EE108a Lab 3: Bike light

Version 2.0 – David Black-Schaffer
Version 2.2 – David Black-Schaffer

Introduction

In lab 3 you are going to get your first taste of sequential logic by building a system of finite state machines, timers, and shifters to create a programmable bike light. To make this lab easy to understand, implement, and test, we've divided up the project into small chunks, but it's up to you to implement and hook up the chunks.

Overview

The bike light project consists of a flashing LED that goes through 6 "master" states when the right button is pressed on the FPGA board. These are:

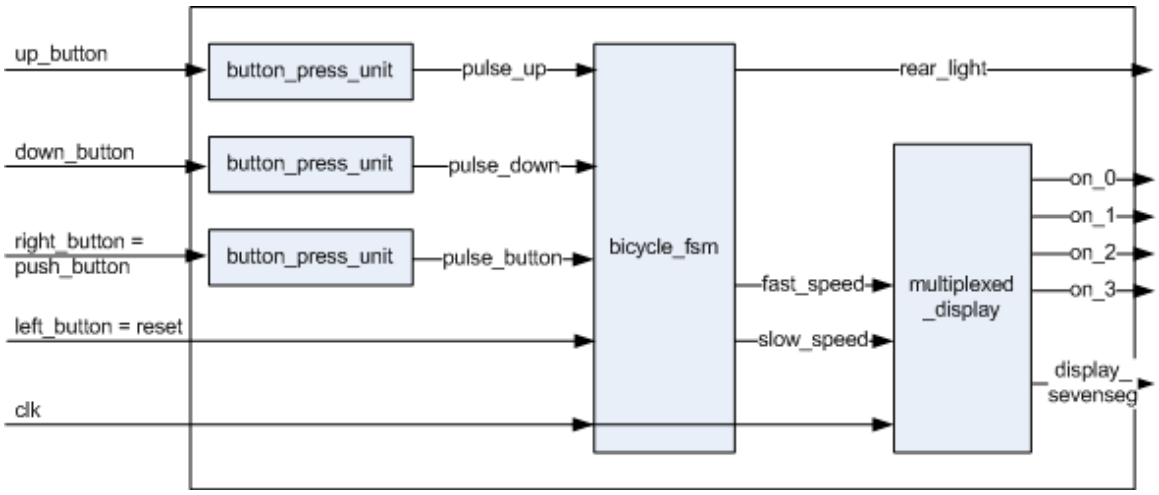
1. Off (resets to here)
2. On
3. Off
4. "Fast" flash
5. Off
6. "Slow" flash

After the 6th state it returns to the 1st state. When the bike light is in the 4th and 5th states the LED flashes at a programmable rate, and the user can press the up and down buttons to change this rate. In addition to the flashing LED output, your bike light will display numbers on the 7-segment displays to let the user know how fast or how slow the "fast" and "slow" states are.

When the module is in the "fast" or "slow" states, pushing the up or down buttons on the FPGA board will change the blinking rate by a factor of two. Both states start blinking once per second (on 1s, off 1s). When the up button is pressed in the "fast" state it will switch to blinking twice per second (on 0.5s, off 0.5s) and so forth. Both "fast" and "slow" modes will have 4 possible speeds, so the total speed will range from 1/8th of a second in the "fast" state to 8 seconds in the "slow" state.

Top-level module

The top-level module has been provided for you for this lab. It takes in the up_button, the down_button, the right_button, and the left_button (used for reset), and the clock, and outputs the rear_light signal (for your LED) and 4 enable signals for the 4 seven-segment displays, and 7 segment signals for each of the segments of the displays.

