Stanford Verilog & Digital System Design Tutorial - Spring 2003
Version 1.3 - 14 April 2003

Introduction

This tutorial is meant to bring you through the process of implementing a project similar to a real EE183 lab. Please expect this tutorial to take between 4 and 6 hours to complete. As this tutorial has been designed to show you the sorts of techniques and problems you may run into in the labs themselves, the more comfortable you become with this material the easier a time you will have with the labs. However, please do not be too worried if you don't completely understand everything: you will by the end of the course…I promise you.

(Also, please excuse any rough edges here as this is the first version of this tutorial. If there is anything which just doesn't make sense please email me.)

Tutorial Project

The tutorial project goal is to design a circuit that will display a pretty color pattern (provided) on a VGA monitor and allow the user to turn on and off the red and green components with the gamepad controller. However, to prevent the user from accidentally turning them on or off, two safety measures will be implemented. The first is that the user will have to press the on button twice in a row to enable a given color while a single off button press time will disable it and require two new on presses to enable it. Secondly, there will be a global enable button that will toggle a "lock-out" of the color on/off buttons and display its status on an LED on the XSA board. When the system is "locked-out" no on/off presses will have any affect on the display.

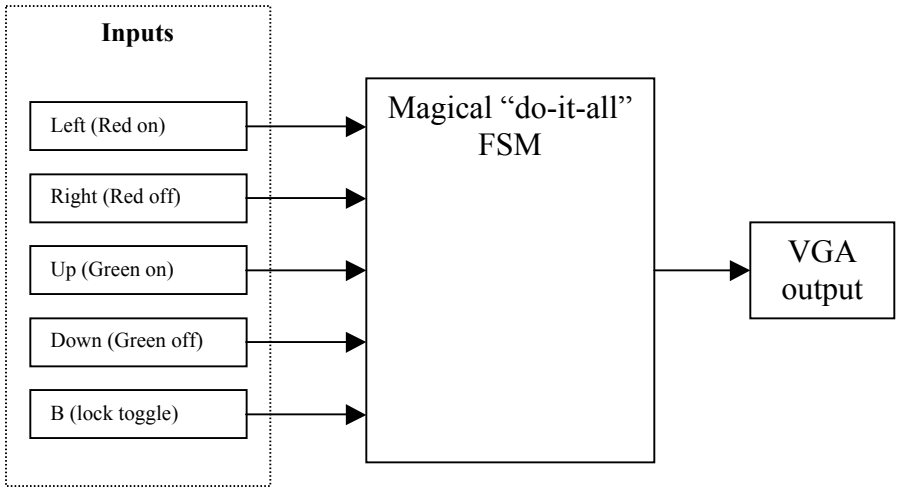
At this point you should be thinking about inputs/outputs and FSMs for dividing up this project. Let's take a look at some.

Contents

Part I: Initial Design

In doing this design we are going to focus on **hierarchical design** of intercommunicating finite state machines (FSMs) and **modular design and testing**. As the behavior of the design is already set above, we are going to now define our inputs and outputs:

Inputs:
Left button (`left_in`) controls red on.
Right button (`right_in`) controls red off.
Up button (`up_in`) controls green on.
Down button (`down_in`) controls green off.
A/B button (`ab_in`) controls lock-out toggle.
Outputs:
Lock-out status LED (`dot_led`) on seven-segment display.
VGA signals color/synch (synch signals are provided).

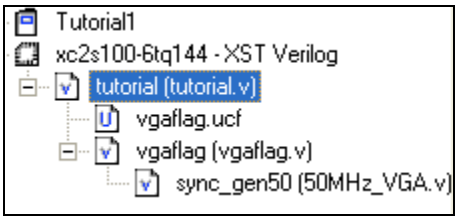


This block diagram should look wrong to you. Too many functions are being done in that one big FSM and it isn't at all clear how to get the VGA to display what we want based on the button presses unless you've already done a lot with the VGA.

So before we go further let's take a look at what's provided for completing this project, and, in particular, let's take a look at our inputs (buttons), outputs (VGA), development tools (Xilinx) and some basic verilog.

