EE2A Digital Systems

Asynchronous Finite State Machine Implementation

Electronic Lock

1.0 Objectives

The primary objective of this experiment is to implement a sophisticated asynchronous digital circuit on a programmable logic device. Students will be required to design and build an asynchronous finite state machine that will act as a five-digit electronic lock.

This experiment should take about two weeks. This can only be achieved when adequate preparatory work is undertaken before the laboratory class. The preparatory work for this experiment will typically take between three and nine hours.

2.0 Learning Outcomes

Deliverables:

i) A completed design for a five-digit electronic lock including state diagrams and transition equations suitable for implementation on a sum-of-products device.

ii) A working circuit and thorough simulation and verification procedures targeted as written evidence during legal proceedings.

iii) A log-book showing design decisions and testing methodologies.

What will be assessed:

i) The design of an asynchronous finite state machine with adequate commenting of the different stages.

ii) Evidence that the student has consulted relevant text books related to asynchronous finite state machine design.

iii) Evidence that the student understands the concept of race hazards, single-bit code changes and the transition through unused states.

iv) A working lock that is "Teaching Assistant and lecturer proof".

v) The adequacy of the log-book as a communications medium including preparatory work and evidence that the log-book is written up as the experiment progresses.

vi) The ability of the log-book to convey testing and verification procedures in a professional manner.

vii) The appropriateness of the printed design material originating from the CAD software to satisfy third-parties in a legal dispute.

3.0 Activities

3.1 Experiment - Implementation of a digital lock

The system diagram for the asynchronous electronic lock is shown in Figure 1. A five-digit code will be entered on a simulated keypad. This keypad will be connected via ten active-high signals to an asynchronous finite state machine. When the correct sequence has been detected, the solenoid door bolt locking the door will be retracted by a signal, *Z*. The operator will be assumed to open the door and this operation will be detected by a switch, *Dr* (causing a return to the idling state). When the lock is first powered-up it will be forced into a safe idling state by the signal *Reset*. The asynchronous finite state machine will move through states encoded by three bits, *Qa*, *Qb* and *Qc*.

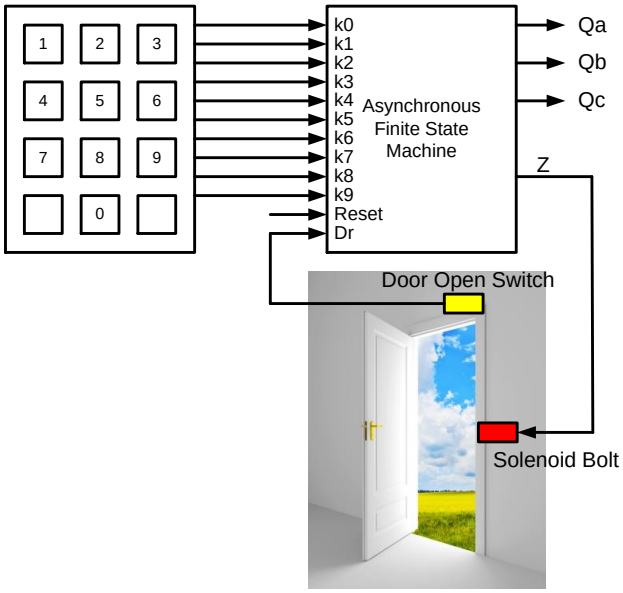


Figure 1. System Diagram

The code sequence for each of the groups numbers defined for the first EE2A1 experiment will be as follows:

Group Number:	Code Sequence:	Group Number:	Code Sequence:	Group Number:	Code Sequence:
1	1,3,5,7,8	12	3,6,7,2,9	23	5,8,7,2,5
2	2,4,6,7,9	13	8,4,7,0,3	24	6,9,0,3,6
3	8,3,5,7,6	14	9,5,7,1,4	25	7,1,8,4,7
4	9,4,6,7,0	15	6,8,7,2,5	26	7,2,9,5,7
5	5,8,6,7,1	16	7,0,9,3,6	27	7,3,6,8,7
6	6,9,7,0,2	17	0,1,4,8,7	28	1,6,7,9,2
7	7,0,8,1,3	18	0,2,5,9,8	29	2,7,0,3,8
8	0,1,9,2,4	19	1,5,7,1,8	30	3,7,1,4,9
9	0,1,3,8,5	20	2,6,7,2,9	31	8,4,0,2,5
10	1,4,7,9,0	21	8,3,7,0,3	32	9,5,1,3,6
11	2,5,7,1,9	22	9,4,7,1,4	33	6,8,2,4,7

It will be assumed that ten keys of a key-pad are simulated by eight switches and two push-buttons mounted on the Digilent Nexys 3 board.

The Digilent Nexys 3 has eight LEDs. These are connected to the XC6LX16-CS324 integrated circuit in the following manner to aid the assessors:

Digilent Net Name	EE2A Experiment Signal	XC6LX16-CS324 I/O Pin Number
Led<0>	*Qa* asynchronous FSM feedback path	U16
Led<1>	*Qb* asynchronous FSM feedback path	V16
Led<2>	*Qc* asynchronous FSM feedback path	U15
Led<3>	Not used in this experiment	V15
Led<4>	Z Output Logic (asynchronous)	M11
Led<5>	Not used in this experiment	N11
Led<6>	Not used in this experiment	R11
Led<7>	Not used in this experiment	T11

The Digilent Nexys 3 has eight switches and four push-buttons. These are connected to the XC6LX16-CS324 integrated circuit in the following manner:

Digilent Net Name	EE2A Experiment Signal	XC6LX16-CS324 I/O Pin Number
sw<0>	Simulated keyboard key, *k7*	T10
sw<1>	Simulated keyboard key, *k6*	T9
sw<2>	Simulated keyboard key, *k5*	V9
sw<3>	Simulated keyboard key, *k4*	M8
sw<4>	Simulated keyboard key, *k3*	N8
sw<5>	Simulated keyboard key, *k2*	U8
sw<6>	Simulated keyboard key, *k1*	V8
sw<7>	Simulated keyboard key, *k0*	T5
btn<0>	Simulated keyboard key, *k9*	B8
btn<1>	Door open switch, *Dr* (active-low)	A8
btn<2>	Simulated keyboard key, *k8*	C4
btn<3>	Power-up reset, *Reset* (active-low)	C9
btn<4>	Not used in this experiment	D9

This hardware mapping will be effected by a suitable '**.ucf*' hardware constraints file.

Design, construct, test, verify and write an effective log book for a five-digit electronic lock. Your design should incorporate features to eliminate key bounce whilst still rejecting attempts by hostile users to unlock the door. As an asynchronous design, no flip-flops may be included; you may only use AND, OR and NOT gates. Particular care should be taken in the avoidance of race hazards.

3.2 Presentation of Deliverables

The presentation of deliverables will consist of a demonstration of the asynchronous finite state machine to the lecturer, or Teaching Assistant, simulation validation (in the form that a lawyer can understand) and a discussion of the log book. Your log book will be analysed for communications and verification effectiveness during the Easter vacation.

4.0 Additional Resources

4.1 Asynchronous Finite State Machine Lock Guidelines

Your state diagram will probably include six or more states. This requires three state variables that could be conveniently implemented by using three output pins - perhaps *Qa*, *Qb* and *Qc*. The output used to control the door solenoid, *Z*, may be implemented using combinatorial output logic (λ logic).

It will be noted that with ten key inputs, one door switch, a reset input and three internal states, that a complete transition table would contain 32768 entries. A more compact form of the transition table will therefore be required. One way of achieving this is to pre-process some of the inputs. For example, we might end up with three different types of pre-processed functions:

k_7	represents key number seven pressed and no other key.
$\widehat{k_7}$	represent any key other than key seven, or a combination of keys pressed.
$\lceil 0 \rceil$	represents no keys pressed.

In this way the transition table might be reduced to perhaps eighteen entries. Do think very carefully about unexpected transitions as an unauthorised user attempts to open the lock.

There are obviously many ways of implementing the electronic lock. The end result should be perhaps three-to-nine hours of preparatory work and schematic entry followed by as little as two hours using the Digilent Nexys 3 board for testing the final product in the laboratory. Compare this with the complexity of the discrete logic implementation of the counter and hopefully you will be sold on the modern programmable logic techniques !

- The outputs of your equations will be functions of the outputs of all the other equations. As this is an asynchronous machine "race hazards" are bound to occur.
- The solution to this problem will not be fully covered in lectures but may be found in a number of text books.
- Substantial credit will be given to asynchronous designs that eliminate potential race hazards.
- Credit will also be given to those students who are able to communicate formal testing and verification methods in a precise and concise manner.
- Do not forget to staple into your log-book the source code, the (well-commented) simulation vectors and proof that the test vectors were accepted by the simulator.