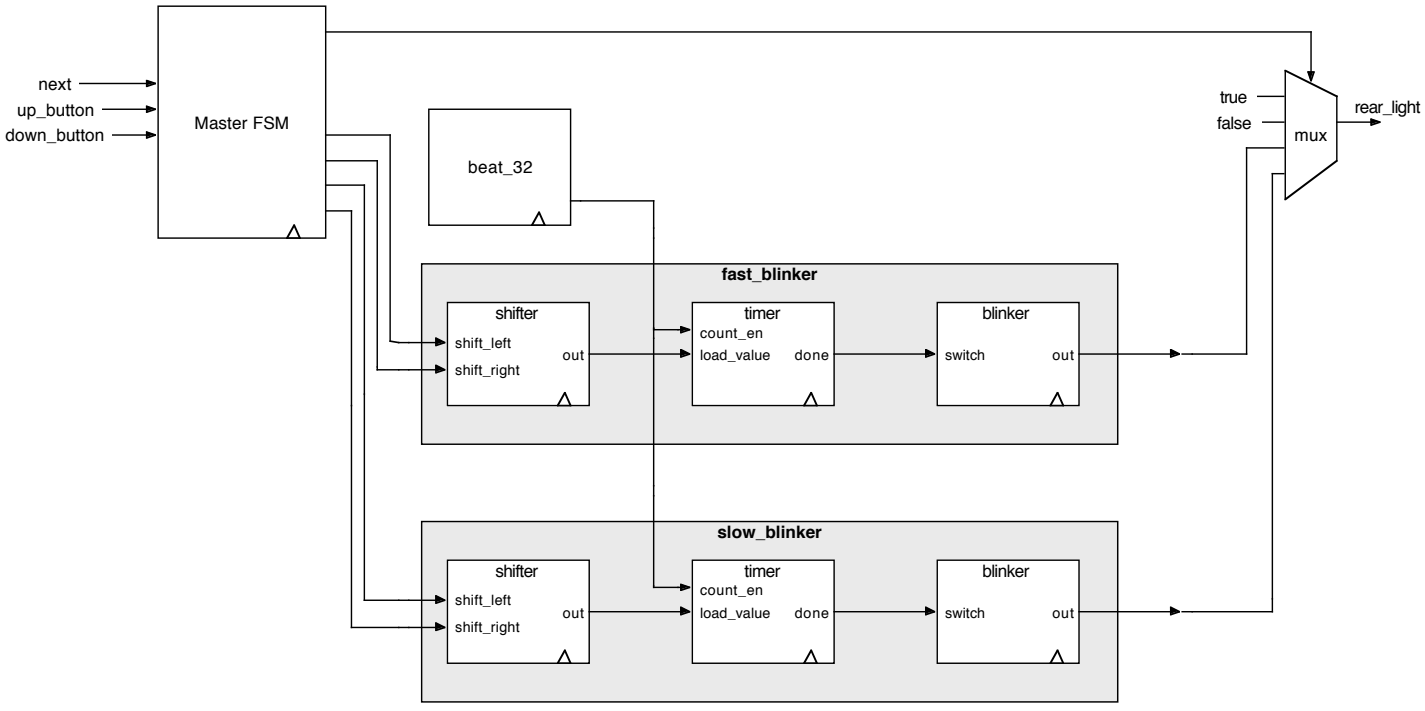


Processing Inputs: The button_press_unit

The button inputs are fed through the provided button_press_unit which does three critical things: synchronizing, debouncing, and one-pulsing. The synchronizer is responsible for making sure that the button press from the user lines up with the 100MHz clock on the FPGA. Since everything else is lined up with the clock edge it would be really bad to have an input that wasn't! (In fact it would be impossible to determine the timing constraints if we didn't know when the signal came in.) The debounce unit takes care of the fact that the switches on the board are actually small mechanical springs which will bounce up and down many times with every push. This module works by looking for the first transition, and then ignoring any other transitions for about 2ms to allow the button to settle down. Note that 2ms at 100MHz is an awful lot of clock cycles, so you probably don't want to simulate the top-level module that includes the button_press_unit or you'll be waiting for a long time. The final thing the button_press_unit does is to one-pulse the input. That is, whenever the (now synchronized and debounced) input goes high, it generates an output that is high for exactly one cycle. This is important because if your FSM is checking for button presses at 100MHz your user would have to hold the button down for 10 nanoseconds or less to make sure the FSM only saw one button press. Pretty unlikely. By one-pulsing the input we guarantee that we get exactly one cycle's worth of input for each button press.

The Bicycle FSM

The bicycle_fsm module shown in the top-level diagram is responsible for implementing all of the functionality outlined above. However, to simplify things we're going to break it apart into several smaller, easy-to-debug modules. Your job is to understand each of them, how they come together to make the whole thing work, and then to build and debug them.



