Physics 364, Fall 2014, Lab #24 **Name:** ____________________

(Field Programmable Gate Arrays: 1)

Monday, November 24 (section 401); Tuesday, November 25 (section 402)

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

This file: http://positron.hep.upenn.edu/wja/p364/2014/files/lab24.pdf

Today's 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 http://www.digilentinc.com/Products/Detail.cfm?Prod=BASYS2 . The BASYS2 manual is at http://www.digilentinc.com/Data/Products/BASYS2/Basys2_rm.pdf .

There is very little for you to write up for today's lab, but there are a few questions here and there, which you should look out for. Also, before you leave class, you need to show us how far you got through the material. There are six parts: try to budget your time so that you get through all of them!

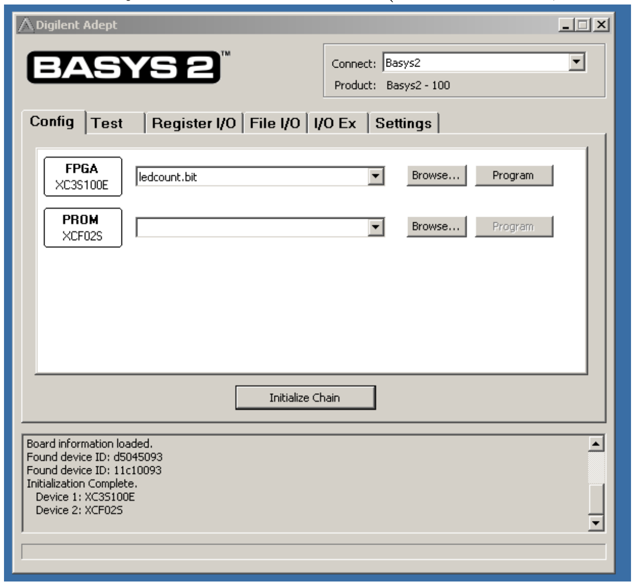
The FPGA is connected to 8 ON/OFF switches (lower edge of board, left side): the corresponding FPGA inputs are HIGH when the switches are UP and LOW when the switches are DOWN. Just above the 8 switches are 8 LEDs: the LEDs are ON when the corresponding FPGA outputs are HIGH and OFF when the outputs 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 we 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, astronomy, and presumably other fields in which scientists build their own instruments.

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 (but probably it is not, if your board has been used before), then you will see a 1111, 2222, 3333, . . ., FFFF pattern count on the 7-segment LED display. If this is what you see, then 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. Since nearly all of our BASYS2 boards were used in the Fall 2012 course, you probably will not need to do anything described in this paragraph.

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
`http://positron.hep.upenn.edu/wja/p364/2014/files/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). Click the **program** button! 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).



Connect: Basys2
Product: Basys2 - 100

FPGA XC3S100E: ledcount.bit

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

(b) Now use your web browser to download
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24.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 `lab24.xise`, 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 `lab24_part1.bit`, which you can now load into the BASYS2 board using the ADEPT software. There is also a pre-compiled copy of `lab24_part1.bit` at
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part1_precompiled.bit

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

```
//
// lab24_part1.v
// Verilog source code for Part 1 of Lab 24
// PHYS 364, UPenn, fall 2014
//

// 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 lab24_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 inputs are currently unused, but I connect them to some
    // unnecessary logic so that the Xilinx compiler doesn't complain
    // about unused inputs.
    wire dummy = ({ja,jb,jc,jd,mclk}==0);

   // 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 = dummy;
   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
```

```verilog

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
```

As you learned in the reading, 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 Prototyping by Verilog Examples* by Pong P. Chu. I learned Verilog by reading Chu's book, so I can highly recommend it. Another book I have been meaning to read (but haven't yet), as it has good reviews on Amazon, is *Verilog by Example* by Blaine Readler.
http://www.amazon.com/FPGA-Prototyping-Verilog-Examples-Spartan-3/dp/0470185325
http://www.amazon.com/Verilog-Example-Concise-Introduction-Design/dp/0983497303

(c) Now let's explore what this Verilog program (called `lab24_part1.v`) is telling the FPGA to do.

(i) 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 `lab24_part1.v` in **sources** tab of ISE to open it up and edit it, if you like.)

