Physics 364, Fall 2014, Lab #25 **Name:** _____

(Field Programmable Gate Arrays: 2)

Monday, December 1 (section 401); Tuesday, December 2 (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/lab25.pdf

Useful files: http://positron.hep.upenn.edu/wja/p364/2014/files/?C=M;O=D

Part 1

(a) Let's start from where we left off last week. We finished off by putting together eight D-type flip-flops (with enable pins) to make an 8-bit counter, whose Q values were displayed on the green LEDs of the BASYS2 board. It eventually became somewhat tedious to have to clock the counter by hand. Let's fix that.

The BASYS2[1] board includes a 50 MHz on-board clock, which is wired to the `mclk` input of the FPGA. 50 MHz is a good speed for a serious electronics project, but it's too fast for us to see (by eye, without using an oscilloscope) what's happening while we're just learning about digital circuits. Let's find a way to slow it down to a more reasonable speed.

Suppose that you clocked your 8-bit counter from last week at frequency $f = 1$ Hz. Then bit 0 of the counter would toggle at frequency $f/2$, bit 1 would toggle at $f/4$, ..., and bit 7 would toggle at $f/256$. For this reason, a single D-type flip-flop whose D input is connected to the NOT of its own Q output is called a "divide-by-two," because it is effectively dividing the clock frequency in half. The 8-bit counter divides the clock frequency by 256.

Let's make an 18-bit counter, to divide 50 MHz down to about 191 Hz, which will let us build circuits whose effects we can begin to see with our eyes. By clocking last week's 8-bit counter with the resulting 191 Hz clock, we will find that bit 7 of this 8-bit counter (which is connected to LED7 on the board) blinks with a period of 1.34 seconds.

```
1  //
2  // lab25_part1.v
3  // Verilog source code for Lab 25 (part 1)
4  // PHYS 364, UPenn, fall 2014
5  //
6
7  `default_nettype none
8
9  module lab25_part1
10   (
11    input        mclk,  // 50 MHz clock built into the BASYS2 board
12    output [6:0] seg,   // red 7-segment display to draw numbers & letters
```

[1] http://www.digilentinc.com/Products/Detail.cfm?Prod=BASYS2

```
    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
    );

     // This counter will count 0..262143 over and over again,
     // using the 50 MHz on-board oscillator as its clock input
     wire [17:0] count18bit;
     counter_18bit mycounter1 (count18bit, mclk);

     // Now we can use bit 17 of the clock to get a frequency
     // (50 MHz) / (262144) = 190.7 Hz
     wire clk191Hz = count18bit[17];

     // Now instead of using the 'Q' output of the SR latch as
     // a clock for the 8-bit counter's flip-flops, as we did
     // last week, let's use our new 191 Hz clock instead.
     wire clock = clk191Hz;

     // The rest of this module is the same as it was at the end of
     // last week's lab.

     // 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;