An asynchronous finite state machine will contain nothing other than combinatorial logic. The equations for a single transition are therefore likely to be of the form:

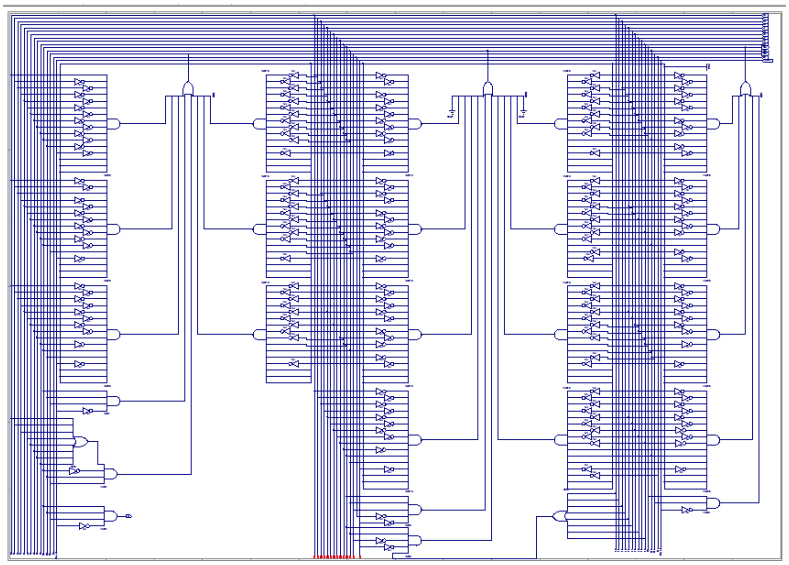
```
Qa = !k0 & k1 & !k2 & !k3 & k4 & !k5 & !k6 & !k7 & !k8 & !k9 & !Qa & Qb & !Qc & !Reset;
```

Many similar terms will be combined using a OR gate to generate each feedback path. Controlling the flow of the machine following a fraudulent entry attempts might be challenging.

4.2 Implementation Guidelines

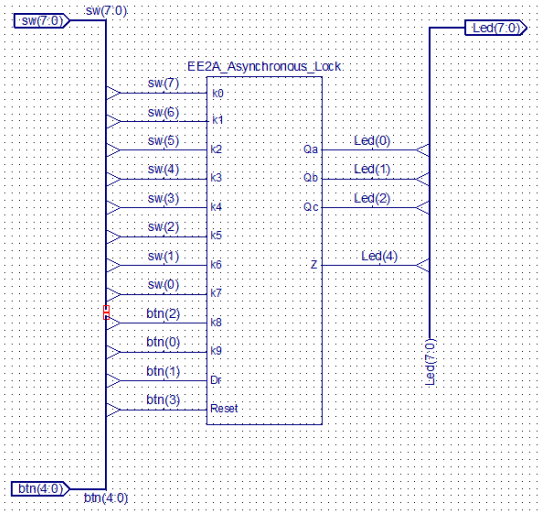
Design your asynchronous finite state machine using a state diagram followed by a direct generation of a transition table. Extract all the prime implicants and write down the transition and output logic equations. From these equations, generate a list of the number and type of gates required – paying particular attention to the number of inputs on each gate (this will save you a significant amount of time as you develop your schematic circuit entry).

Start a new project using the Xilinx ISE using the notes provided in the Appendix. Open a new schematic entry sheet and change the size to the largest available. A sensible starting point is the route all the inputs and feedback paths around the sheet in the shape of an 'E' – this allows you to place a large number of gates with easy access to all the signals. By the time you have completed this experiment, the circuit might look something like (warning – this one has errors):



Most of the AND gates requiring keyboard input signals will be of type '*AND16*'.

Convert the design to a symbol and add this to a new top-level schematic. Add I/0 markers and add nets to the bus entries and exits to match the hardware specification given earlier.



Download the *ucf* constraint file from Canvas and enable the signals that will be used in this experiment.

Generate a VHDL test bed file for thoroughly simulating the design. It would be surprising if this was less than 300 lines by the time you had completed a *vhd* file that would satisfy the lawyers.

Synthesise and implement the design using the green 'play' arrow. Generate the programming file and download this to the Nexys 3 board using '*Configure Target Device*'. Ensure that the hardware functions to specification and think carefully how you will convince someone else that it does so.

Logbook Marking Scheme

The following presents a guide as to the minimum expectations of the logbook. This must not be taken as a prescriptive list – you must develop your own effective documentation techniques.

Item	Mark
• An experimental 'Title' and date the work commenced (for search indexing).	1%
• A brief 'Introduction' section explaining what are the highest-level reasons for committing resources to this activity.	1%
• Probably some method of defining the specification.	1%
• State diagram, transition table, characteristic equations, minimization of multi-variable equations, extraction of prime implicants, ability to maintain single-bit code transitions under fraud conditions, handling of race hazards – all with excellent, concise explanations.	25%
• Schematic circuit entry of complete circuit (printed version enclosed in log book).	15%
• High-level definition of what needs to be tested in order to validate the design.	5%
• Well-commented test bench file to simulate operation of the machine (printed version enclosed in log book). • Phase 1: Testing of normal operation. • Phase 2: Testing of fraudulent operation.	20%
• Material showing that the differences between *simulation* and *verification* are understood. Generation of material that will defeat the most obstinate lawyer – printed paper-trails.	15%
• Practical (hardware) testing and verification strategy.	5%
• Evidence that the claimed deliverables actually existed (photographs, printouts, etc.).	1%
• Brief 'Conclusions' section.	1%
• If necessary, 'Appendices' to include anything you should have included earlier.	-
• The obvious things that are missing from the above list!	10%

Appendix 1 – Example Procedure

To ease the hardware task you will be issued with a ready-built FPGA board, a Digilent Nexys 3, see Figure 2. This device contains a Xilinx Spartan 6 XC6LX16-CS324. It has a two-channel USB connection for JTAG and serial communications, may be powered from a power connector (5V) or USB and has 72 I/O lines. This also device contains a 4 character, 7 segment display, 8 LEDs, 8 slide switches and 5 push buttons.

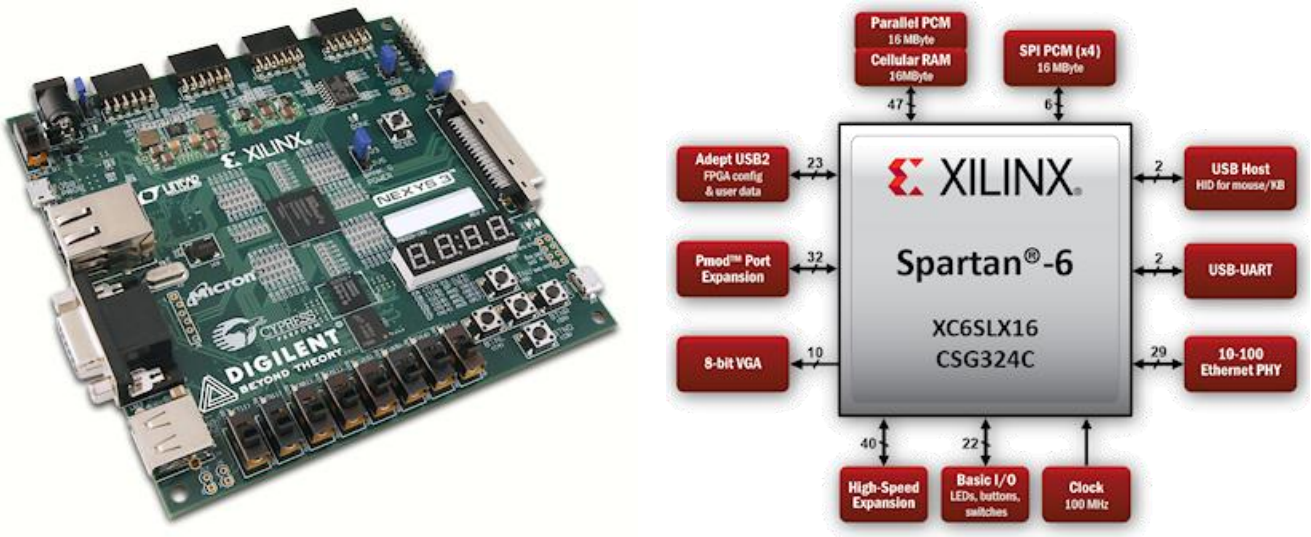


Figure 2. Digilent Nexys 3 board

The Spartan 6 Integrated Circuit

You will be using the Xilinx Spartan 6 (XC6LX16-CS324) integrated circuit. This has a very large number of system gates arranged as 18,224 configurable logic block flip-flops, 576 K bits of block RAM, 32 multipliers and 72 IO lines will be available to you. The typical architecture of the Spartan is shown in Figure 3. You will be using Input/Output Blocks (IOBs) and Configurable Logic Blocks (CLBs) – it is possible that you will also use the Digital Clock Manager (DCM).

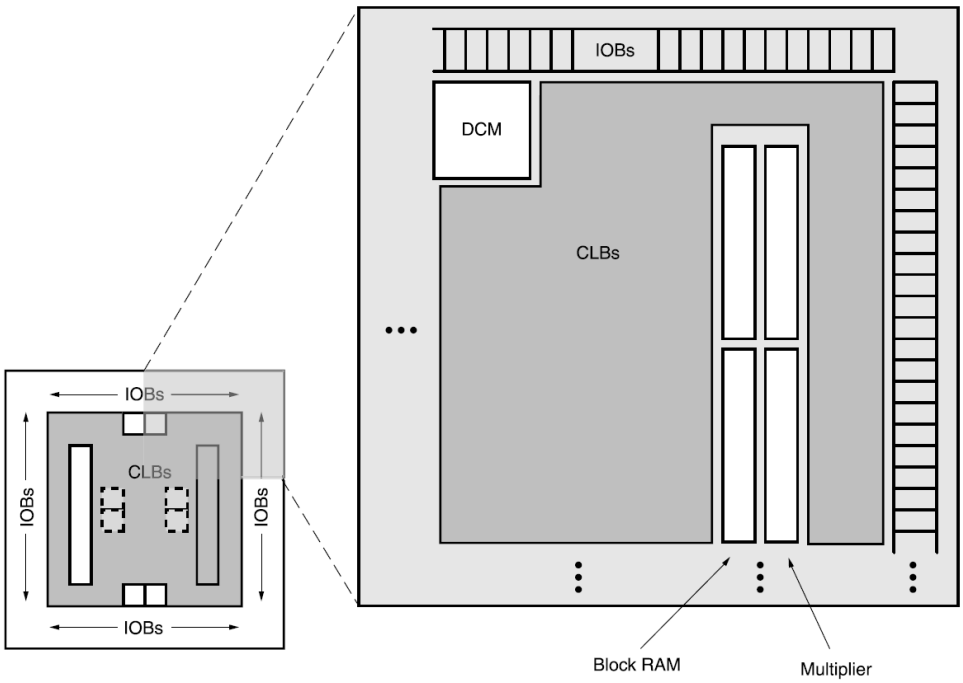


Figure 3: Xilinx Spartan Architecture

Providing second year students with a bare Spartan 6 integrated circuit and expected them to engineer it into a working demonstration might be considered ‘over-ambitious’ by some. Hence you will be provided with a Digilent Nexys 3 board to ease the soldering, programming and power supply tasks.