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

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

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

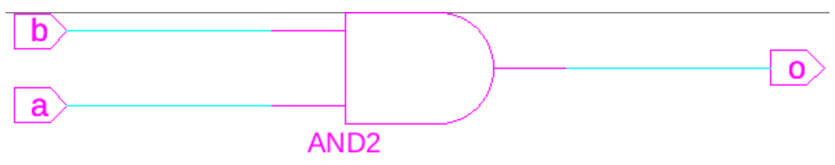
(v) Check that LED 3 is the NOR of switches 0 and 1, and again make sure you see how it is done.

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

(vii) 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.

(viii) 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**. (If you get a dialog box asking to choose between two display options, choose the second option ["Start with a schematic of the top-level block"] and uncheck the "show this dialog" option.) 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. Here is what I see inside `myand1`, comfortingly enough:

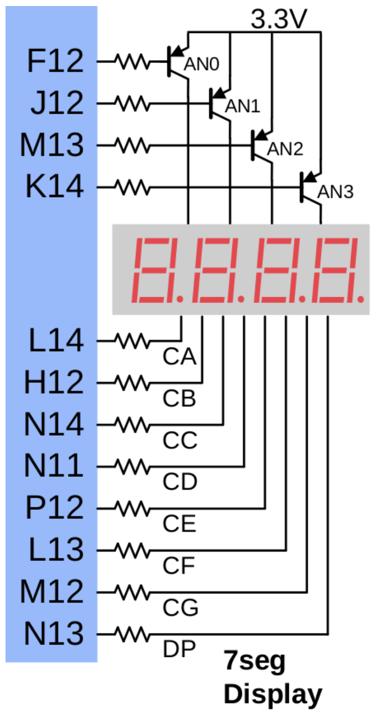


Look inside `mysr1` (both the schematic and the original Verilog) and make sure that you understand what became of the `wire qbar` that connects one NOR gate's output with an input of the other NOR gate.

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 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 `lab24_part1.v` file with the contents of the following `lab24_part2.v` , which you can download from http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part2.v . The easiest way to do this is to edit the contents of your existing `lab24_part1.v` in ISE, then "select all," then delete. Now just click on the `lab24_part2.v` file in your web browser to display its contents. Click "control-A" to select all, then click "control-C" to copy. Now go back to ISE and click "control-V" to paste, then "control-S" to save. The result is that you've overwritten the old contents of `lab24_part1.v` with the online contents of `lab24_part2.v` . Overwriting the old contents of `lab24_part1.v` may be somewhat distasteful, but it is expedient.

```
//
// lab24_part2.v
// Verilog source code for Part 2 of Lab 24
// PHYS 364, UPenn, fall 2014
//

`default_nettype none

module lab24_part2
  (
   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 inputs are currently unused, but I connect them to some
   // unnecessary logic so that the Xilinx compiler doesn't complain
   // about unused inputs.
    wire dummy = ({ja,jb,jc,jd,mclk}==0);

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


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

Then double-click **generate programming file** to make a `lab24_part2.bit` output file. The behavior of this program is exactly the same as the behavior of `lab24_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. (**PLEASE NOTE:** it took me more than one try to get ISE to show me an up-to-date schematic diagram. I ended up doing **Project**→"Cleanup Project Files," then quitting and restarting ISE, then re-running **Synthesize**, and then finally "View RTL Schematic.")

Part 3

This time, let's create a new Xilinx project with the **New Project** wizard. First download these two source-code files:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part3.v
http://positron.hep.upenn.edu/wja/p364/2014/files/basys2.ucf

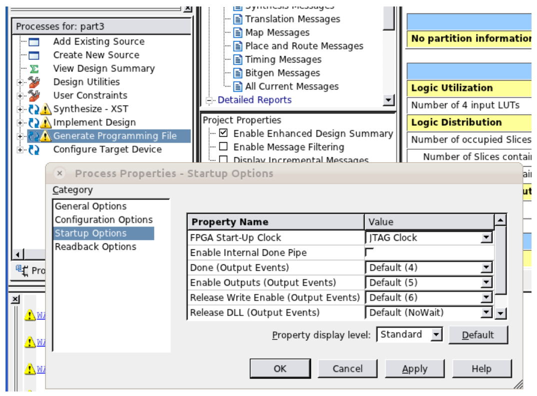
In the Xilinx ISE program, click **File** → **New Project**. Choose a project name and location, and set the top-level source type to "HDL."

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**.

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

Now from the **Project** menu, choose **Add Source**. Add both `lab24_part3.v` and `basys2.ucf` , which you just downloaded a moment ago.

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



Property Name	Value
FPGA Start-Up Clock	JTAG Clock
Enable Internal Done Pipe	
Done (Output Events)	Default (4)
Enable Outputs (Output Events)	Default (5)
Release Write Enable (Output Events)	Default (6)
Release DLL (Output Events)	Default (NoWait)

Now in the **processes** panel, right-click **Generate Programming File** and click **Run**. The result (after a moment) will be a file `lab24_part3.bit` that you can load into the board using the ADEPT software. If you have trouble with the compiler, you can copy my `lab24_part3.bit` from
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part3.bit

(a) The Verilog source code for `lab24_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.