The diagram above includes all the key signals you'll need in your design, but it omits resets and clocks. Don't forget them or you'll have a hard time simulating your design.

Master FSM Module

As you can see in the above diagram, the output from the bicycle_fsm comes from a mux which selects between 4 inputs depending on the state of the Master FSM. It is either off, on, or whatever is coming out of the fast_blinker or the slow_blinker. The Master FSM is also responsible for sending signals to the fast_blinker and

slow_blinker modules to adjust their speed as appropriate. (Remember you can only change them when you are in the right state!)

Beat_32 Module

Our FPGA runs on a 100MHz clock. This means that if you want to blink an LED once a second you need to wait for 100 million clock cycles between each blink. Since it's a pain to deal with counting so high all the time you are going to implement a beat_32 module which generates a one cycle enable pulse every 1/32 of a second. This way we only need to have one big counter. To build this module you simply need a counter that counts up to 100,000,000/32 = 1/32nd of a second, and then resets to zero and starts over again. Each time it resets to zero it should output true for one cycle.

You should think carefully about what this means for your testbenches. The beat_32 module will take 100million/32 cycles before it does anything. You will want to change this when you are simulating so it counts up to a much lower number so your simulations don't take forever! (But remember to put it back for your final version.)

Blinker Modules

The fast_blinker and slow_blinker modules, shown above, are themselves made up of three modules. Their job is to take in a signal that tells them to go faster or slower and then generate the appropriate blinking output. To do this they use the beat signal from the beat_32 so they don't have to count up so high.

Inside the Blinker Modules

From the above diagram you can see that the blinker modules are both very similar. (Indeed they only differ by a few characters total, so you should build and debug one of them completely before you copy it to make the second one.) Each one consists of a shifter which outputs a value to a timer which then uses that value to count down. When the timer expires it tells the blinker module to "blink", that is, if it is on, switch to off, and vice versa. The timer then re-loads the value from the shifter and starts over. This way if the shifter's value changes the timer will count by a different amount on the next blink. Make sure you understand this flow before you dig into the details below.

Shifter

The shifter is responsible for keeping track of how long the blinker module should blink. It does this by storing a 4-bit one-hot value, and shifting it right or left each time the shift_right or shift_left input goes high. Once the value gets down to 4'b0001 or up to 4'b1000 it should remain there even if shift_right or shift_left is asserted again, respectively.

This works very nicely for our design because when the shifter gets a "shift_left" signal it will shift its output one to the left (multiply by two) so the timer will now get an input that is twice as big. Similarly when the shifter shifts to the right it will divide its output by two and the timer will have half as big a value.

Timer

The timer module is very similar to the ones you've seen in class. It simply counts down by one and when it reaches zero it emits a one-cycle pulse on the output, and re-loads itself from the load_value input. However, to make sure our counter doesn't count at 100MHz you will add a count_en (count enable) input which prevents the counter from counting except when the enable is high. By doing so we can hook up the beat_32 output to the counter and end up with a counter that counts down one count every 32nd of a second instead of 100 million

times a second. (Remember that this is an enable and not the clock. You should never put anything other than the one system clock into the clock input!)

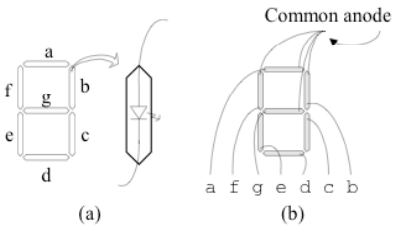
You are responsible for sizing the timer to be able to capture the full range of speeds from 1/8th of a second to 8 seconds. You should reuse the same timer module for both the fast and slow blinkers, so make sure it is big enough for the whole range. Work out how high the counter needs to be able to count before you start building it. Make sure your counter is counting down by 1 and not by the load_value. If you count down by the load_value you'll have problems with wrapping around.

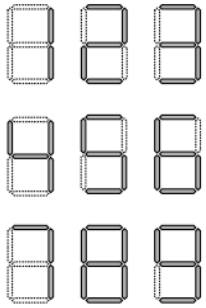
Blinker

The blinker is almost too simple to bother describing. It simply switches from "on" to "off" each time it gets an input. However, this module does have state, so you'd better make sure you're instantiating a flip flop, and you are required to draw out the state diagram for it.

