Physics 364, Fall 2012, Lab #11

(Xilinx FPGA introduction)

one day only: Monday, November 19.

Course materials and schedule are at `positron.hep.upenn.edu/p364`

This one-day lab will introduce you to working with the BASYS2 circuit board, which is made by Digilent, Inc. The key component on the BASYS2 board is a small Field Programmable Gate Array, made by Xilinx, Inc. FPGAs can implement arbitrary combinations of logic gates (AND, OR, XOR, NAND, arithmetic, etc.), as well as flip-flops, memories, and more. The FPGA itself is the 1 cm $\times$ 1 cm chip at the center of the BASYS2 board. This little chip is equivalent to about 100,000 individual NAND gates! The board is both powered and programmed from the USB port of a computer (Windows or Linux, though I managed to get it to work at home on my Mac by using VirtualBox to run a Linux virtual machine). If you're curious for more details, the formal documentation for the BASYS2 board can be found at `www.digilentinc.com/Products/Detail.cfm?Prod=BASYS2` . In particular, the reference manual for the board is at `www.digilentinc.com/Data/Products/BASYS2/Basys2_rm.pdf` .

There is no write-up required for today's lab, but before you leave class, you need to show Bill or Jose how far you got through the material.

The FPGA is connected to 8 ON/OFF switches (lower edge of board, left side): the corresponding FPGA inputs are HIGH when the switches are UP and LOW when the switches are DOWN. Just above the 8 switches are 8 LEDs: the LEDs are ON when the corresponding FPGA outputs are HIGH and OFF when the outputs are LOW. To the right of the switches are 4 buttons: each one corresponds to an FPGA input that is normally LOW but goes HIGH while your finger is on the button. Above the buttons is a set of four 7-segment LED displays, which can be used to display numbers and a few letters, most commonly 0-9 and A–F for hexadecimal display of 16-bit numbers; these 32 LEDs (including the 4 decimal points) are somewhat confusingly controlled by 12 FPGA output pins. An on-board crystal oscillator provides a 25 MHz reference clock to the FPGA, which we will divide down and use at much lower speed. The board can connect to a PS/2 mouse or keyboard and to a VGA monitor, so in principle you could make a little video game, if you were really ambitious. On the top edge of the board are four sets of four user-defined input/output pins, which we will interface to our own breadboards in class next week.

The two main things that I have been aiming for this course to leave you with are (a) conceptual insight into how things work (a.k.a. demystification), and (b) technical skills that may help you to work in a research group. The reason for including the FPGA segment of the course is (a) to gain some conceptual understanding for how, in principle, a computer can be build up out of logic gates (which you already know can be built up out of transistors), and (b) because FPGAs are a big part of the

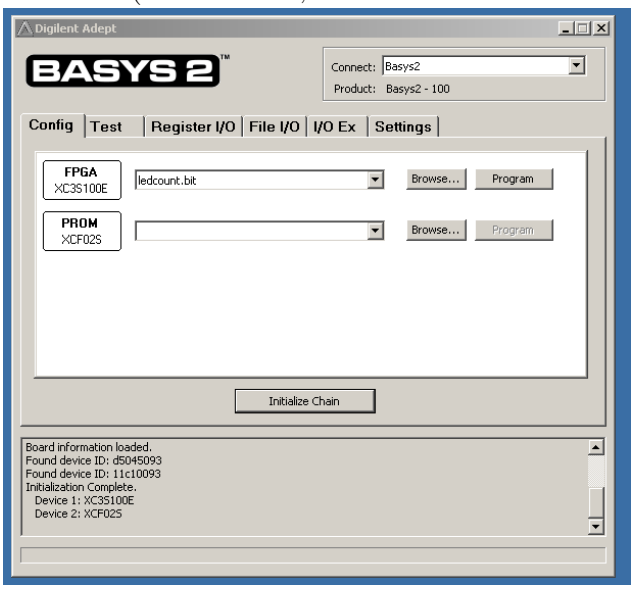
electronics design that we do for experiments in particle physics, medical physics, and presumably other fields in which scientists build their own instruments.