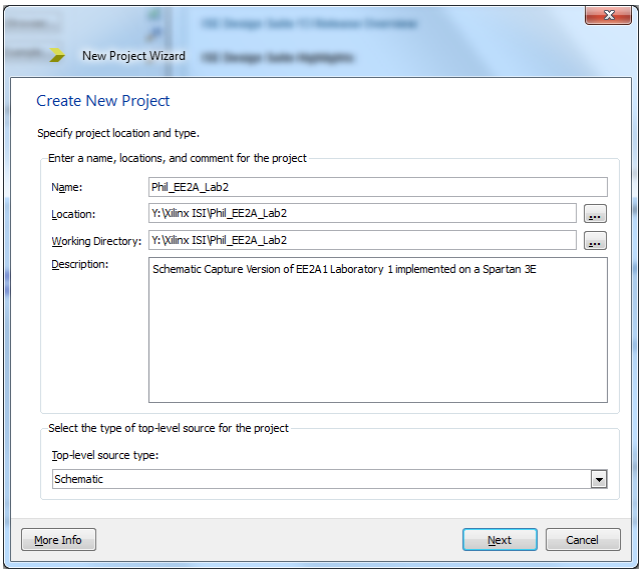
Starting the Xilinx ISE Design Package

The Xilinx ISE design system is based on an integrated environment that includes a schematic capture package as well as a VHDL compiler package. From Windows :

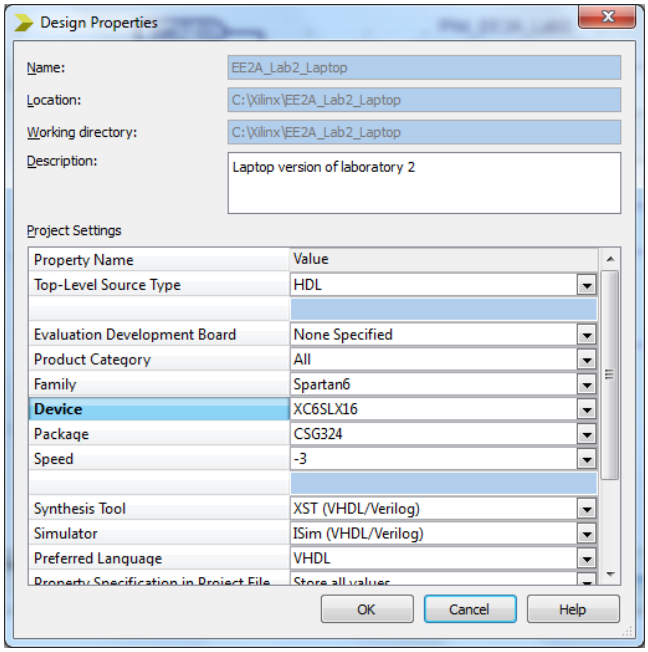
(a) Enter the design system via the menus '*Start > All Programs > Xilinx ISE Design Suite 13.4 > ISE Design Tools > 64-bit Project Navigator*'.

(b) Open up a new FPGA Project via the menus '*File > New Project*' or by using the ‘*New Project*’ icon under the project commands tile.

(c) Enter a name for the project e.g. ‘*Phil_EE2A_Lab6*’ and use the browse dots to the right of ‘Location’ to navigate to somewhere on your C: drive such as the desktop. The Xilimx package does not appear able to use your networked drives, but you must save your files to your networked drive at the end of every session. Xilinx does not allow any spaces within a file name, so use underscores instead. Add a suitable description within the project description text box. Select ‘*schematic*’ under the ‘Top-level source type. Finally click ‘*Next*’. **If a ‘read only’ error message appears at any point, log your computer off and back on again.**

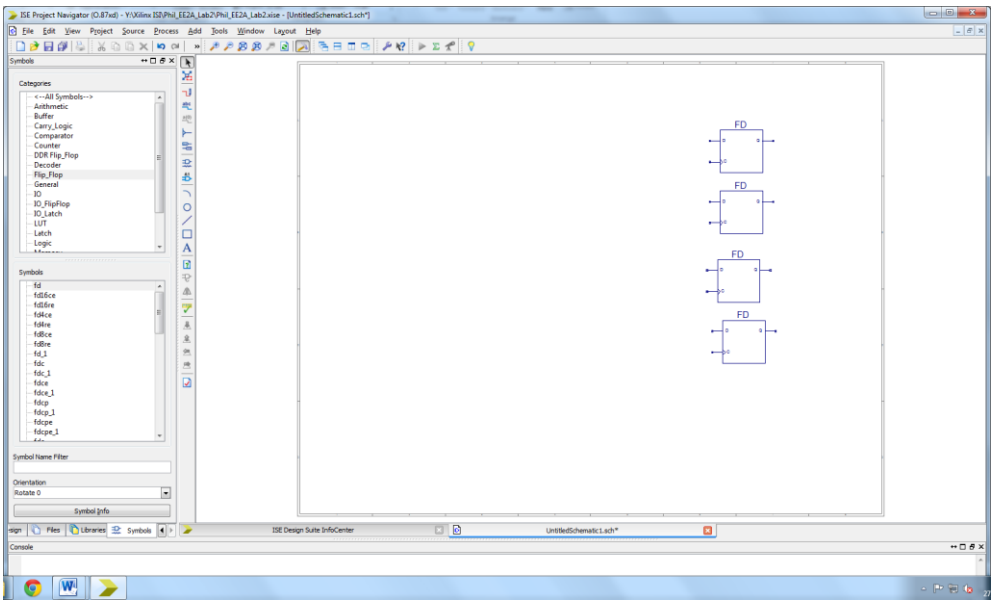


(d) The ‘Design Properties’ window will now appear. Select ‘*Family > Spartan6*’, ‘*Device > XC6SLX16*’, ‘*Package > CSG324*’, ‘*Speed > -3*’ as shown in the screen shot below.

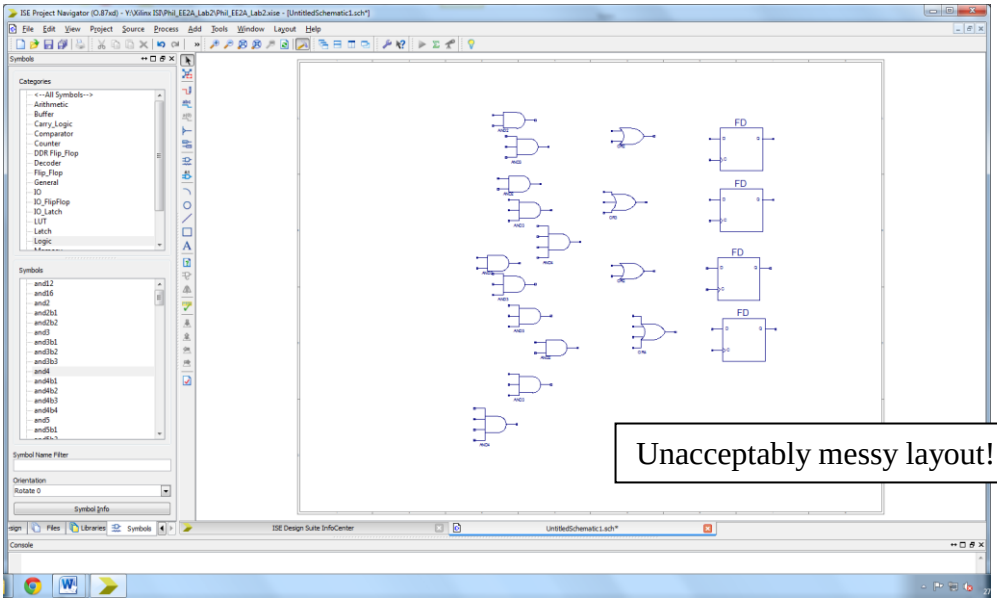


(e) Click '*File > New > Schematic*'.