```
//
// lab24_part3.v
// Verilog source code for Part 3 of Lab 24
// PHYS 364, UPenn, fall 2014
//

`default_nettype none

module lab24_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, cin);
   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

   // Assign dummy values to output pins that are currently unused
   assign {led[7:5],seg[6:0],dp} = 0;
   assign an[3:0] = 4'b1111;
```

```
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
```

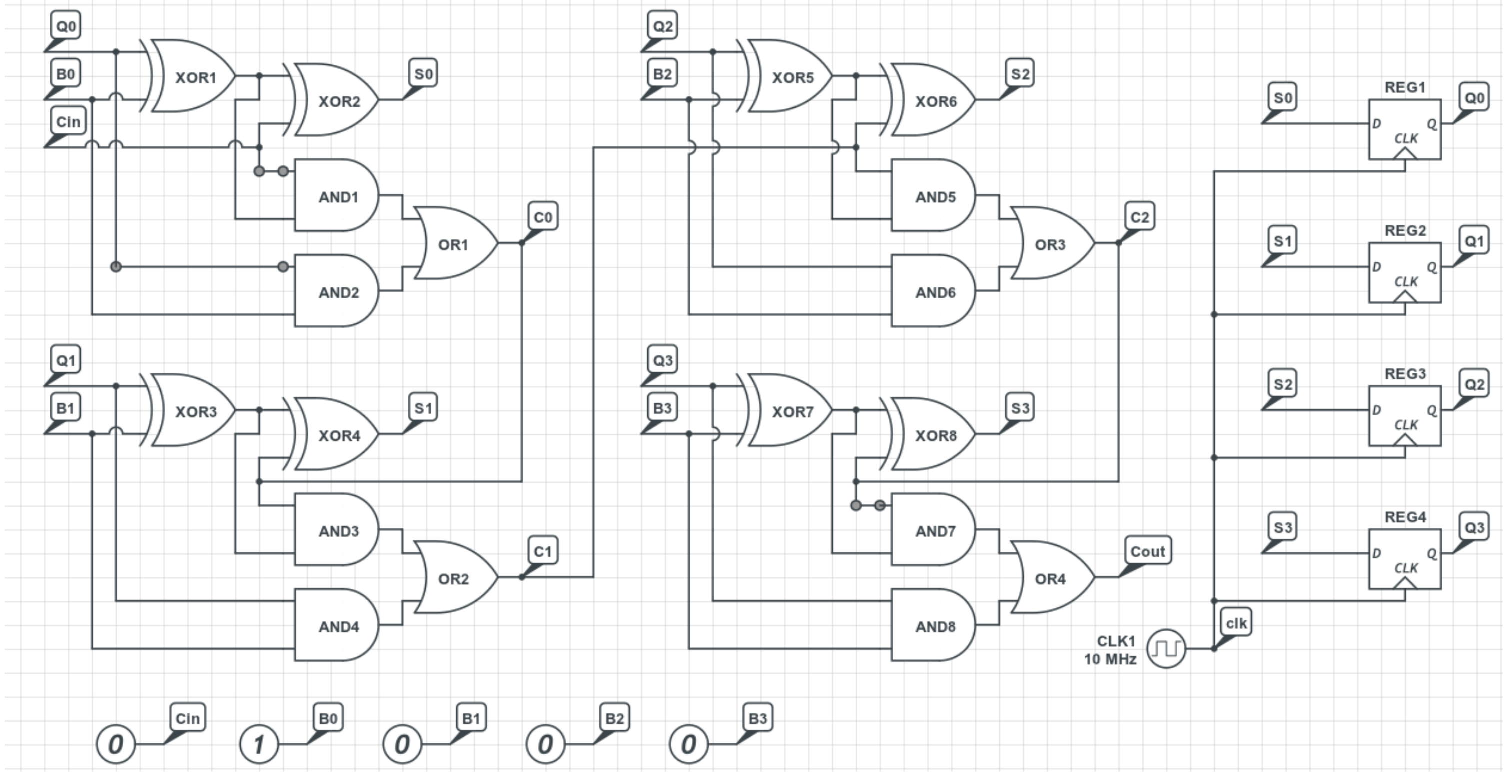
(b) Now look at the lines in the main `lab24_part3` module where the `add4bit` module is *instantiated*, i.e. where the adder is plunked down onto the `lab24_part3` schematic. Notice that the adder inputs `a3 a2 a1 a0` come from switches 3–0 and that the adder inputs `b3 b2 b1 b0` come from swtiches 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.

How did you check that the adder works as expected?

Part 4

Thus far (with just one exception), we've implemented purely **combinational** logic in the Xilinx FPGA, using the Verilog programming language. We've made a 4-bit adder add up the two 4-bit numbers specified by the positions of the 8 sliding switches, and we've displayed the adder's outputs on 5 green LEDs. The one piece of **sequential** logic that we implemented was a single SR latch, which could remember whether the *set* or *reset* button had been pressed more recently, as indicated by lighting (or not lighting) an LED to display the latch's **Q** state.

Let's continue now by programming the FPGA to implement the **4-bit counter** that you so tediously wired up in Lab 23 (though this version will include the carry-in bit as well):



This counter consists of a 4-bit adder and four **D-type flip-flops**. The counter's present value is represented by the states of the four flip-flop outputs: Q_3, Q_2, Q_1, Q_0. On each clock cycle, the constant binary value 0001 is added to the counter's present value.

Here was the verilog code for the adder:

```
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
```

It turns out that the following recipe is how you tell Verilog to make a D-type flip-flop. For the moment, just treat it as a magic incantation. I will put some explanation of what it means into the reading. The basic idea is that whereas a **wire** merely connects two things together (like a wire on your breadboard), a **reg** can maintain an internal state. The statement

```
always @ (posedge clk) q_reg <= d;
```

tells Verilog that every time it detects a positive edge (i.e. a low-to-high transition) on the **clk** input, it should update **q_reg** to reflect the present state of the **d** input. That sounds just like the behavior of a DFF!

```
// 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 dflipflop (output q, input clk, input d);
   reg q_reg;
   always @ (posedge clk) q_reg <= d;
   assign q = q_reg;
endmodule
```

Putting all of the pieces together looks like this:

```
//
// lab24_part4.v
// Verilog source code for Lab 24 part 4
// PHYS 364, UPenn, fall 2014
//

`default_nettype none

module lab24_part4
  (
   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
   );

    // an[3] is Q, btn[0] is SET, btn[1] is RESET, for SR latch
    wire srq;
    srlatch mysr1 (srq, btn[0], btn[1]);
    // make the left-most 7-segment LED light up if Q is true
    assign an[3] = ~srq;

    // We will use the 'Q' output of the SR latch as a clock
    // for the counter's flip-flops, so that we can clock the
    // counter by hand, one clock cycle at a time.
    wire clock = srq;

    // Define wires for the adder's S (sum) and COUT (carry) outputs
    wire s3, s2, s1, s0, cout;
    // The adder's CIN (carry in) will always be zero
    wire cin = 0;
    // The adder's B input will always be binary constant 0001
    wire b3=0, b2=0, b1=0, b0=1;
    // The adder's A inputs will be the flip flops' Q outputs
    wire q3, q2, q1, q0;
    add4bit myadder1 (cout,
              s3, s2, s1, s0,
              q3, q2, q1, q0,
              b3, b2, b1, b0, cin);
    // Each flip-flop's D input will be one of the adder's
    // S (sum) output bits
    dflipflop mydff3 (q3, clock, s3);
    dflipflop mydff2 (q2, clock, s2);
    dflipflop mydff1 (q1, clock, s1);
    dflipflop mydff0 (q0, clock, s0);

