1-SoC CDROM, which should be installed in the C:\altera directory.

[1] This lab was written by Nicole Hamilton.

1

This is a purely combinatorial Verilog project in Quartus.

1. Two 5-bit numbers, A and B, are entered on the 10 switches.

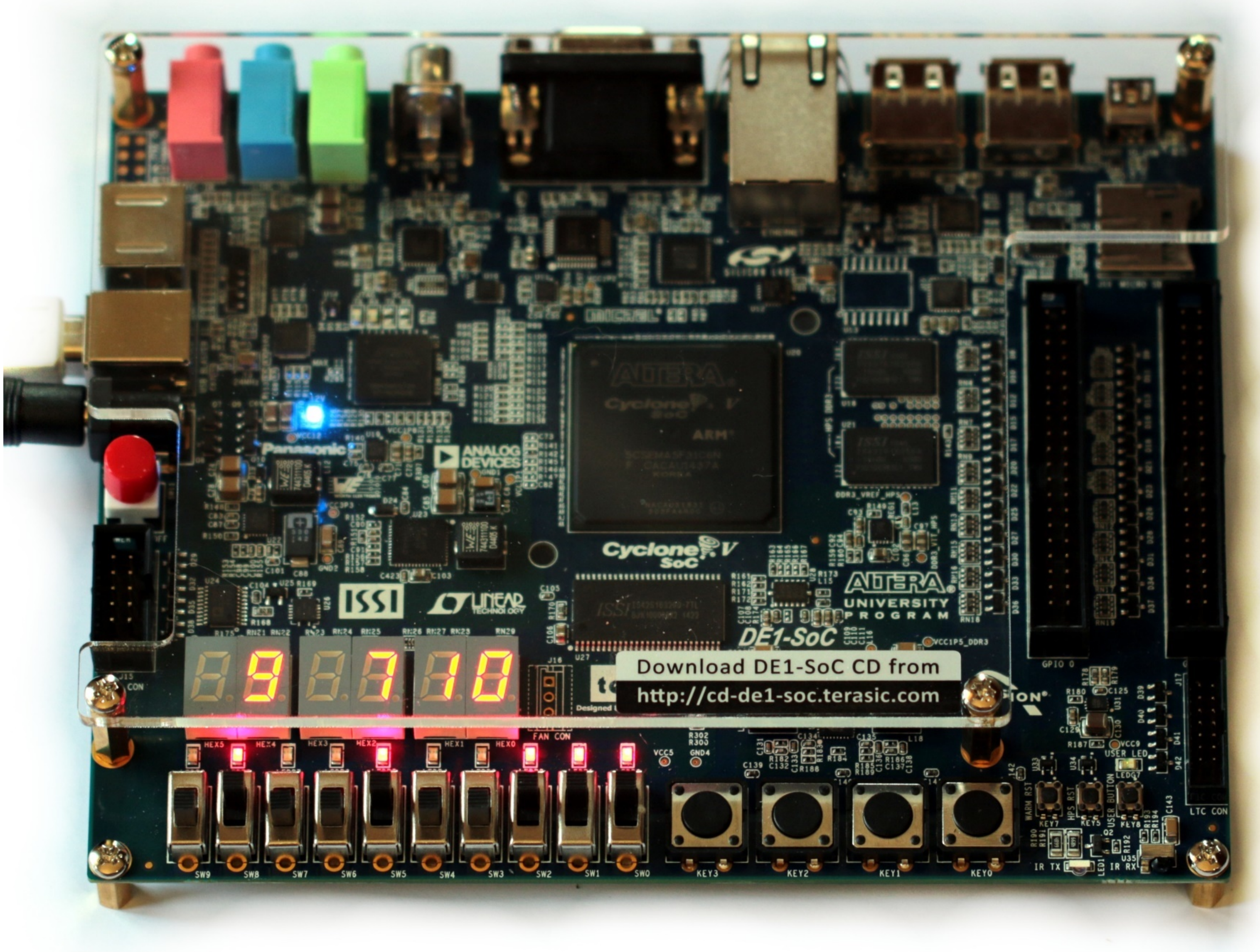
2. A, B and the 6-bit result C = A + B are displayed on pairs of seven segment displays with leading zero suppression.

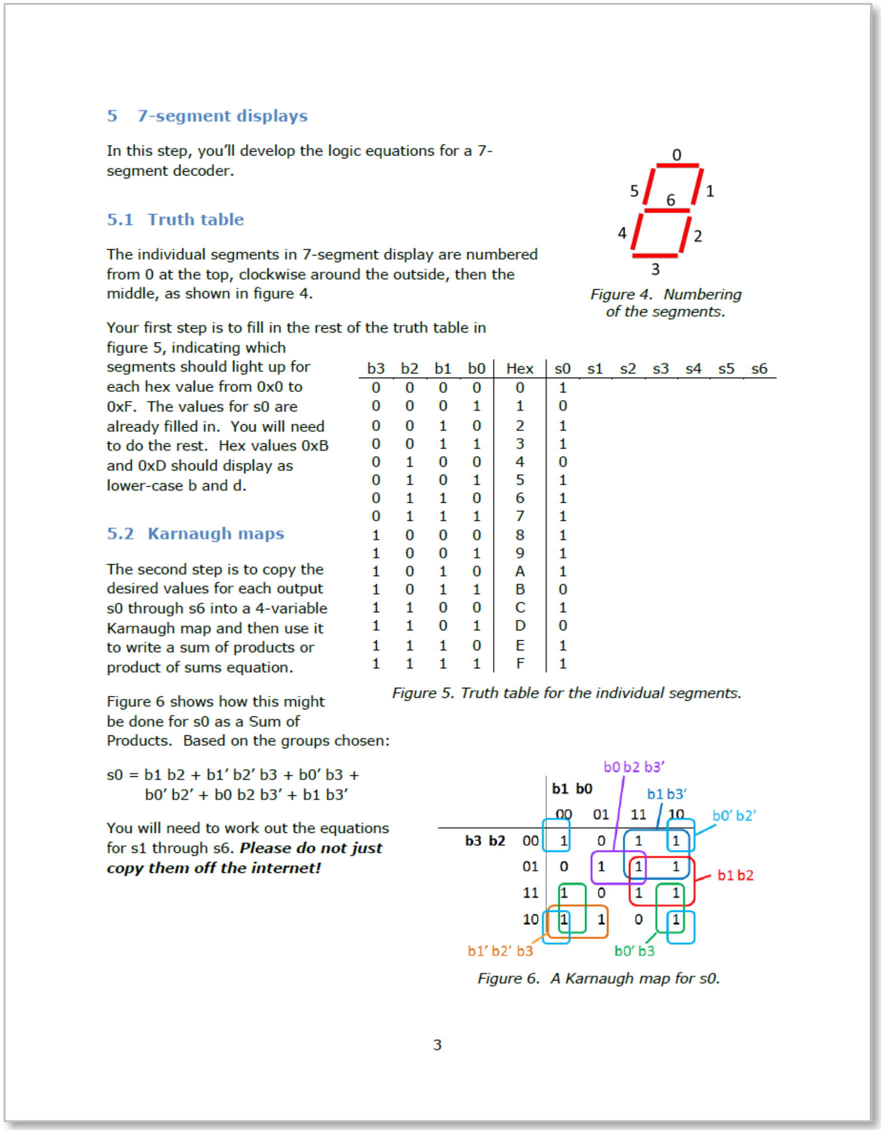
Learning objectives

1. Basic familiarity with binary numbers and hex notation.

2. Ability to create a simple Verilog project in Quartus, compile it and program the FPGA.

A close-up of what they build





5 7-segment displays

In this step, you'll develop the logic equations for a 7-segment decoder.

5.1 Truth table

The individual segments in 7-segment display are numbered from 0 at the top, clockwise around the outside, then the middle, as shown in figure 4.

Figure 4. Numbering of the segments.

Your first step is to fill in the rest of the truth table in figure 5, indicating which segments should light up for each hex value from 0x0 to 0xF. The values for s0 are already filled in. You will need to do the rest. Hex values 0xB and 0xD should display as lower-case b and d.

b3	b2	b1	b0	Hex	s0	s1	s2	s3	s4	s5	s6
0	0	0	0	0	1						
0	0	0	1	1	0						
0	0	1	0	2	1						
0	0	1	1	3	1						
0	1	0	0	4	0						
0	1	0	1	5	1						
0	1	1	0	6	1						
0	1	1	1	7	1						
1	0	0	0	8	1						
1	0	0	1	9	1						
1	0	1	0	A	1						
1	0	1	1	B	0						
1	1	0	0	C	1						
1	1	0	1	D	0						
1	1	1	0	E	1						
1	1	1	1	F	1						

Figure 5. Truth table for the individual segments.

5.2 Karnaugh maps

The second step is to copy the desired values for each output s0 through s6 into a 4-variable Karnaugh map and then use it to write a sum of products or product of sums equation.

Figure 6 shows how this might be done for s0 as a Sum of Products. Based on the groups chosen:

s0 = b1 b2 + b1' b2' b3 + b0' b3 +
b0' b2' + b0 b2 b3' + b1 b3'

You will need to work out the equations for s1 through s6. ***Please do not just copy them off the internet!***

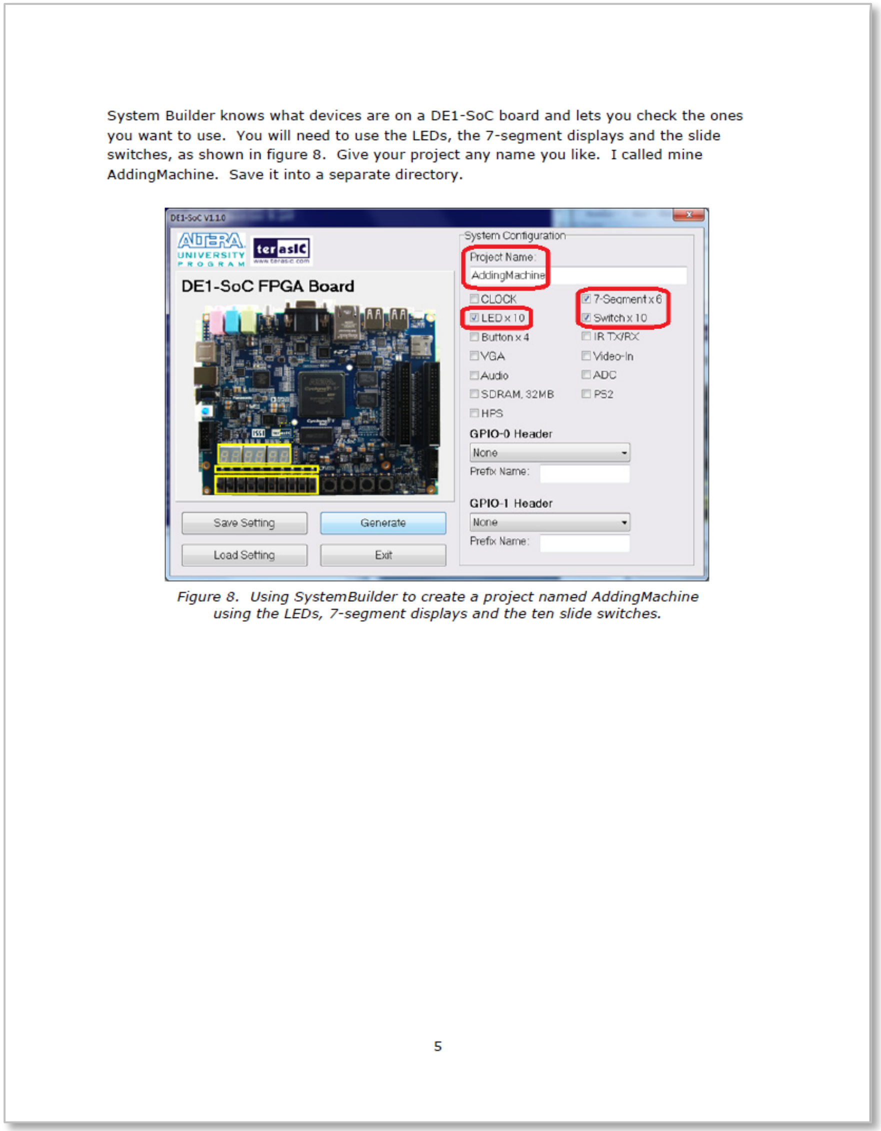
b3 b2	b1 b0 →	00	01	11	10
	00	1	0	1	1
	01	0	1	1	1
	11	1	0	1	1
	10	1	1	0	1

Figure 6. A Karnaugh map for s0.

3. Ability to create a truth table for a desired function.

4. Ability to use a Karnaugh map to create an SOP or POS solution.

(Mercifully, I'm skipping pages.)



System Builder knows what devices are on a DE1-SoC board and lets you check the ones you want to use. You will need to use the LEDs, the 7-segment displays and the slide switches, as shown in figure 8. Give your project any name you like. I called mine AddingMachine. Save it into a separate directory.

DE1-SoC V1.1.0

DE1-SoC FPGA Board

System Configuration

Project Name: AddingMachine

□ CLOCK	☑ 7-Segment x 6
☑ LED x 10	☑ Switch x 10
□ Button x 4	□ IR TX/RX
□ VGA	□ Video-In
□ Audio	□ ADC
□ SDRAM, 32MB	□ PS2
□ HPS	

GPIO-0 Header: None

Prefix Name:

GPIO-1 Header: None

Prefix Name:

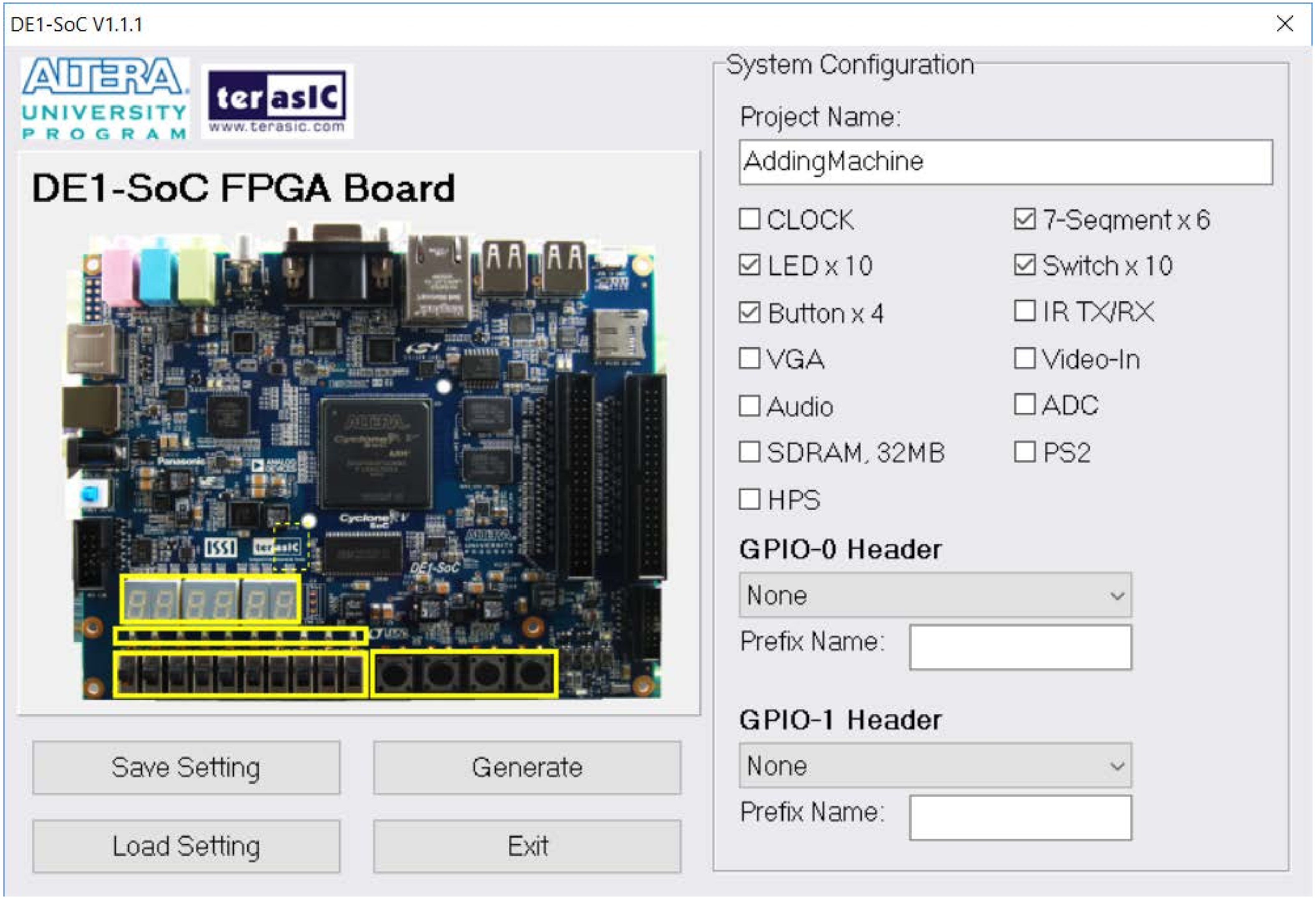
Figure 8. Using SystemBuilder to create a project named AddingMachine using the LEDs, 7-segment displays and the ten slide switches.