```

```
60      // Here is the tricky part:  the first flip flop is always enabled.
61      // The second flip flop is only enabled if the first one is HIGH.
62      // The third flip flop is only enabled if the first two are HIGH.
63      // The fourth is only enabled if the first three are HIGH.  Etc.
64      wire e0 = 1;
65      wire e1 = q0;
66      wire e2 = q0 & q1;
67      wire e3 = q0 & q1 & q2;
68      wire e4 = q0 & q1 & q2 & q3;
69      wire e5 = q0 & q1 & q2 & q3 & q4;
70      wire e6 = q0 & q1 & q2 & q3 & q4 & q5;
71      wire e7 = q0 & q1 & q2 & q3 & q4 & q5 & q6;
72      dffe mydff0 (q0, clock, d0, e0);
73      dffe mydff1 (q1, clock, d1, e1);
74      dffe mydff2 (q2, clock, d2, e2);
75      dffe mydff3 (q3, clock, d3, e3);
76      dffe mydff4 (q4, clock, d4, e4);
77      dffe mydff5 (q5, clock, d5, e5);
78      dffe mydff6 (q6, clock, d6, e6);
79      dffe mydff7 (q7, clock, d7, e7);
80
81      // Display the 8 flip flops' Q values on the 8 green LEDs.
82      assign led[7] = q7;
83      assign led[6] = q6;
84      assign led[5] = q5;
85      assign led[4] = q4;
86      assign led[3] = q3;
87      assign led[2] = q2;
88      assign led[1] = q1;
89      assign led[0] = q0;
90
91      // These outputs are currently unused, but I connect them
92      // anyway to keep the compiler from complaining.
93      assign seg[6:0] = 0;
94      assign dp = 0;
95      assign an[3:0] = ~btn[3:0];
96  endmodule
97
98
99  // This is one way (somewhat tedious) to make an 18-bit
100 // counter, by using 18 DFFE's, each of whose D inputs
101 // is wired to the NOT of its own Q output.
102 module counter_18bit (output [17:0] q, input clk);
103     wire [17:0] enable;
104     assign enable[ 0] = 1;
105     assign enable[ 1] = q[0];
106     assign enable[ 2] = q[ 1:0]=='b11;
```

```
107     assign enable[ 3] = q[ 2:0]==’b111;
108     assign enable[ 4] = q[ 3:0]==’b1111;
109     assign enable[ 5] = q[ 4:0]==’h1f;
110     assign enable[ 6] = q[ 5:0]==’h3f;
111     assign enable[ 7] = q[ 6:0]==’h7f;
112     assign enable[ 8] = q[ 7:0]==’hff;
113     assign enable[ 9] = q[ 8:0]==’h1ff;
114     assign enable[10] = q[ 9:0]==’h3ff;
115     assign enable[11] = q[10:0]==’h7ff;
116     assign enable[12] = q[11:0]==’hfff;
117     assign enable[13] = q[12:0]==’h1fff;
118     assign enable[14] = q[13:0]==’h3fff;
119     assign enable[15] = q[14:0]==’h7fff;
120     assign enable[16] = q[15:0]==’hffff;
121     assign enable[17] = q[16:0]==’h1ffff;
122     // Every ’D’ input is the NOT of the corresponding ’Q’ output
123     wire [17:0] d = ~q[17:0];
124     dffe mydff0  (q[ 0], clk, d[ 0], enable[0]);
125     dffe mydff1  (q[ 1], clk, d[ 1], enable[1]);
126     dffe mydff2  (q[ 2], clk, d[ 2], enable[2]);
127     dffe mydff3  (q[ 3], clk, d[ 3], enable[3]);
128     dffe mydff4  (q[ 4], clk, d[ 4], enable[4]);
129     dffe mydff5  (q[ 5], clk, d[ 5], enable[5]);
130     dffe mydff6  (q[ 6], clk, d[ 6], enable[6]);
131     dffe mydff7  (q[ 7], clk, d[ 7], enable[7]);
132     dffe mydff8  (q[ 8], clk, d[ 8], enable[8]);
133     dffe mydff9  (q[ 9], clk, d[ 9], enable[9]);
134     dffe mydff10 (q[10], clk, d[10], enable[10]);
135     dffe mydff11 (q[11], clk, d[11], enable[11]);
136     dffe mydff12 (q[12], clk, d[12], enable[12]);
137     dffe mydff13 (q[13], clk, d[13], enable[13]);
138     dffe mydff14 (q[14], clk, d[14], enable[14]);
139     dffe mydff15 (q[15], clk, d[15], enable[15]);
140     dffe mydff16 (q[16], clk, d[16], enable[16]);
141     dffe mydff17 (q[17], clk, d[17], enable[17]);
142 endmodule
143
144
145 // Verilog idiom for D-type flip-flop with ENABLE feature
146 module dffe (output q, input clk, input d, input enable);
147     reg q_reg;
148     always @ (posedge clk) if (enable) q_reg <= d;
149     assign q = q_reg;
150 endmodule
```

You can find a `.zip` archive containing a Xilinx ISE project file for this program at the URL http://positron.hep.upenn.edu/wja/p364/2014/files/lab25.zip . You can find just

the Verilog at http://positron.hep.upenn.edu/wja/p364/2014/files/lab25_part1.v . **Compile the above program**, load it into your board, and verify that LED7 blinks with a period of about 1.34 seconds (i.e. check visually that it is a little bit slower than 1 Hz).

(b) Now let's try a slightly different way of making an 8-bit counter. First, we'll define a module called `dffe_8bit`, which will make an 8-bit version of a D-type flip-flop. Add the following module to the bottom of your Verilog file, by typing in this dozen or so lines of code. You can omit the comments, to save a few keystrokes.

```
// Implement an 8-bit D-type flip-flop, with 'enable' and 'reset' inputs
module dffe_8bit (output [7:0] q, input clk, input [7:0] d,
                  input enable, input reset);
    // if 'reset' is asserted, then the data input to the eight
    // flip-flops will be all zeros; otherwise, the data input
    // will just be d[7:0]
    wire [7:0] data = (reset ? 0 : d[7:0]);
    dffe mydff0 (q[0], clk, data[0], enable);
    dffe mydff1 (q[1], clk, data[1], enable);
    dffe mydff2 (q[2], clk, data[2], enable);
    dffe mydff3 (q[3], clk, data[3], enable);
    dffe mydff4 (q[4], clk, data[4], enable);
    dffe mydff5 (q[5], clk, data[5], enable);
    dffe mydff6 (q[6], clk, data[6], enable);
    dffe mydff7 (q[7], clk, data[7], enable);
endmodule
```