    // Let the left four LEDs show the adder's S (sum) outputs
    assign led[7] = s3;
    assign led[6] = s2;
    assign led[5] = s1;
    assign led[4] = s0;

    // Let the right four LEDs show the flip flops' Q outputs
    assign led[3] = q3;
    assign led[2] = q2;
    assign led[1] = q1;
    assign led[0] = q0;

    // Light up all 7 segments (plus decimal) for any 7-segment digit
    // that is activated.  Digits 2,1,0 simply indicate whether buttons
    // 2,1,0 are pressed.  Digit 3 indicates the state of the SR latch
    // (see comment above for an[3]).
    assign seg[6:0] = 0;
    assign dp = 0;
    assign an[2:0] = ~btn[2:0];
endmodule


// four-bit adder
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


// magic incanation to make a D-type flip-flop
module dflipflop (output q, input clk, input d);
    reg q_reg;
    always @ (posedge clk) q_reg <= d;
    assign q = q_reg;
endmodule


// S-R latch (the simplest kind of flip-flop)
module srlatch (output q, input s, input r);
    wire qbar;
    nor mynorgate1 (qbar, q, s);
    nor mynorgate2 (q, qbar, r);
endmodule
```

Download `http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part4.v` and use its contents to replace the present contents of your ISE project's `lab24_part1.v` (or whatever is the name of the Verilog source file in your open ISE project). If you don't already have an ISE project open from Part 3 (maybe you took a break after Part 3), you can find a `.zip` file containing a fresh ISE project at
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part4.zip`
whose contents you can just extract into a new folder, then double-click on `lab24_part4.xise` it that folder to open the ISE project. In case you have technical difficulties with the Xilinx compiler, I've put a pre-compiled version of `lab24_part4.bit` at
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part4.bit`

Notice that we connected one of the adder's two inputs to the binary constant 0001 and the other input to the **Q** outputs of the four flip-flops. On each rising edge of the clock signal, the sum increments by 1.

Notice that the 4-bit output of the adder is displayed on LEDs 7,6,5,4, and notice that the 4 flip-flop **Q** outputs are displayed on LEDs 3,2,1,0.

For a clock signal, we are using the **Q** output of an SR latch, whose **reset** input is **button 1** and whose **set** input is **button 0**. You can monitor the present value of the latch's **Q** output by looking at the left-most digit of the 7-segment display. By alternatively pressing buttons 1 and 0, you can manually clock the counter.

Try clocking the counter by pushing (alternately) buttons 1 and 0.

Notice that the four **Q** bits from the DFFs are always one clock cycle behind the four **S** bits of the adder. In other words, the value displayed on the right 4 LEDs just after the clock edge is the value that had been displayed on the left 4 LEDs just before the clock edge.

Take the time to understand how the Verilog code is equilvalent to the schematic diagram shown above.

Now **modify the program** so that the **B** input bits to the adder are the switch values `sw[3]`, `sw[2]`, `sw[1]`, `sw[0]` instead of the constant values 0, 0, 0, 1. Indicate below how you modified the program.

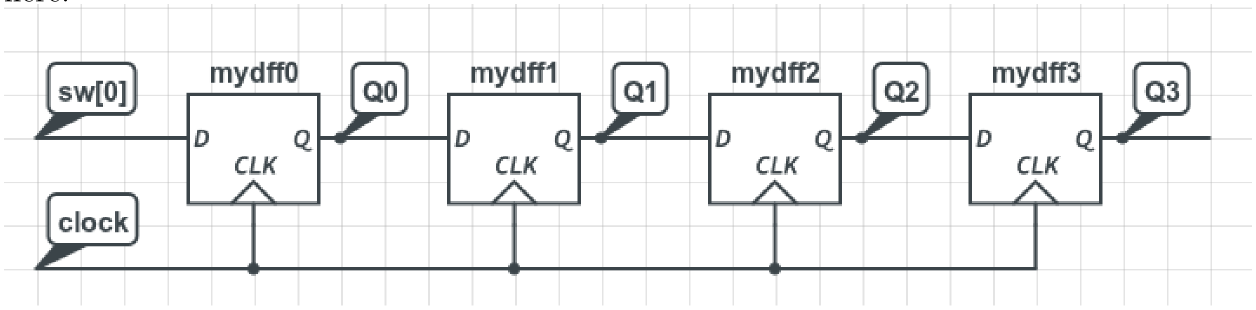
Now you will need to set the right four switches to DOWN, DOWN, DOWN, UP to get the counter to count up by ones. Give it a try! Can you make the counter count up by twos? By threes?

What settings should you use on the four switches to make your counter count DOWN by ones? Hint: do you remember the 4-bit two's complement representation for -1?

Part 5

Now we're going to connect 8 D-type flipflops together into a **shift register**. This configuration is also known as a **pipeline**, because data flows from one flip-flop to the next on each clock cycle, as if it were moving through a pipe. The motion is also analogous to that of a factory assembly line.

The schematic diagram looks something like this, though I only drew 4 of the 8 flip-flops here:



Here is the Verilog code that is equilvalent to the above schematic (but with 8 DFFs instead of just 4):