5

5. Use the SystemBuilder tool to create a template project.

(We do this as a group in the lab.)

The DE1-SoC SystemBuilder utility



DE1-SoC V1.1.1

System Configuration

Project Name: AddingMachine

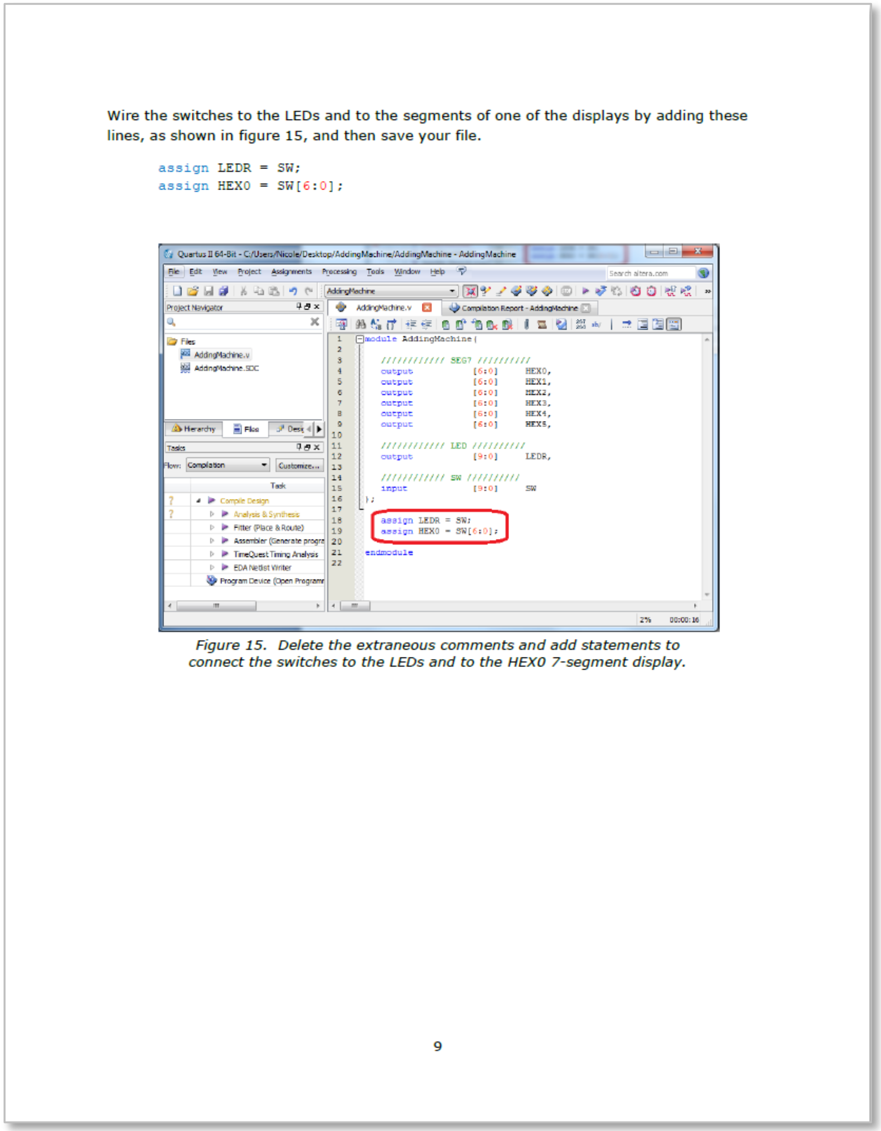
□ CLOCK
☑ LED x 10
☑ Button x 4
□ VGA
□ Audio
□ SDRAM, 32MB
□ HPS
☑ 7-Segment x 6
☑ Switch x 10
□ IR TX/RX
□ Video-In
□ ADC
□ PS2

GPIO-0 Header: None

Prefix Name:

GPIO-1 Header: None

Prefix Name:



Wire the switches to the LEDs and to the segments of one of the displays by adding these lines, as shown in figure 15, and then save your file.

```
assign LEDR = SW;
assign HEX0 = SW[6:0];
```

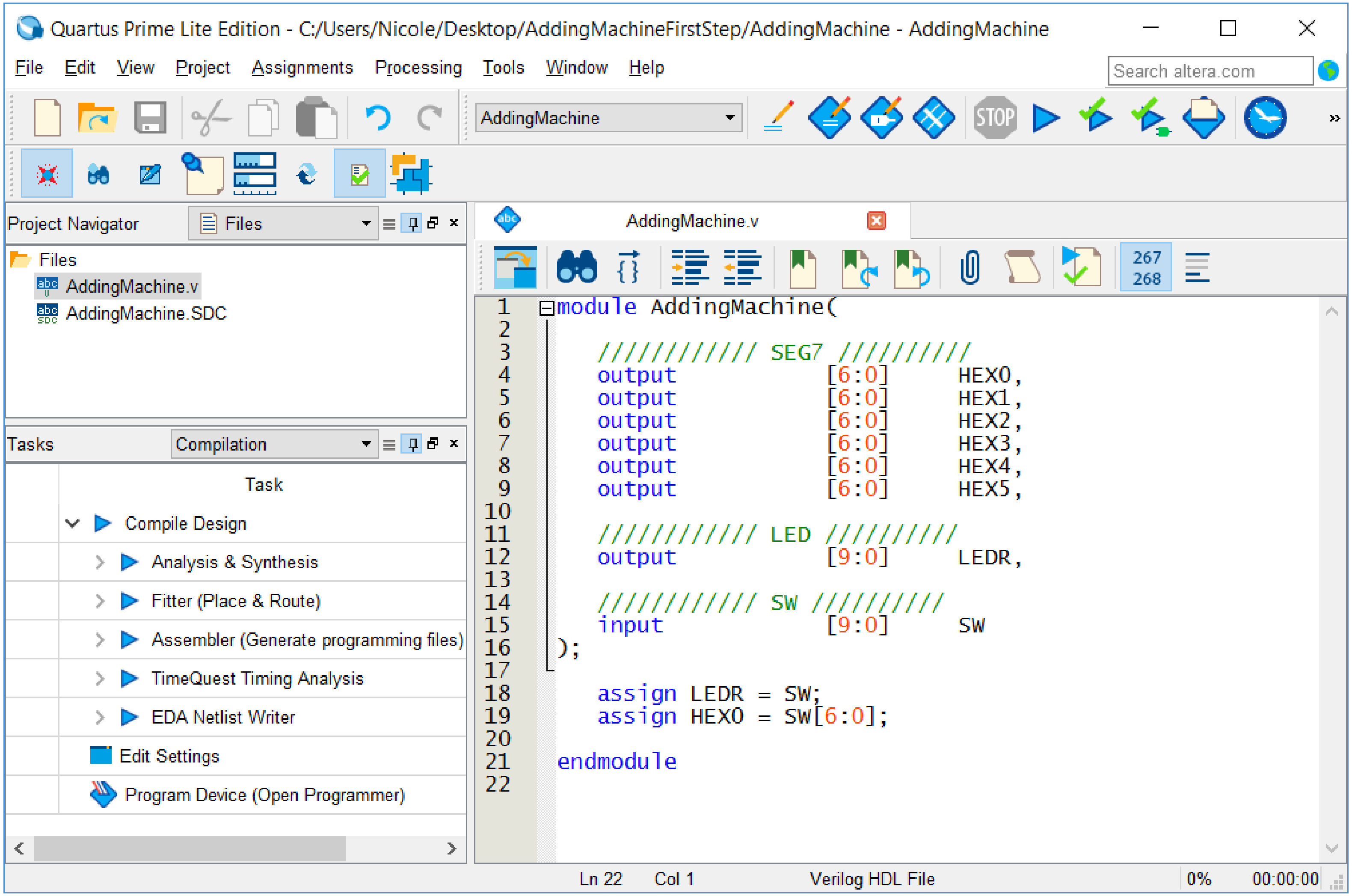
```
module AddingMachine(

    ///////////// SEG7 //////////
    output          [6:0]       HEX0,
    output          [6:0]       HEX1,
    output          [6:0]       HEX2,
    output          [6:0]       HEX3,
    output          [6:0]       HEX4,
    output          [6:0]       HEX5,

    ///////////// LED //////////
    output          [9:0]       LEDR,

    ///////////// SW //////////
    input           [9:0]       SW
);

    assign LEDR = SW;
    assign HEX0 = SW[6:0];

endmodule
```

Quartus II 64-Bit - C:/Users/Nicole/Desktop/AddingMachine/AddingMachine - AddingMachine
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
AddingMachine
Project Navigator
Files
AddingMachine.v
AddingMachine.SDC
Hierarchy
Files
Desi
Tasks
Flow: Compilation
Customize...
Task
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate progra
TimeQuest Timing Analysis
EDA Netlist Writer
Program Device (Open Programm
AddingMachine.v
Compilation Report - AddingMachine
2%
00:00:16

Figure 15. Delete the extraneous comments and add statements to connect the switches to the LEDs and to the HEX0 7-segment display.

6. Open a project in Quartus and add the Verilog file.
7. Wire the switches to the LEDs and to one of the 7-segment displays.

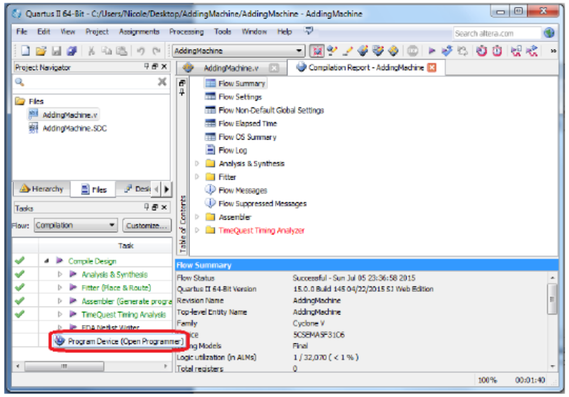
Switches wired to the LEDs



Quartus Prime Lite Edition - C:/Users/Nicole/Desktop/AddingMachineFirstStep/AddingMachine - AddingMachine
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
AddingMachine
Project Navigator
Files
Files
AddingMachine.v
AddingMachine.SDC
Tasks
Compilation
Task
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
TimeQuest Timing Analysis
EDA Netlist Writer
Edit Settings
Program Device (Open Programmer)
AddingMachine.v
267
268
Ln 22
Col 1
Verilog HDL File
0%
00:00:00

```verilog
module AddingMachine(

    //////////// SEG7 //////////
    output             [6:0]      HEX0,
    output             [6:0]      HEX1,
    output             [6:0]      HEX2,
    output             [6:0]      HEX3,
    output             [6:0]      HEX4,
    output             [6:0]      HEX5,

    //////////// LED //////////
    output             [9:0]      LEDR,

    //////////// SW //////////
    input              [9:0]      SW
);

    assign LEDR = SW;
    assign HEX0 = SW[6:0];

endmodule

```

8. Successfully compile their code and program the FPGA.

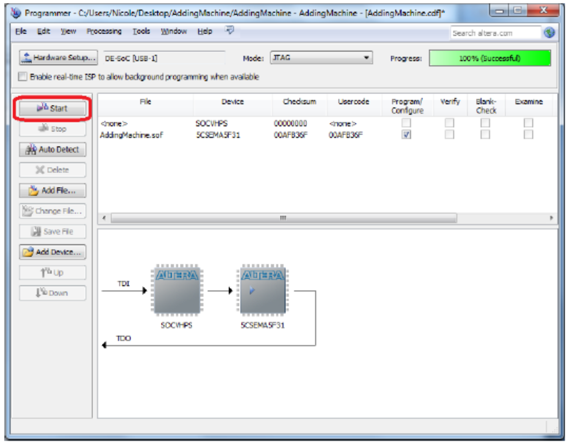

Quartus II 64-Bit - C:/Users/Nicole/Desktop/AddingMachine/AddingMachine - AddingMachine
File Edit View Project Assignments Processing Tools Window Help
Project Navigator
Files
AddingMachine.v
AddingMachine.SDC
Compilation Report - AddingMachine
Flow Summary
Flow Settings
Flow Non-Default Global Settings
Flow Elapsed Time
Flow OS Summary
Flow Log
Analysis & Synthesis
Fitter
Flow Messages
Flow Suppressed Messages
Assembler
TimeQuest Timing Analyzer
Tasks
Flow: Compilation
Customize...
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate progra
TimeQuest Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)

Flow Summary

Flow Status	Successful - Sun Jul 05 23:36:58 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	AddingMachine
Top-level Entity Name	AddingMachine
Family	Cyclone V
ice	5CSEMA5F31C6
ng Models	Final
Logic utilization (in ALMs)	1 / 32,070 (< 1 %)
Total registers	0

100% 00:01:40

Figure 18. After successful compile, program the device.