Next, replace the last few dozen lines of the main `lab25_part1` module (starting from just below the `wire clock = clk191Hz;` line, and ending with the first `endmodule` line) with the following code. Again, you can omit the comments if you wish.

```
    // The rest of this module WAS the same as it was at the end of
    // Lab 24, BUT WE ARE REPLACING IT NOW

    // Use 8-bit-wide D-type flip-flop to make an 8-bit counter
    // The 'enable' is controlled by a sliding switch
    wire slowcount_enable = !sw[0];

    // A push-button can 'reset' the counter back to zero
    wire slowcount_reset = btn[0];

    // These wires hold the current counter value
    wire [7:0] slowcount;

    // These wires hold the next value to be loaded into the counter
    wire [7:0] slowcount_nextvalue = slowcount + 1;

    // Instantiate the 8-bit-wide flip-flop
    dffe_8bit mydffe8bit1 (slowcount[7:0],       // Q outputs
```

```
                            clk191Hz,             // clock input
                            slowcount_nextvalue,  // D inputs
                            slowcount_enable,     // 'enable' input
                            slowcount_reset);     // 'reset' input

    // Display the 8 flip flops' Q values on the 8 green LEDs.
    assign led[7:0] = slowcount[7:0];

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

By defining an 8-bit-wide flip-flop, we can make a counter with much less hassle than with eight individual DFFE's. On each new clock cycle, the `D[7:0]` inputs for the flip-flop are just given by the new value (`Q[7:0] + 1`). We can do this because, as we saw in the reading, Verilog knows how to express the addition of two integers in terms of the corresponding logic gates.

Recompile this new program and load it into your board. Check that it still counts, with the slowest LED blinking a bit slower than 1 Hz.

Also, check that when you slide one of the switches (which one?) to a certain position (what position?), the counter stops counting. It resumes counting when you slide the switch back.

Also, check that when you press one of the buttons (which one?), the counter is "reset" back to zero. Look inside the `dffe_8bit` module to make sure you understand how we got this reset feature to work (basically by using a multiplexer).

Can you see how all of this works inside the Verilog code?

Can you envision what the internal schematic diagram for all of this would look like, if built up out of individual logic gates? (Don't try too hard at this. Just try to imagine vaguely how the Verilog turns into logic gates.)

Part 2

Modify your program from Part 1 so that LED7 blinks at 1.00 Hz, i.e. get the frequency to within about a percent of exactly 1 Hz. Here's a hint: modify the rule for the next value that the slow counter will take. If the current value of the counter is 190, you want the next value to be 0. Otherwise, you want the next value to be the current value plus one, as it is now. You can achieve this by "multiplexing," using the "ternary operator" from the reading, i.e. the expression `(condition ? valueA : valueB)` .

Compile your program, load it into the board, and check that you get a frequency of 1 Hz from LED7. Briefly summarize your modifications here:

Now define a new wire called `clk1Hz` and connect it to bit 7 of your "slow" counter, like this:

```
wire clk1Hz = slowcount[7];
```

Now let's use this `clk1Hz` signal as the clock for another 8-bit counter, so that it will count up at a frequency of 1 Hz: one mississippi, two mississippi, three mississippi, etc.

The code should look something like the following, which would go just below the code that instantiates `mydffe8bit1`. Remember not to assign the `led[7:0]` outputs twice. The last line replaces the existing LED assignment.

