EE/CE/TE 1202– Introduction to Electrical Engineering II - Dr. Diana Cogan

Lab 2: Designing, simulating and implementing a 4-bit ALU

Assigned: 09/26/2016

Parts 1, 2 and 3 due: 10/05/2016 (3 points)
Parts 4, 5, 6 and 7 due: ~~10/12/2016~~ (3 points)
Mon 10/17

Contents

1. Part 1: Expected duration – 30 minutes
   Building the logic extender component.
2. Part 2: Expected duration – 30 minutes
   Building the arithmetic extender component.
3. Part 3: Expected duration – 30 minutes
   Building the ALU.
4. Part 4: Expected duration – 50 minutes
   The 4-bit–to–7-segment converter.
5. Part 5: Expected duration – 30 minutes
   Everything comes together.
6. Part 6: Expected duration – 15 minutes
   Generating the UCF file.
7. Part 7: Expected duration – 15 minutes
   Programming the board.

In the previous lab assignment you constructed an ALU that operated on 8-bits numbers. It was rather limited, though, in that it was only capable of performing two operations: addition and subtraction.

This time around, you will construct a 4-bit ALU that performs a total of 8 operations (4 logic operations and 4 arithmetic operations). Like the previous ALU, this one also builds upon the full-adder. If you remember last time, an XOR gate was used to switch between addition and subtraction, and a carry-in signal was also tweaked for that purpose – in effect, this was a way to prepare the inputs of the full adder for each of the supported operations. This time, in order to perform all 8 operations you will design 2 devices that will connect to the half-adder's inputs: a logic extender and an arithmetic extender.

Then, you will put it all together, along with a binary-to-7-segment module that you will also design. The finished ALU will have two inputs, X(3:0) and Y(3:0), 2 outputs, (F3,F2,F1,F0) and the carry signal Cout, and 3 control signals, M, S0 and S1. The control signals specify the operation you would like to perform.

After simulating and making sure it all works in theory, you will synthesize your code into the Digilent Baysys2 board to see how well it does in practice.

Part 1 - Building a Logic Extender

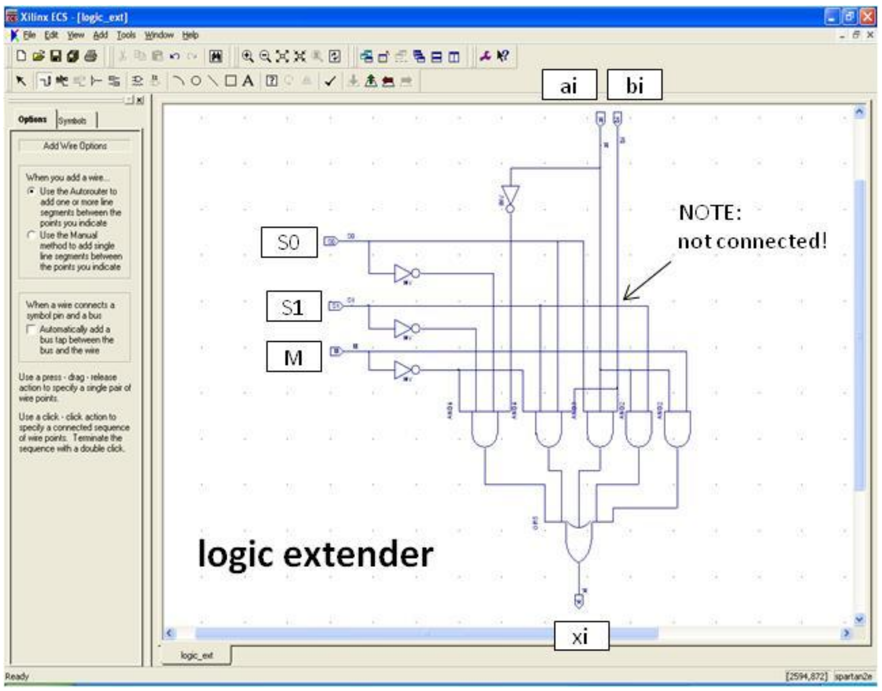
First, you are going to build a logic extender (LE). The LE is a device that performs a logic operation on either one or both of its inputs, and outputs the result to the full-adder (FA) from the previous assignment. The other input of the full-adder is then set to 0, so that output of the LE is in reality forwarded to the output of the FA. The LE must therefore function as described on the following table:

M	**S1**	**S0**	**Fun. Name**	**F**	$\mathbf{x_i}$
0	0	0	Complement	A’	a_i’
0	0	1	AND	A AND B	a_ib_i
0	1	0	Identity	A	a_i
0	1	1	OR	A OR B	$a_i + b_i$
1	--	--	--	--	a_i

As you can see, the signal **M** controls whether the logic extender should be utilized or bypassed. The signals **S0** and **S1** are used to specify the operation that should performed on $\mathbf{a_i}$ and $\mathbf{b_i}$. For example, when **S0**, **S1** are **0**, **0**, the logic extender should output the complement of $\mathbf{a_i}$ (and ignore $\mathbf{b_i}$).

Here is how you will build it:

1. Open the Xilinx ISE Project Navigator and create a **New Source** of type schematic. Call it **logic_ext.sch**.
2. Draw the following schematic, which was derived in class:



Add Wire Options

When you add a wire...
- (●) Use the Autorouter to add one or more line segments between the points you indicate
- ( ) Use the Manual method to add single line segments between the points you indicate

When a wire connects a symbol pin and a bus
- [ ] Automatically add a bus tap between the bus and the wire

Use a press - drag - release action to specify a single pair of wire points.

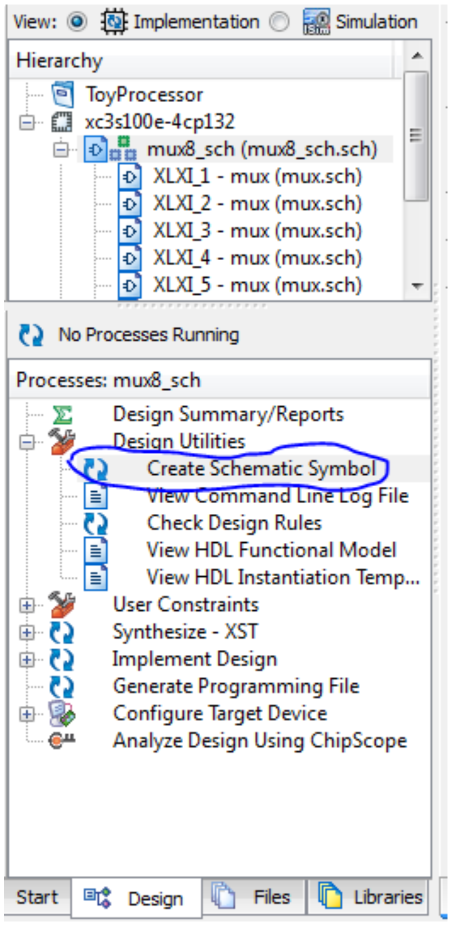
Use a click - click action to specify a connected sequence of wire points. Terminate the sequence with a double click.