You should probably leave your BASYS2 board in the lab at the end of each class, unless you are so eager to learn FPGA programming at home that you can endure the substantial hassle of downloading (multi GB) and installing the Xilinx software on your own computer (Windows or Linux). We have a total of 17 boards, which doesn't leave any spares, so you might as well share a board with your lab partner unless you have a strong preference for not doing so.

Part 1

(a) Connect your board to the lab computer's USB port and turn it on by sliding the switch on the lower-left corner of the board to the UP position. If the blue "MODE" jumper on the top-right corner of the board is on the "ROM" (right-hand) position, then you will see a 1111, 2222, 3333, ..., FFFF pattern count on the 7-segment LED display. Now power the board OFF, then move the blue jumper to the "PC" (left-hand) position, and power the board back ON. You should see a bright red LED illuminate near the power switch, but nothing else.

Now start up the Digilent ADEPT software on the lab PC. The software should detect the BASYS2 board. From your web browser, download the file `positron.hep.upenn.edu/wja/P364_2012/ledcount.bit` to your PC (you'll need to pick a folder you're allowed to save to). Now in the ADEPT software's **config** tab, choose this `ledcount.bit` file as the program to load into the FPGA (XC3S100E). If all goes well, you should see the eight green LEDs counting in binary, updating once per second. If that doesn't work, ask for help! This is just a check that you can load a program onto the board. The ADEPT software (from Digilent, who make the board) is a tool for loading a `*.bit` file onto the ADEPT board. The `*.bit` file itself is created by the ISE software (from Xilinx, who make the FPGA chip).



(b) Now use your web browser to download `positron.hep.upenn.edu/wja/P364_2012/lab11_part1.zip` which is an archived version of a Xilinx ISE project. (Again, you need to save the file in a location you are allowed to write to on the PC.) Now from Windows, right-click the file and extract its contents to a folder. Now go into that folder and double-click on `lab11_part1.ise`, which should open up the Xilinx ISE software. In the ISE software's **processes** panel, right-click on **generate programming file** and choose **run**. After a moment, you should see the message `Process "Generate Programming File" completed successfully`. The result of all this is a file called `part1.bit`, which you can now load into the BASYS2 board using the ADEPT software. (There is also a pre-compiled copy of `part1.bit` located at `positron.hep.upenn.edu/wja/P364_2012/part1.bit` .)