Programmer - C:/Users/Nicole/Desktop/AddingMachine/AddingMachine - AddingMachine - [AddingMachine.cdf]*
File Edit View Processing Tools Window Help
Search altera.com
Hardware Setup...
Start
Stop
Auto Detect
Delete
Add File...
Change File...
Save File
Add Device...
Up
Down
TDI
SOCVHPS
5CSEMA5F31
TDO

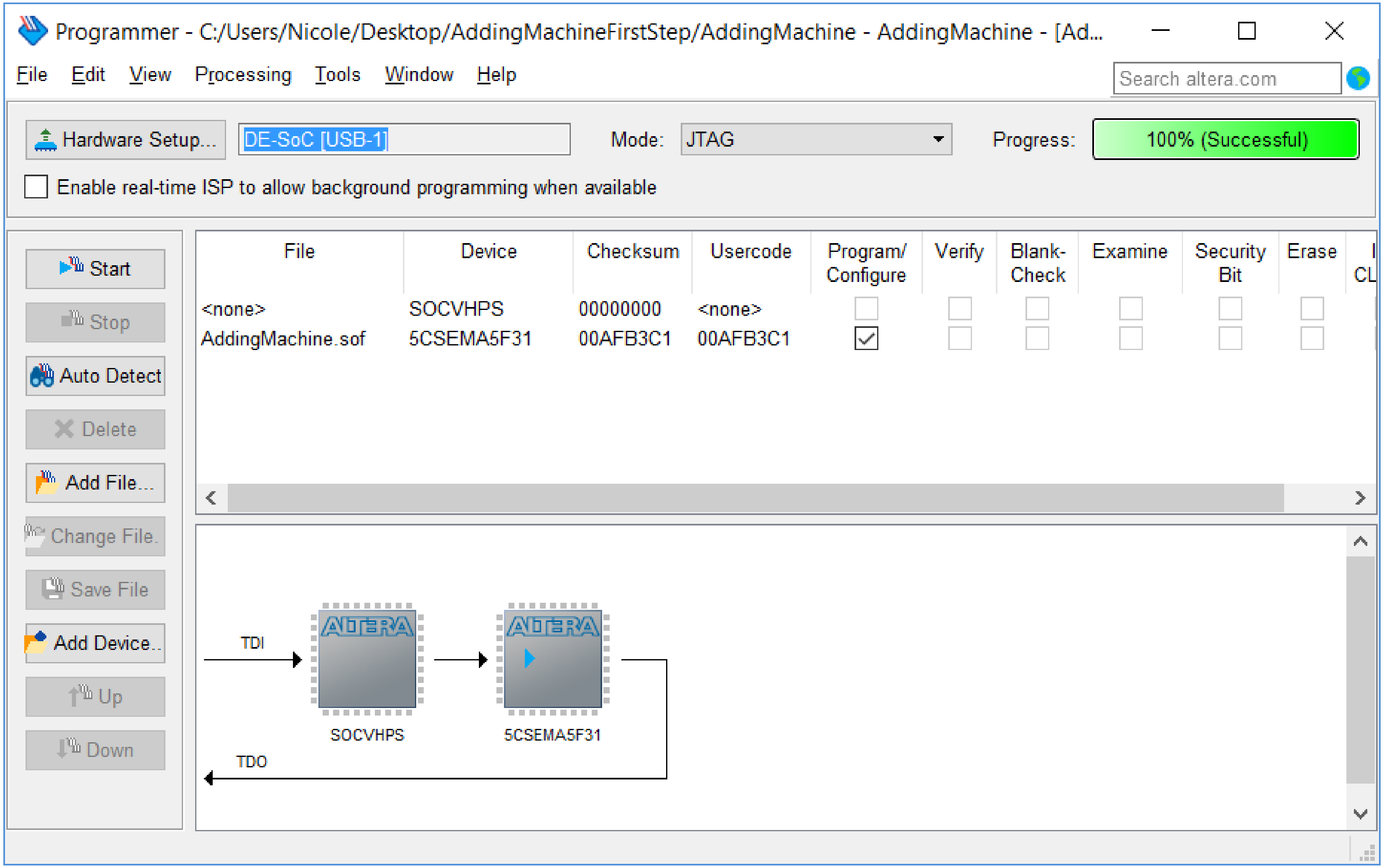
DE-SoC [USB-1] Mode: JTAG Progress: 100% (Successful)

□ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
<none>	SOCVHPS	00000000	<none>	□	□	□	□
AddingMachine.sof	5CSEMA5F31	00AFB36F	00AFB36F	☑	□	□	□

Figure 18. Device programmer.

We get to here as a group



Programmer - C:/Users/Nicole/Desktop/AddingMachineFirstStep/AddingMachine - AddingMachine - [Ad...

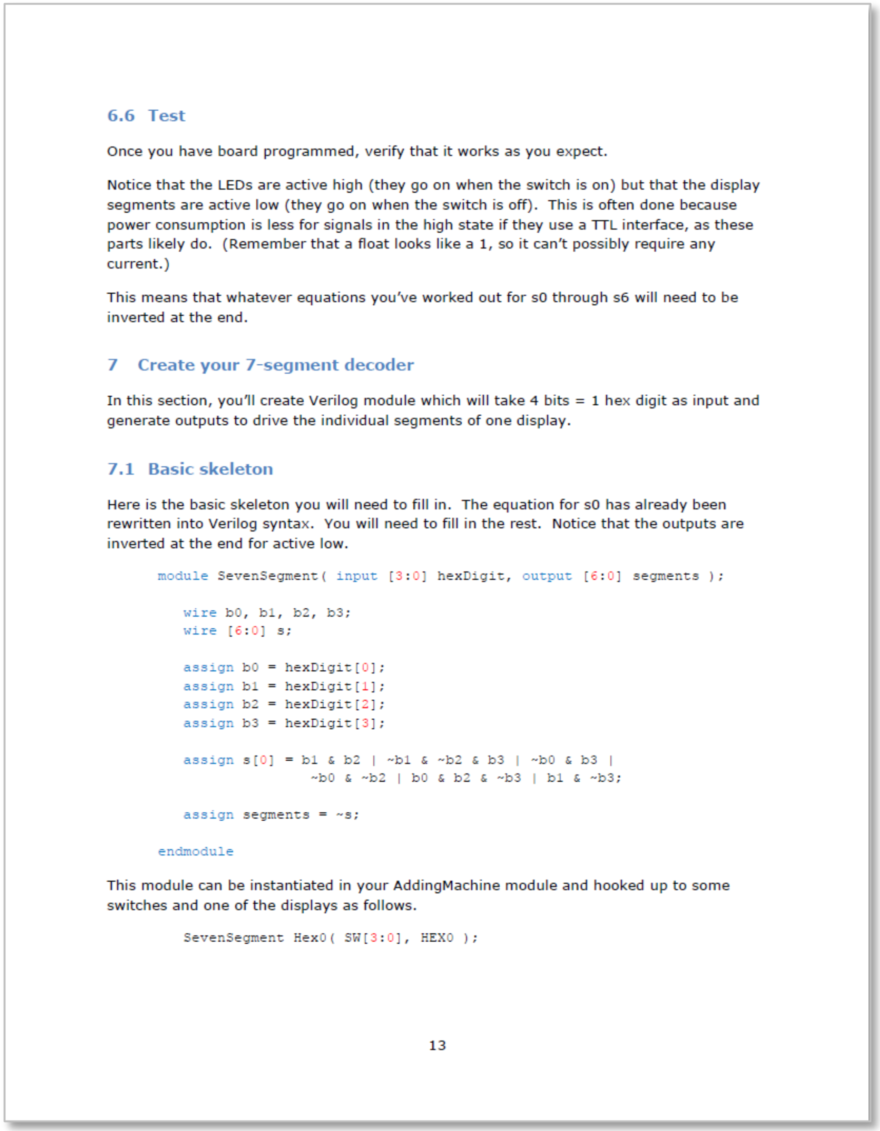
DE-SoC [USB-1]

Mode: JTAG

Progress: 100% (Successful)

□ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase	I CL
<none>	SOCVHPS	00000000	<none>	□	□	□	□	□	□	
AddingMachine.sof	5CSEMA5F31	00AFB3C1	00AFB3C1	☑	□	□	□	□	□	



6.6 Test

Once you have board programmed, verify that it works as you expect.

Notice that the LEDs are active high (they go on when the switch is on) but that the display segments are active low (they go on when the switch is off). This is often done because power consumption is less for signals in the high state if they use a TTL interface, as these parts likely do. (Remember that a float looks like a 1, so it can't possibly require any current.)

This means that whatever equations you've worked out for s0 through s6 will need to be inverted at the end.

7 Create your 7-segment decoder

In this section, you'll create Verilog module which will take 4 bits = 1 hex digit as input and generate outputs to drive the individual segments of one display.

7.1 Basic skeleton

Here is the basic skeleton you will need to fill in. The equation for s0 has already been rewritten into Verilog syntax. You will need to fill in the rest. Notice that the outputs are inverted at the end for active low.

```
module SevenSegment( input [3:0] hexDigit, output [6:0] segments );

   wire b0, b1, b2, b3;
   wire [6:0] s;

   assign b0 = hexDigit[0];
   assign b1 = hexDigit[1];
   assign b2 = hexDigit[2];
   assign b3 = hexDigit[3];

   assign s[0] = b1 & b2 | ~b1 & ~b2 & b3 | ~b0 & b3 |
                 ~b0 & ~b2 | b0 & b2 & ~b3 | b1 & ~b3;

   assign segments = ~s;

endmodule
```

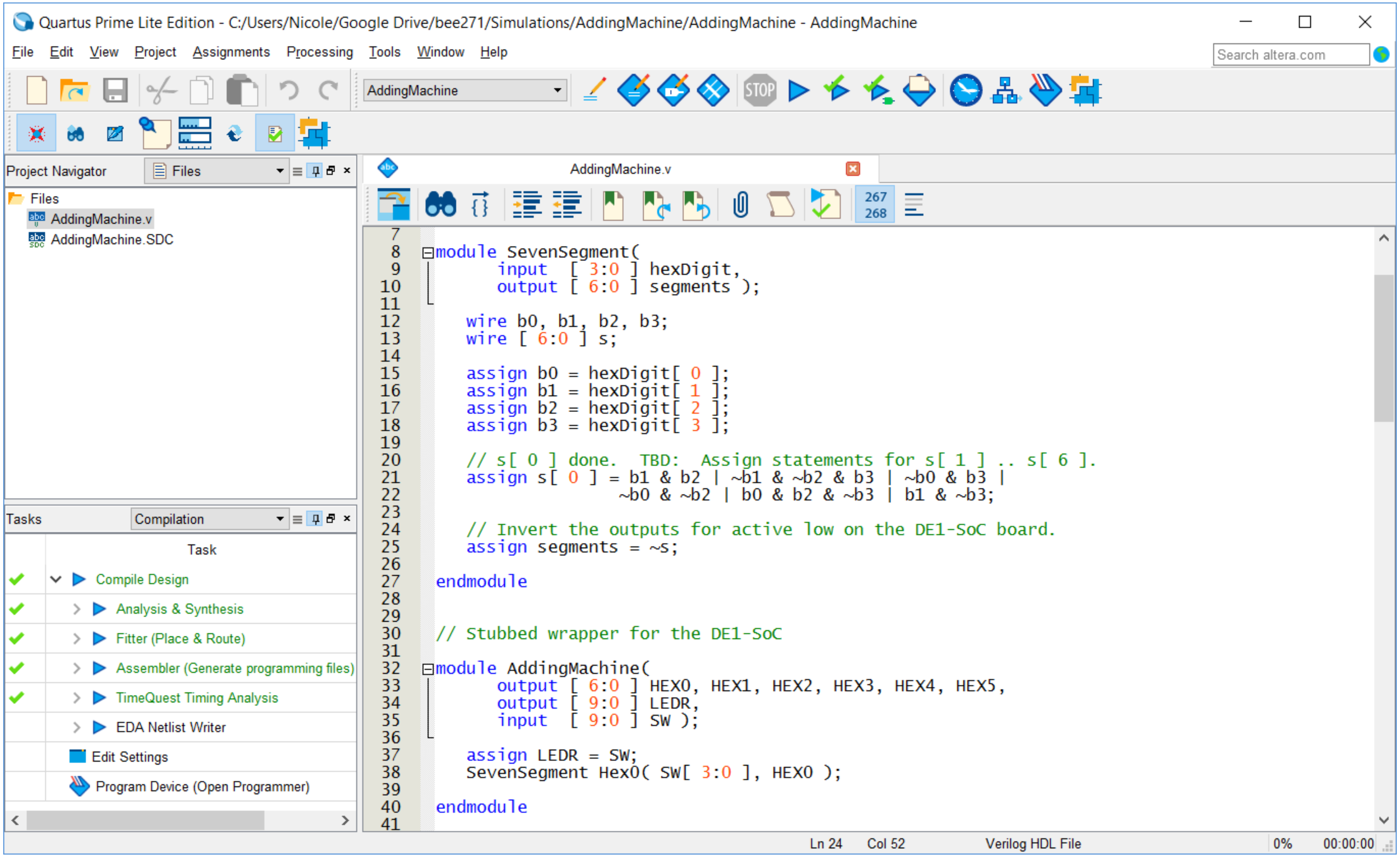
This module can be instantiated in your AddingMachine module and hooked up to some switches and one of the displays as follows.

```
   SevenSegment Hex0( SW[3:0], HEX0 );
```

9. Add the basic skeleton for the seven segment decoder and instantiate one copy.

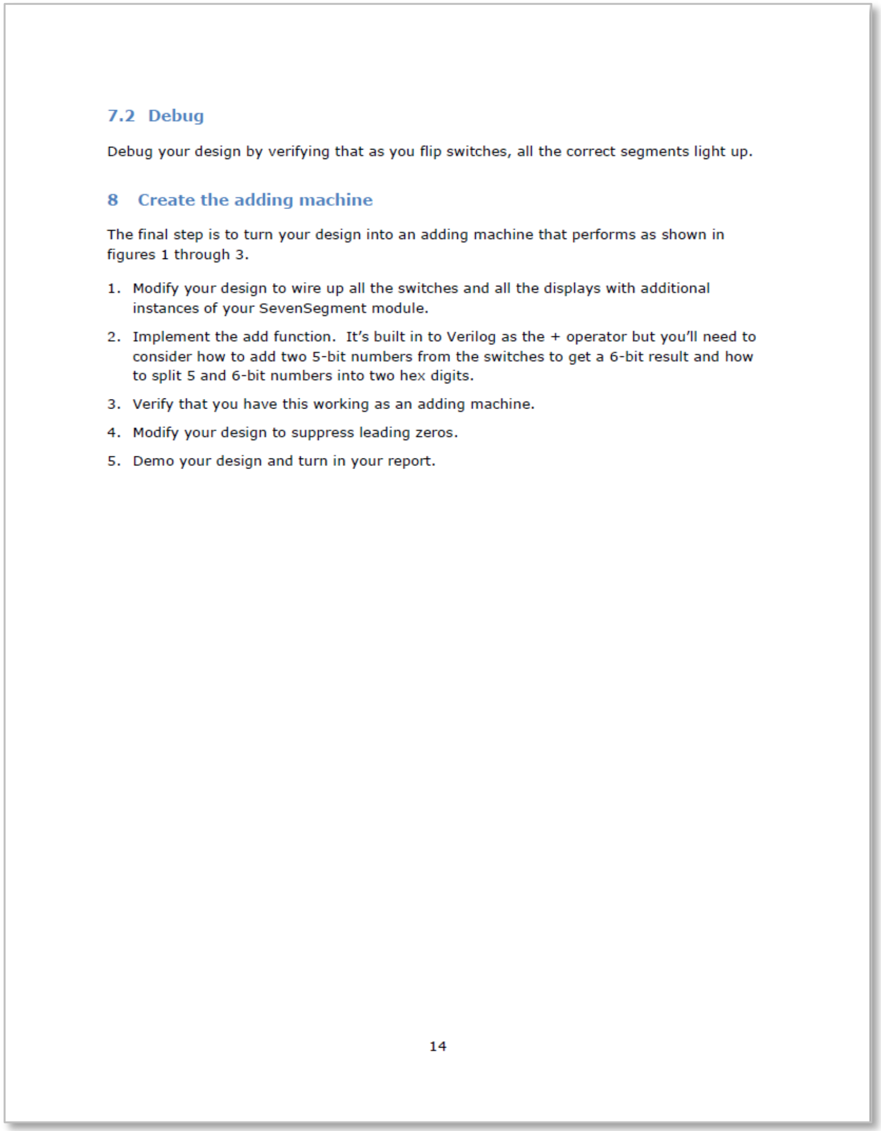
10. Fill in equations for segments 1 through 6.

Basic skeleton



```verilog

module SevenSegment(
      input  [ 3:0 ] hexDigit,
      output [ 6:0 ] segments );

   wire b0, b1, b2, b3;
   wire [ 6:0 ] s;

   assign b0 = hexDigit[ 0 ];
   assign b1 = hexDigit[ 1 ];
   assign b2 = hexDigit[ 2 ];
   assign b3 = hexDigit[ 3 ];

   // s[ 0 ] done.  TBD:  Assign statements for s[ 1 ] .. s[ 6 ].
   assign s[ 0 ] = b1 & b2 | ~b1 & ~b2 & b3 | ~b0 & b3 |
                 ~b0 & ~b2 | b0 & b2 & ~b3 | b1 & ~b3;

   // Invert the outputs for active low on the DE1-SoC board.
   assign segments = ~s;

endmodule


// Stubbed wrapper for the DE1-SoC

module AddingMachine(
      output [ 6:0 ] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   assign LEDR = SW;
   SevenSegment Hex0( SW[ 3:0 ], HEX0 );

endmodule

```

On their own to solve remaining design problems



7.2 Debug

Debug your design by verifying that as you flip switches, all the correct segments light up.