3. Create a test bench called **logic_ext_tb.v** (code given below).

```
//logic_ext_tb.v

`timescale 1ns/1ps

module logic_ext_tbw_tb_0;
   reg ai = 1'b0;
   reg bi = 1'b0;
   reg M = 1'b0;
   reg S0 = 1'b0;
   reg S1 = 1'b0;
   wire xi;
   integer i = 0;
   parameter num_inputs = 5;
   parameter max_count = (1<<num_inputs);

   logic_ext UUT (
      .ai(ai),
      .bi(bi),
      .M(M),
      .S0(S0),
      .S1(S1),
      .xi(xi));

initial begin
  #100;                                    //Wait 100ns for initial inputs to settle.
  for (i=0; i<max_count; i=i+1)
    begin
      {M,S1,S0,ai,bi} = i;                 //Cycle through all 5 input combinations.
      #100;                                //Wait 100ns. How many combinations won't be
    end                                    //tested because of the 3000ns time constraint?
end

endmodule
```

4. Simulate and make sure your component functions as specified by the table above.

5. Close **ISim**. Then, back in the ***Project Navigator***, create a symbol for your LE.



View: ● Implementation ○ Simulation

Hierarchy

- ToyProcessor
- xc3s100e-4cp132
  - mux8_sch (mux8_sch.sch)
    - XLXI_1 - mux (mux.sch)
    - XLXI_2 - mux (mux.sch)
    - XLXI_3 - mux (mux.sch)
    - XLXI_4 - mux (mux.sch)
    - XLXI_5 - mux (mux.sch)

No Processes Running

Processes: mux8_sch

- Design Summary/Reports
- Design Utilities
  - Create Schematic Symbol
  - View Command Line Log File
  - Check Design Rules
  - View HDL Functional Model
  - View HDL Instantiation Temp...
- User Constraints
- Synthesize - XST
- Implement Design
- Generate Programming File
- Configure Target Device
- Analyze Design Using ChipScope

***Useful Note:** To create a symbol, change the view to Implementation, highlight the name of the schematic file and double-click **Create Schematic Symbol** from the **Design Utility** in the **Process View** tab. This creates a black-box type symbol for the logic extender. The default name given to it by **Project Manager** is **your_schematic_name.sym.** Or you can use the Symbol Wizard under Tools in schematic view of the component.*

Part 2 - Building an Arithmetic Extender

Now you are going to build the arithmetic extender (AE). The AE is a component similar to the LE, except it sets up the inputs of the full-adder for arithmetic calculations rather than logic ones. Its operation is described on the following table:

M	**S1**	**S0**	**Fun. Name**	**F**	y_i	c_0
1	0	0	Decrement	A – 1	1	0
1	0	1	Add	A + B	b_i	0
1	1	0	Subtract	A + B’ + 1	b_i’	1
1	1	1	Increment	A + 1	0	1
0	--	--	--	--	0	0

Note that the carry-in signal (c_0) is included here only for informative purposes. It is not part of the AE, but helps you understand the operations that are taking place.

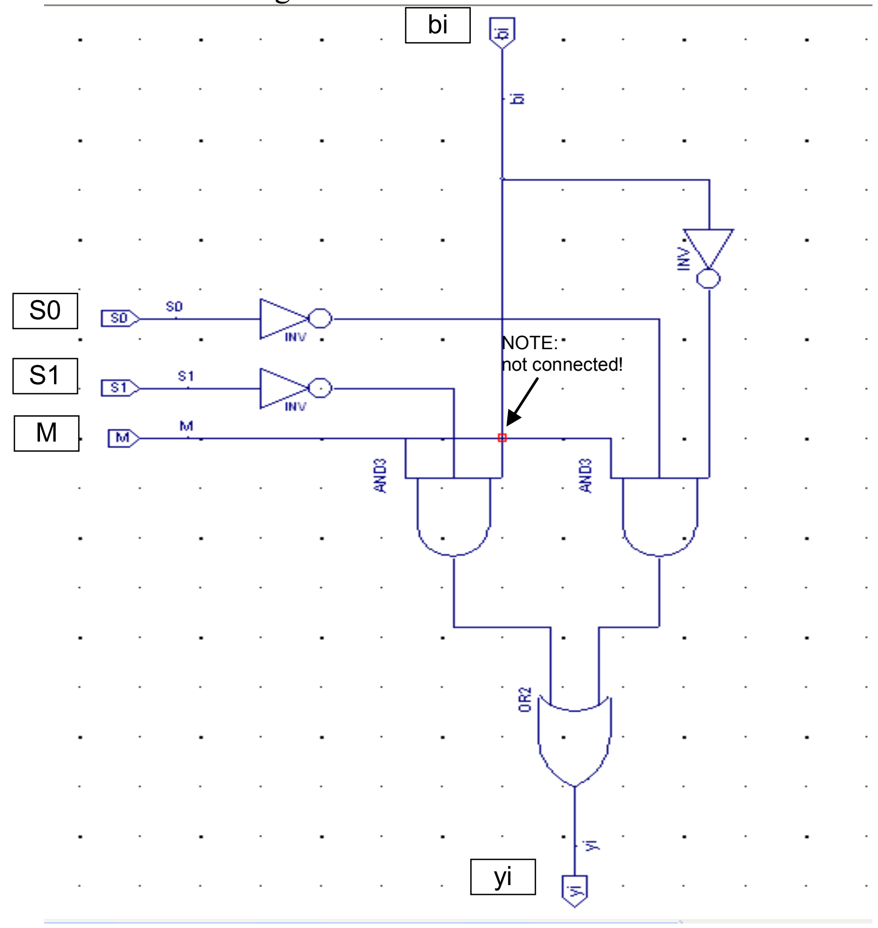
From the table it is easy to see how this works:

- To decrement X, the Y input is set to -1 (all 1s in two's complement);
- To add X and Y there is no preparation required, so X and Y are merely forwarded to the output.
- To subtract Y from X, the two's complement representation of Y is accomplished by inverting Y and setting the carry to 1.
- To increment X, the carry is set to 1 while Y is 0

Now you are ready to begin building the AE:

1. With ***Project Manager*** still open, create a **New Source** of type schematic. Call it **arith_ext.sch**. It is going to be the arithmetic extender, a basic component of our ALU.

2. Draw the following schematic:



arithmetic extender

3. Run "check schematic".
4. Create a test bench called **arith_ext_tb.v** (code shown below)**.**

```
//arith_ext_tb.v
`timescale 1ns/1ps
module arith_ext_tbw_tb_0;
   reg bi = 1'b0;
   reg M = 1'b0;
   reg S0 = 1'b0;
   reg S1 = 1'b0;
   wire yi;
   integer i = 0;
   parameter num_inputs = 4;
   parameter max_count = (1<<num_inputs);

arith_ext UUT (
   .bi(bi),
   .M(M),
   .S0(S0),
   .S1(S1),
   .yi(yi));
```

```
initial begin
   #100;                                  //Wait 100ns for initial inputs to settle.
   for (i=0; i<max_count; i=i+1)
      begin
         {M,S1,S0,bi} = i;                //Cycle through all 4 input combinations.
         #100;                            //Wait 100ns between new inputs.
      end
end

endmodule
```

5. Simulate and make sure your component functions as specified by the table above.

6. Close **ISim** and create a symbol for the AE using ***Project Manager.***

Part 3: Building the ALU

At last, you are going to put everything together and build a 4-bit ALU. This is done by stacking the LE and AE onto the full-adder.