Generating Outputs: the multiplexed_display

In addition to flashing the LED on the FPGA board you will display the speed for the fast and slow modes on the 7-segment displays. The left two will display the fast speed and the right two the slow speed. We've provided you with a truth table to decode binary numbers (figure d below, in the hexadecimal_display module) and the multiplexer for two of the displays.

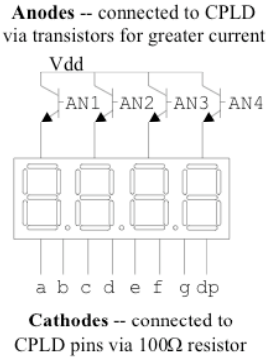


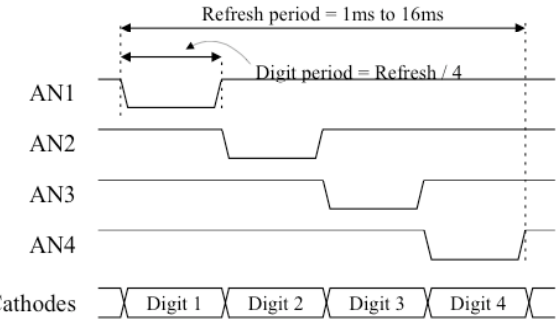


Digit Shown	Illuminated Segment a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

(d)

Figure 4. (a) Seven segment display detail. (b) common anode display configuration. (c) segement illumination patterns for decimal digits. (d) segment illumination truth table.





The 7-segment displays on the FPGA board each have 8 inputs: 1 anode and 7 cathodes. When you pull the anode low and one of the cathodes high it lights up that segment. (I.e., anode 0 and cathode g would light up the middle segment to make a "-" on the leftmost display.) The tricky part is that all 4 of the displays share the same 7 cathodes. If you want them to all display the same thing this is not a problem, but if you want to display different things on each one you have to run the display in a multiplexed mode. That is, you turn on each anode sequentially, and while it is on you output its cathode value to all the displays. (See the timing diagram above from the Digilent DI/O 4 board manual.) If you do this fast enough the human eye can't see that you're turning

them on one-after-another, and it appears that they are all on. For this lab we've provided you with a module which will drive 2 of the 4 displays and you need to modify it to drive all 4 and output the fast and slow counts to each side. Remember to look in the UCF file (that's the file that controls where the signals in your top-level module go on the FPGA – the format should be quite easy to figure out from the provided file and the one from lab 0) to make sure that you uncomment the two remaining displays so the tools will connect it correctly.

Note that the fast and slow modes each only have 4 bits of speed and that we're asking you to design a 7-segment multiplexer for 4 displays, each of which is capable of displaying 4 bits (hex 0-F) of data. (That's 16-bits of display data when you only have 8-bits to display.) This means two of them will always display 0, but you will need this module for lab 4. Your output should display the result of the shifter – i.e., it should start out displaying "0101" and then when fast goes up it will display "0201" then "0401" then "0801", similarly for slow.

Inferred state

We've warned you several times about not inferring latches in ee108a but this is the first time it will actually be a problem, so here's an overview of what to do and what to not do. Remember, **whenever you need state storage (counters, FSMs, etc.) you must explicitly instantiate a flip flop from the provided ff_lib.v file**. This file explicitly infers state to generate a flip flop. To understand what you need to do to avoid inferring state, let's take a look inside the flip flop library:

The EE108a Flip Flop Module:

```
module dff (d, clk, q);
      parameter WIDTH = 1;
      input clk;
      input [WIDTH-1:0] d;
      output [WIDTH-1:0] q;
      reg [WIDTH-1:0] q;

      always @ (posedge clk)
            q <= d;
endmodule
```

You should note two things here that you haven't seen before in Verilog. The first is the @ (posedge clk) and the second is the <= operator. (You are not allowed to use either of these in ee108a so that's why you haven't seen them. The reason we don't teach them is that it is important that you learn to distinguish the combinational and sequential portions of your design so you understand what the tools are building from your code. If you write Verilog using these constructs it is a lot harder to understand what you're building.)