```
1   // Make a new clock that ticks at just 1 Hz
2   wire clk1Hz = slowcount[7];
3   // Make a new counter that counts at 1 Hz
4   wire [7:0] count1Hz;
5   wire [7:0] count1Hz_next = count1Hz + 1;
6   dffe_8bit mydffe8bit2 (count1Hz[7:0],       // Q outputs
7                          clk1Hz,              // clock input
8                          count1Hz_next[7:0],  // D inputs
9                          1,                   // 'enable' input
10                         btn[1]);             // 'reset' input
11  // Display the VERY slow counter on the 8 green LEDs.
12  assign led[7:0] = count1Hz[7:0];
```

Assuming that you did as I did and connected `btn[1]` to the reset line for `count1Hz`'s `dffe_8bit`, while leaving `btn[0]` connected to the reset line for `slowcount`, what do these two reset buttons do?

Can you see the effect of pressing `btn[0]` about three-quarters of a second after the LEDs have changed value? Why does this happen? What if you hold down `btn[0]`?

Do you notice something odd about using `btn[1]` to reset the count displayed by the LEDs? How long do you have to old it down before it takes effect? Why does it generally not work if you only press it for a small fraction of a second?

Use an SR latch to make `btn[1]` reliably reset `count1Hz`, even if you only press the button for an instant. The latch should be set by the button's being pressed and reset by `(count1Hz==0)`. The output of this SR latch should then be the reset signal for `count1Hz`'s `dffe_8bit`. Describe your solution below. Here's how we coded an SR latch last week:

```
1 module sr_latch (output q, input s, input r);
2    wire qbar;
3    nor mynor1 (q, r, qbar);
4    nor mynor2 (qbar, s, q);
5 endmodule
```

Part 3

Now let's display the lowest four bits of `count1Hz` on the 7-segment LED display. The connections for the 7-segment display look something like this image from the BASYS2 reference manual:

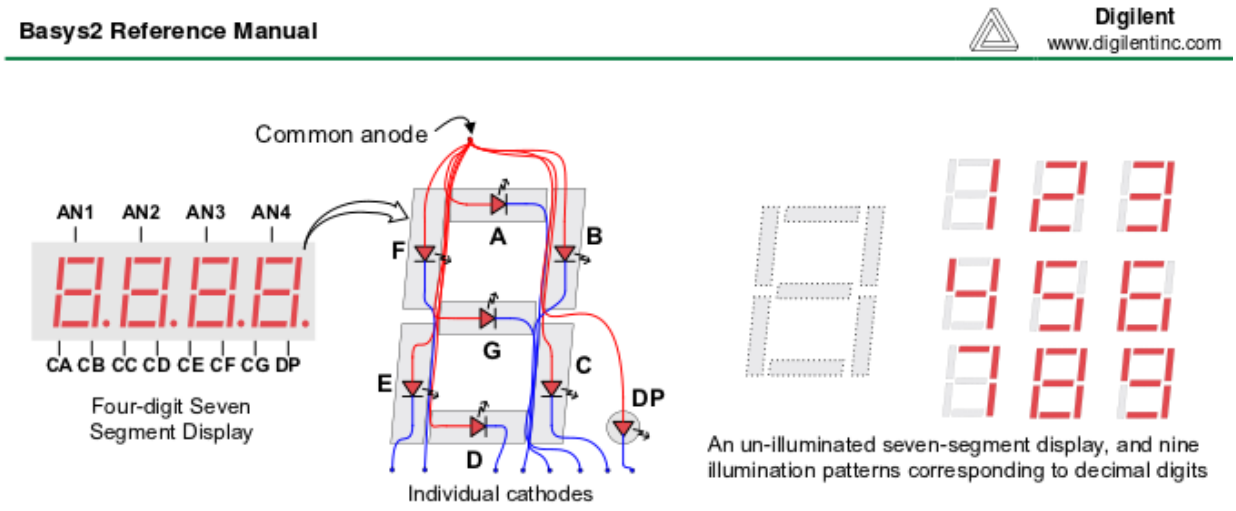
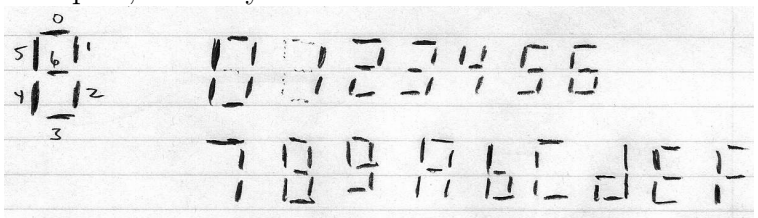


Figure 7. Seven-segment display

The top segment ("A" on the diagram) is driven by `led[0]`. The upper-right segment ("B") is driven by `led[1]`. And so on for C, D, E, F, G. Note that `assign led[0] = 1;` turns OFF the top segment, and `assign led[3] = 0;` turns ON the bottom segment, etc. This is because the FPGA outputs are connected to the lower-voltage sides (cathodes) of the diodes, not the higher-voltage sides (anodes). The higher-voltage sides (anodes) are conected to the `an[3:0]` FPGA pins, in a way that we will see below.



Modify the assignment for `an[3:0]` to this (I'll explain below):