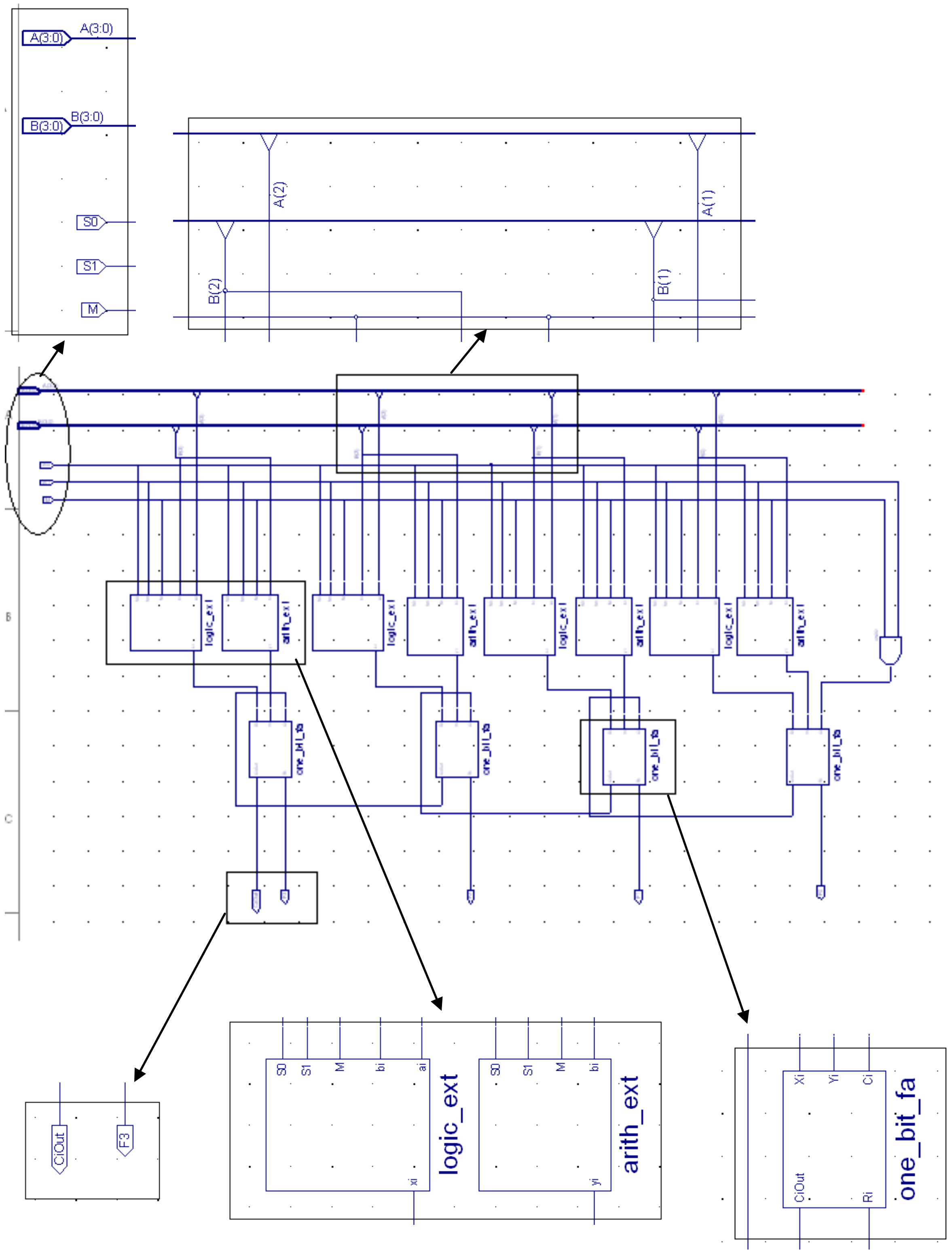
You should follow the same familiar steps:

1. Add a **New Source** of type schematic to the project. Name it **alu4bit_sch.sch.**
2. Draw the schematic of a 4-bit ALU as shown on the next page.
3. Create a test bench called **alu4bit_tb.v** to test the functionality of our ALU. The result you get should verify the functionalities specified on the 2 tables above (code shown below).

```
// alu4bit_tb.v

`timescale 1ns/1ps

module alu4bit_tbw_tb_0;
  reg [3:0] A = 4'b0000;
  reg [3:0] B = 4'b0000;
  reg M = 1'b0;
  reg S0 = 1'b0;
  reg S1 = 1'b0;
  wire CiOut;
  wire F0;
  wire F1;
  wire F2;
  wire F3;
  integer i = 0;
  parameter num_inputs = 3;
  parameter max_count = (1<<num_inputs);

alu4bit_sch UUT (
  .A(A),
  .B(B),
  .M(M),
  .S0(S0),
  .S1(S1),
  .CiOut(CiOut),
  .F0(F0),
  .F1(F1),
  .F2(F2),
  .F3(F3));

initial begin
  #100;                          //Wait 100ns before beginning test.
  for (i=0; i<max_count; i=i+1)
    begin
      {M,S1,S0} = i;             //Cycle through all control input combinations.
      A = 4'b0101;               //Choose a value for A.
      B = 4'b0100;               //Choose a value for B.
      #100;                      //Wait 100ns between control signal changes.
    end
  #100;                          //Wait 100ns before next test cycle.
  for (i=0; i<max_count; i=i+1)
    begin
      {M,S1,S0} = i;             //Cycle through all control input combinations.
      A = 4'b1010;               //Choose a different value for A
      B = 4'b0101;               //Choose a different value for B.
      #100;                      //Wait 100ns between control signal changes.
    end
end

endmodule
```

4. Simulate and make sure your ALU functions work as expected. Simulate interesting cases, such as the ones where there is long carry-signal propagation by choosing different values for A and B.

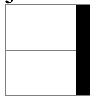
Make sure you understand how the ALU works, and the parts played by the AE and LE.