(f) For demonstration purposes, the first thing we will add is four D-type flip-flops on the right-hand side of the page. Click '*Add > Symbol > Flip-Flop > FD*' or cntrl+m. Both the '*Flip-Flop*' and '*FD*' appear in the panes on the left side of the screen. Use multiple mouse clicks to add the four flip-flops to the sheet. You will notice that there is a '*Symbol Info*' button to the lower-left of the screen – this gives you invaluable information about the functionality of the logic block you are about to add. **You will not need any flip-flops in your asynchronous design.**

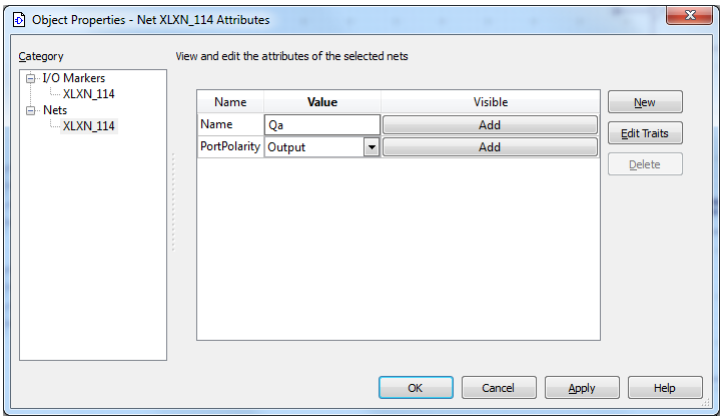


(g) Before continuing further, ensure that your log book contains an accurate record of **all** the gates required (e.g. the number of AND and OR gates required and the number of inputs associated with each gate). Use a large number of inverters connected to the inputs of the AND gates – lay the circuit out exactly as you sums-of-products Boolean equations dictate. **Trust the lecturer: Failure to have a list of the number of inputs required to every gate when using the schematic capture editor will add several hours to your laboratory experiment.** Click '*Add > Symbol > Logic > OR2..9*' to add your four OR gates. Click '*Add > Symbol > Logic > AND2..16*' to add your AND gates. **By this stage, your gates are likely to be poorly and irregularly placed – you should now spend ten minutes tidying the circuit by moving the gates with the mouse, such that they form a regularly aligned and elegant layout.**

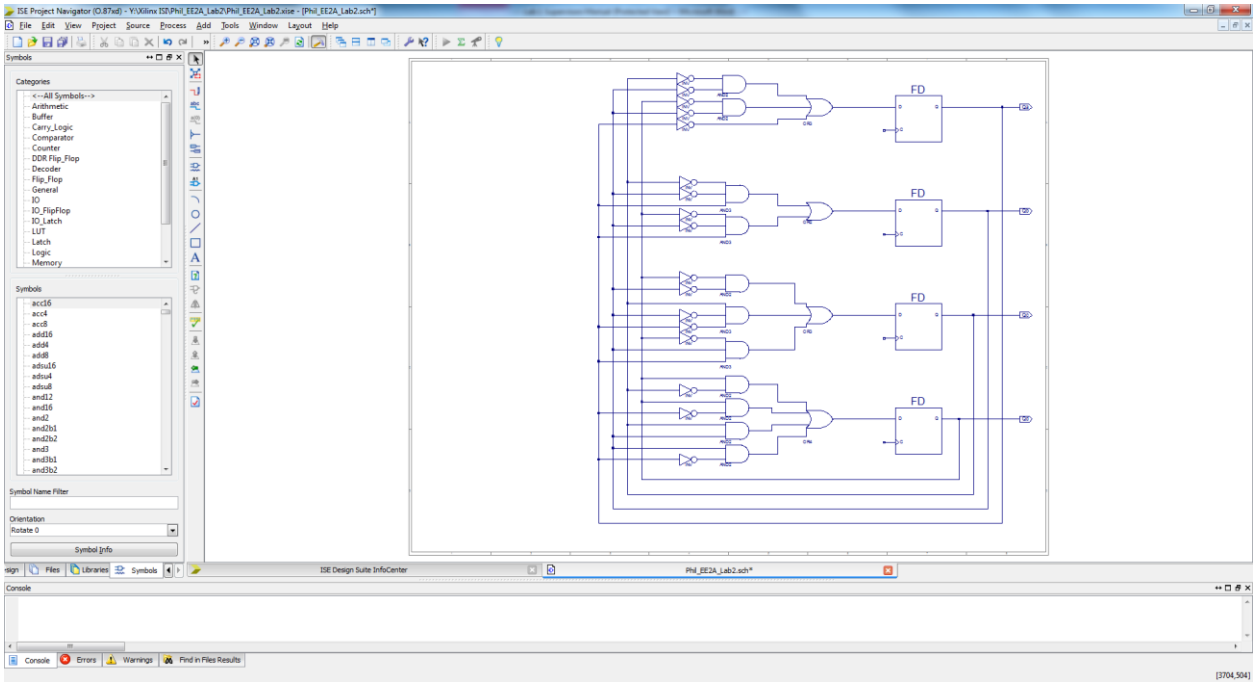


(g) Now use '*Add* > *Symbol* > *Logic* > *inv*' to add any inverters required in the circuit. Place the interconnections using '*Add* > *Wire*' – it is best to select the radio button '*line segments between the points to indicate*' in order to achieve an elegant circuit. It is best to click on the '*Options*' pane and then '*Select the line segment*' to allow you to delete small portions of inter-connects.

(f) Use '*Add* > *I/O Marker*' and click the radio button to add an output marker to place an output pin on the output of each flip-plop. Double click each marker and then click on '*Nets*' to change the name to that used in your design (e.g. *Qa*).



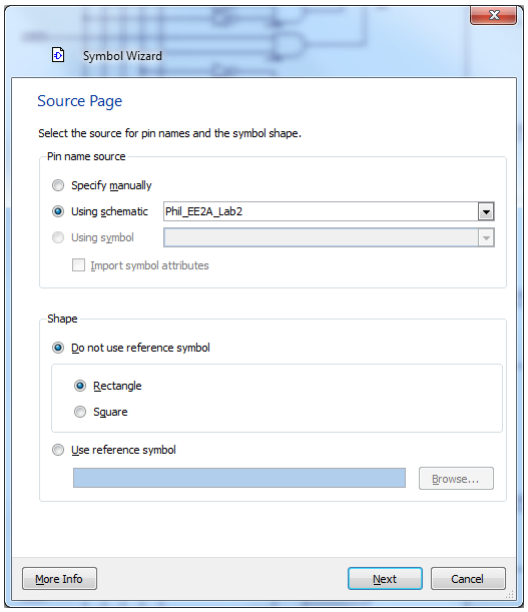
After a few more minutes of effort you will end up with a schematic circuit diagram looking something like (but implementing a synchronous rather than a synchronous machine):



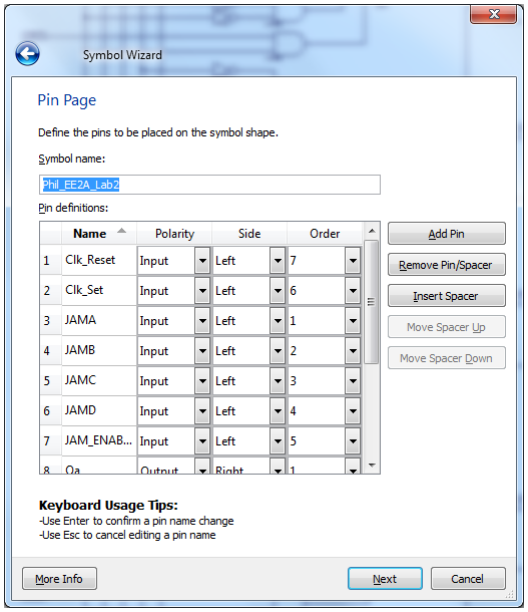
Hierarchical vs. Flat Designs

The work you have completed so far is known as a flat design – i.e. it is entered onto a single sheet. Single sheets are useful for very simple designs. However, after graduation you will be working on designs that are too big to fit on a single sheet. The golden rule of all high-quality designs is that the documentation should fit onto a single sheet. This implies that single sheets become 'modules' in a much larger design – known as a hierarchical design.

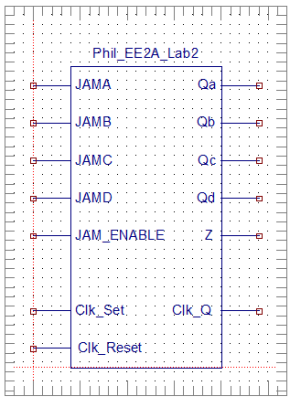
To make your single sheet design into a hierarchical symbol click '*Tools* > *Symbol Wizard* > *Pin name source*' and select your schematic file name from the pull-down list.



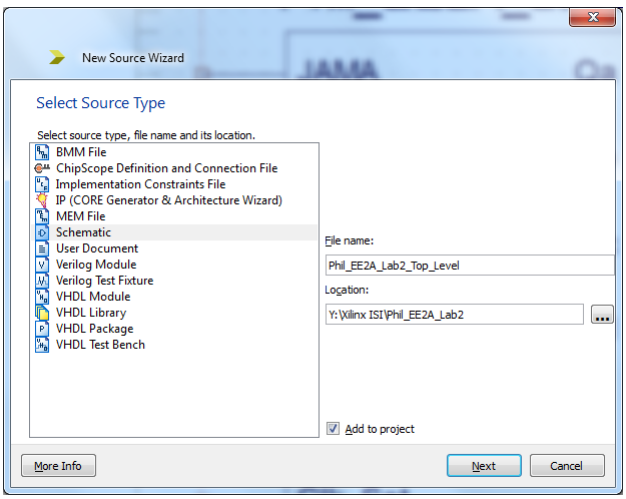
Click '*Next*'. Now ensure that inputs appear on the left and outputs appear on the right. Change the ordering such that inputs and outputs are aligned and use the '*Insert Spacer*' button to add spaces between signals of different functionality grouping.