8 Create the adding machine

The final step is to turn your design into an adding machine that performs as shown in figures 1 through 3.

1. Modify your design to wire up all the switches and all the displays with additional instances of your SevenSegment module.
2. Implement the add function. It's built in to Verilog as the + operator but you'll need to consider how to add two 5-bit numbers from the switches to get a 6-bit result and how to split 5 and 6-bit numbers into two hex digits.
3. Verify that you have this working as an adding machine.
4. Modify your design to suppress leading zeros.
5. Demo your design and turn in your report.

14

1. Instantiate copies of the 7-segment decoder for each of the displays.
2. Implement C = A + B.
3. Split the bits for A, B and C across the 6 displays.
4. Suppress leading zeros.

Demo and submit Karnaugh maps and code.

14 pages.

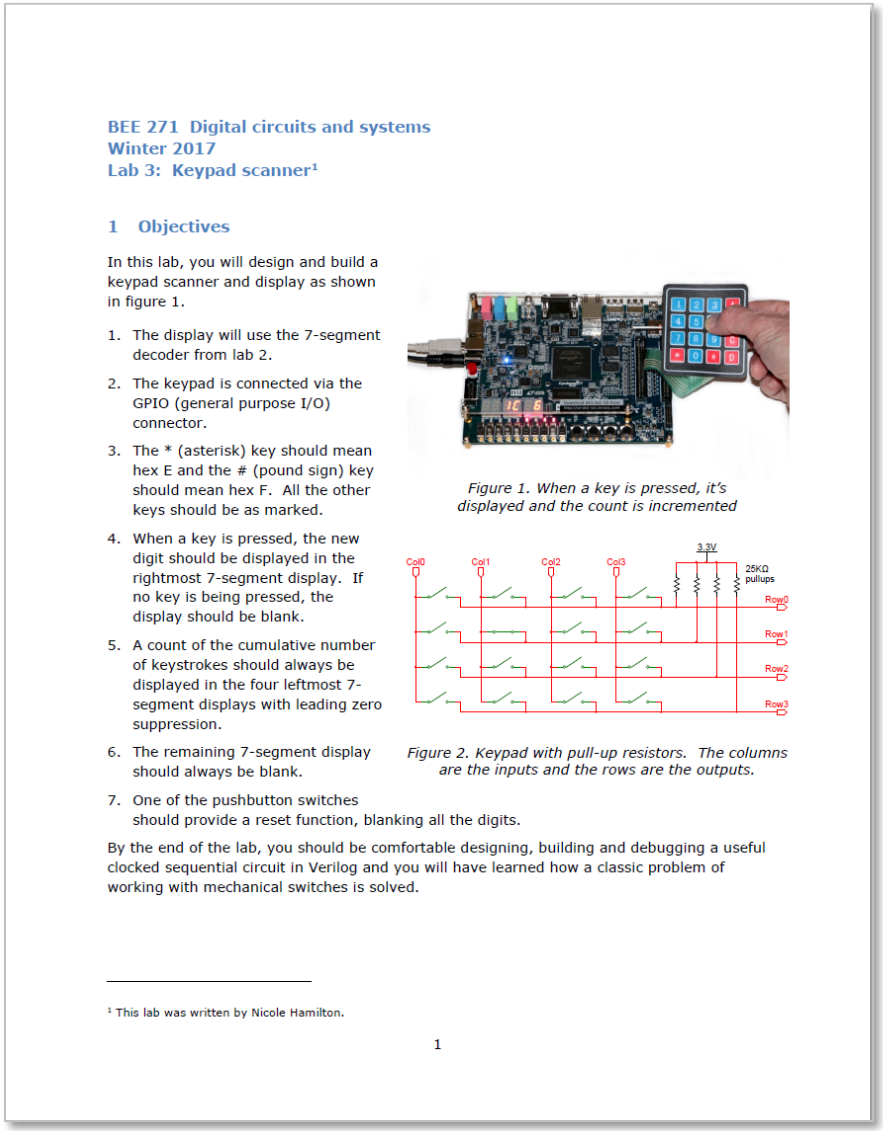
Lab 2 reflections

1. The students like the exercise and most complete it without much help.

2. They like building something and that it’s more satisfying than homework as a way to learn Karnaugh maps. It works or it doesn’t and they can keep at it until it does.

3. Compiles are very slow and often hang. We need much faster lab machines.

Lab 3 Keypad scanner



BEE 271 Digital circuits and systems
Winter 2017
Lab 3: Keypad scanner[1]

1 Objectives

In this lab, you will design and build a keypad scanner and display as shown in figure 1.

1. The display will use the 7-segment decoder from lab 2.
2. The keypad is connected via the GPIO (general purpose I/O) connector.
3. The * (asterisk) key should mean hex E and the # (pound sign) key should mean hex F. All the other keys should be as marked.
4. When a key is pressed, the new digit should be displayed in the rightmost 7-segment display. If no key is being pressed, the display should be blank.
5. A count of the cumulative number of keystrokes should always be displayed in the four leftmost 7-segment displays with leading zero suppression.
6. The remaining 7-segment display should always be blank.
7. One of the pushbutton switches should provide a reset function, blanking all the digits.

Figure 1. When a key is pressed, it's displayed and the count is incremented

Figure 2. Keypad with pull-up resistors. The columns are the inputs and the rows are the outputs.

By the end of the lab, you should be comfortable designing, building and debugging a useful clocked sequential circuit in Verilog and you will have learned how a classic problem of working with mechanical switches is solved.

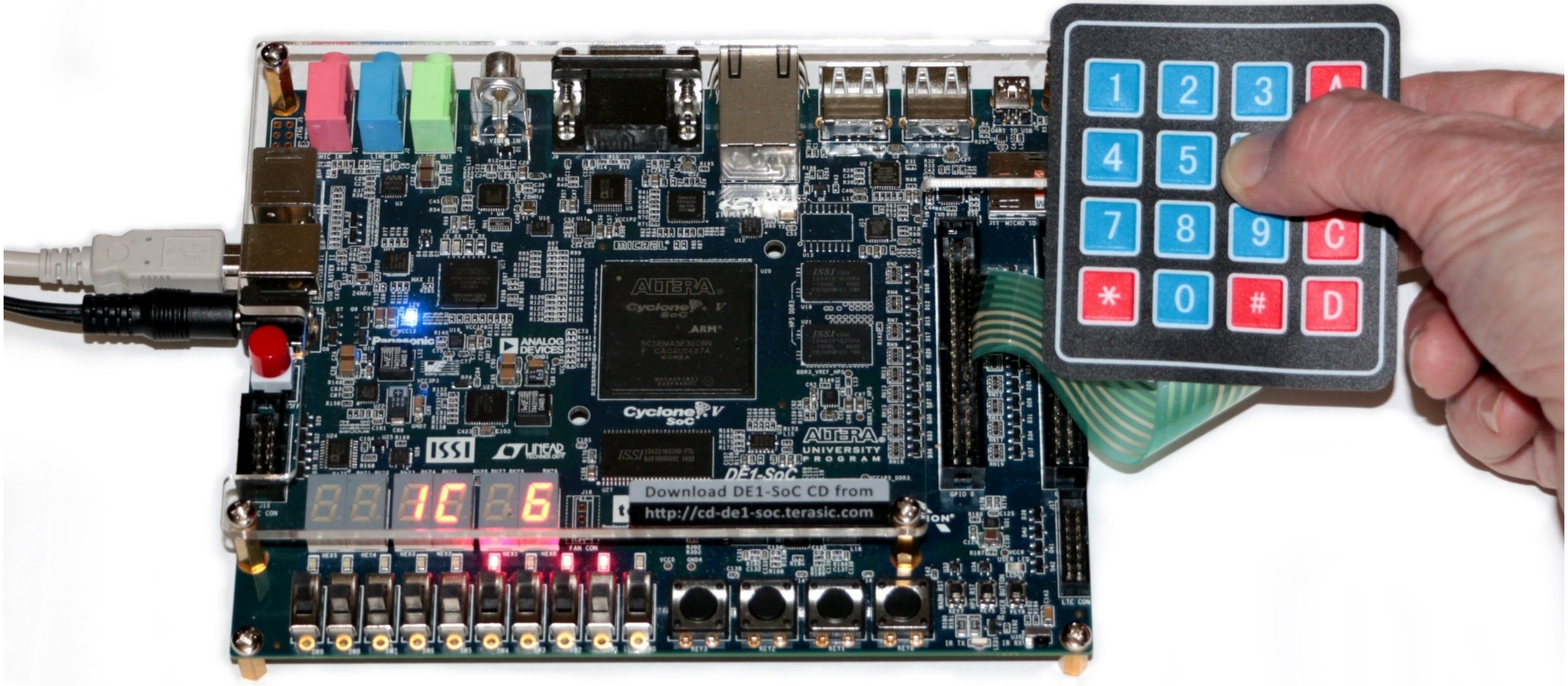
[1] This lab was written by Nicole Hamilton.

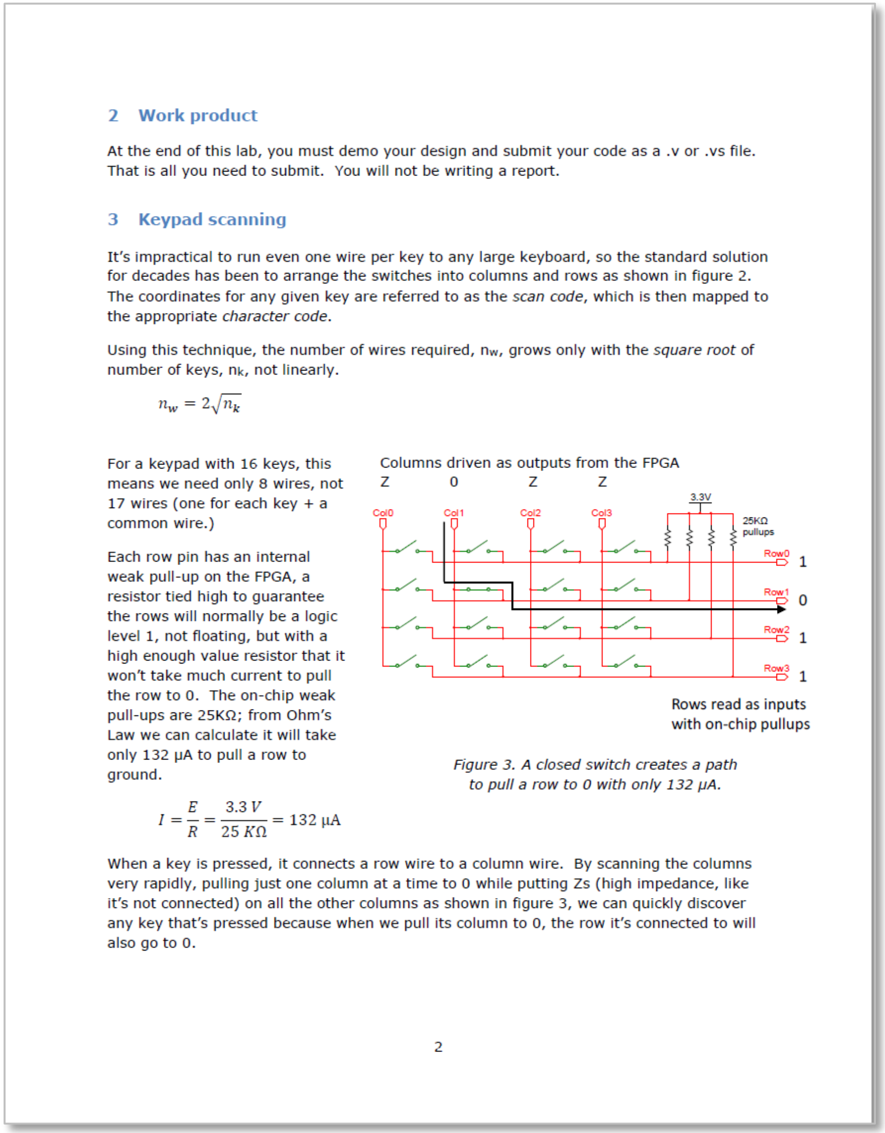
This is a sequential logic project.

1. Scan a hex keypad to determine when a key has been pressed.
2. When a key is pressed, display it and increment a 16-bit count.

Learning objectives

1. Ability to solve a simple sequential logic problem.
2. Understanding of an actual application, how a keypad is read.


1C
6
Download DE1-SoC CD from
http://cd-de1-soc.terasic.com
DE1-SoC
Cyclone V SoC
ALTERA UNIVERSITY PROGRAM
1
2
3
A
4
5
7
8
9
C
*
0
#
D



2 Work product

At the end of this lab, you must demo your design and submit your code as a .v or .vs file. That is all you need to submit. You will not be writing a report.

3 Keypad scanning

It's impractical to run even one wire per key to any large keyboard, so the standard solution for decades has been to arrange the switches into columns and rows as shown in figure 2. The coordinates for any given key are referred to as the *scan code*, which is then mapped to the appropriate *character code*.

Using this technique, the number of wires required, n_w, grows only with the *square root* of number of keys, n_k, not linearly.

$$n_w = 2\sqrt{n_k}$$

For a keypad with 16 keys, this means we need only 8 wires, not 17 wires (one for each key + a common wire.)