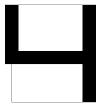
Complete ALU Schematic

Part 4 – The 4-bit-to-7-segment converter

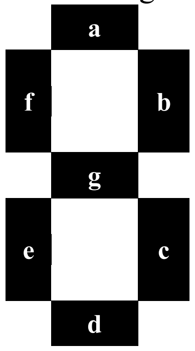
In order to interface the ALU with the 7-segment displays on the FPGA board that we will be using, we need to design a circuit that converts 4-bit binary numbers to the appropriate signals to illuminate the seven segments of the display, as we discussed in class. The 7-segment display contains 7 "lines", one for each segment that it can display. Digits are constructed by illuminating the appropriate segments. For the number "1", for example, we just need to illuminate the 2 rightmost lines, as shown below:



And for the number "4", you need to turn on 4 of the seven lines as shown below:



The 7-segment display is operated by 7 inputs, each representing one of the possible "lines" of a decimal digit. They are mapped like this:



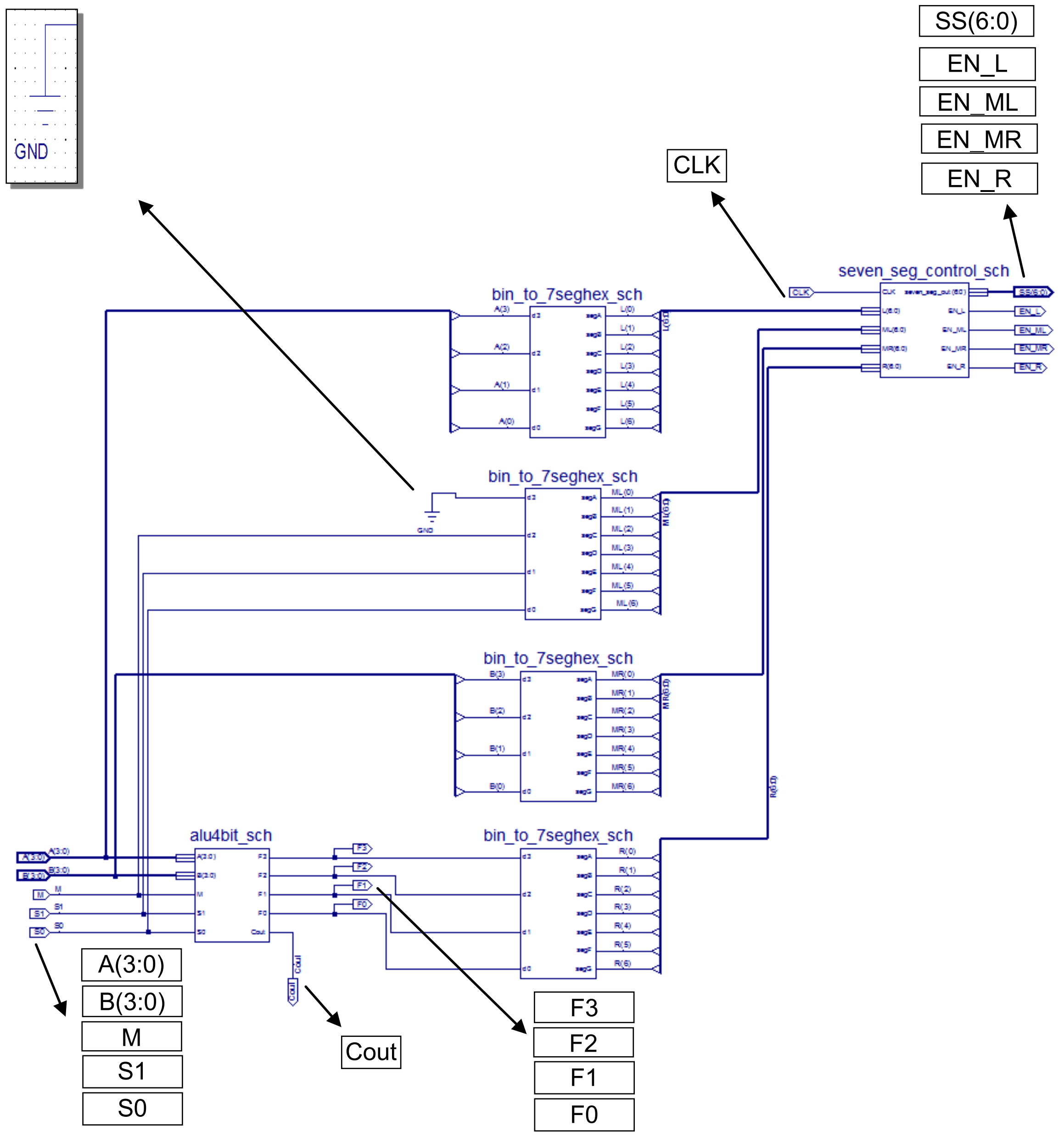
The inputs **a, b, c, d, e, f, g** are logic-low, that is, you illuminate them by sending a 0 and turn them off with a 1. Thus, you must make a converter that operates according to the following table:

4-Bit Binary Input	**Digit To be Shown on 7-Seg Display**	**Values Needed for Segment Signals**						
		a	**b**	**c**	**d**	**e**	**f**	**g**
0000	0	0	0	0	0	0	0	1
0001	1	1	0	0	1	1	1	1
0010	2	0	0	1	0	0	1	0
0011	3	0	0	0	0	1	1	0
0100	4	1	0	0	1	1	0	0
0101	5	0	1	0	0	1	0	0
0110	6	0	1	0	0	0	0	0
0111	7	0	0	0	1	1	1	1
1000	8	0	0	0	0	0	0	0
1001	9	0	0	0	1	1	0	0
1010	A	0	0	0	1	0	0	0
1011	B	1	1	0	0	0	0	0
1100	C	0	1	1	0	0	0	1
1101	D	1	0	0	0	0	1	0
1110	E	0	1	1	0	0	0	0
1111	F	0	1	1	1	0	0	0

Build the converter schematic (this was done in class), naming it “bin_to_7seghex_sch”, then simulate it to make sure it works (use the table above to assess that). You should understand and be able to explain how the converter works. When you are done, make a symbol for the converter.

Part 5 – Everything comes together

Now it's time to put everything together, but first you should download a component that we are providing from the class website, in the **Lab** section. Download the file called “seven_seg_controller.zip” and unzip its contents into your project folder. Then, open your project in the Xilinx ISE, click on the **Project** menu, then **Add Source**. Select the files that start with “mux4” and “seven_seg_control” (you can use the **Ctrl** key to select multiple files at once). Now press **Open** and then **OK**. Connect all components as shown in the schematic below. Name the schematic “alu4bit_board_sch”. Then, as usual, simulate it. Given the control signals M, S0 and S1 (of the LE and AE) you should be able to execute different logic and arithmetic functions on the inputs X and Y. The output should be in the format detailed in the truth table for the 4-bit-to-7-digit-converter.



Notice there is an input port here that we haven't mentioned yet: the CLK. We will connect (in Part 6) the CLK port to a clock signal that is provided by the Pegasus board.

Also notice that there are 4 output ports, signals EN_L, EN_ML, EN_MR, and EN_R. These signals enable each of the 4 seven-segment displays of the Pegasus board. As we explained in class, all 4 seven-segment displays are driven by the same 7 lines SS[6:0] and these 4 signals enable in a round-robin fashion each of the displays while sending the appropriate values on the SS[6:0] bus. In other words, all 4 seven-segment displays see the values to be displayed but only one is activated. By cycling through the 4 seven-segment displays fast enough, we create the illusion that they are continuously illuminated, while in reality they are not. This is what the seven_seg_control_sch does.