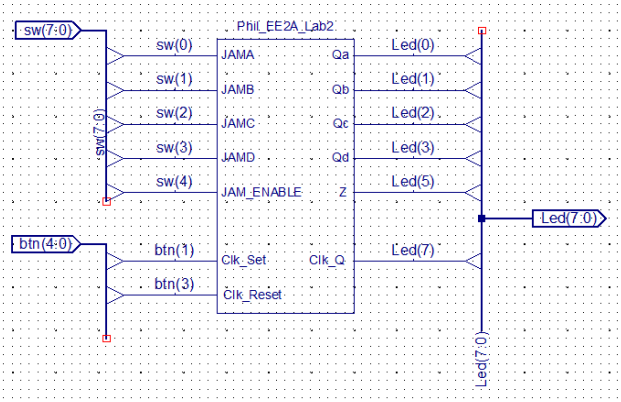
Click '*Next*' and '*Next*' changing anything that you think will add to the elegance of your documentation. Eventually you will end up with a symbol looking something like (but only for a synchronous implementation of Experiment 1 – yours will have very different variable names):



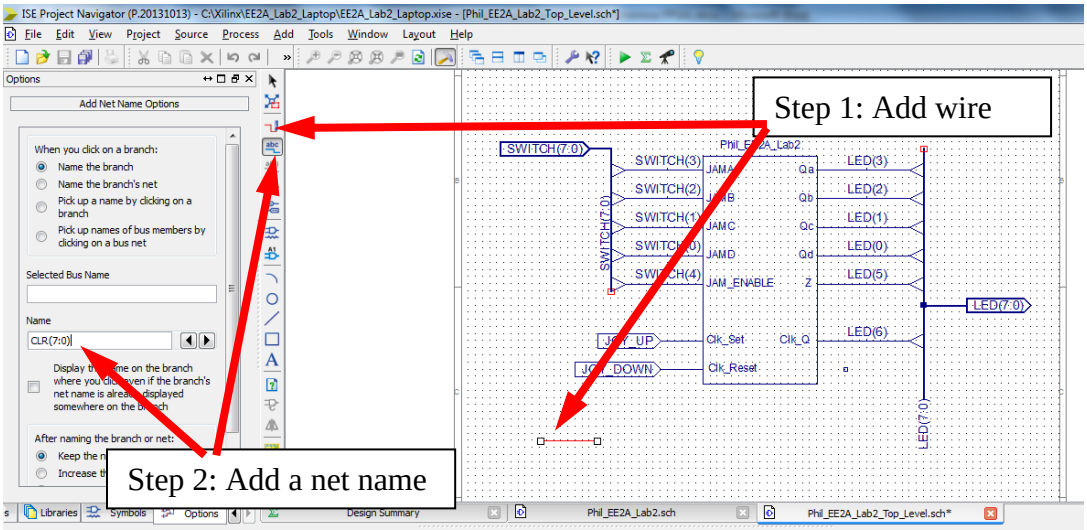
Now the symbol can be used in as many designs as you wish. Create a master schematic sheet by clicking '*Project* > *New source* > *Schematic*' and adding an appropriate file name (without spaces).



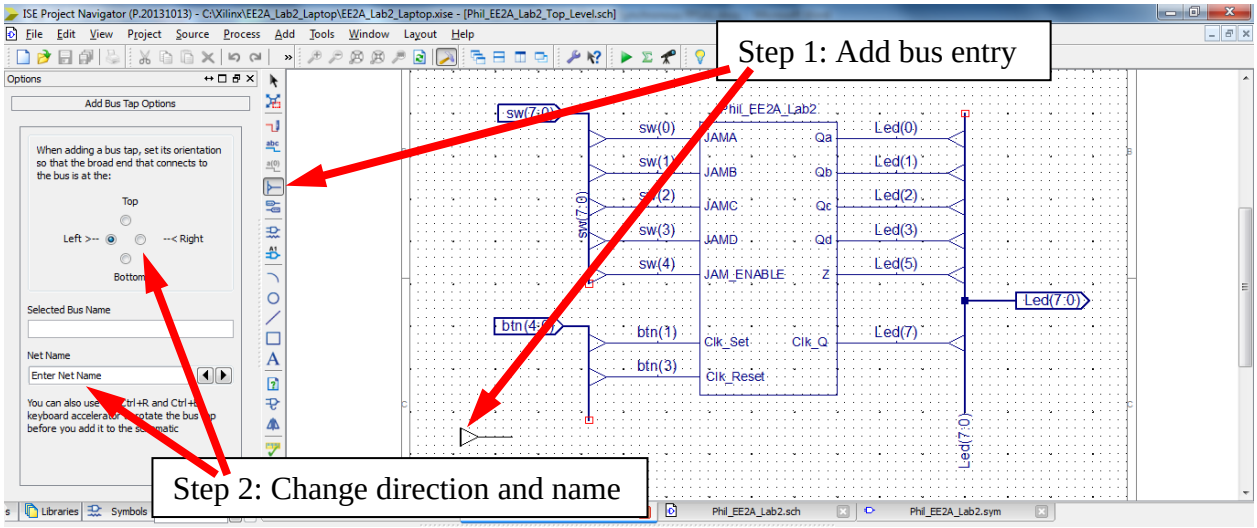
A blank schematic sheet will appear. Click the '*symbol*' icon and you will notice that the symbol you have created appears at the top of the left-hand selection pane. Add your new symbol to the sheet. The real reason that you have performed this function is to allow a mapping from a logic variable (e.g. *Qa*) to a hardware pin definition (e.g. *Led(3)*).



Now this stage also requires you to learn about busses (groups of wires). The designer of the Digilent Nexys 3 constraint file decided to group the eight LEDs together as a bus (called Led(7:0) in this example) rather than giving each LED an individual name (e.g. Led_A, Led_B, etc.). To add a bus to a drawing, first add a wire where you want the bus to run. Then add a '*net name*' by clicking on the icon and entering a bus name such as Led<7:0> or sw<7:0> – use the mouse to place this name on the wire, thus transforming it into a bus.



The next stage is to add taps onto the bus and connecting wires to each of the pins on your symbol. Again use the '*Add Net Name*' icon and entry box on the left to name each bus entry and exit with names such as Led<5> to match the hardware. You will need to experiment with the direction of the bus entry, or exit.



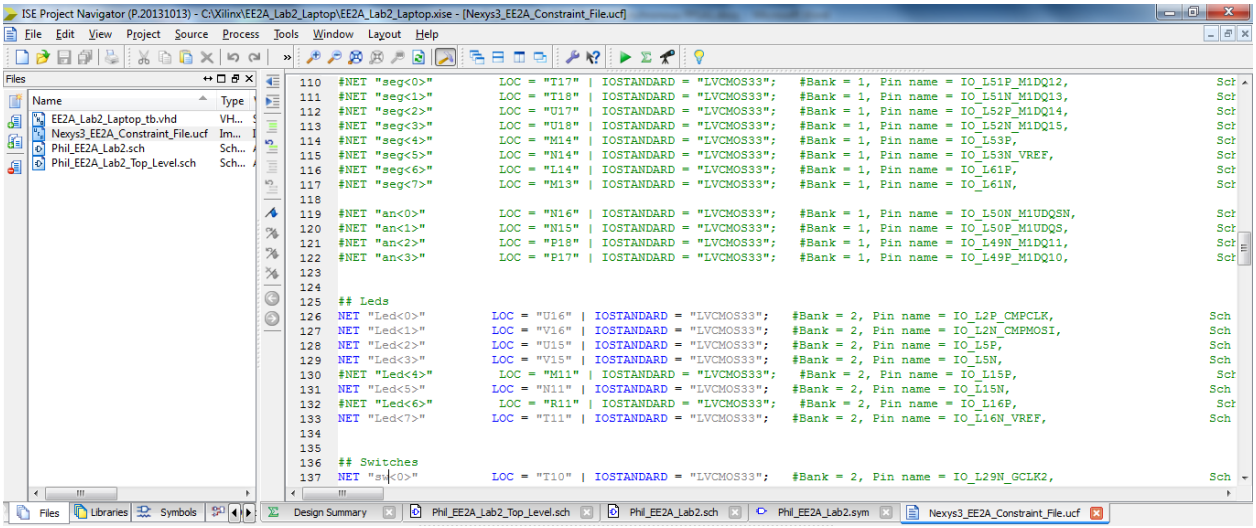
Finally, use the '*Add I/O Marker*' icon to add the pins that exist on the hardware (e.g. Led<7:0>, sw<7:0>, etc.)

What you have just implemented is a 'bottom-up' hierarchical design – starting with the low-level technical detail and fusing the design into a simple high-level design. Your Technical Director, or you in twenty years' time, will generate 'top-down' designs – they will assume that the big-picture takes precedence over the technical detail. In reality, all engineering projects use both 'top-down' and 'bottom-up' elements at the same time – ideally fusing seamlessly somewhere in the middle.

Constraint Files

At some point a mapping is required between the hardware pins available and the schematic logic design. This is achieved by using a hardware constraint file (*.ucf). Versions have been written for the Digilent Nexys 3 board and a downloadable copy is available on Canvas. Click '*Project > Add Source > Nexys3_Master.ucf*' – this file should have been obtained from Canvas and placed in your working directory. You may open this file to see the names of the nets available to you.

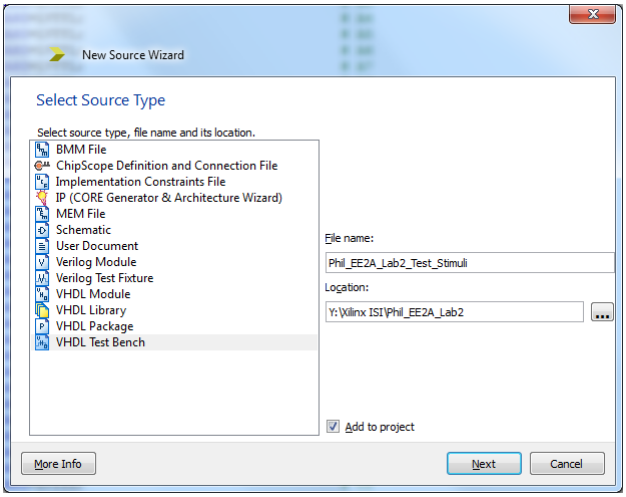
This file contains the name of every hardware pin available on the Digilent Nexys 3 and many will not be required during this experiment. Thus all pins are initially deactivated by placing a comment character '#' at the start of the lines. You need to remove the leading # on the signals you require.