So what is that always block doing? Well, because it is not using always @* it will only evaluate when @ (posedge clk) is true. This means that on every rising edge of the clock (and only on the rising edges of the clock) this block will be evaluated. The q <= d statement says that q will get the last value of d, and will only change to that value when everything else is done being evaluated. (That way the output of the flip flop will only change once even if the input changes multiple times during simulation.)

Now the real question is what is the value of q when we're not on the rising edge of the clock? What happens is that Verilog infers a latch for q, and remembers its value whenever the clock is not a rising edge. This is exactly the behavior we want for a flip flop, but we don't want this behavior anywhere else in our designs.

The ff_lib.v module also has two other flipflops you can use: dffr and dffre. The "r" here stands for reset, with each of these flipflops having a reset input which will set them to zero when asserted. If you want your logic to reset to zero then you don't have to do anything other than use a dffr and put the reset signal into all your flipflops. (If you need it to reset to something else, say 1000, you will need to explicitly build logic to set the flipflop's input when reset is asserted.) The dffre also has an enable signal. When enable it true the flipflop will latch in the input value on the rising edge of the clock. When enable is false it won't.

Mistakes that infer state

Now that we've seen how to build a flip flop on purpose, let's see how we can build them by accident.

Verilog will only infer state in your design if you don't build combinational logic in always blocks. That is, **as long as every single output is defined for every single combination of inputs you will never infer state**. This should make sense: if all the outputs are defined for all the inputs then the block is purely combinational (the outputs only depend on the current inputs). If an output is not defined for a given set of inputs then Verilog will infer a latch to remember the previous output to use in that case. You are not allowed to do this ee108a and we will take off points if you do.

Let's look at an example:

```
reg result;
reg status;
always @* begin
      if (input_button) begin
            status = 1'b1;
            result = 1'b0;
      end
      else if (done_signal) begin
            status = 1'b0;
      end
end
```

This code does not define every output for every combination of inputs. There are two different problems here. The first is that we don't define the value of the result output for the case where done_signal is true. The second is that we don't define the value of either status or result in the case where both done_signal and input_button are false. As a result this will infer latches for both result and status.

The correct code is as follows, with changes underlined:

```
reg result;
reg status;
always @* begin
      if (input_button) begin
            status = 1'b1;
            result = 1'b0;
      end
      else if (done_signal) begin
            status = 1'b0;
            result = 1'b1;
      end
      else begin
            status = 1'b0;
            result = 1'b0;
      end
end
```

This version is purely combinational because it defines all outputs for all combinations of inputs. The same thing can happen with a case statement where you don't include a default.

The basic rules to avoid this problem are:

1. Always have an else for every if
2. Always define the same set of signals in all cases or if clauses
3. Always have a default for every case
4. Always read the warnings in Xilinx ISE about inferred latches.

You will only see warnings about this in Xilinx because ModelSim assumes that you just wanted to build a latch and so it just gives you one. When you run Xilinx you need to look at the output from the synthesis step and make sure you don't see any warnings except where you have instantiated a flip flop from the ff_lib.v module. E.g., the following warning is not allowed in this class:

```
Synthesizing Unit <lab0_top>.
    Related source file is "../lab0_top.v".
WARNING:Xst:737 - Found 4-bit latch for signal <result>.
    Found 4-bit adder for signal <added_result>.
    Summary:
      inferred   1 Adder/Subtractor(s).
Unit <lab0_top> synthesized.
```

You must make sure you have run your prelab through Xilinx ISE before you submit it and that you did not receive any warnings about inferred latches for anything other than flip flops. If we see problems like this in your prelab code (and we're pretty good at recognizing them) you will lose points.

Your Mission, should you choose to accept it…

(Ha! Just kidding. You don't really have a choice, but this is the first really cool lab, and lab 4 is even cooler. :)

Methodology

1. Draw state diagrams for the master FSM, the shifter, and the blinker.
2. Draw block diagrams for the timer and the beat_32.
3. Determine the inputs and outputs for each module. (Remember that anything that has a clock input should also have a reset input.)
4. Implement the master FSM, shifter, blinker, timer, and beat_32 modules. Use case statements for the FSMs and remember to include default cases.