Part 6 – Generating the UCF file

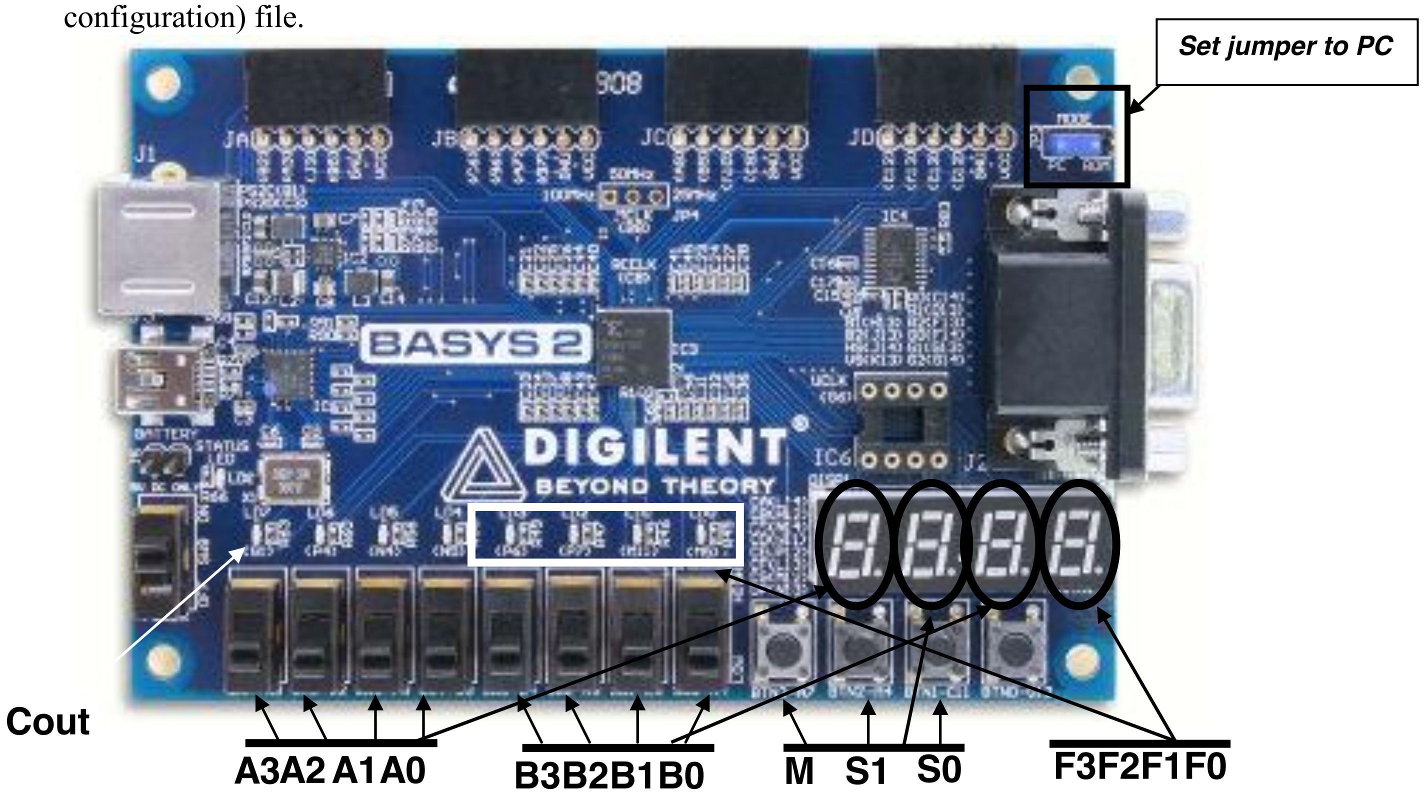
In the previous parts of this assignment you put together an ALU and it is now time to finally program it onto the Pegasus FPGA board. But this ALU is now a black box with input and output ports, and before you download it to the board you must tell very precisely where these ports must be connected. This is where the UCF (User Constraints File) comes in: it tells the FPGA which pins on the chip should be connected to which inputs and outputs of your schematic.

A few things to note:

1. You cannot assign one input/output port to multiple pins. To do that, you will have to duplicate the input/output signal on your schematic to go to the number of ports you want.
2. Two pages below on this document you will see an Excel sheet that helps generate a UCF file. This Excel sheet can be downloaded from the lab webpage. Once you have it, all you have to do is enter your signal names on the left column, and the UCF will be automatically generated on the right. If you didn't have this, you'd have to read the datasheet of the Pegasus board to find the pin names that you may utilize.
3. On the UCF file, you may insert comments using a hash (#) in the beginning of the line.
4. A pin on the Pegasus is denoted as P*XX*, where XX is the pin number (i.e. P13 for pin 13).

As shown in the figure below, you should use the 8 switches for the number inputs. Number A(3:0) will be specified via switches sw7—sw4. B(3:0) will be specified via switches sw3—sw0. The ALU operation will be specified by connecting Signals (M, S1, S0) to Push Buttons (BTN3, BTN2, and BTN1), respectively. A(3:0) should be displayed in hexadecimal format at the left 7–segment display, the ALU operation should be displayed in the middle left 7–segment display (it's a 3-bit entity, so we will connect the most significant bit to the GND), B(3:0) should be displayed at the middle right 7–