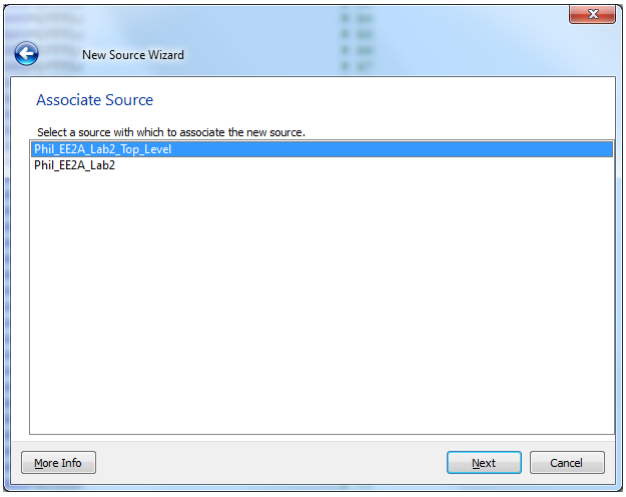
Simulating the Design

This is the most important, useful and difficult part of the experiment to perform. A simulator requires both a description of the equivalent circuit and a set of stimulus simulation input values. The ISE design package automatically generates a Hardware Description Language (HDL) file every time the schematic is saved. For this experiment you are going to use a text file based input of stimulus test vectors – you will find that this is the 'industry preferred' method of simulating large designs and is a more defendable method of protecting yourself in a court-of-law against negligence claims.

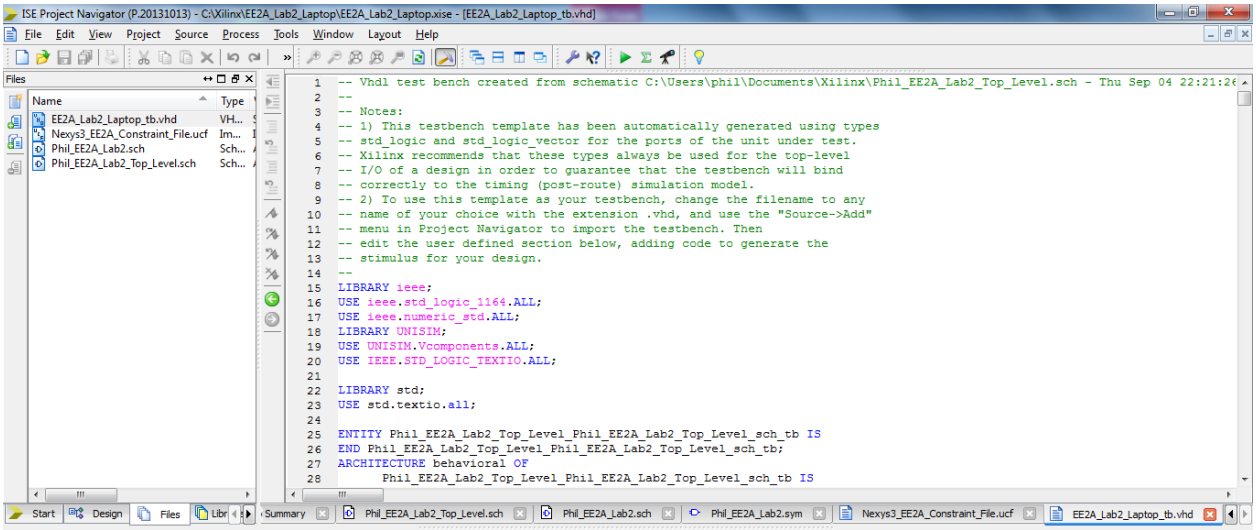
To add a stimulus file click '*Project > New Source > VHDL Test Bench*' and add a suitable file name (that does not include spaces).



Click '*Next*' and highlight the top level of your design. Click '*Next*' and '*Finish*'.

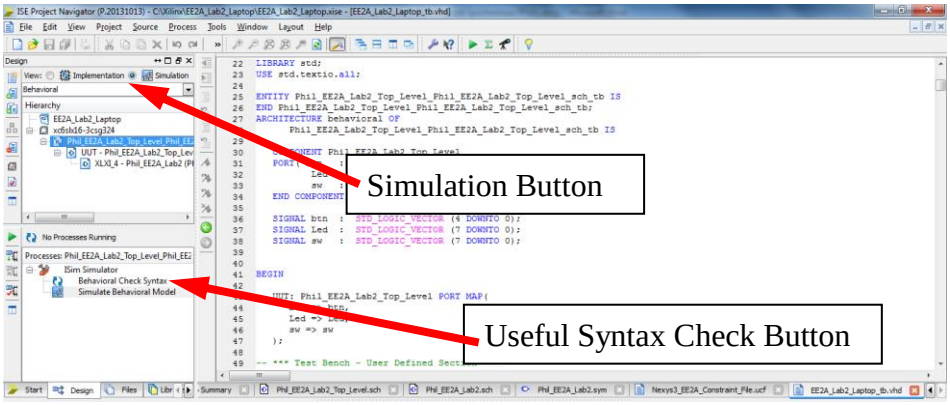


A basic VHDL test bench template appears – this should have been populated with the signals used in the top-level of the design hierarchy.



You now have the task of teaching yourself enough VHDL in a few minutes in order to be able to thoroughly simulate the design. It is suggested that you write simple linear, procedural code (avoiding loops, conditional statements, sub-routines, etc.) and this stage.

Make sure you click on the '*Simulation*' button.



Next, decide how you are going to test every transition on the state diagram and prove that you have done so to a lawyer (the critical verification process). You will then waterfall down to the next level. Finally, what operations are required to test fraudulent attempts at access?

There are very few VHDL commands you need to know about.

```
btn(1) <= '1'; -- You can never include enough comments for a lawyer
```

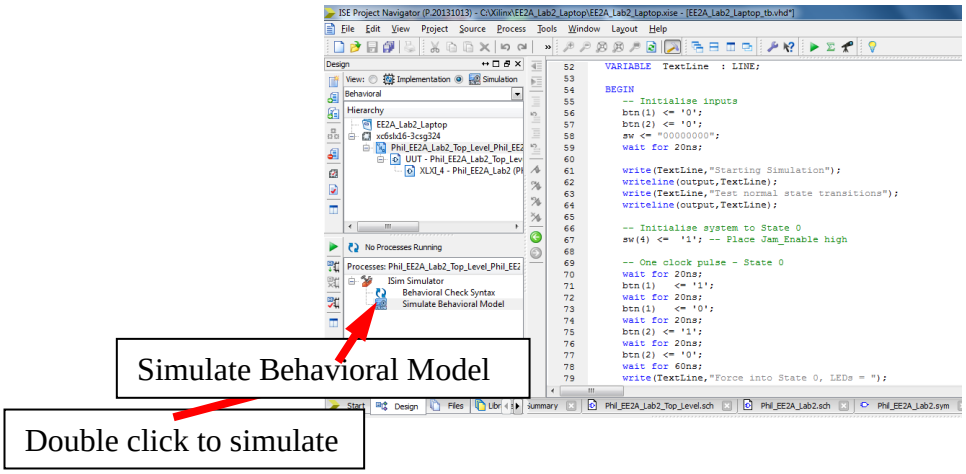
Now you know about assigning a logical value to a signal and adding a comment.

```
wait for 20ns;
```

Now you know about adding time delays between procedural actions.

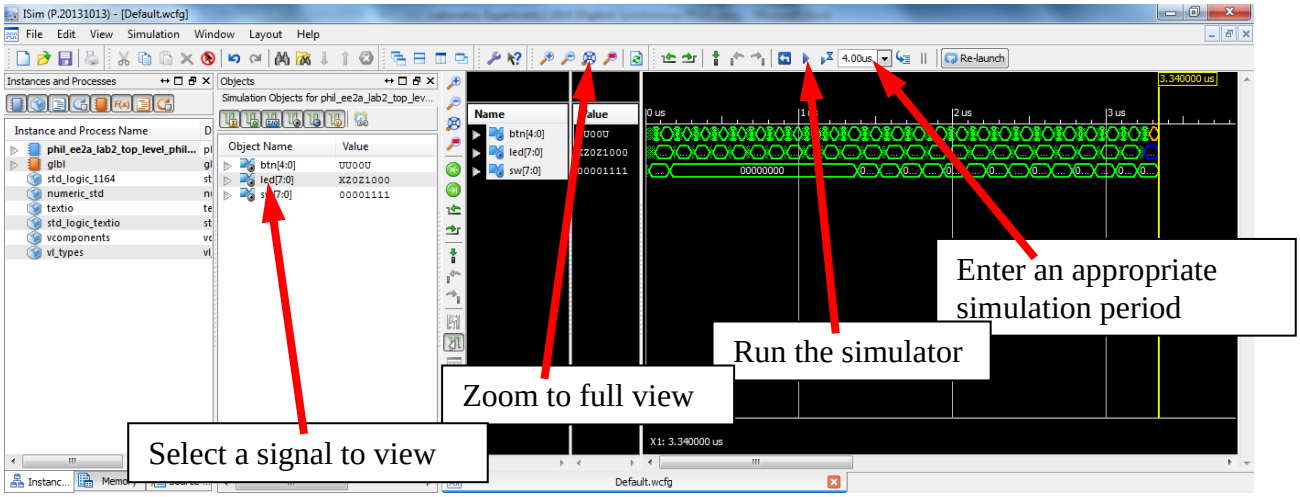
Helpful Hint: Only add perhaps five lines of code at a time and then sort out the simulation errors introduced.