Each row pin has an internal weak pull-up on the FPGA, a resistor tied high to guarantee the rows will normally be a logic level 1, not floating, but with a high enough value resistor that it won't take much current to pull the row to 0. The on-chip weak pull-ups are 25KΩ; from Ohm's Law we can calculate it will take only 132 μA to pull a row to ground.

$$I = \frac{E}{R} = \frac{3.3\ V}{25\ K\Omega} = 132\ \mu\text{A}$$

Columns driven as outputs from the FPGA

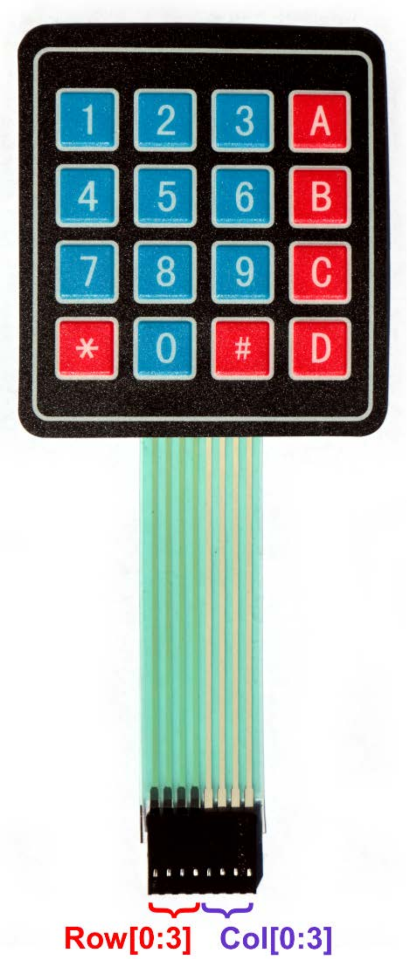
Z 0 Z Z

Rows read as inputs with on-chip pullups

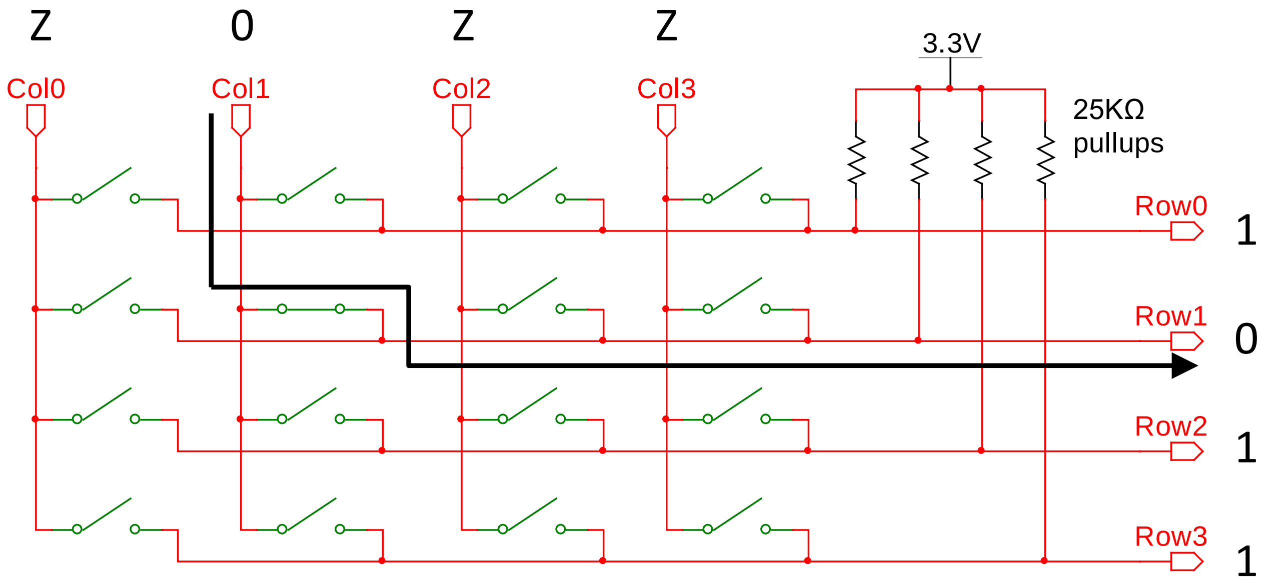
Figure 3. A closed switch creates a path to pull a row to 0 with only 132 μA.

When a key is pressed, it connects a row wire to a column wire. By scanning the columns very rapidly, pulling just one column at a time to 0 while putting Zs (high impedance, like it's not connected) on all the other columns as shown in figure 3, we can quickly discover any key that's pressed because when we pull its column to 0, the row it's connected to will also go to 0.

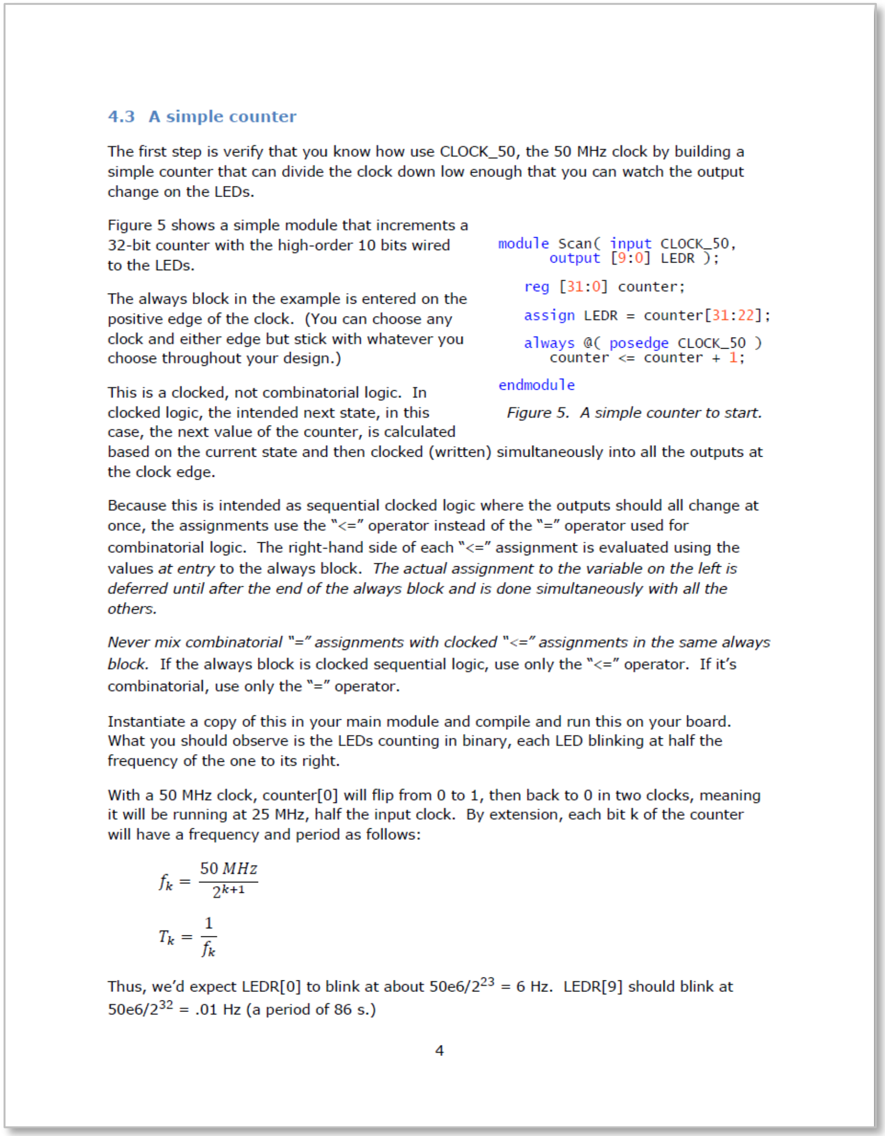
3. Rotate a 0 across the columns at some reasonable scan rate.
4. If a row goes to zero, a key at the intersection has been pressed.



Columns driven as outputs from the FPGA



Rows read as inputs
with on-chip pullups



4.3 A simple counter

The first step is verify that you know how use CLOCK_50, the 50 MHz clock by building a simple counter that can divide the clock down low enough that you can watch the output change on the LEDs.

Figure 5 shows a simple module that increments a 32-bit counter with the high-order 10 bits wired to the LEDs.

The always block in the example is entered on the positive edge of the clock. (You can choose any clock and either edge but stick with whatever you choose throughout your design.)

```
module Scan( input CLOCK_50,
      output [9:0] LEDR );

   reg [31:0] counter;

   assign LEDR = counter[31:22];

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

endmodule
```

Figure 5. A simple counter to start.

This is a clocked, not combinatorial logic. In clocked logic, the intended next state, in this case, the next value of the counter, is calculated based on the current state and then clocked (written) simultaneously into all the outputs at the clock edge.

Because this is intended as sequential clocked logic where the outputs should all change at once, the assignments use the "<=" operator instead of the "=" operator used for combinatorial logic. The right-hand side of each "<=" assignment is evaluated using the values *at entry* to the always block. *The actual assignment to the variable on the left is deferred until after the end of the always block and is done simultaneously with all the others.*

Never mix combinatorial "=" assignments with clocked "<=" assignments in the same always block. If the always block is clocked sequential logic, use only the "<=" operator. If it's combinatorial, use only the "=" operator.

Instantiate a copy of this in your main module and compile and run this on your board. What you should observe is the LEDs counting in binary, each LED blinking at half the frequency of the one to its right.

With a 50 MHz clock, counter[0] will flip from 0 to 1, then back to 0 in two clocks, meaning it will be running at 25 MHz, half the input clock. By extension, each bit k of the counter will have a frequency and period as follows:

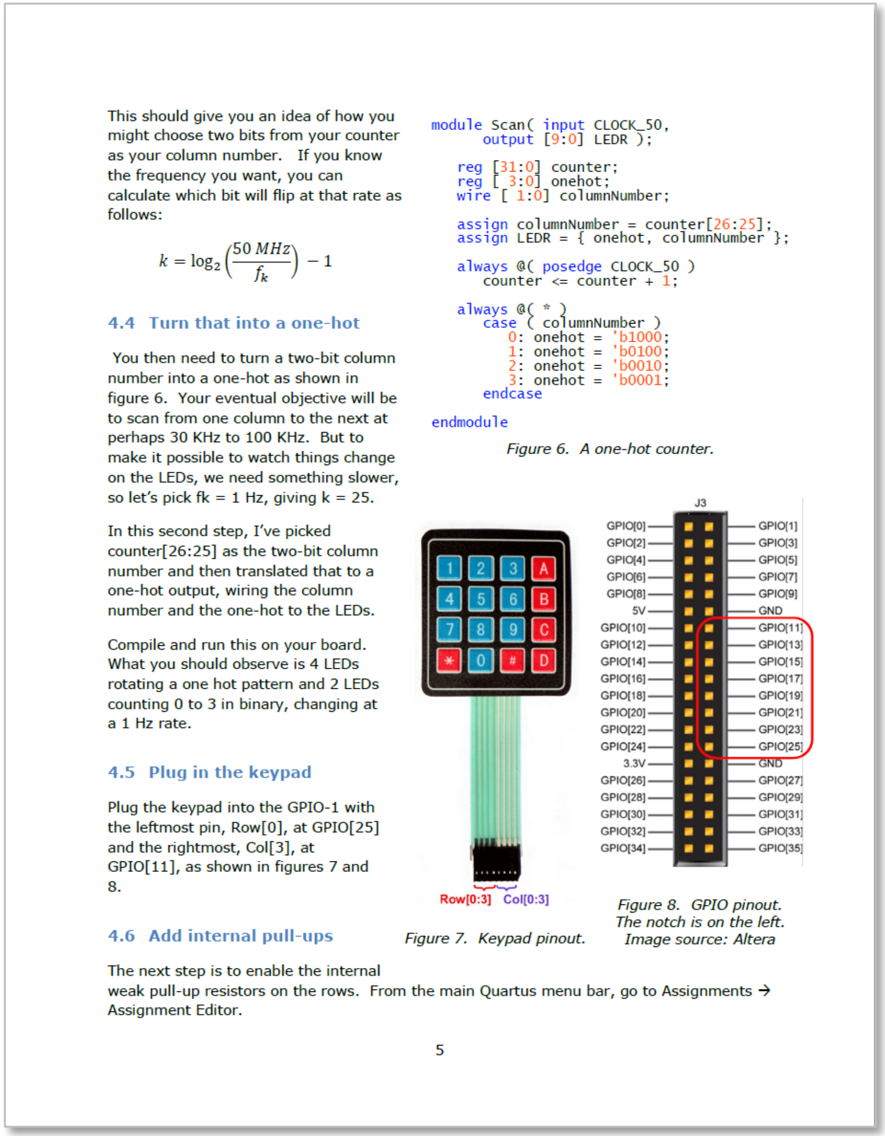
$$f_k = \frac{50\ MHz}{2^{k+1}}$$

$$T_k = \frac{1}{f_k}$$

Thus, we'd expect LEDR[0] to blink at about 50e6/2^{23} = 6 Hz. LEDR[9] should blink at 50e6/2^{32} = .01 Hz (a period of 86 s.)

5. Starts with a simple counter tied to the LEDs and discussion of how fast they'll blink.

```
module Scan( input CLOCK_50,
      output [9:0] LEDR );

   reg  [ 31:0 ] counter;

   assign LEDR = counter[ 26:25 ];

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

endmodule
```



This should give you an idea of how you might choose two bits from your counter as your column number. If you know the frequency you want, you can calculate which bit will flip at that rate as follows:

$$k = \log_2\left(\frac{50\ MHz}{f_k}\right) - 1$$

4.4 Turn that into a one-hot

You then need to turn a two-bit column number into a one-hot as shown in figure 6. Your eventual objective will be to scan from one column to the next at perhaps 30 KHz to 100 KHz. But to make it possible to watch things change on the LEDs, we need something slower, so let's pick fk = 1 Hz, giving k = 25.

In this second step, I've picked counter[26:25] as the two-bit column number and then translated that to a one-hot output, wiring the column number and the one-hot to the LEDs.

Compile and run this on your board. What you should observe is 4 LEDs rotating a one hot pattern and 2 LEDs counting 0 to 3 in binary, changing at a 1 Hz rate.

4.5 Plug in the keypad

Plug the keypad into the GPIO-1 with the leftmost pin, Row[0], at GPIO[25] and the rightmost, Col[3], at GPIO[11], as shown in figures 7 and 8.

4.6 Add internal pull-ups

The next step is to enable the internal weak pull-up resistors on the rows. From the main Quartus menu bar, go to Assignments → Assignment Editor.

```
module Scan( input CLOCK_50,
   output [9:0] LEDR );

   reg [31:0] counter;
   reg [ 3:0] onehot;
   wire [ 1:0] columnNumber;

   assign columnNumber = counter[26:25];
   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

   always @( * )
      case ( columnNumber )
         0: onehot = 'b1000;
         1: onehot = 'b0100;
         2: onehot = 'b0010;
         3: onehot = 'b0001;
      endcase

endmodule
```

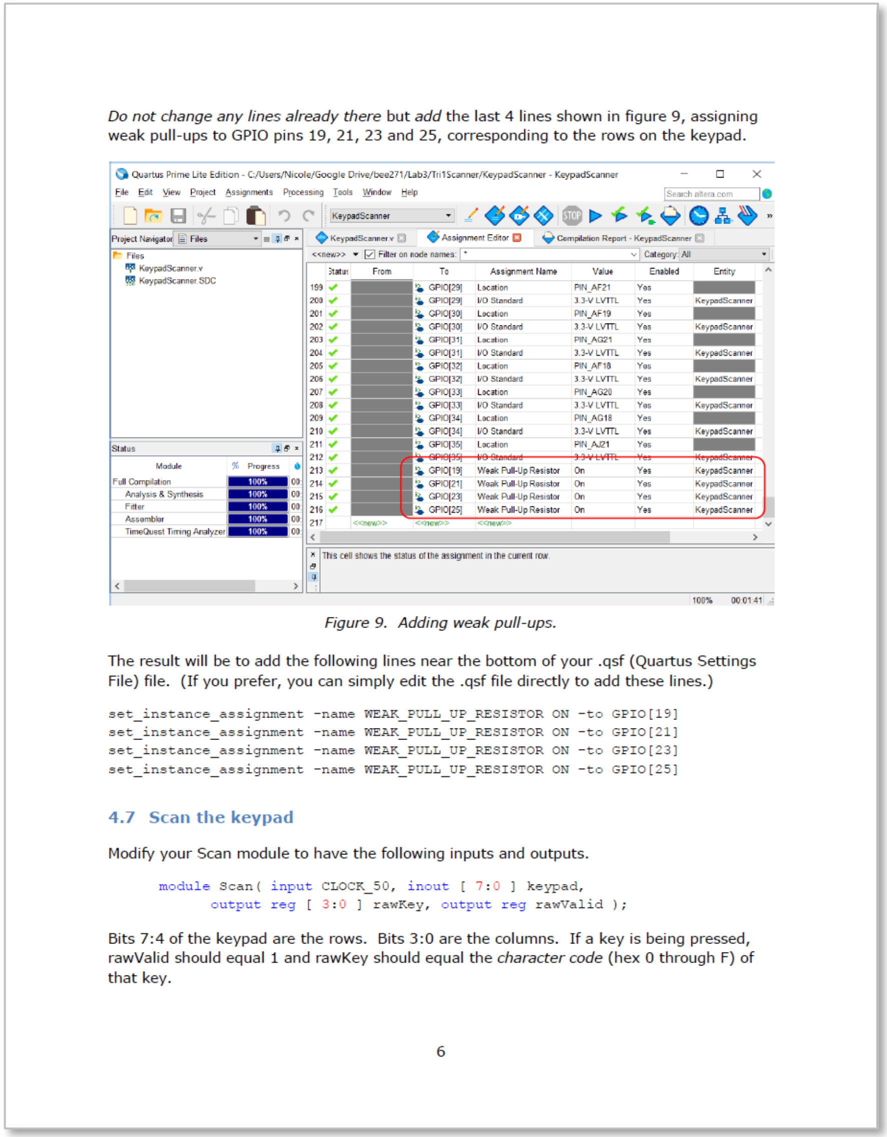
Figure 6. A one-hot counter.