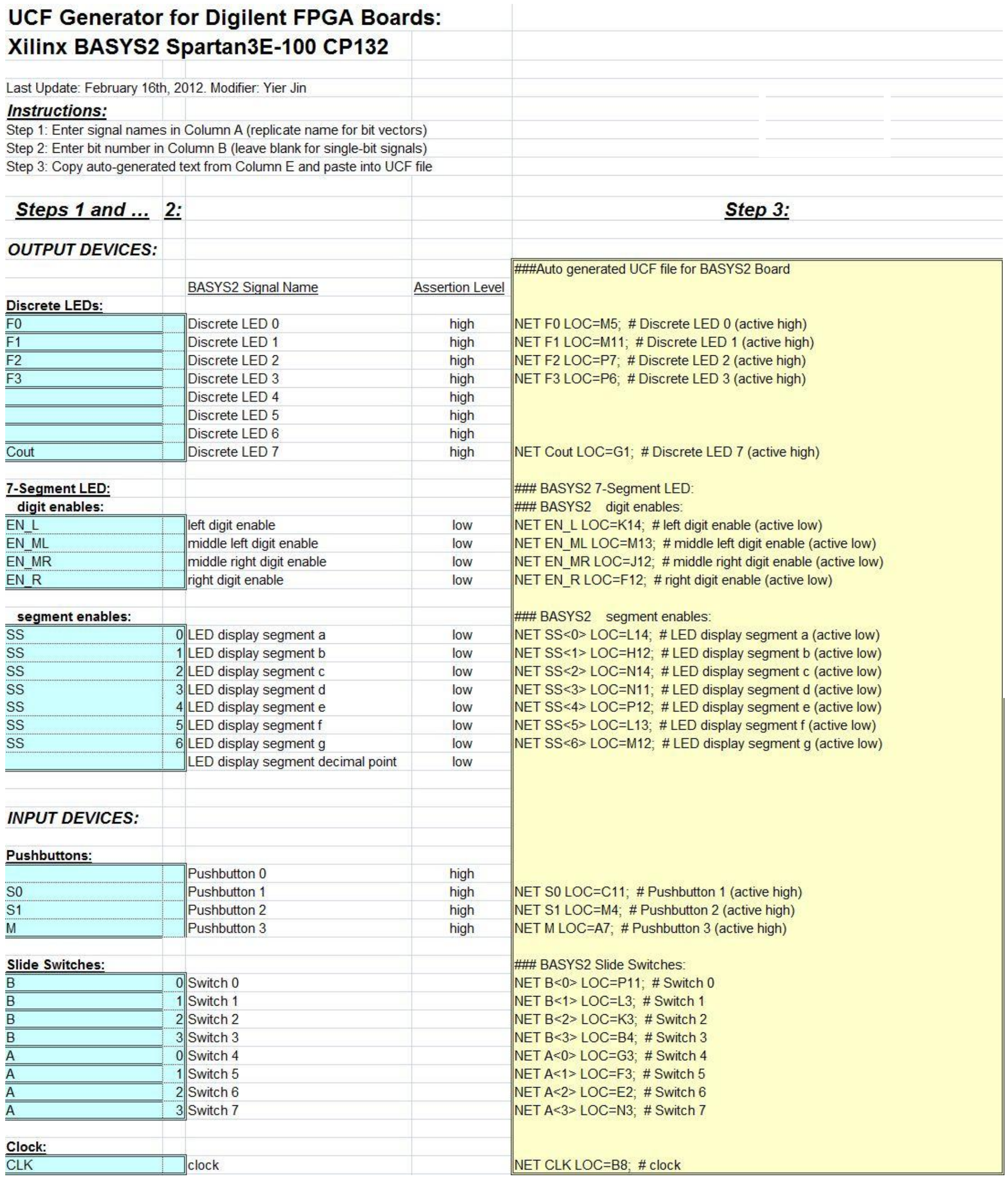
segment display, and the result (F3,F2,F1,F0) should be displayed at the right 7–segment display. Also, F3, F2, F1, F0 should be displayed in discrete LEDs (LD3:LD0) and the Cout signal should be displayed at discrete LED LD7. The following figure should clarify these assignments, which will be made through writing an appropriate UCF (user configuration) file.



After you are done modifying the Excel sheet, you should copy/paste the UCF file on the right column into a program like **notepad**. Then save the file under your project folder (probably c:\Xilinxworks\ToyProcessor) with the name “alu4bit_board_ucf.ucf”. Now you must go back to the **Xilinx ISE Project Navigator** and add the UCF file as a source in your project. This is done by going to the “Project” menu, clicking on “Add source” and choosing the “alu4bit_board_ucf.ucf” file.

Another route:

You could automatically assign the pins using the **Xilinx ISE Project Navigator** by double-clicking **Create Timing Constraints** under the **User Constraints** process. The tool is intuitive. A UCF file is then automatically generated. This is just like using the Excel sheet except that the Excel file indicates in plain English which pin (e.g. “middle digit of 7-segment LED”) you are assigning.



UCF Generator for Digilent FPGA Boards:
Xilinx BASYS2 Spartan3E-100 CP132

Last Update: February 16th, 2012. Modifier: Yier Jin

Instructions:

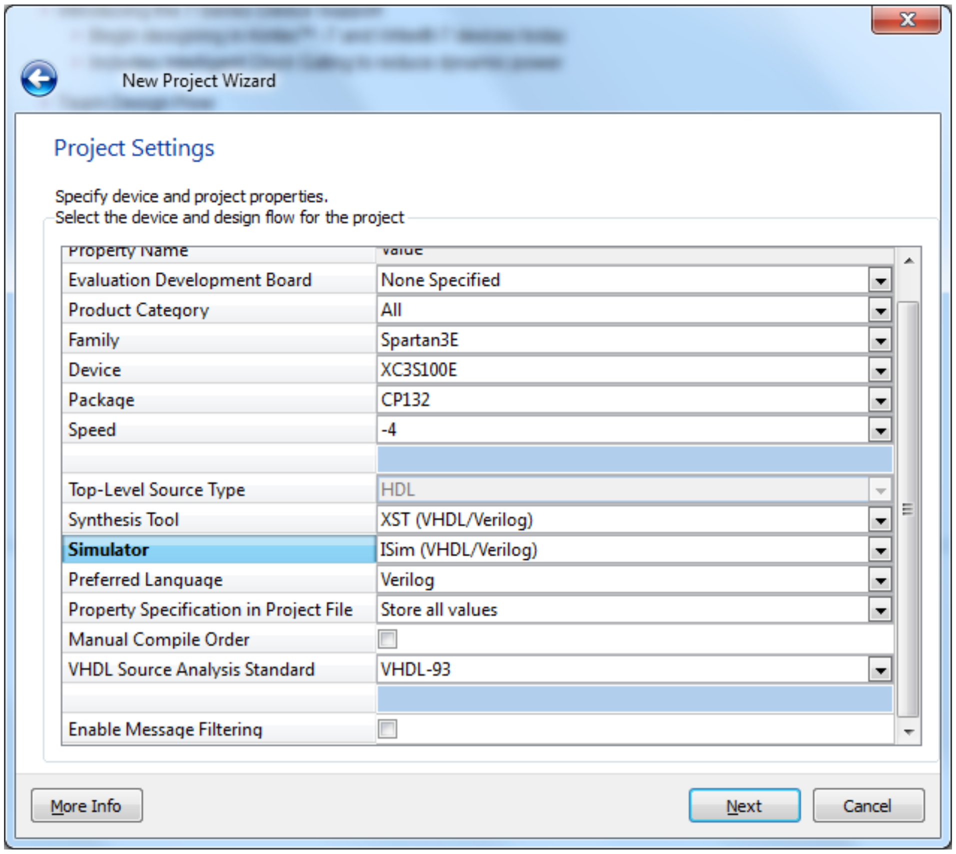
Step 1: Enter signal names in Column A (replicate name for bit vectors)
Step 2: Enter bit number in Column B (leave blank for single-bit signals)
Step 3: Copy auto-generated text from Column E and paste into UCF file