Check the syntax of you *vhd* file by double clicking on the '*Behavioral Check Syntax*' icon. To simulate the design, double click on the '*Simulate Behavioral Model*' in the lower-left pane. Make sure that you do not attempt to run two instances of the simulator at the same time – logging off your computer will be the only way of recovering if you attempt this.



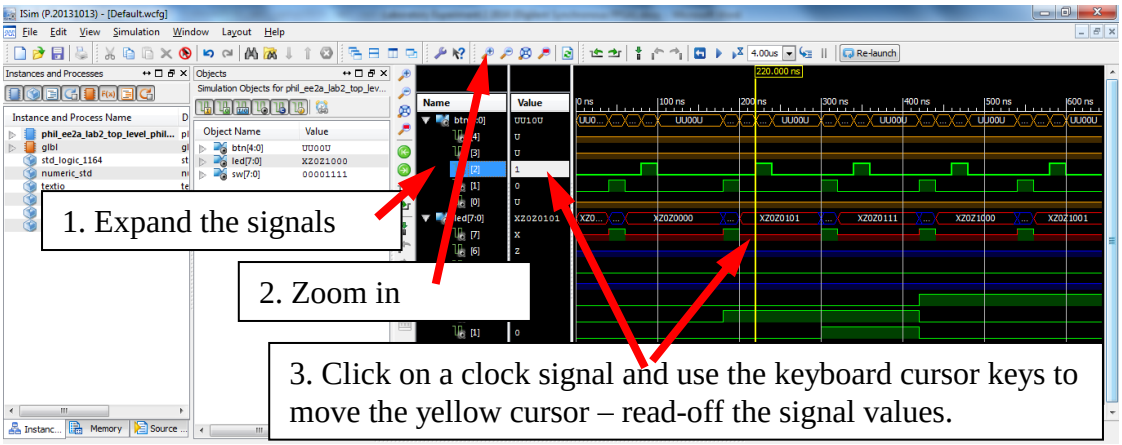
You are almost certainly going to obtain a number of error messages – read these, comprehend what they are telling you (search using Google), fix the problem and try again. 'Warnings' can usually be ignored, but 'Errors' must be rectified before continuing. By the time you have exhaustedly tested the design, you will probably have entered of the order of 1000+ lines in the simulation input file.

On double clicking on the '*Simulate Behavioral Model*' icon, the ISim window will open. Enter an appropriate period over which to simulate the circuit (typically 4 us). Run the simulator by clicking the blue 'play' arrow. Click on the '*Zoom to full view*' icon. Click on the data you would like to display (e.g. Led[7:0]) – you can now move the yellow cursor with the left and right keyboard arrows.



Stop – Think. By this point, you have obtained a very pretty timing diagram of the output of your design. **Only you will be able to understand it – a lawyer will tear you apart! You need to find a better way!**

The first approach is to expand the signals, such that you can read off the data at the cursor.



Stop – Think. You have now obtained a slightly more useful ***simulation*** of your design. **You are still the only person able to understand it – a lawyer will still tear you apart! You still need to find a better way!**

Now you need to start thinking about ***verification*** rather than ***simulation***. This probably involves printed textual documents rather than graphical representations of waveforms. The functionality of VHDL may be extended by adding libraries for textual read and write operations.

You will notice that the vhd file already loads an IEEE library and 'uses' a number of components within this library. First add '`USE IEEE.STD_LOGIC_TEXTIO.ALL;`', then two additional lines to make use of components within the standard library.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- Add the following lines
USE IEEE.STD_LOGIC_TEXTIO.ALL;

LIBRARY std;
USE std.textio.all;
```

In VHDL we make use of a variable of type line. For example, a variable called '*TextLine*' may be defined immediately after the test bench process:

```
-- *** Test Bench - User Defined Section ***
```

```
   tb : PROCESS
      VARIABLE  TextLine  : LINE;
```

Textual data is appended to this variable and then flushed to the console, or a file. For example, a header may be generated by:

```
write(TextLine,"Starting Simulation");
writeline(output,TextLine);   --  Write the contents of TextLine to the console
writeline(output,TextLine);   --  Write a blank line to the console
write(TextLine,"Test normal state transitions");
writeline(output,TextLine);
```

We can now clock the finite state machine and display the results in textual form, e.g.

```
-- One clock pulse - State 5
wait for 20ns;
btn(1) <= '1';
wait for 20ns;
btn(1) <= '0';
wait for 20ns;
btn(2) <= '1';
wait for 20ns;
btn(2) <= '0';
wait for 60ns;
write(TextLine,"Clocked to State 5, LEDs = ");
write(TextLine,Led(0));   -- Append LED signals in order
write(TextLine,Led(1));
write(TextLine,Led(2));
write(TextLine,Led(3));
write(TextLine,"   Z = ");  -- Append the Z output signal
write(TextLine,Led(5));
writeline(output,TextLine);  -- Write Textline variable to the console
```

Thus we can get an output on the console such as:

```
Starting Simulation
Test normal state transitions
Force into State 0, LEDs = 0000 Z = 0
Clocked to State 5, LEDs = 1010 Z = 0
Clocked to State 7, LEDs = 1110 Z = 0
Clocked to State 8, LEDs = 0001 Z = 0
Clocked to State 9, LEDs = 1001 Z = 0
Clocked to State 11, LEDs = 1101 Z = 0
Clocked to State 12, LEDs = 0011 Z = 0
Clocked to State 13, LEDs = 1011 Z = 0
Clocked to State 3, LEDs = 1100 Z = 0
Clocked to State 4, LEDs = 0010 Z = 0
Clocked to State 5, LEDs = 1010 Z = 0

Test not-allowed state transitions
Force into State 0, LEDs = 0000 Z = 0
Clocked to State 5, LEDs = 1010 Z = 0

Force into State 1, LEDs = 0001 Z = 0
Clocked to State 7, LEDs = 1001 Z = 0
```

You can obviously improve on this approach significantly! Instead of writing to the console, it will probably appease a lawyer more if the data is written to a date-stamped file. First open a file with:

```
file OutputFile : text is out "EE2A_Lab6_Verification_File.txt";  --declare output file
```

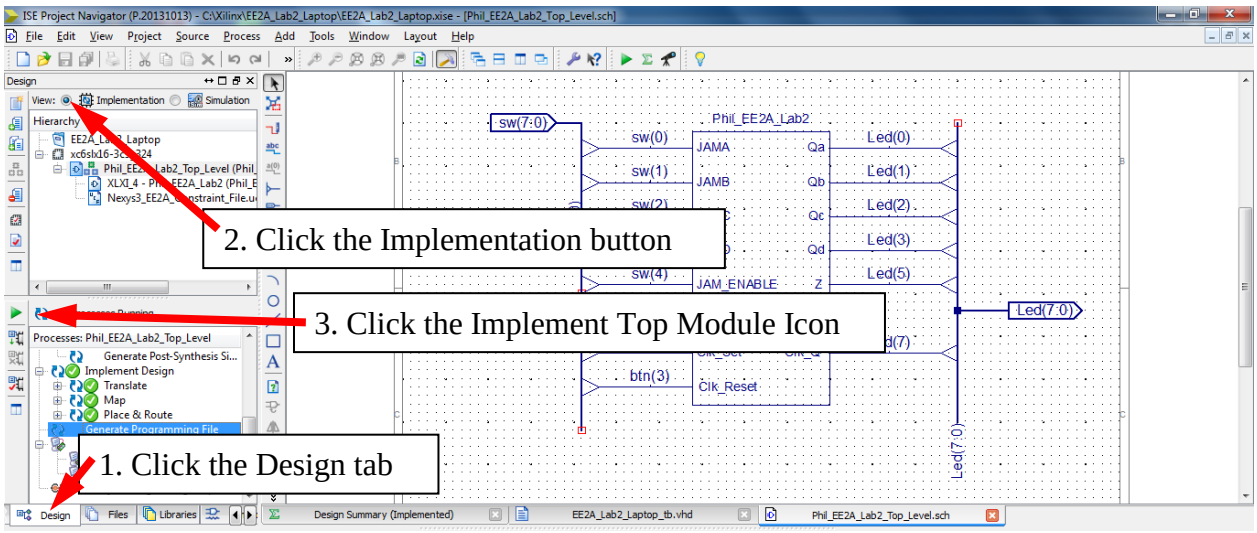
Now change every '*writeline*' to contain the file identifier '*OutputFile*', e.g.