Part II: Getting Started with Verilog

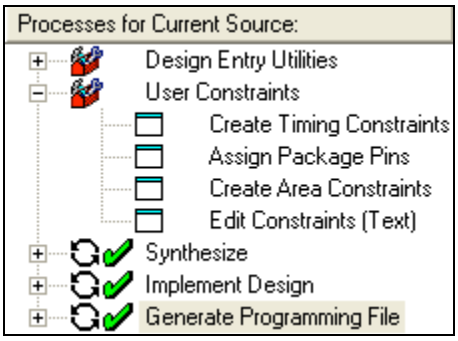
1. Log onto a lab computer or your own machine with version 5.1 of the Xilinx software installed.
2. Go to the Handouts section of the EE183 web page and download Part 1 of the tutorial to your computer. Note that you may have to download it to a directory with no spaces in the path to get this to work. (I.e., the "My Documents" directory, or any sub-directory thereof, may not work.)
3. Expand the archived project. (Double-click on it on the lab machines.)
4. Run the Xilinx Project Navigator.
5. Open the Project with File->Open Project.



You will now be presented with the Xilinx IDE window with the project loaded. You should see in the upper left corner under "Sources in Project" the type of chip being used (xc2s100-6tq144, i.e., a Spartan 2 with speed grade –6 and package tq144) the type of language (Verilog) and the project files. The hierarchy here tells you what modules use other modules. In this case the tutorial.v file is the top level and the other files are included in it. The vgaflag.ucf is the constraints file which defines the pin-outs for the I/O connections. The vgaflag.v file contains the code to draw the colored pattern and the sync_gen50 is the module which generates the VGA sync signals for you.

Let's first compile and test out this project.

1. Right-click on "Generate Programming File" in the "Processes for Current Source" window below the hierarchy display. Make sure the "tutorial (tutorial.v)" file is selected above as your top level or you'll compile the wrong part of the project.
2. Choose "Rerun All" to process all the files.
3. The tools will now Synthesize and Implement the design and then create a bit file for downloading. You should see green check marks next to each of those three steps.
4. Once it has finished, run gxsload and load the resulting tutorial.bit file to the XSA board with the VGA cable plugged into the board (not the Xtend board).
5. You should now see the colored pattern on the display.
6. Press the button on the XSA board and see what happens.



Now let's take a look at the code.

Open the tutorial module by double-clicking on "tutorial (tutorial.v)" in the sources pane in the upper left. You will notice how short this module is. Since it is the top-level module it should do nothing more than wire together other modules and connect them to the inputs and outputs. As you can see this is all it does. The module starts out defining itself with:

```
module tutorial(
                    clock,
                    vga_hsync, vga_vsync, vga_red0, vga_green0, vga_blue0,
                    vga_red1, vga_green1, vga_blue1,
                    push_button
                    );
```

which tells verilog that this module is called tutorial and that it has the ports listed between (and). Note that this first line ends with a ;. Also, the module itself is closed with the `endmodule` tag at the end.

Next in the module we tell verilog how to interpret the ports in the module definition:

```
// *** INPUTS ***
input clock;
input push_button;

// *** OUTPUTS ***
output        vga_hsync, vga_vsync; // sync signals for monitor
output        vga_red0, vga_green0, vga_blue0, vga_red1, vga_green1, vga_blue1;
```

This is pretty straight forward. Inputs are defined as such and outputs as well. You can have `inout` ports, but those require tri-stating which is not something we will be doing in 183.

Next comes the guts of this module, instantiating the VGA flag pattern you see on the screen:

```
// Instantiate our VGA output with the push_button wired directly to it
vgaflag vgaOutput(
                            .clock(clock),
                            .vga_hsync(vga_hsync), .vga_vsync(vga_vsync),
                            .vga_red0(vga_red0), .vga_green0(vga_green0),
.vga_blue0(vga_blue0),
                            .vga_red1(vga_red1), .vga_green1(vga_green1),
.vga_blue1(vga_blue1),
                            .enable(push_button)
                            );
```

This section **instantiates a copy** of the `vgaflag` module and calls it `vgaOutput`. It is very important to realize that **this is NOT a function** call. Every time you instantiate something like this you are replicating all of its logic on the FPGA. These copies operate in parallel. Let me say that again so there is no confusion: When you instantiate a module you are replicating it so you have twice as much logic operating in parallel. This is not a function call. Keep this in mind when simulating.