```
// The first flip-flop's D input will be sw[0] (the right-most
// switch).  Each subsequent flip-flop's D input will be the
// previous flip-flop's Q output.
wire q7, q6, q5, q4, q3, q2, q1, q0;
dflipflop mydff0 (q0, clock, sw[0]);
dflipflop mydff1 (q1, clock, q0);
dflipflop mydff2 (q2, clock, q1);
dflipflop mydff3 (q3, clock, q2);
dflipflop mydff4 (q4, clock, q3);
dflipflop mydff5 (q5, clock, q4);
dflipflop mydff6 (q6, clock, q5);
dflipflop mydff7 (q7, clock, q6);
```

Notice that each flip-flop's **D** input is connected to the previous flip-flop's **Q** output. We also send all of the **Q** outputs to the 8 green LEDs:

```
// Display the 8 flip flops' Q values on the 8 green LEDs.
assign led[7] = q7;
assign led[6] = q6;
assign led[5] = q5;
assign led[4] = q4;
assign led[3] = q3;
assign led[2] = q2;
assign led[1] = q1;
assign led[0] = q0;
```

Here is the complete program, which you should download from
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part5.v You can paste the contents into the ISE editor in place of your previous contents of `lab24_part4.v` .

```
//
// lab24_part5.v
// Verilog source code for Lab 24 (part 5)
// PHYS 364, UPenn, fall 2014
//

‘default_nettype none

module lab24_part5
  (
   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
   );

    // an[3] is Q, btn[0] is SET, btn[1] is RESET, for SR latch
    wire srq;
    srlatch mysr1 (srq, btn[0], btn[1]);
    // make the left-most 7-segment LED light up if Q is true
    assign an[3] = ~srq;

    // We will use the ’Q’ output of the SR latch as a clock
    // for the counter’s flip-flops, so that we can clock the
    // counter by hand, one clock cycle at a time.
    wire clock = srq;

    // The first flip-flop’s D input will be sw[0] (the right-most
    // switch).  Each subsequent flip-flop’s D input will be the
    // previous flip-flop’s Q output.
    wire q7, q6, q5, q4, q3, q2, q1, q0;
    dflipflop mydff0 (q0, clock, sw[0]);
    dflipflop mydff1 (q1, clock, q0);
    dflipflop mydff2 (q2, clock, q1);
    dflipflop mydff3 (q3, clock, q2);
    dflipflop mydff4 (q4, clock, q3);
    dflipflop mydff5 (q5, clock, q4);
    dflipflop mydff6 (q6, clock, q5);
    dflipflop mydff7 (q7, clock, q6);

    // Display the 8 flip flops' Q values on the 8 green LEDs.
    assign led[7] = q7;
    assign led[6] = q6;
    assign led[5] = q5;
    assign led[4] = q4;
    assign led[3] = q3;
    assign led[2] = q2;
    assign led[1] = q1;
    assign led[0] = q0;

    // Light up all 7 segments (plus decimal) for any 7-segment digit
    // that is activated.  Digits 2,1,0 simply indicate whether buttons
    // 2,1,0 are pressed.  Digit 3 indicates the state of the SR latch
    // (see comment above for an[3]).
    assign seg[6:0] = 0;
    assign dp = 0;
    assign an[2:0] = ~btn[2:0];
endmodule