Here is the Verilog source code for the program that you just loaded:

```
//
// part1.v
// Verilog source code for Part 1 of Lab 11
// PHYS 364, UPenn, fall 2012
//

// This line forces you to declare the names of all wires explicitly;
// otherwise, if you use a name that the compiler is not aware of, it
// will quietly assume that it is a new wire name.  In Verilog, a "wire"
// connects the output of one part of your circuit to the input of
// another part of your circuit, just like a physical wire.
`default_nettype none

// In Verilog, a "module" is like a schematic diagram that describes how
// a component of your circuit behaves.  This "top-level" module represents
// the schematic diagram for the entire FPGA.  Its inputs represent the
// signals coming in to the FPGA, and its outputs represent the signals
// going out of the FPGA.  The signals with square brackets [] by their
// names are multi-bit signals:  for example, there are 4 buttons and
// 8 switches among the FPGA's inputs, and there are 8 LEDs among the
// FPGA's outputs.  The FPGA itself is the 1cm x 1cm chip sitting at the
// center of the BASYS2 printed circuit board.  An FPGA is capable of
// implementing arbitrary digital logic functions, so it is a handy way
// for you to see what can be done with a very large collection of AND,
// OR, XOR gates, flip-flops, etc.
module part1
  (
   input        mclk,  // 25 MHz clock built into the BASYS2 board
   output [6:0] seg,   // red 7-segment display to draw numbers & letters
   output       dp,    // decimal point for 7-seg (0=ON, 1=OFF)
```

```
output [3:0] an,    // shared LED anode signal per 7-seg digit (0=ON)
output [7:0] led,   // 8 green LEDs, just above sliding switches
input  [7:0] sw,    // 8 sliding switches (up=1, down=0)
input  [3:0] btn,   // 4 push buttons (normal=0, pushed=1)
input  [4:1] ja,    // 4 pins (JA, NW corner) can connect to breadboard
input  [4:1] jb,    // 4 pins (JB) can connect to breadboard
input  [4:1] jc,    // 4 pins (JC) can connect to breadboard
input  [4:1] jd     // 4 pins (JD, NE corner) can connect to breadboard
);
// We define modules below called 'and_gate', 'nand_gate', etc.  For
// each of these modules, the first argument is the output pin, and
// the next two arguments are the input pins.  When I write the line
// 'and_gate myand1(led[0], sw[0], sw[1]);' it is equivalent to drawing
// an AND gate on my schematic diagram, writing the name 'myand1' to
// label this AND gate (so that it has a unique identity to distingish
// two copies of the AND gate, e.g. 'myand1' and 'myand2'), and then
// connecting the output of the AND gate to LED #0 and the two inputs
// of the AND gate to switches #0 and #1.
and_gate  myand1  (led[0], sw[0], sw[1]);

// Just to show you why we give each instance of 'and_gate' a unique
// name, here is a second copy of the same circuit (i.e. a second instance
// of the same 'and_gate' module), which functions equivalently to the
// first instance, but its output is connected to LED #7 and its input
// is connected to switches #6 and #7.
and_gate  myand2  (led[7], sw[6], sw[7]);

// LED #1 should display the NAND of switches #0 and #1.  Note that
// the switches and LEDs are numbered from 0 to 7, where 0 is on the
// right-hand side and 7 is on the left-hand side.
nand_gate mynand1 (led[1], sw[0], sw[1]);

// LED #2 should display the OR of the two switches.
or_gate   myor1   (led[2], sw[0], sw[1]);

// LED #3 should display the NOR of the two switches.
nor_gate  mynor1  (led[3], sw[0], sw[1]);

// LED #4 should display the XOR of the two switches.
xor_gate  myxor1  (led[4], sw[0], sw[1]);

// LED #5 is the 'Q' output of a SR latch, which is the most basic
// kind of flip-flop we studied.  Button #0 is the SET input, and
// button #1 is the RESET input.
sr_latch  mysr1   (led[5], btn[0], btn[1]);
```

```
   assign led[6] = btn[3];

   // These outputs are currently unused, but I connect them anyway to
   // keep the compiler from complaining.
   assign seg[6] = sw[6];
   assign seg[5] = sw[5];
   assign seg[4] = sw[4];
   assign seg[3] = sw[3];
   assign seg[2] = sw[2];
   assign seg[1] = sw[1];
   assign seg[0] = sw[0];
   assign dp = 0;
   assign an[3] = ~btn[3];
   assign an[2] = ~btn[2];
   assign an[1] = ~btn[1];
   assign an[0] = ~btn[0];
endmodule


// This 'and_gate' module tells the compiler what I mean when I ask it
// to put an 'and_gate' down on the top-level schematic.  I define a module
// with one output pin called 'o' and two input pins called 'a' and 'b',
// to correspond with a real two-input AND gate.
module and_gate (output o, input  a, input  b);
   // Verilog uses a C-like operator syntax.  The '&' operator means the
   // 'AND' operation.  The assign statement tells the compiler to take
   // whatever is on the right-hand side of the '=' sign and to wire it up
   // to whatever is on the left-hand side.  In this case, we 'AND' the two
   // inputs and connect the result to the output.
   assign o = a & b;
endmodule

module nand_gate (output o, input a, input b);
   // We can also declare a 'wire' inside a module, which is like drawing
   // a new wire on your schematic diagram.  Writing it this way is like
   // making a new wire called 'obar' which is connected to the AND of
   // inputs 'a' and 'b'.  Then the wire 'obar' is used as the input to
   // a NOT gate (a.k.a. an inverter), whose output we connect to 'o'.  In
   // Verilog (as in C), the bit-wise NOT operator is a tilde (~).
   wire obar = a & b;
   assign o = ~obar;
endmodule

module or_gate (output o, input a, input b);
   // In Verilog, as in C, the bit-wise OR operator is the veritical bar '|'.
```

```
   assign o = a | b;
endmodule

module nor_gate (output o, input a, input b);
   assign o = ~(a | b);
endmodule

module xor_gate (output o, input a, input b);
   // In Verilog, as in C, you use a carat '^' for exclusive OR.
   assign o = a ^ b;
endmodule

// This module implements the SR latch that we studied in Reading #11.
// You draw two NOR gates.  The output of the first NOR gate is called 'q'.
// The output of the second NOR gate is called 'qbar' (i.e. the opposite
// of 'q').  The first NOR gate's inputs are connected to r (reset) and qbar.
// The second NOR gate's inputs are connected to s (set) and q.  Normally,
// neither r nor s is asserted, and q keeps its previous state.  If you
// assert r (but not s), then q goes to 0.  If you assert s (but not r), then
// q goes to 1.  An undefined state results if r and s are asserted together.
module sr_latch (output q, input s, input r);
   // We define the wire 'qbar' here, but we don't connect it to anything,
   // because using it as the output argument of the second nor_gate will
   // connect it.  You can see that the wire called 'qbar' connects the
   // output of the second nor_gate to an input of the first nor_gate,
   // just as you would do if you were drawing a schematic diagram of an
   // SR latch.
   wire qbar;
   nor_gate mynor1 (q, r, qbar);
   nor_gate mynor2 (qbar, s, q);
endmodule
```