```
1 assign an[3] = slowcount[6:5]!=3;
2 assign an[2] = slowcount[6:5]!=2;
3 assign an[1] = slowcount[6:5]!=1;
4 assign an[0] = slowcount[6:5]!=0;
```

Also modify the bottom dozen or so lines of the `lab25_part1` module (starting just after the line that reads `assign led[7:0] = count1Hz[7:0];`) so that they look like the following:

```
    // Digit will display low 4 bits of the very slow counter
    wire [3:0] digit = count1Hz[3:0];

    // Create wires that are '1' when corresponding segment is ON
    wire [6:0] seg_is_on;
    assign seg_is_on[0] = (digit==0)   || (digit==2)   || (digit==3)   ||
                          (digit==5)   || (digit==6)   || (digit==7)   ||
                          (digit==8)   || (digit==9)   || (digit=='hA) ||
                          (digit=='hC) || (digit=='hE) || (digit=='hF);
    // You need to fill in the answers for 1, 2, 3, 4, 5, 6
    assign seg_is_on[6:1] = 0;

    // Each LED actually lights up when 'seg' is LOW not HIGH
    assign seg[6:0] = ~seg_is_on[6:0];

    // We're not using the decimal point now, so just light it up
    assign dp = 0;

    // This is the mysterious anode wiring -- explained later
    assign an[3] = slowcount[6:5]!=3;
    assign an[2] = slowcount[6:5]!=2;
    assign an[1] = slowcount[6:5]!=1;
    assign an[0] = slowcount[6:5]!=0;
endmodule
```

I wrote the logic for you for `seg_on[0]`. You need to write your own logic for `seg_on[1]` ... `seg_on[6]`. When you think you have it, compile the program and load it into your board. **Briefly summarize here** the lines you added to the program.

Part 4

The next modification we'll make is to take four digits' worth of input bits (i.e. 16 bits) and to use them to drive all four digits of your 7-segment LED displays. Since time may be running short, I've provided a complete Verilog file for this purpose at
http://positron.hep.upenn.edu/wja/p364/2014/files/lab25_part4.v .

```
//
// lab25_part4.v
// Verilog source code for Lab 25 (part 4)
// PHYS 364, UPenn, fall 2014
//

`default_nettype none

module lab25_part4
  (
   input        mclk,  // 50 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
   );

    // This counter will count 0..262144 over and over again,
    // using the 50 MHz on-board oscillator as its clock input
    wire [17:0] count18bit;
    counter_18bit mycounter1 (count18bit, mclk);

    // Now we can use bit 17 of the clock to get a frequency
    // (50 MHz) / (262144) = 190.7 Hz
    wire clk191Hz = count18bit[17];

    // Now instead of using the 'Q' output of the SR latch as a clock
    // for the 8-bit counter's flip-flops, we use our new 191 Hz clock.
    wire clock = clk191Hz;

    // Use 8-bit-wide D-type flip-flop to make an 8-bit counter
    //
    // The 'enable' is controlled by a sliding switch
    wire slowcount_enable = !sw[0];
    // A push-button can 'reset' the counter back to zero
    wire slowcount_reset = btn[0];
    // These wires hold the current counter value
    wire [7:0] slowcount;
    // These wires hold the next value to be loaded into the counter
```