5. Write test benches for each module and simulate them. (The modules are small so this will be easy.)
6. Implement the fast_blinker module and write a test bench for it.
7. Copy the fast_blinker module and change it to create the slow_blinker module. (You should use the same testbench for each.)
8. Hook everything up inside the biycle_fsm module and simulate it.
9. Create a Xilinx ISE project, import your files (except your testbench) and the UCF file and generate a .bit file. (Remember you'll need to modify the .UCF file to include the extra two anodes for the 7-segment display and the LED output. You will need to look up the pin for the LEDs either on the Xilinx web site (search for "Xilinx XUP board manual") or you can look and see what they were in lab 0. If you've forgotten how to create an ISE project please go back to lab 0 and follow those instructions.)

A Warning

This lab is significantly more involved than any of the previous labs. Don't put off starting on this until the weekend.

Prelab Questions

1. If you have a de-bounced button (that is one that only goes on when you push it and immediately turns off when you let go) as an input to a 100MHz FSM, you would have to hold down the button for about 10ns to have it only advance one state in the FSM. This is obviously rather impractical. Describe the states of a "one-pulse" FSM that outputs a 1-cycle pulse every time the button is pressed, regardless of how long it is held down.
2. Assume there is an error in a one-hot encoded FSM and a bit is flipped such that either two bits are high or zero bits are high. What will happen to the design if it was written as a Verilog case statement with no default entry? (Hint: the other states are now handled like don't cares in Kmaps.) What if there was a default entry?

Submission

Prelab

Please turn in the following for this week's prelab, following the submission template on the website. Remember that all submissions for prelabs are electronic.

1. The answers to the prelab questions above.
2. FSM state diagrams for the master FSM, the shifter, and the blinker.
3. Block diagrams of the timer and beat_32 modules.
4. The complete project with modules of all logic blocks for your design and test benches. Make sure you do not submit hundreds of temporary simulation files along with these files.
5. Simulation results (i.e. annotated wave forms) showing that your design works. You must clearly describe what the simulation results show and how that indicates it works. You need a simulation **for each of the modules you wrote**.
6. Extract timing information for the *whole* design from the timing analysis report. This tool is hidden under Implement Design -> Place & Route -> Generate Post-Place& Route Static Timing -> Analyze Post-Place & Route Static Timing (Timing Analyzer). When you run it click the button for "Analyze Against Auto-Generated Constraints" and accept the defaults. The report consists of paths sorted from slowest (largest delay) to fastest (shortest delay). Each path starts with "Delay:" and then describes the source and destination, and lists the delay of each net (or wire) in-between. At the end it reports the total delay broken up into % logic and % routing. Report the fastest speed the design will run and the critical path. (The critical path is the one that keeps the design from going faster, which is the path at the top of the report.) Does this path make any sense to you?

7. Find the resource utilization for the *whole* design. (Click on the "Design Summary" tab in the right-hand pane in Xilinx ISE.) How many Slice Flip Flops are used? How many 4 input LUTs and how many occupied Slices are there.
8. Repeat 4 and 5 above for *just the bicycle_fsm* portion of the design as specified above. (I.e., select the "bicycle_fsm" module in Xilinx ISE and re-run the Timing Analyzer using that as the top-level. Comment on the differences.
9. Look at your *whole* design in the Xilinx FPGA Editor and include a screenshot with the master FSM's state flip-flops highlighted. (Select the flip flops on the left side, and then zoom in so they and their connections are visible.)

Remember, you must submit your project whether or not it works correctly on the due date or you will receive no credit.

Prelab 3 Submission Template

Student names:
Group number:
Project name:
Notes about the design: *whether or not the design is successful, how you testing it, etc.*

Prelab questions

* Answer the two prelab questions *

Design Diagrams

* Show your FSMs and your block diagrams for the modules specified above. *

Verilog Code

* Include all Verilog modules used in this lab. Paste it into Word and make the font small enough so it fits across the page. *

Testbenches

* Include all testbenches used to verify the function of the project *

Simulation Waveforms

* Include all simulated waveforms, complete with detailed annotations and explanation indicating how the waveforms show successful function *

Timing Information

* Give the requested timing information for both the overall design and the individual bike_light module. Comment on differences *

Resource Utilization

* Give the requested resource utilization information for both the overall design and the individual bike_light module. Comment on differences *

Design Screenshot

* Include a screenshot of the whole design from the Xilinx FPGA Editor, with state-flip-flops high-lighted *