As you can see from this example, the `vgaflag` module has ports named `clock`, `vga_hsync`, `vga_vsync`, `vga_red0`, `vga_red1`, `vga_green0`, `vga_green1`, `vga_blue0`, `vga_blue1`, and `enable`. It doesn't matter in what order you specify these ports; just use the `.portname(signal)` notation to connect them. It is also not clear from here whether these are inputs or outputs. What is important to note is that the `tutorial` module's inputs and outputs are simply passed down into this instantiation of the `vgaflag` module. So here we see that the `push_button` input is being passed to the `enable` input on the `vgaflag`. This makes sense because when we pushed the push button on the working design it disabled the VGA display. (Remember that the button is active low!)

This should get you thinking about our overall design. Maybe we could modify the `vgaflag` to take in an enable for red (`enable_r`) and an enable for green (`enable_g`) instead of just one global enable. Then our control circuitry would only have to generate those two enables and the `vgaflag` would handle the rest. Sounds good to me. Let's take a look at how we might do that.

Open the "vgaflag (vgaflag.v)" module. Inside you'll see the same sort of module definition you saw in the tutorial module, but you'll notice that in addition to the `inputs` and `outputs` there are some `wires` defined:

```
	wire	[9:0] YPos;	// [0..479]
	wire	[10:0] XPos;	// [0...1287]
	wire	vga_valid;
	wire	Valid;
```

Wires are, well, just what you would think they are: wires. They serve to connect two signals together in a static manner. (I.e., they can not be used in a procedural assignment block where the tools determine the logic, but we'll get to that later.) This means that

they are not variables. They do not "store" values. They merely connect an output to an input in some manner. Any "storing" of values **must** be done using flip-flops in EE183.

In this case the wire `YPos` is defined to be a bus with 10-bits (9,8,7,6,5,4,3,2,1,0) while the `vga_valid` and `Valid` wires are a single bit wide, by default. The tools can be somewhat flaky about realizing how many bits are in a wire, so it is best to always be explicit when using wires with more than 1 bit.

It is good practice to define all your `inputs`, `outputs`, and `wires` at the top of each module to keep it clean. Please do this unless defining a wire locally makes the code significantly easier to read.

Below those definitions you'll see that the `vgaflag` module instantiates the `sync_gen50` module and hooks it up. (This is the guts of the VGA stuff but we won't be dealing with it now. It will be covered in lecture.)

```
sync_gen50 syncVGA( .clk(clock), .CounterX(XPos), .CounterY(YPos),
       .Valid(vga_valid),.vga_h_sync(vga_hsync), .vga_v_sync(vga_vsync));
```

Below this is the guts of the module. You will notice that the `sync_gen50` module gives out an X (`XPos`) and Y (`YPos`) coordinate and a valid signal (`vga_valid`). The rest of this module merely sets the color when the VGA is valid and is in the correct location.

Note: The VGA, as we are using it, has 6-bits of color (2 for red, 2 for green, and 2 for blue). You turn them on by setting `red0`, `red1`, `green0`, etc. high, and low to turn them off. That is, white is all on, black is all off, bright green is just `green1` and `green0` on and dark green is just `green0` on. (`green1` on by itself will give medium green.)