Figure 7. Keypad pinout.

Figure 8. GPIO pinout. The notch is on the left. Image source: Altera

6. Example turned into a one-hot, which has to be modified by the student.

```
module Scan( input CLOCK_50,
      output [9:0] LEDR );

   reg  [ 31:0 ] counter;
   reg  [  3:0 ] onehot;
   wire [  1:0 ] columnNumber;

   assign columnNumber = counter[ 26:25 ];
   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

   always @( * )
      case ( columnNumber )
         0: onehot = 'b1000;
         1: onehot = 'b0100;
         2: onehot = 'b0010;
         3: onehot = 'b0001;
      endcase

endmodule
```



Do not change any lines already there but *add* the last 4 lines shown in figure 9, assigning weak pull-ups to GPIO pins 19, 21, 23 and 25, corresponding to the rows on the keypad.

Quartus Prime Lite Edition - C:/Users/Nicole/Google Drive/bee271/Lab3/Tri1Scanner/KeypadScanner - KeypadScanner

	Status	From	To	Assignment Name	Value	Enabled	Entity
199	✓		GPIO[29]	Location	PIN_AF21	Yes	
200	✓		GPIO[29]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
201	✓		GPIO[30]	Location	PIN_AF19	Yes	
202	✓		GPIO[30]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
203	✓		GPIO[31]	Location	PIN_AG21	Yes	
204	✓		GPIO[31]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
205	✓		GPIO[32]	Location	PIN_AF18	Yes	
206	✓		GPIO[32]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
207	✓		GPIO[33]	Location	PIN_AG20	Yes	
208	✓		GPIO[33]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
209	✓		GPIO[34]	Location	PIN_AG18	Yes	
210	✓		GPIO[34]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
211	✓		GPIO[35]	Location	PIN_AJ21	Yes	
212	✓		GPIO[35]	I/O Standard	3.3-V LVTTL	Yes	KeypadScanner
213	✓		GPIO[19]	Weak Pull-Up Resistor	On	Yes	KeypadScanner
214	✓		GPIO[21]	Weak Pull-Up Resistor	On	Yes	KeypadScanner
215	✓		GPIO[23]	Weak Pull-Up Resistor	On	Yes	KeypadScanner
216	✓		GPIO[25]	Weak Pull-Up Resistor	On	Yes	KeypadScanner
217		<<new>>	<<new>>	<<new>>			

Status

Module	% Progress	
Full Compilation	100%	00:
Analysis & Synthesis	100%	00:
Fitter	100%	00:
Assembler	100%	00:
TimeQuest Timing Analyzer	100%	00:

This cell shows the status of the assignment in the current row.

Figure 9. Adding weak pull-ups.

The result will be to add the following lines near the bottom of your .qsf (Quartus Settings File) file. (If you prefer, you can simply edit the .qsf file directly to add these lines.)

```
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[19]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[21]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[23]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[25]
```

4.7 Scan the keypad

Modify your Scan module to have the following inputs and outputs.

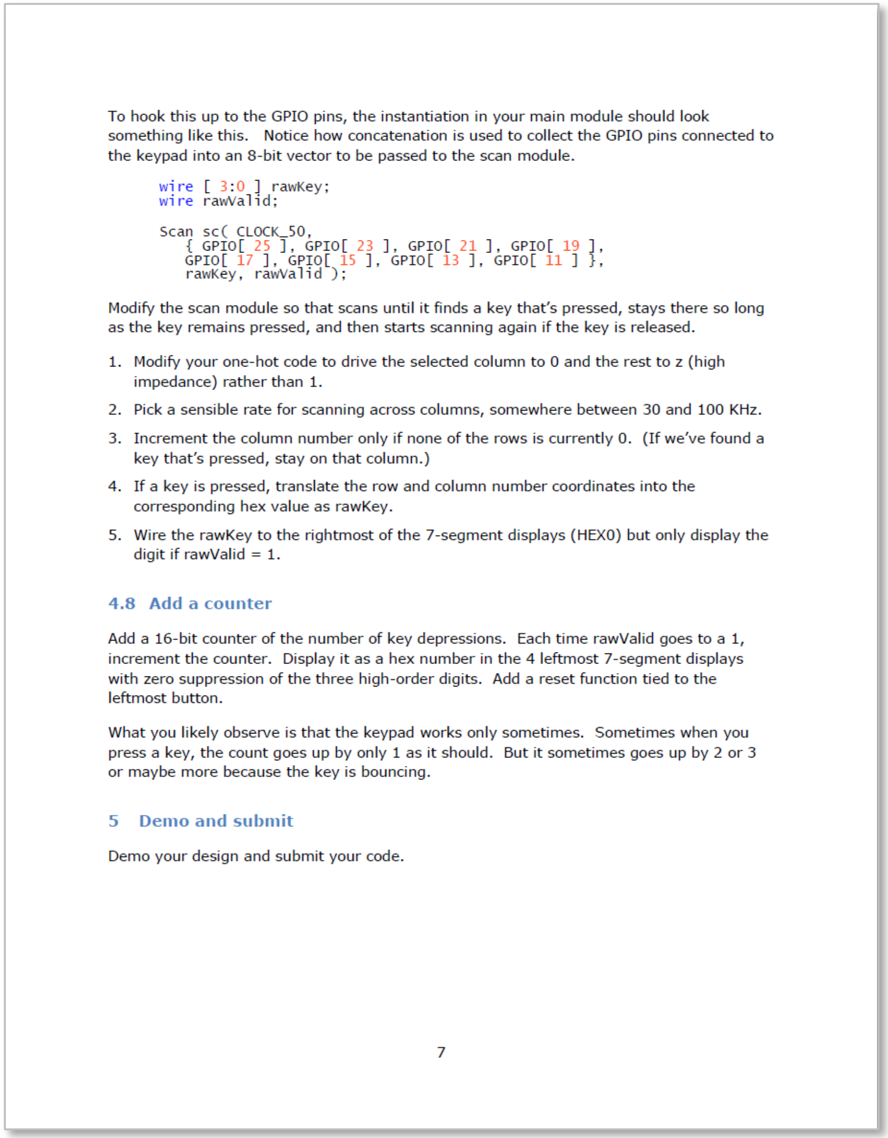
```
    module Scan( input CLOCK_50, inout [ 7:0 ] keypad,
          output reg [ 3:0 ] rawKey, output reg rawValid );
```

Bits 7:4 of the keypad are the rows. Bits 3:0 are the columns. If a key is being pressed, rawValid should equal 1 and rawKey should equal the *character code* (hex 0 through F) of that key.

7. On-chip pullups are added with the Assignment Editor.

8. Create a Scan module.

```
module Scan( input CLOCK_50,
      inout [ 7:0 ] keypad,
      output reg [ 3:0 ] rawKey,
      output reg rawValid );
```



To hook this up to the GPIO pins, the instantiation in your main module should look something like this. Notice how concatenation is used to collect the GPIO pins connected to the keypad into an 8-bit vector to be passed to the scan module.

```
wire [ 3:0 ] rawKey;
wire rawValid;

Scan sc( CLOCK_50,
   { GPIO[ 25 ], GPIO[ 23 ], GPIO[ 21 ], GPIO[ 19 ],
   GPIO[ 17 ], GPIO[ 15 ], GPIO[ 13 ], GPIO[ 11 ] },
   rawKey, rawValid );
```

Modify the scan module so that scans until it finds a key that's pressed, stays there so long as the key remains pressed, and then starts scanning again if the key is released.

1. Modify your one-hot code to drive the selected column to 0 and the rest to z (high impedance) rather than 1.
2. Pick a sensible rate for scanning across columns, somewhere between 30 and 100 KHz.
3. Increment the column number only if none of the rows is currently 0. (If we've found a key that's pressed, stay on that column.)
4. If a key is pressed, translate the row and column number coordinates into the corresponding hex value as rawKey.
5. Wire the rawKey to the rightmost of the 7-segment displays (HEX0) but only display the digit if rawValid = 1.

4.8 Add a counter

Add a 16-bit counter of the number of key depressions. Each time rawValid goes to a 1, increment the counter. Display it as a hex number in the 4 leftmost 7-segment displays with zero suppression of the three high-order digits. Add a reset function tied to the leftmost button.

What you likely observe is that the keypad works only sometimes. Sometimes when you press a key, the count goes up by only 1 as it should. But it sometimes goes up by 2 or 3 or maybe more because the key is bouncing.

5 Demo and submit

Demo your design and submit your code.

9. Instantiate their Scan module and tie it to the GPIO.

```
wire [ 3:0 ] rawKey;
wire rawValid;

Scan sc( CLOCK_50,
   { GPIO[ 25 ], GPIO[ 23 ],
     GPIO[ 21 ], GPIO[ 19 ],
     GPIO[ 17 ], GPIO[ 15 ],
     GPIO[ 13 ], GPIO[ 11 ] },
   rawKey, rawValid );
```

10. Add a counter, 7-segment displays and a reset.

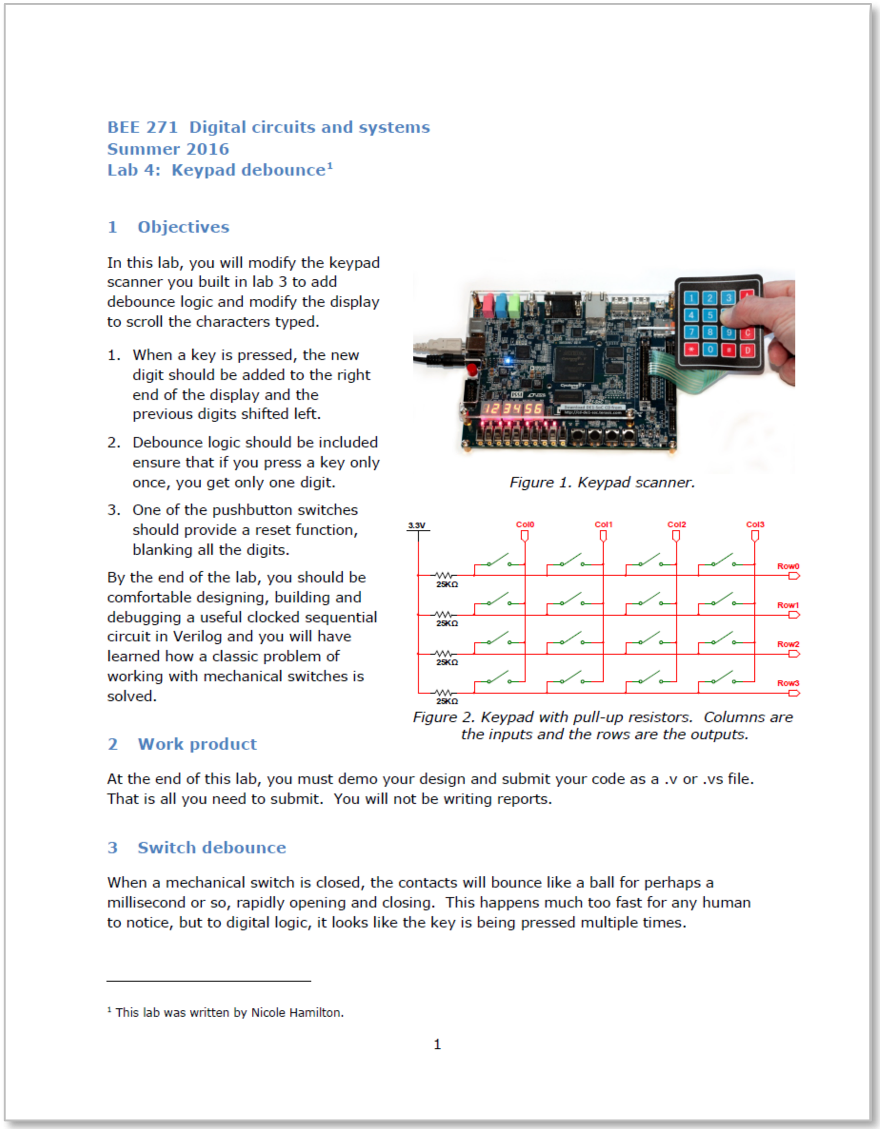
11. They should observe that keys sometimes bounce.

12. Demo and submit the code.

Lab 3 reflections

1. The students like the exercise and are intrigued to learn how a simple keypad works but find it deceptively difficult.
2. If they have trouble, it’s in:
   a) How to stop advancing to the next column if they’ve discovered a key that’s pressed.
   b) How to translate row and column coordinates into a hex key code.
3. Sadly, some of the keypads don’t bounce very much.

Lab 4 Keypad debounce



BEE 271 Digital circuits and systems
Summer 2016
Lab 4: Keypad debounce[1]

1 Objectives

In this lab, you will modify the keypad scanner you built in lab 3 to add debounce logic and modify the display to scroll the characters typed.

1. When a key is pressed, the new digit should be added to the right end of the display and the previous digits shifted left.
2. Debounce logic should be included ensure that if you press a key only once, you get only one digit.
3. One of the pushbutton switches should provide a reset function, blanking all the digits.

By the end of the lab, you should be comfortable designing, building and debugging a useful clocked sequential circuit in Verilog and you will have learned how a classic problem of working with mechanical switches is solved.

Figure 1. Keypad scanner.

Figure 2. Keypad with pull-up resistors. Columns are the inputs and the rows are the outputs.

2 Work product

At the end of this lab, you must demo your design and submit your code as a .v or .vs file. That is all you need to submit. You will not be writing reports.

3 Switch debounce

When a mechanical switch is closed, the contacts will bounce like a ball for perhaps a millisecond or so, rapidly opening and closing. This happens much too fast for any human to notice, but to digital logic, it looks like the key is being pressed multiple times.

[1] This lab was written by Nicole Hamilton.