Today we're going to learn a bit of Verilog by immersion, and soon I'll give you some tutorial material to read. Anyway, you'll mainly be making minor edits to snippets of Verilog code that I provide, so you're not required to learn too many details of Verilog. Verilog is a programming language that is used to describe digital logic systems. It is widely used for both FPGA programming and the design of integrated circuits. In this course, we'll just dabble in Verilog. If you're eager to learn more of the details when the course is over, an excellent introductory book is *FPGA Programming by Verilog Examples* by Pong P. Chu.

(c) Now let's explore what this Verilog program (called `part1.v`) is telling the FPGA to do.

The 8 green LEDs are numbered 0 to 7, from right to left. The 8 switches below the LEDs are also numbered 0 to 7, from right to left. Each switch is 1=UP, 0=DOWN. Move switches 0 and 1 back and forth to check that LED 0 shows the AND of switches 0 and 1. Check the behavior of the LED in response to the switches, and relate it to the Verilog source code. (You can also double-click `part1.v` in **sources** tab of ISE to open it up and edit it, if you like.)

Check that LED 7 is the AND of switches 6 and 7.

Now check that LED 1 is the NAND of switches 0 and 1.

Check that LED 2 is the OR of switches 0 and 1.

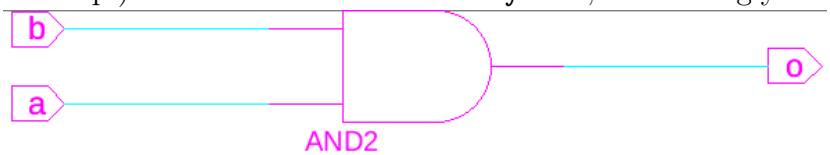
Check that LED 3 is the NOR of switches 0 and 1, and again make sure you see how it is done.

Check that LED 4 is the XOR of switches 0 and 1, and again look at the Verilog syntax.

Now notice the SR latch (the first flip-flop-like device that we studied in today's reading), implemented using two NOR gates. LED 5 displays the state, **Q**, of the SR latch. Button 0 is the **set** input, and button 1 is the **reset** input. Check that the set and reset features work as you expect, and check that LED 5 remembers whether the most recent command was set vs. reset.

Also notice that button 3 is connected directly to LED 6, so that you can see that the buttons are only on while you hold them down.