```
46      wire [7:0] slowcount_nextvalue = (slowcount==191 ? 0 : slowcount+1);
47      // Instantiate the 8-bit-wide flip-flop
48      dffe_8bit mydffe8bit1 (slowcount[7:0],       // Q outputs
49                             clk191Hz,             // clock input
50                             slowcount_nextvalue,  // D inputs
51                             slowcount_enable,     // 'enable' input
52                             slowcount_reset);     // 'reset' input
53
54      // Make a new clock that ticks at just 1 Hz
55      wire clk1Hz = slowcount[7];
56      // Make a new counter that counts at 1 Hz
57      wire [7:0] count1Hz;
58      wire [7:0] count1Hz_next = count1Hz + 1;
59      dffe_8bit mydffe8bit2 (count1Hz[7:0],        // Q outputs
60                             clk1Hz,               // clock input
61                             count1Hz_next[7:0],   // D inputs
62                             1,                    // 'enable' input
63                             btn[1]);              // 'reset' input
64      // Display the VERY slow counter on the 8 green LEDs.
65      assign led[7:0] = count1Hz[7:0];
66
67      // Instantiate 'display4digits' module to drive 7-segment display
68      wire [3:0] digit0, digit1, digit2, digit3, dots;  // each is a 4-bit vector
69      wire [1:0] count0123;
70      display4digits myd4d1 (seg[6:0], dp,   // module outputs to drive the FPGA
71                             an[3:0],        //   pins for 7-segment display
72                             count0123[1:0], // 2-bit counter to ping-pong digits
73                             digit0[3:0],    // value to display on right digit
74                             digit1[3:0],    //   each digit can display a
75                             digit2[3:0],    //   4-bit number in hexadecimal
76                             digit3[3:0],    // value to display on left digit
77                             dots[3:0]);     // what to send to the 4 'dp' dots
78      // Connect useful values to the inputs of 'myd4d1'
79      assign count0123[1:0] = slowcount[5:4];
80      assign digit0[3:0] = 1;
81      assign digit1[3:0] = 2;
82      assign digit2[3:0] = 3;
83      assign digit3[3:0] = 4;
84      assign dots[3:0] = btn[3:0];
85  endmodule
86
87
88  // This module drives the FPGA outputs ('seg', 'dp', 'an') that connect
89  // to the 4-digit 7-segment LED display.  The display actually contains
90  // 32 separate LEDs (7 segments plus decimal, for each of 4 digits), but
91  // uses just 12 FPGA pins to drive them.  The trick is to illuminate only
92  // one of the four digits at a time, but to alternate between the four
93  // digits so quickly that the human eye doesn't notice.  The wire 'an[i]'
94  // connects to the anodes (positive sides) of all 8 LEDs on segment #i,
95  // while the wire 'seg[j]' connects to the cathodes (negative sides) of
96  // segment #j for all four digits.  The 'dp' wire connects to the cathodes
97  // of all four decimal points.
98  module display4digits(output [6:0] seg,          // 7-segment LED cathodes
99                        output       decimalpoint, // decimal-point cathodes
```

```
100                       output [3:0] anode,        // one anode per LED digit
101                       input  [1:0] count0123,    // which digit to light up now
102                       input  [3:0] digit0,       // bits for right-hand digit
103                       input  [3:0] digit1,       //   each digit value goes
104                       input  [3:0] digit2,       //   from 0 to F (hexadecimal)
105                       input  [3:0] digit3,       // bits for left-hand digit
106                       input  [3:0] dots);        // inputs for 4 DP dots
107    assign decimalpoint = (count0123==0) ? ~dots[0] :
108                          (count0123==1) ? ~dots[1] :
109                          (count0123==2) ? ~dots[2] : ~dots[3];
110    // Each digit's anode is driven HIGH (by an external PNP transistor)
111    // whenever the corresponding an[] bit is driven LOW by the FPGA; by
112    // illuminating one digit at a time in sequence, we can make all four
113    // digits appear to be always lit, but appearing to be displaying
114    // distinct digit values simultaneously
115    assign anode[0] = ~(count0123==0);
116    assign anode[1] = ~(count0123==1);
117    assign anode[2] = ~(count0123==2);
118    assign anode[3] = ~(count0123==3);
119    // We need to go ping-pong (4-way ping-pong) between the four input
120    // digits, to decide which digit we are currently converting into
121    // the corresponding 7 segments
122    wire [3:0] digit = (count0123==0) ? digit0 :
123                       (count0123==1) ? digit1 :
124                       (count0123==2) ? digit2 : digit3 ;
125    // Create wires that are '1' when corresponding segment is ON:
126    //   seg 0=north, 1=NE, 2=SE, 3=south, 4=SW, 5=NW, 6=center
127    wire [6:0] seg_is_on =
128               // I thought you might enjoy this different way of decoding
129               // the numbers 0..F into the 7-segment ON/OFF pattern.  But
130               // there is a bug here in the display of the 'F' digit: can
131               // you find a way to fix it?
132               (digit==0)   ? 'b0111111 :
133               (digit==1)   ? 'b0000110 :
134               (digit==2)   ? 'b1011011 :
135               (digit==3)   ? 'b1001111 :
136               (digit==4)   ? 'b1100110 :
137               (digit==5)   ? 'b1101101 :
138               (digit==6)   ? 'b1111101 :
139               (digit==7)   ? 'b0000111 :
140               (digit==8)   ? 'b1111111 :
141               (digit==9)   ? 'b1101111 :
142               (digit=='hA) ? 'b1110111 :
143               (digit=='hb) ? 'b1111100 :
144               (digit=='hC) ? 'b0111001 :
145               (digit=='hd) ? 'b1011110 :
146               (digit=='hE) ? 'b1111001 :
147                              'b1000111 ;
148    // Each digit's corresponding LED segment lights up when its
149    // 'seg' bit is LOW, which drives that LED's cathode to ground
150    assign seg[6:0] = ~seg_is_on[6:0];
151 endmodule
152
153
```