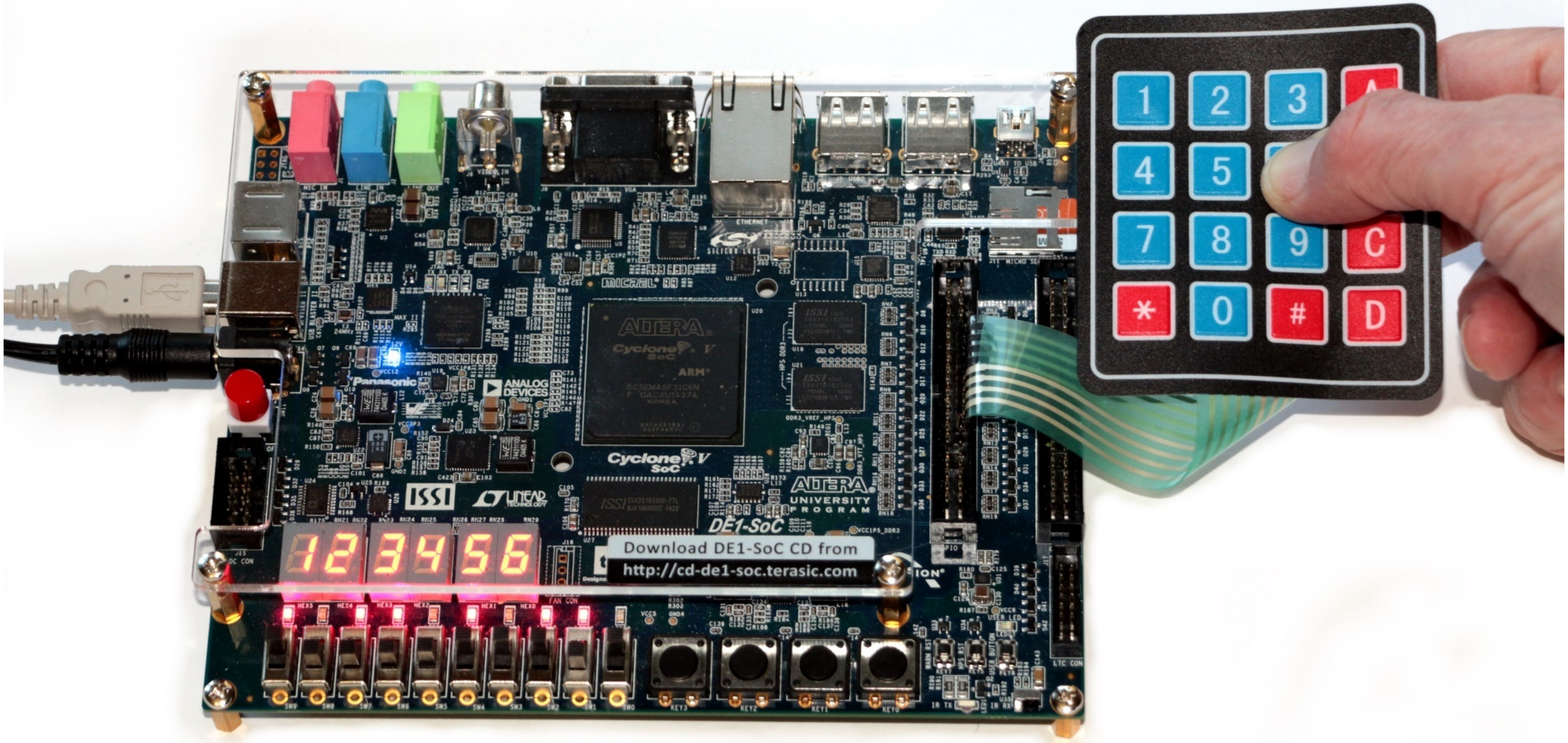
1

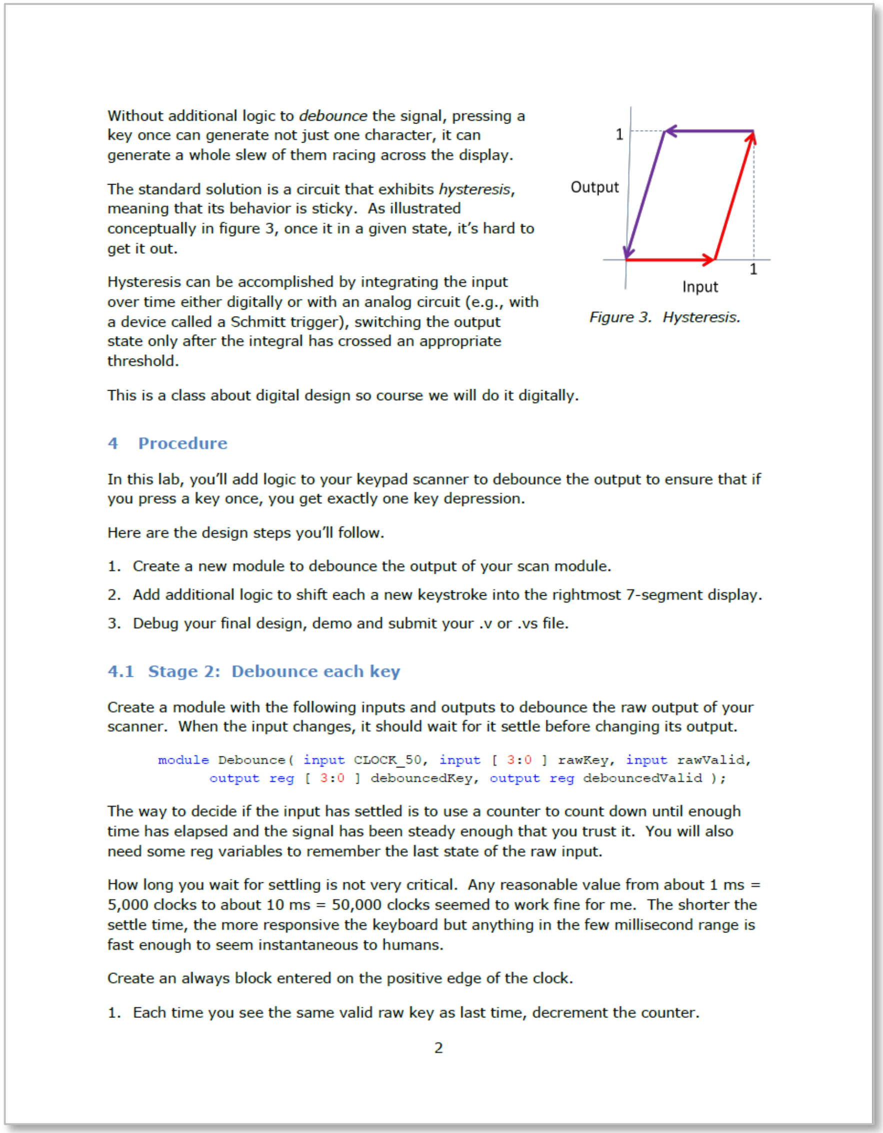
This is a sequential logic project.

1. Build a new module that will debounce the raw keys.
2. When a key is pressed, shift the displays and add it.

Learning objectives

1. Ability to solve a slightly more difficult sequential logic problem.
2. Understanding of how to use hysteresis to debounce a signal.


1 2 3 A
4 5
7 8 9 C
* 0 # D
123456
Download DE1-SoC CD from
http://cd-de1-soc.terasic.com
ALTERA Cyclone V SoC ARM
DE1-SoC
ALTERA UNIVERSITY PROGRAM



Without additional logic to *debounce* the signal, pressing a key once can generate not just one character, it can generate a whole slew of them racing across the display.

The standard solution is a circuit that exhibits *hysteresis*, meaning that its behavior is sticky. As illustrated conceptually in figure 3, once it in a given state, it's hard to get it out.

Hysteresis can be accomplished by integrating the input over time either digitally or with an analog circuit (e.g., with a device called a Schmitt trigger), switching the output state only after the integral has crossed an appropriate threshold.

Figure 3. Hysteresis.

This is a class about digital design so course we will do it digitally.

4 Procedure

In this lab, you'll add logic to your keypad scanner to debounce the output to ensure that if you press a key once, you get exactly one key depression.

Here are the design steps you'll follow.

1. Create a new module to debounce the output of your scan module.
2. Add additional logic to shift each a new keystroke into the rightmost 7-segment display.
3. Debug your final design, demo and submit your .v or .vs file.

4.1 Stage 2: Debounce each key

Create a module with the following inputs and outputs to debounce the raw output of your scanner. When the input changes, it should wait for it settle before changing its output.

```
module Debounce( input CLOCK_50, input [ 3:0 ] rawKey, input rawValid,
     output reg [ 3:0 ] debouncedKey, output reg debouncedValid );
```

The way to decide if the input has settled is to use a counter to count down until enough time has elapsed and the signal has been steady enough that you trust it. You will also need some reg variables to remember the last state of the raw input.

How long you wait for settling is not very critical. Any reasonable value from about 1 ms = 5,000 clocks to about 10 ms = 50,000 clocks seemed to work fine for me. The shorter the settle time, the more responsive the keyboard but anything in the few millisecond range is fast enough to seem instantaneous to humans.

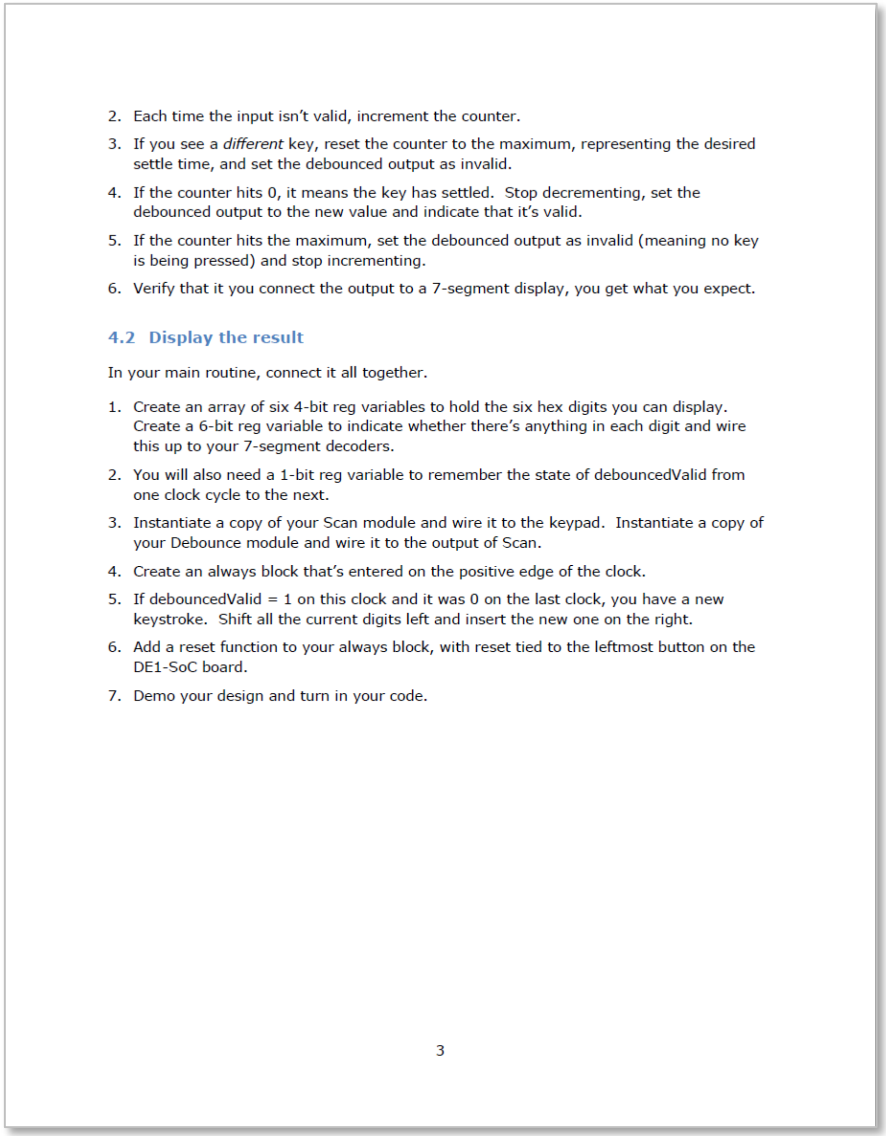
Create an always block entered on the positive edge of the clock.

1. Each time you see the same valid raw key as last time, decrement the counter.

3. Understand the concept of hysteresis as a way of debouncing a signal.

4. Create a debounce module that can take in a raw key and output a debounced key.

```
module Debounce(
      input CLOCK_50,
      input [ 3:0 ] rawKey,
      input rawValid,
      output reg [ 3:0 ] debouncedKey,
      output reg debouncedValid );
```



2. Each time the input isn't valid, increment the counter.
3. If you see a *different* key, reset the counter to the maximum, representing the desired settle time, and set the debounced output as invalid.
4. If the counter hits 0, it means the key has settled. Stop decrementing, set the debounced output to the new value and indicate that it's valid.
5. If the counter hits the maximum, set the debounced output as invalid (meaning no key is being pressed) and stop incrementing.
6. Verify that it you connect the output to a 7-segment display, you get what you expect.

4.2 Display the result

In your main routine, connect it all together.

1. Create an array of six 4-bit reg variables to hold the six hex digits you can display. Create a 6-bit reg variable to indicate whether there's anything in each digit and wire this up to your 7-segment decoders.
2. You will also need a 1-bit reg variable to remember the state of debouncedValid from one clock cycle to the next.
3. Instantiate a copy of your Scan module and wire it to the keypad. Instantiate a copy of your Debounce module and wire it to the output of Scan.
4. Create an always block that's entered on the positive edge of the clock.
5. If debouncedValid = 1 on this clock and it was 0 on the last clock, you have a new keystroke. Shift all the current digits left and insert the new one on the right.
6. Add a reset function to your always block, with reset tied to the leftmost button on the DE1-SoC board.
7. Demo your design and turn in your code.

Only 3 pages.

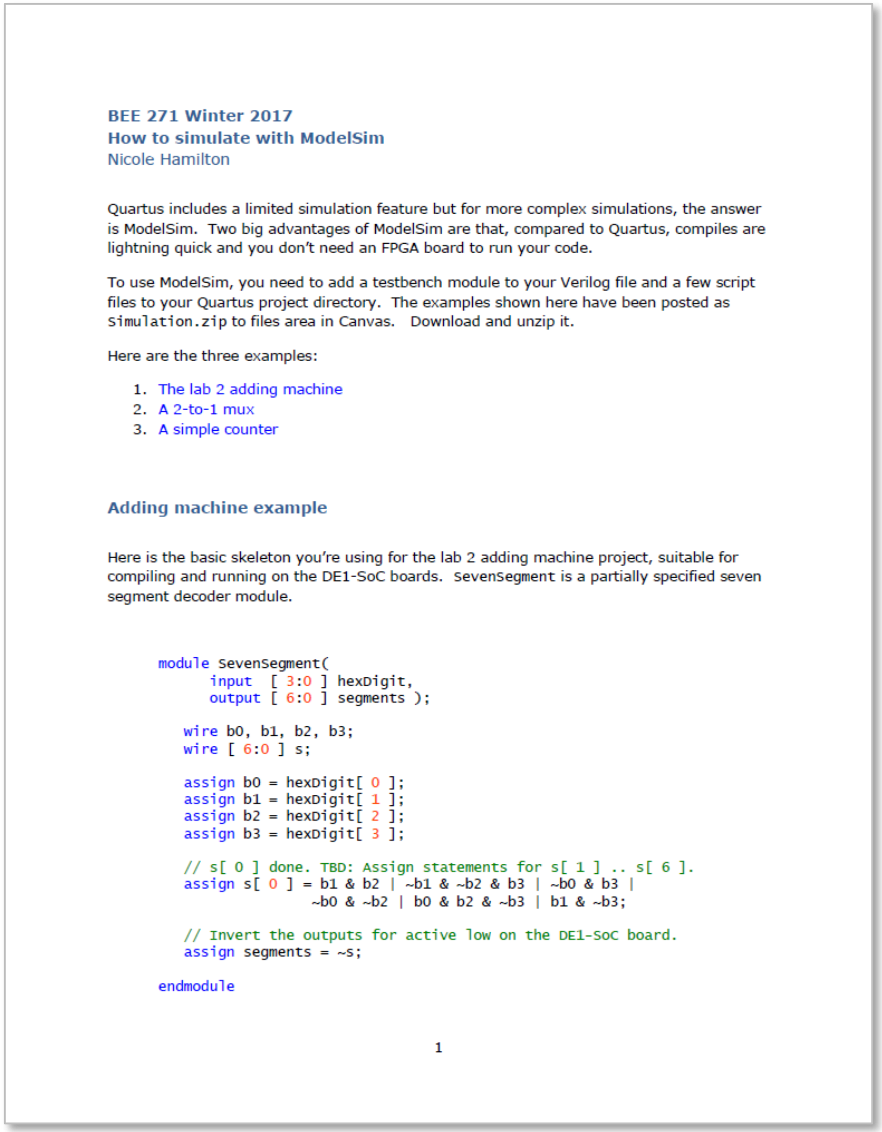
5. Demo and submit your code.