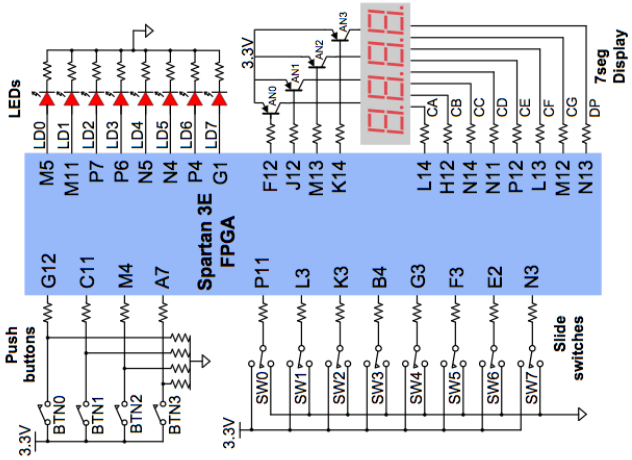
(d) Now let's ask the Xilinx ISE software to show us in schematic form its interpretation of this Verilog program. In the **processes** panel, click the **(+)** plus sign next to **Synthesize**, then double-click **View RTL Schematic**. You will see a box indicating the inputs and outputs of the top-level FPGA program. Double-click the box to see what is inside. You will see what looks like a schematic diagram, with boxes for the various module instances like `myand1`. Double-click on each of these boxes to look inside. (When you're done, right-click and choose **pop to the calling schematic** to navigate back up.) Here is what I see inside `myand1`, comfortingly enough:



Notice that when you look inside `mynand1`, you see the DeMorgan's equivalent of a NAND gate: instead of an AND with a bubble on its output, you see an OR with bubbles on both inputs.

Look inside `mysr1` (both the schematic and the original Verilog) and make sure that you understand what became of the `wire qbar` that connects one NOR gate's output with an input of the other NOR gate.

Now hold down button 3 while you slide switches 0–6 up and down one at a time, and notice what happens to the segments of the red 7-segment display. Notice that each of the display's 7 segments is controlled by one switch. Can you find a set of switch positions to spell out the numeral 5? Now about the numeral 2? Also notice what happens when you hold buttons 2 and 3 at the same time. The figure below may help to explain what is going on. Notice that a single PNP transistor provides the power to each of the four 7-segment displays, while the 7 wires controlling the 7 segments of each LED are in fact shared between the four 7-segment displays. This is a trick that the manufacturer uses in order to reduce the number of required input/output pins. (I/O pins are often a precious resource in electronics.) If you want to use all four digits separately, you need to quickly (e.g. once per millisecond) alternate between the four PNP transistors, turning only one of them on at a time, while quickly wiggling the 7 segments for each LED digit. I'll do this for you next week.