Steps 1 and …	***2:***			***Step 3:***
OUTPUT DEVICES:				
				###Auto generated UCF file for BASYS2 Board
		BASYS2 Signal Name	Assertion Level	
Discrete LEDs:				
F0		Discrete LED 0	high	NET F0 LOC=M5; # Discrete LED 0 (active high)
F1		Discrete LED 1	high	NET F1 LOC=M11; # Discrete LED 1 (active high)
F2		Discrete LED 2	high	NET F2 LOC=P7; # Discrete LED 2 (active high)
F3		Discrete LED 3	high	NET F3 LOC=P6; # Discrete LED 3 (active high)
		Discrete LED 4	high	
		Discrete LED 5	high	
		Discrete LED 6	high	
Cout		Discrete LED 7	high	NET Cout LOC=G1; # Discrete LED 7 (active high)
7-Segment LED:				### BASYS2 7-Segment LED:
digit enables:				### BASYS2 digit enables:
EN_L		left digit enable	low	NET EN_L LOC=K14; # left digit enable (active low)
EN_ML		middle left digit enable	low	NET EN_ML LOC=M13; # middle left digit enable (active low)
EN_MR		middle right digit enable	low	NET EN_MR LOC=J12; # middle right digit enable (active low)
EN_R		right digit enable	low	NET EN_R LOC=F12; # right digit enable (active low)
segment enables:				### BASYS2 segment enables:
SS	0	LED display segment a	low	NET SS<0> LOC=L14; # LED display segment a (active low)
SS	1	LED display segment b	low	NET SS<1> LOC=H12; # LED display segment b (active low)
SS	2	LED display segment c	low	NET SS<2> LOC=N14; # LED display segment c (active low)
SS	3	LED display segment d	low	NET SS<3> LOC=N11; # LED display segment d (active low)
SS	4	LED display segment e	low	NET SS<4> LOC=P12; # LED display segment e (active low)
SS	5	LED display segment f	low	NET SS<5> LOC=L13; # LED display segment f (active low)
SS	6	LED display segment g	low	NET SS<6> LOC=M12; # LED display segment g (active low)
		LED display segment decimal point	low	
INPUT DEVICES:				
Pushbuttons:				
		Pushbutton 0	high	
S0		Pushbutton 1	high	NET S0 LOC=C11; # Pushbutton 1 (active high)
S1		Pushbutton 2	high	NET S1 LOC=M4; # Pushbutton 2 (active high)
M		Pushbutton 3	high	NET M LOC=A7; # Pushbutton 3 (active high)
Slide Switches:				### BASYS2 Slide Switches:
B	0	Switch 0		NET B<0> LOC=P11; # Switch 0
B	1	Switch 1		NET B<1> LOC=L3; # Switch 1
B	2	Switch 2		NET B<2> LOC=K3; # Switch 2
B	3	Switch 3		NET B<3> LOC=B4; # Switch 3
A	0	Switch 4		NET A<0> LOC=G3; # Switch 4
A	1	Switch 5		NET A<1> LOC=F3; # Switch 5
A	2	Switch 6		NET A<2> LOC=E2; # Switch 6
A	3	Switch 7		NET A<3> LOC=N3; # Switch 7
Clock:				
CLK		clock		NET CLK LOC=B8; # clock

Part 7 – Programming the board

Now you are ready to program the board. To do this you must first generate a bit stream file and then download it into the FPGA. This is how you generate the bit stream:

First of all, you must inform the Xilinx ISE that the Xilinx board that you will be programming has a SPARTAN XC3S100E FPGA. To do this, go to the ***Project*** and choose "Design Properties". Make sure you project is set like this:

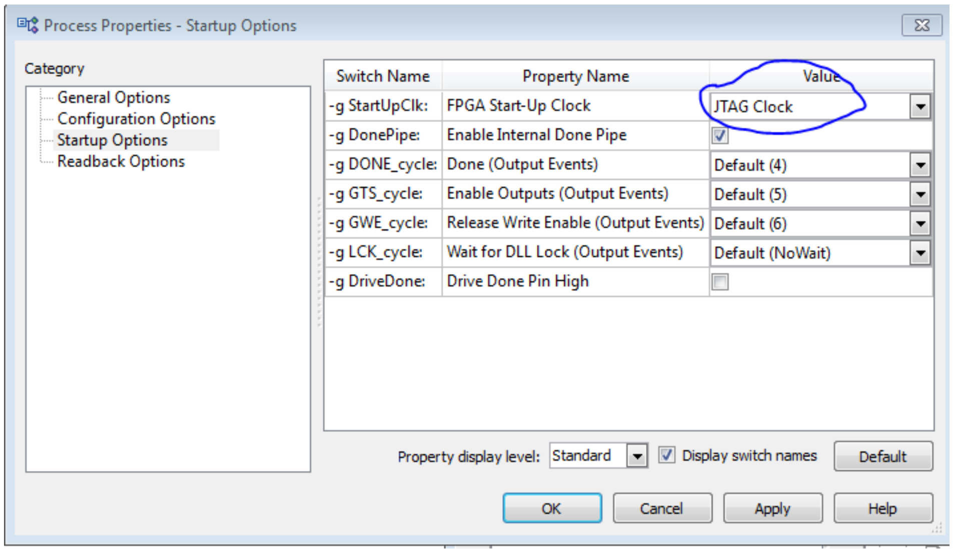


Project Settings

Specify device and project properties.
Select the device and design flow for the project

Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Spartan3E
Device	XC3S100E
Package	CP132
Speed	-4
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	☐
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	☐