Lab 4 reflections

1. Least amount of how-to guidance.

2. Deceptively difficult for most students.

ModelSim group exercise



BEE 271 Winter 2017
How to simulate with ModelSim
Nicole Hamilton

Quartus includes a limited simulation feature but for more complex simulations, the answer is ModelSim. Two big advantages of ModelSim are that, compared to Quartus, compiles are lightning quick and you don't need an FPGA board to run your code.

To use ModelSim, you need to add a testbench module to your Verilog file and a few script files to your Quartus project directory. The examples shown here have been posted as `Simulation.zip` to files area in Canvas. Download and unzip it.

Here are the three examples:

1. The lab 2 adding machine
2. A 2-to-1 mux
3. A simple counter

Adding machine example

Here is the basic skeleton you're using for the lab 2 adding machine project, suitable for compiling and running on the DE1-SoC boards. `SevenSegment` is a partially specified seven segment decoder module.

```
module SevenSegment(
      input  [ 3:0 ] hexDigit,
      output [ 6:0 ] segments );

   wire b0, b1, b2, b3;
   wire [ 6:0 ] s;

   assign b0 = hexDigit[ 0 ];
   assign b1 = hexDigit[ 1 ];
   assign b2 = hexDigit[ 2 ];
   assign b3 = hexDigit[ 3 ];

   // s[ 0 ] done. TBD: Assign statements for s[ 1 ] .. s[ 6 ].
   assign s[ 0 ] = b1 & b2 | ~b1 & ~b2 & b3 | ~b0 & b3 |
                  ~b0 & ~b2 | b0 & b2 & ~b3 | b1 & ~b3;

   // Invert the outputs for active low on the DE1-SoC board.
   assign segments = ~s;

endmodule
```

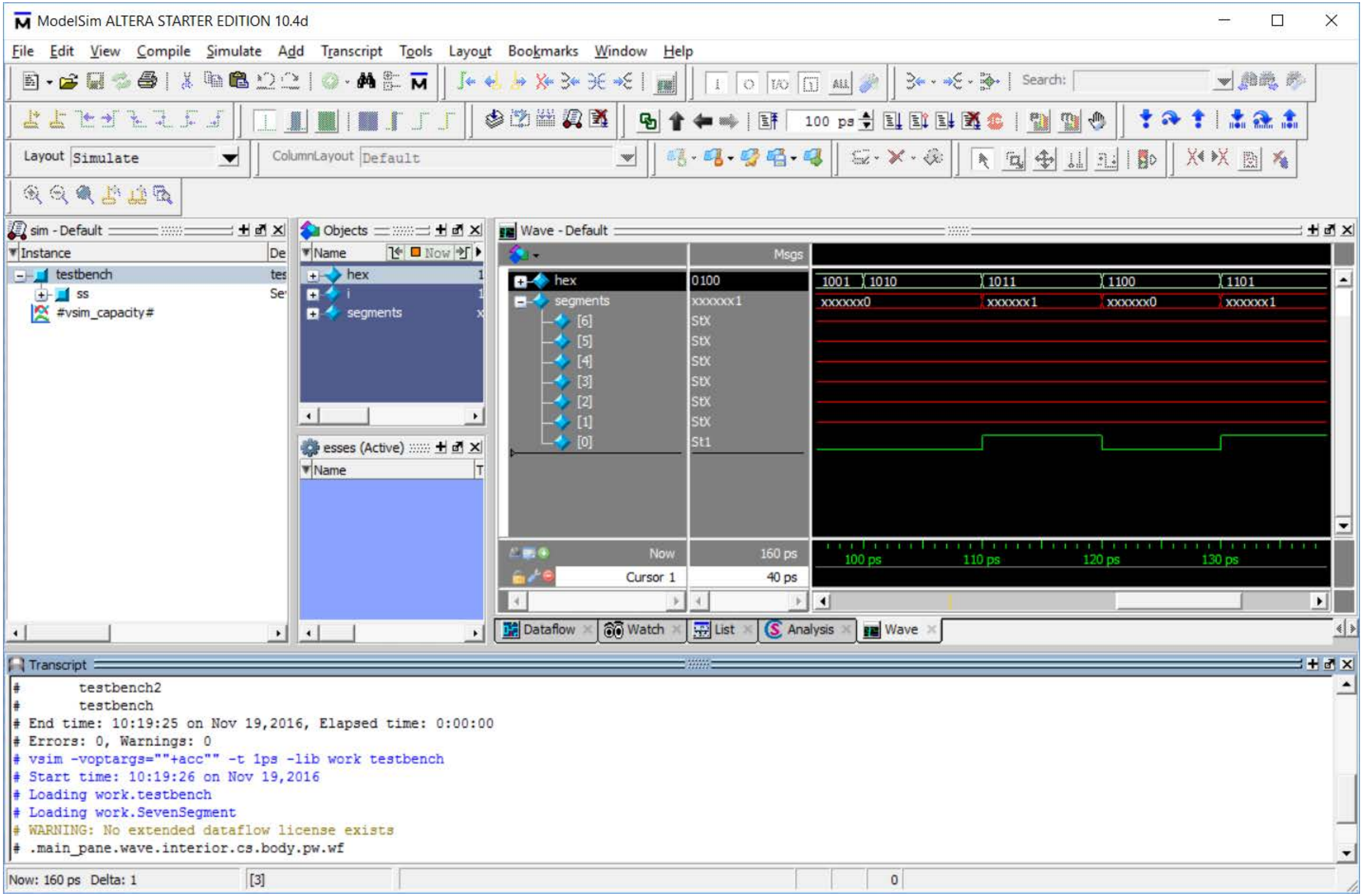
ModelSim is Altera's Verilog simulator.

When I taught EE 271 in Seattle, I liked the way Scott Hauck was using ModelSim in his labs.

What he didn't have was a stand-alone turnkey tutorial.

Three examples as a .zip file.

1. The seven segment decoder from the lab 2 adding machine
2. Several 2-to-1 muxes
3. Two simple counters

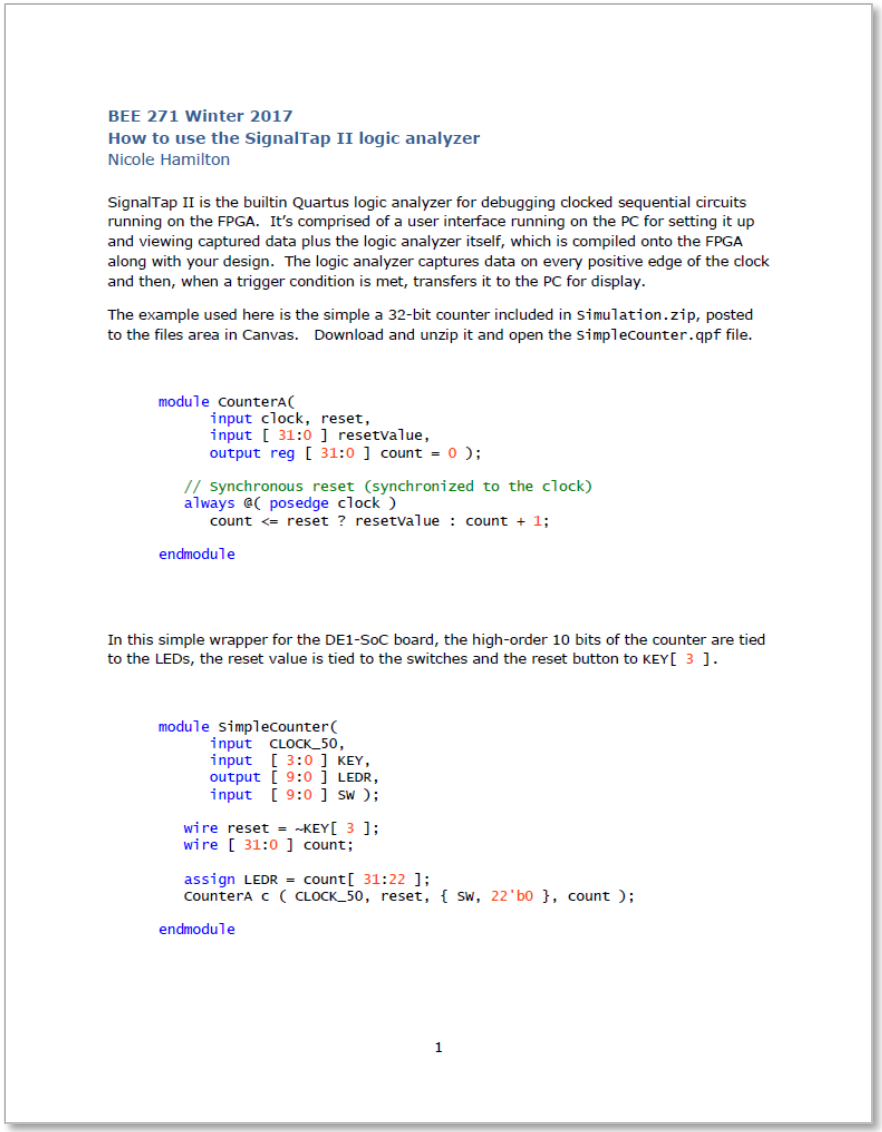


```
hex          0100      1001 | 1010 | 1011 | 1100 | 1101
segments     xxxxxx1   xxxxxx0 | xxxxxx1 | xxxxxx0 | xxxxxx1
  [6]        StX
  [5]        StX
  [4]        StX
  [3]        StX
  [2]        StX
  [1]        StX
  [0]        St1

Now          160 ps    100 ps   110 ps   120 ps   130 ps
Cursor 1     40 ps
```

```
#       testbench2
#       testbench
# End time: 10:19:25 on Nov 19,2016, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# vsim -voptargs=""+acc"" -t 1ps -lib work testbench
# Start time: 10:19:26 on Nov 19,2016
# Loading work.testbench
# Loading work.SevenSegment
# WARNING: No extended dataflow license exists
# .main_pane.wave.interior.cs.body.pw.wf
```

Now: 160 ps Delta: 1   [3]   0

SignalTap II group exercise



BEE 271 Winter 2017
How to use the SignalTap II logic analyzer
Nicole Hamilton

SignalTap II is the builtin Quartus logic analyzer for debugging clocked sequential circuits running on the FPGA. It's comprised of a user interface running on the PC for setting it up and viewing captured data plus the logic analyzer itself, which is compiled onto the FPGA along with your design. The logic analyzer captures data on every positive edge of the clock and then, when a trigger condition is met, transfers it to the PC for display.

The example used here is the simple a 32-bit counter included in `Simulation.zip`, posted to the files area in Canvas. Download and unzip it and open the `SimpleCounter.qpf` file.

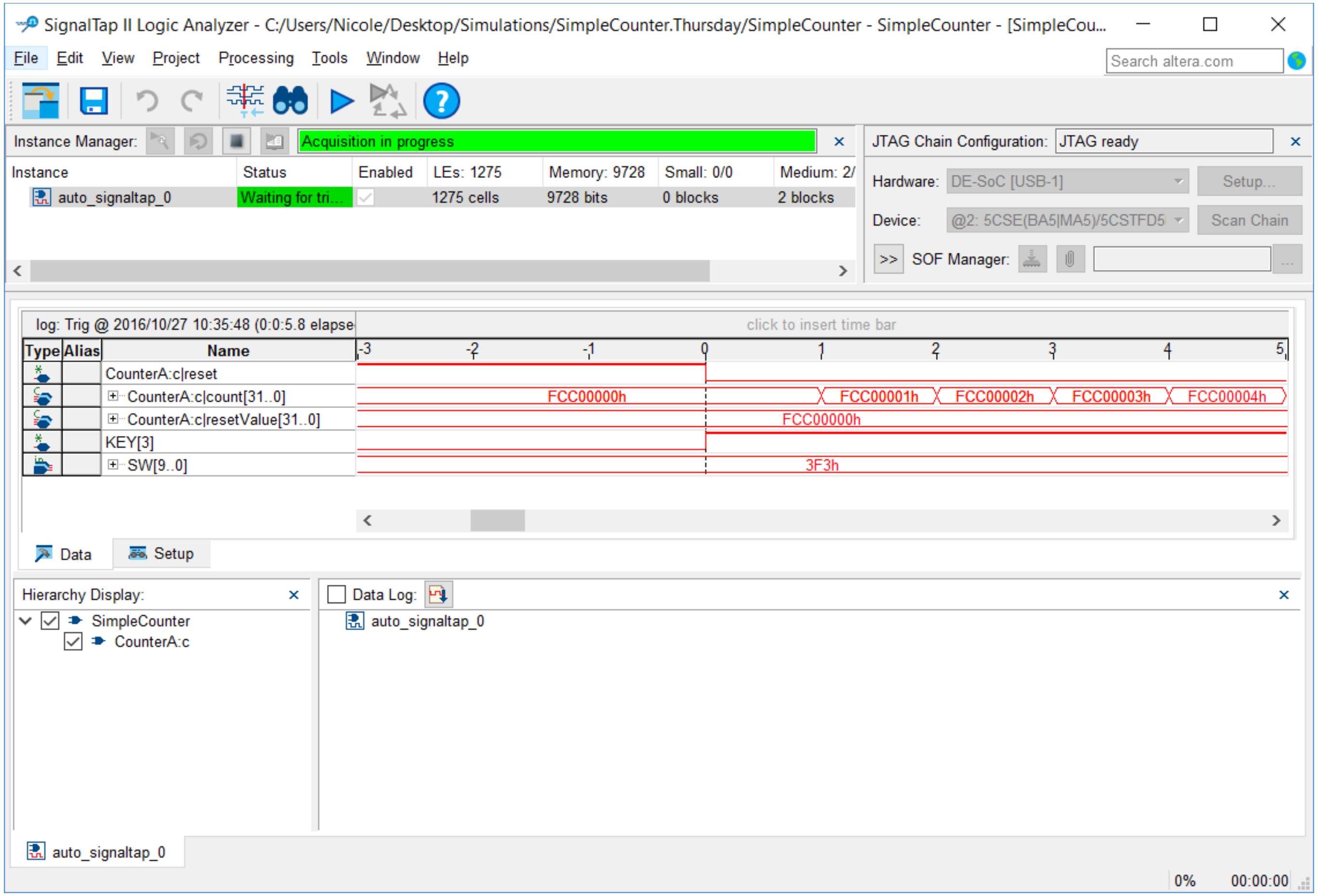
```
module CounterA(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule
```

In this simple wrapper for the DE1-SoC board, the high-order 10 bits of the counter are tied to the LEDs, the reset value is tied to the switches and the reset button to `KEY[ 3 ]`.

```
module SimpleCounter(
      input  CLOCK_50,
      input  [ 3:0 ] KEY,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   wire reset = ~KEY[ 3 ];
   wire [ 31:0 ] count;

   assign LEDR = count[ 31:22 ];
   CounterA c ( CLOCK_50, reset, { SW, 22'b0 }, count );

endmodule
```

On-chip logic analyzer that can be compiled onto the FPGA along with your design.

SignalTap II



SignalTap II Logic Analyzer - C:/Users/Nicole/Desktop/Simulations/SimpleCounter.Thursday/SimpleCounter - SimpleCounter - [SimpleCou...

Instance Manager: Acquisition in progress

Instance	Status	Enabled	LEs: 1275	Memory: 9728	Small: 0/0	Medium: 2/
auto_signaltap_0	Waiting for tri...	✓	1275 cells	9728 bits	0 blocks	2 blocks

JTAG Chain Configuration: JTAG ready

Hardware: DE-SoC [USB-1]

Device: @2: 5CSE(BA5|MA5)/5CSTFD5

log: Trig @ 2016/10/27 10:35:48 (0:0:5.8 elapse

Type	Alias	Name
		CounterA:c|reset
		⊞ CounterA:c|count[31..0]
		⊞ CounterA:c|resetValue[31..0]
		KEY[3]
		⊞ SW[9..0]

Hierarchy Display:

- ☑ SimpleCounter
  - ☑ CounterA:c

□ Data Log:

auto_signaltap_0