Part 2

(To save time, you might just want to read through part 2 without actually doing anything. This is the least important part of today's lab.) Now replace the contents of your `part1.v` file with the contents of the following `part2.v` , which you can download from `positron.hep.upenn.edu/wja/P364_2012/part2.v` .

```
//
// part2.v
// Verilog source code for Part 2 of Lab 11
// PHYS 364, UPenn, fall 2012
//

`default_nettype none
```

```
module part1
  (
   input        mclk,  // 25 MHz clock built into the BASYS2 board
   output [6:0] seg,   // red 7-segment display to draw numbers & letters
   output       dp,    // decimal point for 7-seg (0=ON, 1=OFF)
   output [3:0] an,    // shared LED anode signal per 7-seg digit (0=ON)
   output [7:0] led,   // 8 green LEDs, just above sliding switches
   input  [7:0] sw,    // 8 sliding switches (up=1, down=0)
   input  [3:0] btn,   // 4 push buttons (normal=0, pushed=1)
   input  [4:1] ja,    // 4 pins (JA, NW corner) can connect to breadboard
   input  [4:1] jb,    // 4 pins (JB) can connect to breadboard
   input  [4:1] jc,    // 4 pins (JC) can connect to breadboard
   input  [4:1] jd     // 4 pins (JD, NE corner) can connect to breadboard
   );
   assign led[0] = sw[0] & sw[1];     // LED0 is the AND of SW0 and SW1
   assign led[7] = sw[6] & sw[7];     // LED7 is the AND of SW6 and SW7
   assign led[1] = ~(sw[0] & sw[1]);  // LED1 is the NAND of SW0 and SW1
   assign led[2] = sw[0] | sw[1];     // LED2 is the OR of SW0 and SW1
   assign led[3] = ~(sw[0] | sw[1]);  // LED3 is the NOR of SW0 and SW1
   assign led[4] = sw[0] ^ sw[1];     // LED4 is the XOR of SW0 and SW1
   assign led[6] = btn[3];            // LED6 does whatever button 3 does

   // LED5 is Q, BTN0 is SET, BTN1 is RESET, for SR latch
   srlatch mysr1 (led[5], btn[0], btn[1]);

   // These outputs are currently unused, but I connect them anyway to
   // keep the compiler from complaining.
   assign seg[6] = sw[6];
   assign seg[5] = sw[5];
   assign seg[4] = sw[4];
   assign seg[3] = sw[3];
   assign seg[2] = sw[2];
   assign seg[1] = sw[1];
   assign seg[0] = sw[0];
   assign dp = 0;
   assign an[3] = ~btn[3];
   assign an[2] = ~btn[2];
   assign an[1] = ~btn[1];
   assign an[0] = ~btn[0];
endmodule


module srlatch (output q, input s, input r);
  wire qbar = ~(q | s);
```

```
  assign q = ~(qbar | r);
endmodule
```

Then click **generate programming file** to make a new `part1.bit` output file. The behavior of this program is exactly the same as the behavior of `part1.v` . I mainly just want you to see two different ways of writing the same program — in one case, using many different modules to implement the various features, and in the other case, writing the equations directly in one module.

The other thing worth looking at here is the Xilinx compiler's schematic diagram representing this Verilog file. Again, in the **processes** panel, click the (+) plus sign next to **Synthesize**, then double-click **View RTL Schematic**. Then double-click the box representing the top-level FPGA program. Now the AND and OR gates are sitting at the top level. There are some inverters to convert the outputs into NAND and NOR.

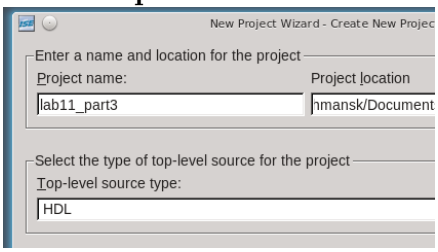
Part 3

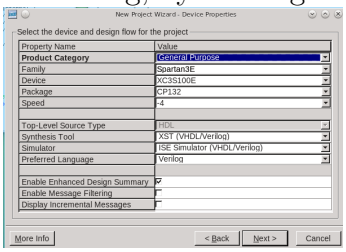
This time, let's create a new Xilinx project with the **New Project** wizard. First download source-code files `part3.v` and `basys2.ucf` from `positron.hep.upenn.edu/wja/P364_2012` .

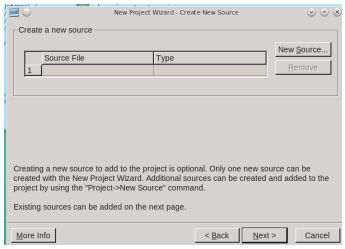
In the Xilinx ISE program, click **File** → **New Project**. Choose a project name and location, and set the top-level source type to "HDL."

In the next dialog, set **Product Category** to "General Purpose," set **Family** to "Spartan3E," set **Device** to "XC3S100E," set **Package** to "CP132," and set **Speed** to "-4." Click **next**.

Just **skip** the **Create New Source** dialog, by clicking **next**.