Now how does this all work? Well, let's take a look:

```
wire red0 = Valid &&       ((XPos <200 ) ||
                            (YPos > 80 ) );
wire red1 = Valid &&       ((XPos > 200) && (XPos  < 400) ||
                            (YPos > 80 ) && (YPos  < 160) );

assign vga_red0   = red0 ? 1'b1 :1'b0;

assign vga_red1   = red1 ? 1'b1: 1'b0;
```

(I've skipped the similar lines for green and blue and the first definition of `Valid` for now.) The module defines several wires (`red0`, `red1`, etc.) as the logical AND (&&) of a `Valid` signal and some criteria on the position. In the first line we see that red0 will be true (high = on = dark red) if `Valid` is true AND the `XPos` < 200 or the `YPos` > 80. The other colors are defined similarly. Below that the actual outputs are defined. In the first case `vga_red0` is defined to be 1 if `red0` is 1 and 0 if `red0` is 0. Note several things here:

First, this is silly. Why not just say `vga_red0 = red0`? No good reason, personal preference. The tools will minimize/eliminate this.

Second, note the way constants are defined: 1'b1 says "a 1-bit number that has a binary representation of 1." Similarly, I could say, 4'hf to get a 4-bit number that has the hexadecimal representation of 0xF (i.e., 16 in base 10). Here's the gotcha: what does 15 mean? Well, it depends. If it's a 4-bit bus then it means 1111, if it is an 8-bit bus then it should mean 00001111, but you'd have to trust the tools to give you that. Bottom line: always explicitly define the widths of your constants.

Third, note that the assign operator is wiring up the ouput `vga_red0` in a static fashion. In this case it is defining a MUX based on `red0` which selects between 1 and 0. This is a good way to define MUXes.

Fourth, note that the `wires` above were defined in-place. This is the same as if they had been defined as:

```
wire   red0;
assign red0 = Valid &&     ((XPos <200 ) ||
                            (YPos > 80 ) );
```

Use whatever is more readable.

Now, what is going on with the `Valid` signal?

```
// Turn the display on and off based on the enable input
assign Valid = vga_valid && enable;
```

It looks like `Valid` is defined to be true whenever the VGA is valid (`vga_valid`) AND the `enable` is true. Since the color output is ANDed with this `Valid` signal that means that the color will shutoff whenever the VGA is not valid or whenever the `enable` signal is false. This is cool. Remember that the `enable` signal is coming in from the `push_button` in the `tutorial` module so whenever the `push_button` goes low (i.e., it's pressed) the VGA will shutoff. Now we see how the display goes away when the button is pressed.

That's fun, but our project requires us to turn the red and green on and off separately. So let's modify the `vgaflag` module so it will do this.

There are a few steps to modify vgaflag to incorporate these changes:

1. We need to add the new inputs enable_r, enable_g, and (optionally) enable_b to the module port definition and input list.
2. We need to remove the generic enable from the module port definition and input list.
3. We need to change the code so that the red, green, and blue each obey their own valid signals.
4. We need to modify the tutorial module so that it passes in the correct signals.

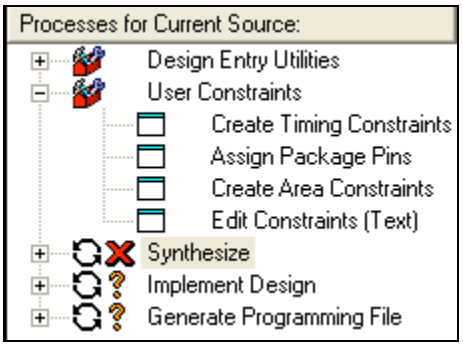
Let's start out with 1 and 2. Change the module definition to list the three color enables above instead of the single enable and update the input list. Now save the file.

Now we need to make the logic actually work. This part is up to you. You will need to modify some combination of the `wire` and `assign` statements at the bottom so the reds are