```
154 // 18-bit counter: internally uses 18-bit-wide D-type flip-flop
155 module counter_18bit (output [17:0] q, input clk);
156     wire [17:0] d = q + 1;
157     dffe_18bit myff (.q(q), .clk(clk), .d(d), .enable(1), .reset(0));
158 endmodule
159
160
161 // 8-bit D-type flip-flop, with 'enable' and 'reset' inputs
162 module dffe_8bit (output [7:0] q, input clk, input [7:0] d,
163                   input enable, input reset);
164     // if 'reset' is asserted, then the data input to the eight
165     // flip-flops will be all zeros; otherwise, the data input
166     // will just be d[7:0]
167     wire [7:0] data = (reset ? 0 : d);
168     reg  [7:0] q_reg;
169     always @ (posedge clk) if (enable) q_reg <= data;
170     assign q = q_reg;
171 endmodule
172
173
174 // 18-bit D-type flip-flop, with 'enable' and 'reset' inputs
175 module dffe_18bit (output [17:0] q, input clk, input [17:0] d,
176                    input enable, input reset);
177     wire [17:0] data = (reset ? 0 : d);
178     reg  [17:0] q_reg;
179     always @ (posedge clk) if (enable) q_reg <= data;
180     assign q = q_reg;
181 endmodule
```

Compile the above Verilog program and load it into your board. You should see `4321` displayed on the four digits of the 7-segment display.

Carefully read through the comments in the new `display4digits` module and look again at the illustrations (on this page and on the next page) for how the 7-segment display is connected. See if you can understand how the FPGA manages to drive all four digits. **Explain here very briefly how it works:**

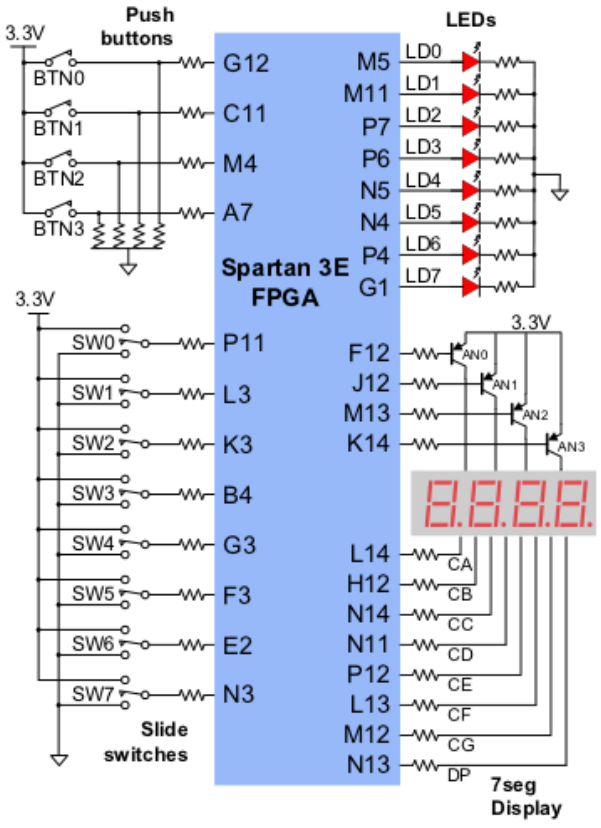


Figure 6. Basys2 I/O circuits

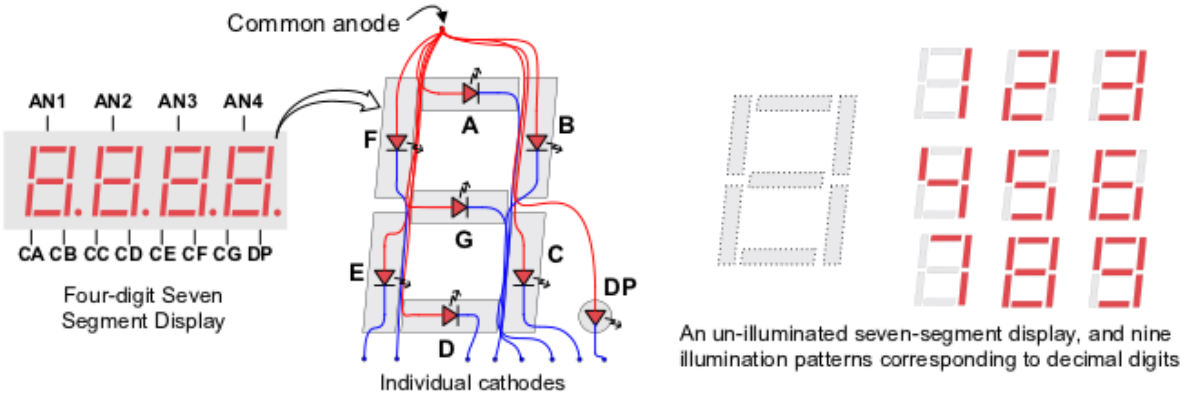
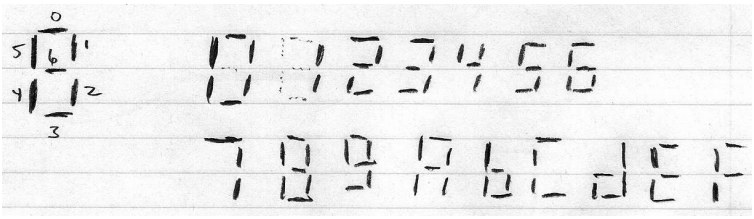


Figure 7. Seven-segment display



Now make one small change to the assignment of `count0123` in the top-level module (where it says "Connect useful values to the inputs of `myd4d1`") so that the FPGA alternates between the four different digits so quickly that your eye can't even detect the motion: make it update so quickly that it looks as if all four digits are displayed simultaneously. Hint: making it update about sixteen times as fast is a very easy way to do this. Recompile and see if "`4321`" now seems to be constantly illuminated.

Briefly describe what you did:

Now connect the `digit0...digit3` inputs of the `display4digits` module so that the right-hand pair of digits displays the 8-bit counter that is ticking at 1 Hz (the same counter that drives the green LEDs), and so that the left-hand pair of digits displays the 8-bit value set by the eight sliding switches.

When you load this into your board, you should see the right two digits ticking at 1 Hz. You should be able to manipulate the sliding switches to change the left two digits. But notice that `sw[0]` is still also connected to the `slowcount_enable` wire, so sliding that switch up will stop the clock, which stops all of the action.

Notice that there is a bug in my logic for displaying the `F` digit to the 7-segment display. As I have it currently coded, the `F` digit displays backwards. **Find and fix my bug**, and make the letter F display correctly. **Describe your fix.**

Now that we have all four digits working correctly, we will make use of this display in the upcoming labs.

Extra challenge!

If you have time, see if you can modify this circuit so that it displays, on the 16-bit 4-digit hexadecimal display, the sequence of Fibonacci numbers, from 1 (hexadecimal `0001`) through 46368 (hexadecimal `B520`). To make this work, you will probably need a pair of 16-bit-wide D-type flip-flops. Don't worry about having your output make sense once the numbers get too big to fit into 16 bits. Instead, use one of the buttons to provide a `reset` input for your circuit. Display each new Fibonacci number for one second.

Here is *Mathematica*'s calculation of the expected output of your program, shown both in decimal and in hexadecimal.