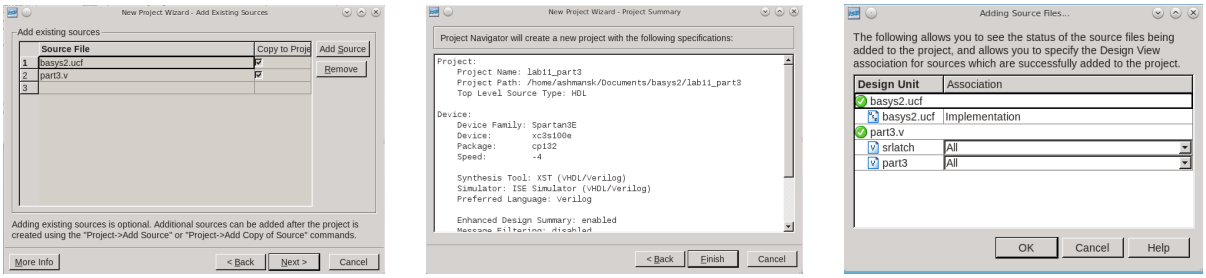




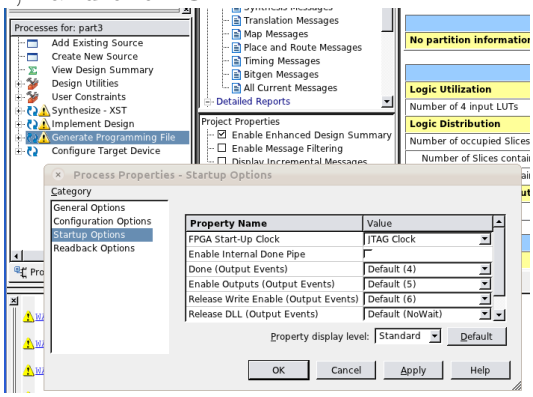
In the **Add Existing Sources** dialog, click **Add Source**, and add your `basys2.ucf` file and your `part3.v` file. Then click **next**.

In the **Project Summary** dialog, just click **finish**.

Finally, in the **Adding Source Files** dialog, just click **OK**.



Now in the **processes panel**, right-click **Generate Programming File** and click **Properties**. Under the **Startup Options** category, change **FPGA Start-Up Clock** to "JTAG Clock," and click **OK**.



Now in the **processes** panel, right-click **Generate Programming File** and click **Run**. The result (after a moment) will be a file `part3.bit` that you can load into the board using the ADEPT software. If you have trouble with the compiler, you can copy my `part3.bit` from `positron.hep.upenn.edu/wja/P364_2012` .

(a) The Verilog source code for `part3.v` is included below. Look through the code for the `add4bit` module and see if you can relate it to the 4-bit adder that we discussed in today's reading. Each bit of the sum is a 3-way XOR of two input bits and one carry bit, so that it is HIGH only if an odd number of these three bits are HIGH. Each carry bit is a 3-way OR, each term of which is an AND, such that the carry bit is HIGH only if two or more of these three bits are HIGH. Stare at this (and ask questions) until this makes sense.