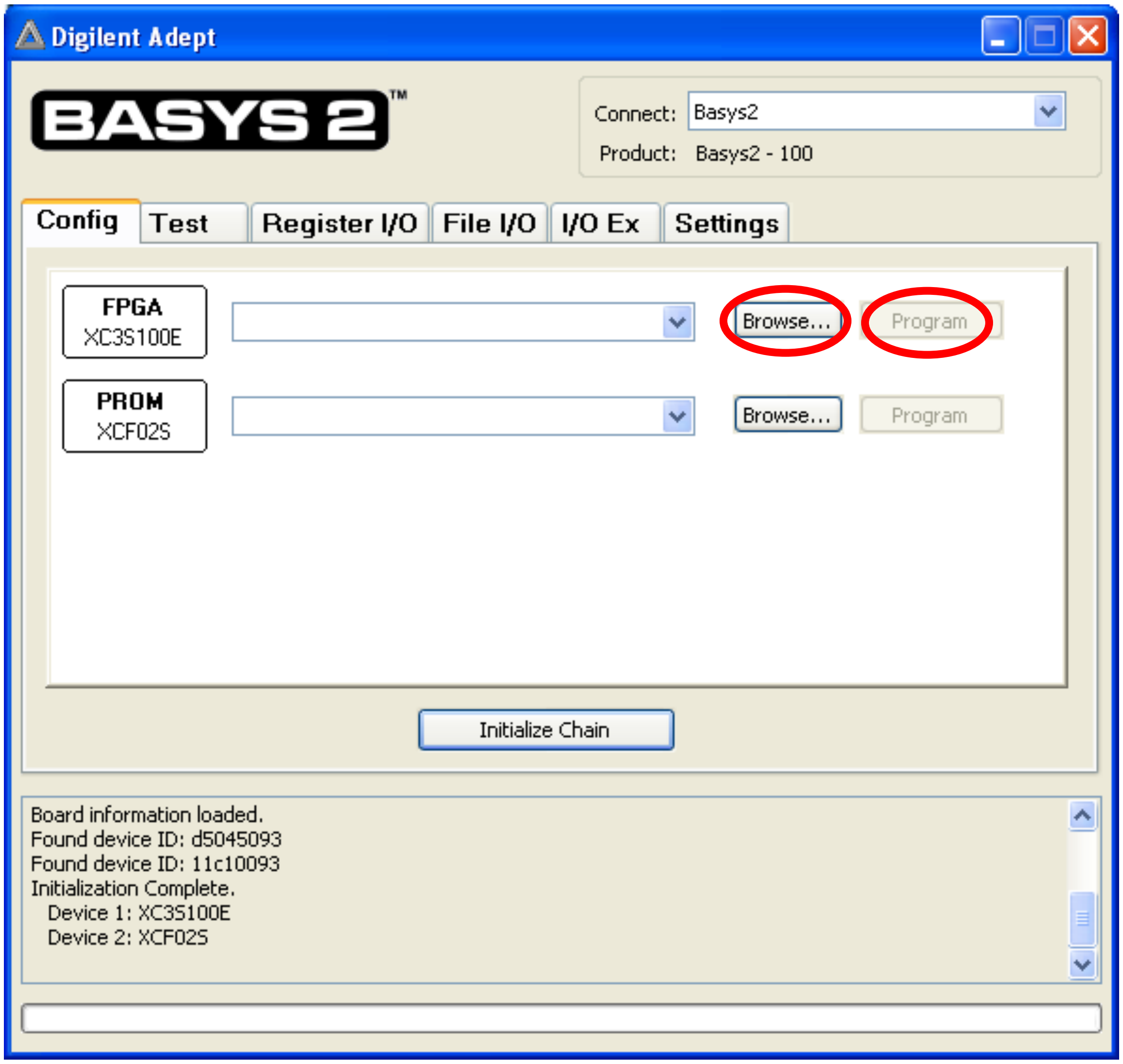
Now, on the ***Design*** window, make sure the ***view*** is set to "Implementation", right click on the 4-bit ALU schematic file and choose "Set as top module". With the ALU schematic still selected, go to the "Process" window below, right-click on "Generate Programming File" and choose "Process Properties". Click on the "Startup Options" category and make sure the FPGA Start-Up Clock is set to "JTAG Clock":



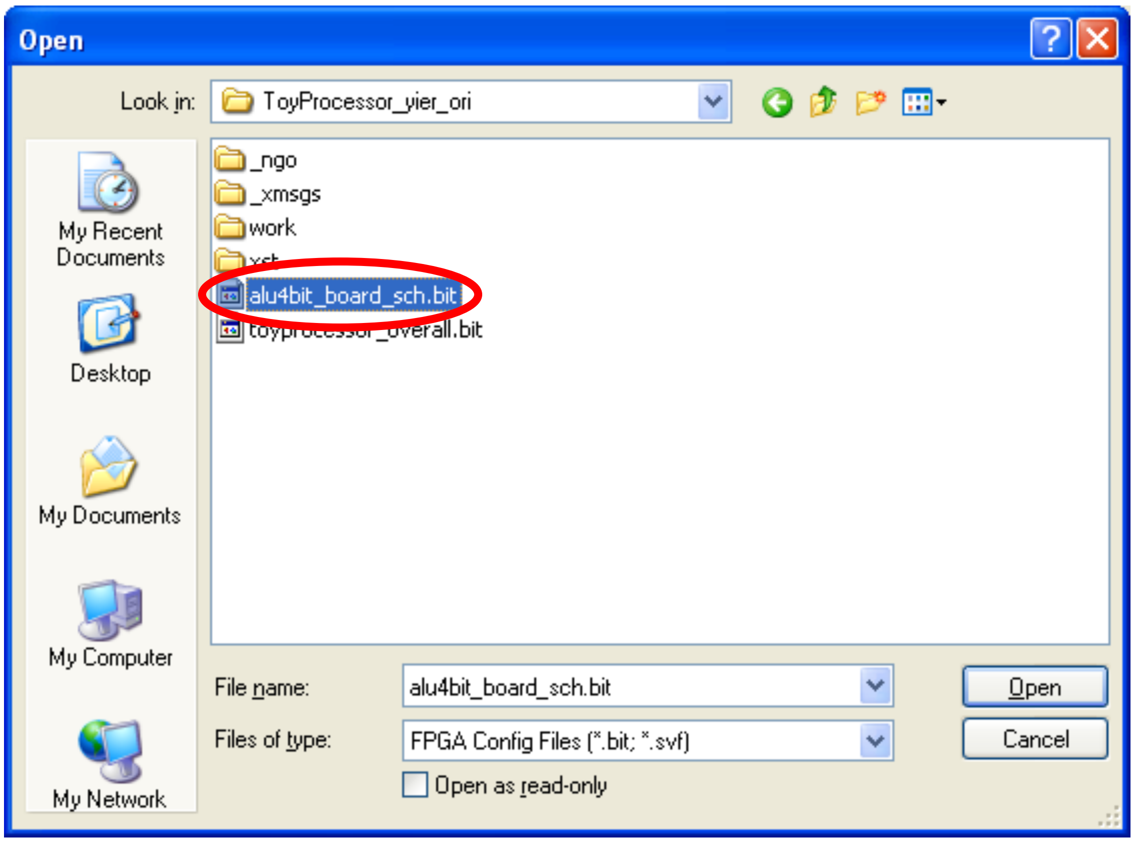
Switch Name	Property Name	Value
-g StartUpClk:	FPGA Start-Up Clock	JTAG Clock
-g DonePipe:	Enable Internal Done Pipe	✓
-g DONE_cycle:	Done (Output Events)	Default (4)
-g GTS_cycle:	Enable Outputs (Output Events)	Default (5)
-g GWE_cycle:	Release Write Enable (Output Events)	Default (6)
-g LCK_cycle:	Wait for DLL Lock (Output Events)	Default (NoWait)
-g DriveDone:	Drive Done Pin High	☐

Press OK. Now double-click the “Generate Programming File” tree in the processes window. This will take a while to complete. Check the console window in the bottom of the screen to see what is going on.

Once successful a .bit file will be generated in your program folder. You will download your bitstream to the board using Adept. Plug in your board using the USB adapter, and make sure the jumper on the top right of the board is **switched to "PC" rather than "ROM"**. Go to Start -> Program Files -> Digilent -> Adept. Your board should show up when the program opens. Click "Browse" that corresponds with the FPGA (the top one), select your bitstream, and press "Program" with the procedure shown below.



```
Board information loaded.
Found device ID: d5045093
Found device ID: 11c10093
Initialization Complete.
  Device 1: XC3S100E
  Device 2: XCF02S
```



Open

Look in: ToyProcessor_yier_ori

_ngo

_xmsgs

work

xst

alu4bit_board_sch.bit

toyprocessor_overall.bit

File name: alu4bit_board_sch.bit

Files of type: FPGA Config Files (*.bit; *.svf)

Now play with the buttons and flip switches to make sure everything is working correctly.