// D-type flip-flop (treat this as a magic incantation for now)
module dflipflop (output q, input clk, input d);
    reg q_reg;
    always @ (posedge clk) q_reg <= d;
    assign q = q_reg;
endmodule


// SR latch
module srlatch (output q, input s, input r);
    wire qbar;
    nor mynorgate1 (qbar, q, s);
    nor mynorgate2 (q, qbar, r);
endmodule
```

Once you have the program compiled and loaded into your FPGA board, try clocking the flip-flops by alternately pressing buttons 1 and 0 (which alternately sets and resets the SR latch whose **Q** output supplies the clock signal). If you do this with SW0 in the DOWN position, you should see nothing very interesting happen.

Now set SW0 to the UP position. You should see the LEDs light up one-by-one.

Try setting SW0 back to the DOWN position. You should see darkness shift left clock-by-clock in the same way as you saw light shifting left before.

Keep clocking in ZEROs until all of the LEDs are off. Now clock in a single ONE bit. In other words, flip SW0 to UP, then do a single clock cycle, then flip SW0 back to DOWN, then keep clocking. Do you see how the single ONE bit shifts from right to left? **Briefly explain what is happening.**

A shift register is sometimes used to delay data so that they can be kept around while a lengthy computation completes. The length of a shift register is analogous to the length of the *pipeline* on your home computer's microprocessor.

Part 6

Now here is a slightly different kind of D-type flip-flop. It has an additional input called `enable`. This is the type of flip-flop that I use most commonly in my own elecronics-design work.

```
// This magic incantation is how you tell the Xilinx compiler to make
// a D-type flip-flop with the ENABLE feature.
module dffe (output q, input clk, input d, input enable);
    reg q_reg;
    always @ (posedge clk) if (enable) q_reg <= d;
    assign q = q_reg;
endmodule
```

Now we're going to modify the main part of the Part 5 program to look like this instead:

```
    // Now we set up 8 D-type flip-flops that have an additional
    // input wire called ENABLE.  If ENABLE is TRUE, then the
    // flip-flop behaves exactly as an ordinary DFF.  But if ENABLE
    // is FALSE, then the DFF does nothing.

    // Now we connect each flip flop's D input to the NOT of its own
    // Q output.  So if ENABLE is asserted, the flip-flop just toggles
    // back and forth between 0 and 1.  But if ENABLE is deasserted,
    // the flip-flop does nothing.
    wire q7, q6, q5, q4, q3, q2, q1, q0;
    wire d7 = ~q7;
    wire d6 = ~q6;
    wire d5 = ~q5;
    wire d4 = ~q4;
    wire d3 = ~q3;
    wire d2 = ~q2;
    wire d1 = ~q1;
    wire d0 = ~q0;

    // Here is the tricky part:  the first flip flop is always enabled.
    // The second flip flop is only enabled if the first one is HIGH.
    // The third flip flop is only enabled if the first two are HIGH.
    // The fourth is only enabled if the first three are HIGH.  Etc.
    wire e0 = 1;
    wire e1 = q0;
    wire e2 = q0 & q1;
    wire e3 = q0 & q1 & q2;
    wire e4 = q0 & q1 & q2 & q3;
    wire e5 = q0 & q1 & q2 & q3 & q4;
    wire e6 = q0 & q1 & q2 & q3 & q4 & q5;
    wire e7 = q0 & q1 & q2 & q3 & q4 & q5 & q6;
    dffe mydff0 (q0, clock, d0, e0);
    dffe mydff1 (q1, clock, d1, e1);
    dffe mydff2 (q2, clock, d2, e2);
    dffe mydff3 (q3, clock, d3, e3);
    dffe mydff4 (q4, clock, d4, e4);
    dffe mydff5 (q5, clock, d5, e5);
    dffe mydff6 (q6, clock, d6, e6);
    dffe mydff7 (q7, clock, d7, e7);
```

Can you guess what this program will do?! See if you can figure out what it is doing and why it works.

You can find the complete source code at
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part6.v Download this file and study it until it makes sense to you. Then paste its contents into your ISE editor in place of the contents of the Part 5 Verilog file that you have open. Then compile it and load the resulting `lab24_part6.bit` into your board. If you run into technical difficulties, you can find a pre-compiled `.bit` file at
http://positron.hep.upenn.edu/wja/p364/2014/files/lab24_part6.bit