```
Map[Function[i, {Fibonacci[i], BaseForm[Fibonacci[i], 16]}],
    Range[24]]
```

$$\begin{pmatrix} 1 & 1_{16} \\ 1 & 1_{16} \\ 2 & 2_{16} \\ 3 & 3_{16} \\ 5 & 5_{16} \\ 8 & 8_{16} \\ 13 & \mathrm{d}_{16} \\ 21 & 15_{16} \\ 34 & 22_{16} \\ 55 & 37_{16} \\ 89 & 59_{16} \\ 144 & 90_{16} \\ 233 & \mathrm{e9}_{16} \\ 377 & 179_{16} \\ 610 & 262_{16} \\ 987 & \mathrm{3db}_{16} \\ 1597 & \mathrm{63d}_{16} \\ 2584 & \mathrm{a18}_{16} \\ 4181 & 1055_{16} \\ 6765 & \mathrm{1a6d}_{16} \\ 10946 & \mathrm{2ac2}_{16} \\ 17711 & \mathrm{452f}_{16} \\ 28657 & \mathrm{6ff1}_{16} \\ 46368 & \mathrm{b520}_{16} \end{pmatrix}$$

If you make this work, email it to `ashmansk@hep.upenn.edu` for extra credit. Be sure to CC your lab partner if you worked together.