```
writeline(OutputFile,TextLine);
```

You should now be able to view the results of your exhaustive efforts in a text file *'EE2A_Lab6_Verification_File.txt*'.

Fitting the Design and Programming the Device

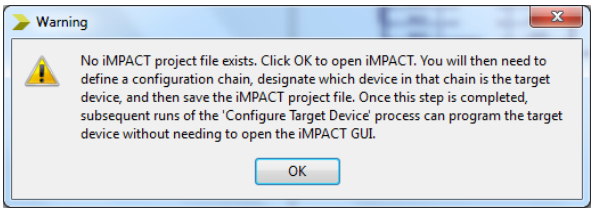
Having proved that the behaviour of the device has been adequately verified, you now need to 'fit' the design onto the silicon. Return to the '*Design*' tab and click the '*Implementation*' button. Make sure the highest level of your design is highlighted within the '*Hierarchy*' pane. Finally, click the green '*Implement Top Module*' icon.



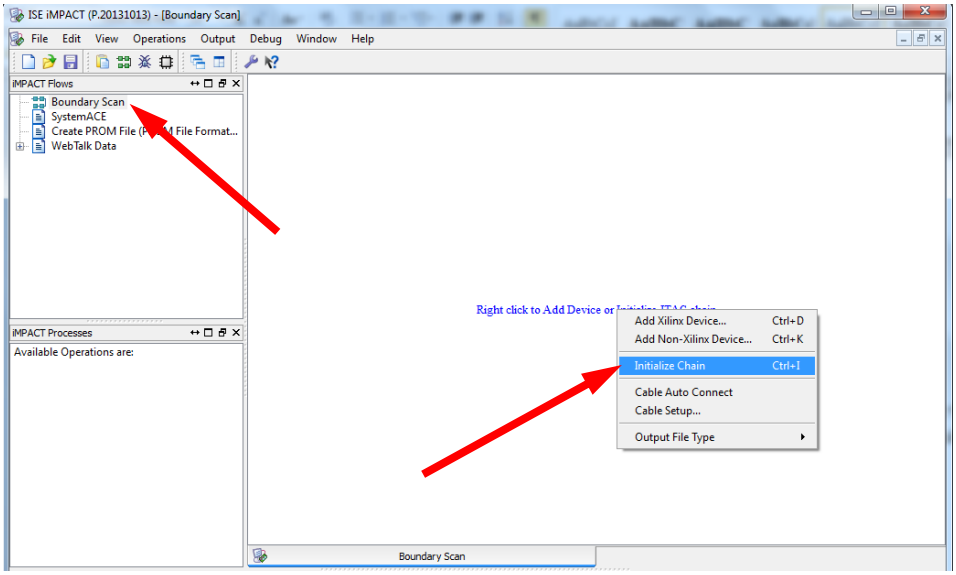
The implementation of the design may well take several minutes. Check the console for error messages and take appropriate remedial action if errors are encountered. Next double-click the '*Generate Programming File'* icon in the lower-left pane. At any time, you may need to tidy-up the large number of intermediate files generated by clicking '*Project > Cleanup Project Files*'.

Ensure that the Digilent Nexys 3 board is plugged into the USB socket at that the power switch near the USB connector is switched to on (lights will appear).

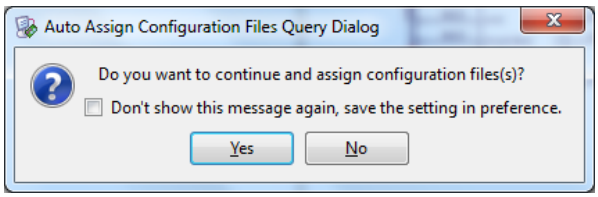
Double-click the '*Configure Target Device'* icon in the lower-left pane. A '*No iMPACT*' warning message will appear – click '*OK*' to continue.



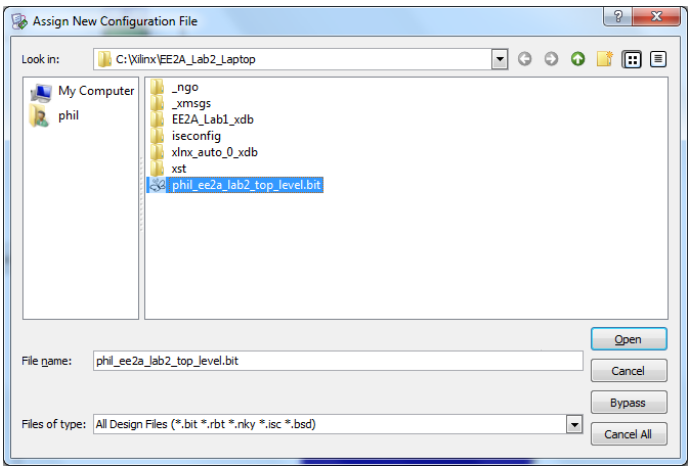
A new window will appear. Double-click '*Boundary Scan*' in the top-left pane. Move the cursor to the main boundary scan window and right-click '*Right click to Add Device or Initialize chain*' and then right-click '*Initialize Chain*'**.**



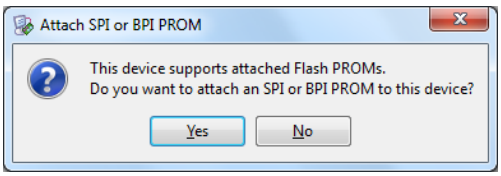
A new '*Auto Assign Configuration Files Query Dialog*' message box will appear – click '*Yes*'.



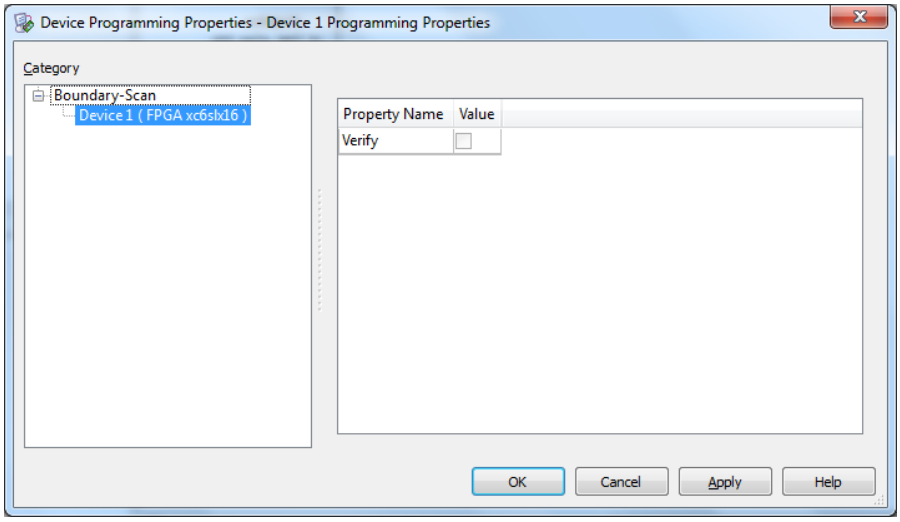
A file selection panel will appear. Navigate to the location of your files and select the '*bit*' file generated by your project, then click '*Open*'.



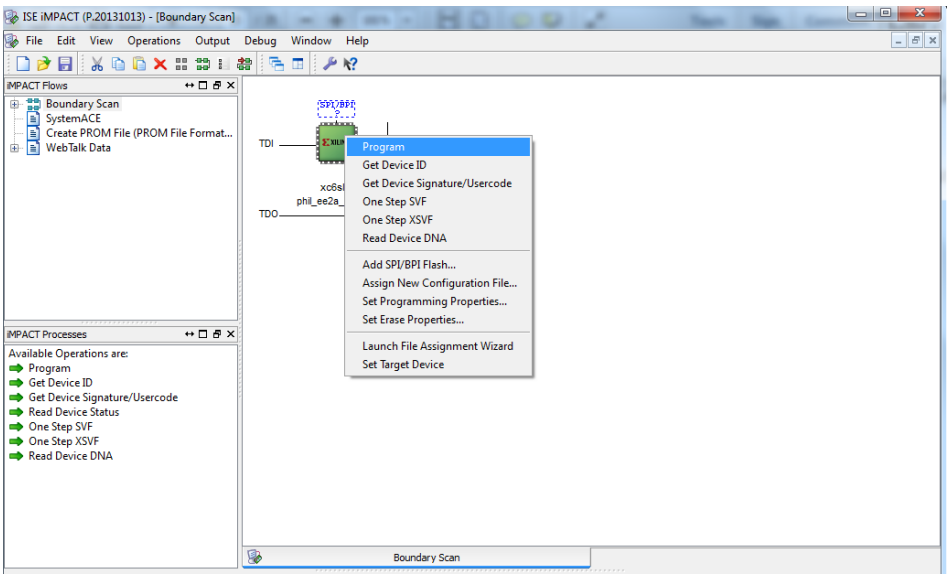
A PROM controller might be attached to the FPGA, click '*No*' as you will temporarily load the FPGA configuration memory.



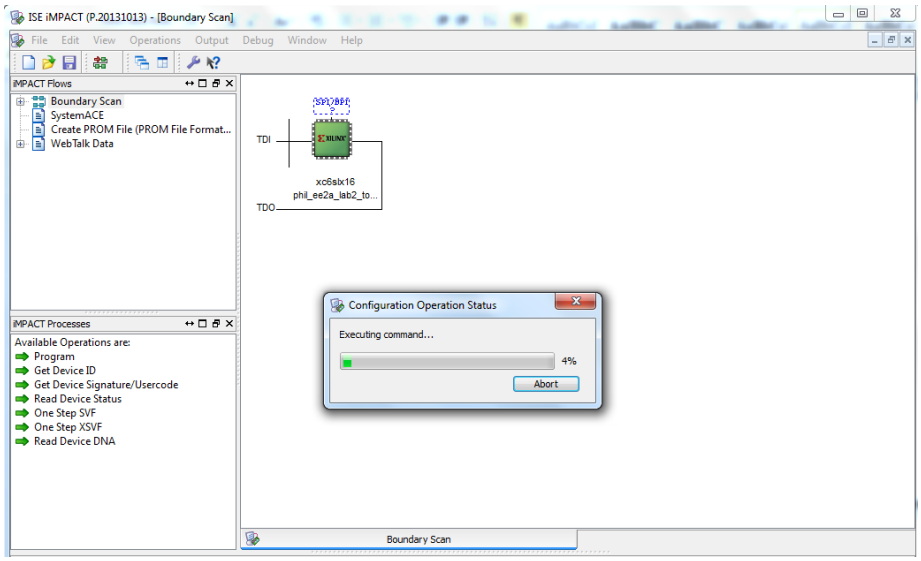
In the next box, click 'OK' to perform a boundary scan programming of the FPGA device.



Now right click on the green integrated circuit icon and click '*Program*'.



A pop-up programming box should appear and device will be programmed.



Once programmed, you can manipulate the buttons and switches to demonstrate that the board works as specified.