(b) Now look at the lines in the main `part3` module where the `add4bit` module is *instantiated*, i.e. where the adder is plunked down onto the `part3` schematic. Notice that the adder inputs `a3 a2 a1 a0` come from switches 3–0 and that the adder inputs `b3 b2 b1 b0` come from swtiches 7–4. Then notice that the adder outputs `cout s3 s2 s1 s0` go to LEDs 4–0. Slide the switches up and down to spell out two 4-bit binary numbers A and B, and check that the adder displays the sum $A + B$ on the LEDs. You can also use push-button 3 to assert the `cin` input of the adder if you like. The reason that an adder has a `cin` input is so that you can e.g. chain together two 4-bit adders to make an 8-bit adder, and so on.

```
//
// part3.v
// Verilog source code for Part 3 of Lab 11
// PHYS 364, UPenn, fall 2012
//

`default_nettype none

module part3
  (
   input        mclk,  // 25 MHz clock built into the BASYS2 board
   output [6:0] seg,   // red 7-segment display to draw numbers & letters
   output       dp,    // decimal point for 7-seg (0=ON, 1=OFF)
   output [3:0] an,    // shared LED anode signal per 7-seg digit (0=ON)
   output [7:0] led,   // 8 green LEDs, just above sliding switches
   input  [7:0] sw,    // 8 sliding switches (up=1, down=0)
   input  [3:0] btn,   // 4 push buttons (normal=0, pushed=1)
   input  [4:1] ja,    // 4 pins (JA, NW corner) can connect to breadboard
   input  [4:1] jb,    // 4 pins (JB) can connect to breadboard
   input  [4:1] jc,    // 4 pins (JC) can connect to breadboard
   input  [4:1] jd     // 4 pins (JD, NE corner) can connect to breadboard
   );

   // These wires give understandable names to the inputs of
   // the 4-bit adder, just to make the connections more clear.
   wire a0 = sw[0];    // switches 0-3 will be a0, a1, a2, a3
   wire a1 = sw[1];
   wire a2 = sw[2];
   wire a3 = sw[3];
   wire b0 = sw[4];    // switches 4-7 will be b0, b1, b2, b3
   wire b1 = sw[5];
   wire b2 = sw[6];
   wire b3 = sw[7];
   wire cin = btn[3];  // push-button 3 will be carry-in
   wire cout, s3, s2, s1, s0;  // for the outputs of the 4-bit adder
   add4bit myadder1 (cout, s3, s2, s1, s0,
                     a3, a2, a1, a0,
                     b3, b2, b1, b0);
   assign led[0] = s0;    // LEDs 0-3 are the adder's SUM bits
   assign led[1] = s1;
   assign led[2] = s2;
   assign led[3] = s3;
   assign led[4] = cout;  // LED4 is carry-out of the ader

   // LED5 is Q, BTN0 is SET, BTN1 is RESET, for SR latch
```

```
   srlatch mysr1 (led[5], btn[0], btn[1]);

   // These outputs are currently unused, but I connect them anyway to
   // keep the compiler from complaining.
   assign led[6] = 0;
   assign led[7] = 0;
   assign seg[6] = sw[6];
   assign seg[5] = sw[5];
   assign seg[4] = sw[4];
   assign seg[3] = sw[3];
   assign seg[2] = sw[2];
   assign seg[1] = sw[1];
   assign seg[0] = sw[0];
   assign dp = 0;
   assign an[3] = ~btn[3];
   assign an[2] = ~btn[2];
   assign an[1] = ~btn[1];
   assign an[0] = ~btn[0];
endmodule


// This is the 4-bit adder circuit that we saw in
// the reading assignment.  It's the second version,
// which includes carry-in and carry-out pins, so that
// in principle you can chain two of them together to
// make an 8-bit adder, etc.
module add4bit (output cout, output s3, output s2, output s1, output s0,
input a3, input a2, input a1, input a0,
input b3, input b2, input b1, input b0, input cin);
   wire   c0   = (a0 & b0) | (a0 & cin) | (b0 & cin);
   wire   c1   = (a1 & b1) | (a1 & c0)  | (b1 & c0);
   wire   c2   = (a2 & b2) | (a2 & c1)  | (b2 & c1);
   assign cout = (a3 & b3) | (a3 & c2)  | (b3 & c2);
   assign s0   = a0 ^ b0 ^ cin;
   assign s1   = a1 ^ b1 ^ c0;
   assign s2   = a2 ^ b2 ^ c1;
   assign s3   = a3 ^ b3 ^ c2;
endmodule


// This implements an SR latch, using two NOR gates,
// as we saw in the reading assignment.
module srlatch (output q, input s, input r);
  wire qbar = ~(q | s);
  assign q = ~(qbar | r);
```

```
endmodule


// This magic incantation is how you tell the Xilinx
// compiler to make a D-type flip-flop.  Just accept
// it as an idiom, rather than trying at this stage
// to follow exactly what the syntax means.  But the
// jist of it is that whenever a positive edge of the
// clk signal is seen, the contents of d are copied to
// the new contents of 'q_reg', which in turn is wired
// to the 'q' output of this circuit.  We will use this
// module in parts 4 and 5.
module dff (output q, input clk, input d);
   reg q_reg;
   always @ (posedge clk) q_reg <= d;
   assign q = q_reg;
endmodule
```