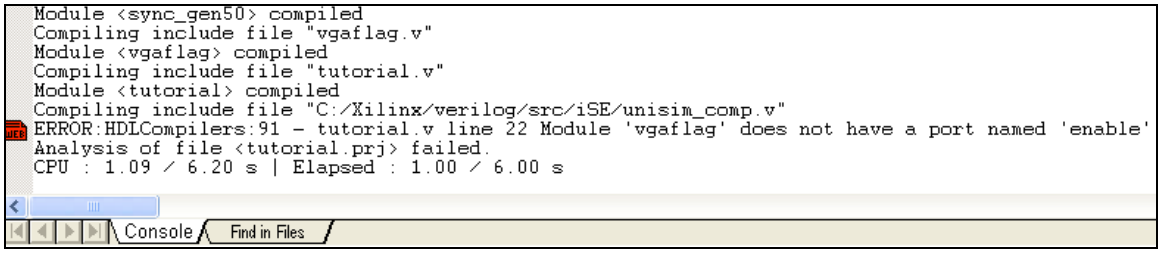
only true when the vga is valid AND the position is correct AND the `enable_r` is valid. There are two easy ways to do this: one is to just add a second `enable_r &&` to the `wire red =` assignments and the other is to change the `assign vga_red0 = red0 ? 1'b1:1'b0;` to instead assign the value of the `enable_r` if `red0` is true. Either way is fine, but remember that you need to change the `Valid` signal so it is the `vga_valid` and not the AND of that and the generic (and now removed) `enable`.

Once you've made these changes, make sure your top-level file "tutorial (tutorial.v)" is selected in the sources pane and right-click on "Generate Programming File" in the "Processes for Current Source" pane and chose "Rerun all."

You will notice that the compilation fails this time.



So scroll up in the "Console" window at the bottom until you find the first error:



You will see that the error is in the `tutorial` module and that the problem is that we changed the port list for the `vgaflag` module but we didn't update the `tutorial` module. If you right-click on the small red "web" icon to the left of the error, you can open a somewhat helpful webpage on the error message.

Let's go back and update the `tutorial` module to pass the `push_button` into the `enable_r` and pass in a constant `1'b0` and `1'b1` to the `enable_g` and `enable_b`, respectively. Don't forget that you need a . before each signal name. Now you should have this in the `tutorial` module: (Notice the extra , after the last port name in the module. This will cause all sorts of cryptic error messages when you compile, so don't include it!)

```
// Instantiate our VGA output with the push_button wired directly to it
vgaflag vgaOutput(
              .clock(clock),
              .vga_hsync(vga_hsync), .vga_vsync(vga_vsync),
              .vga_red0(vga_red0), .vga_green0(vga_green0),
.vga_blue0(vga_blue0),
              .vga_red1(vga_red1), .vga_green1(vga_green1),
.vga_blue1(vga_blue1),
              .enable_r(push_button), .enable_g(1'b0), .enable_b(1'b1),
                    );
```

Re-synthesize and implement the design and download it to the FPGA. Now try pushing the button and you should see that the red turns on and off with the button, the green is always off, and the blue is always on. You'll notice that you get some warnings about the fact that the green lines are never used. You should expect this because with the enable hard-wired to 0 the logic reduces to 0 all the time, so those connections will be removed by the tools. **Whenever you see a warning indicator you should always check to make sure what it is warning you about is okay**. A lot of nasty things can pass by as "warnings."

This is great. Now we know how to deal with an input from a push button and how to control the color going out to the VGA module. Now it's time for some system-level design and simulation work.

This is a good time to take a break.

Part III: System Design

Now that we have some feeling for how the inputs (buttons) and outputs (VGA) work, we need to determine how the signals will interact based on the specifications for the project. This is the point where we are going to decide how to break up our design into several small FSMs and what each FSM will do.

There are a few key things to notice in this project. The first is that the "lock-out" function controls the whole design and is a very simple function (toggle on/off) which just enables the color controls. The second is that the logic for controlling both the green and red enables is the same. This leads us to divide up our design into one small FSM for the lock-out and two identical FSMs for the color enables in the VGA module.

FSMs:

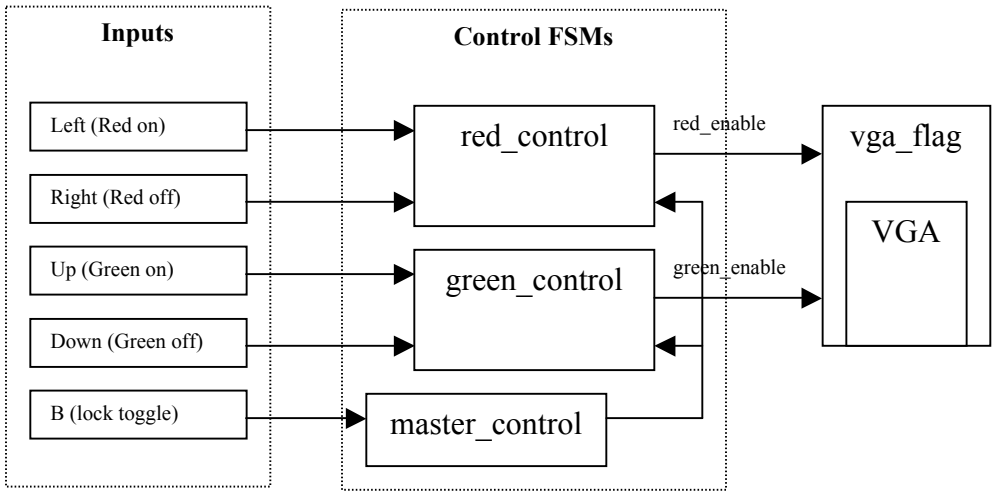
`master_control` – Handles the lock-out toggle button and tells the other FSMs when they can operate.

`color_control` – One module for each color which handles the color enabling. We'll have two of these, so we'll call one `red_control` and the other `green_control`.

Other elements:

We have our 5 inputs and the VGA module we just designed in the first section. That's about all we need to know to get started.

So here is our design currently:

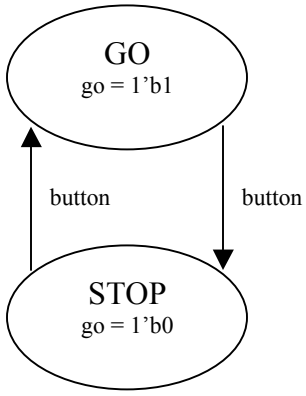


Now we are going to need to modify the top level of the design to have those five inputs instead of the push button at some point, but right now we'll focus on designing and debugging the two FSMs.

We'll start out with the `master_control` FSM. This state machine has two states: one for enabling the `red_control` and `green_control` FSMs (we'll call that state `GO`) and one for

disabling them (we'll call it `STOP`). The FSM has one input from the button and one output, called 'go' which is high when the other FSMs should be enabled.

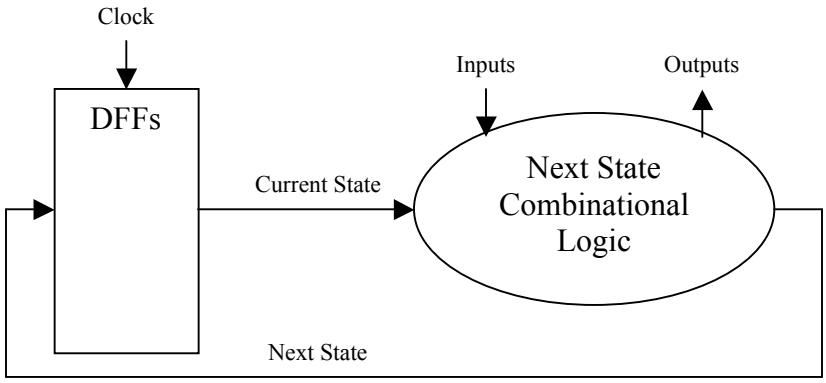
Here is a bubble diagram for this FSM:



In either state, if the button is pressed we transition to the next state. The output is dependent only on the current state, so this is a Moore state machine. We have two states so we will use one FF to store the state as `GO=1'b1` and `STOP =1'b0`.

An Overview of FSMs (This stuff is REALLY important.)

Before we dig into the verilog for defining an FSM let's review how all FSMs work. We have a state register which is just a bunch of D-flip-flops holding/defining/outputting the current state, some combinational logic which takes in the inputs to the FSM and the current state and calculates the next state and feeds it back into the DFFs. This way we calculate the new next state at every clock based on the current stat and the inputs.



You should keep this layout in mind every single time you design an FSM because this is the logic you are synthesizing. In EE183 you are required to explicitly instantiate your FFs and every output for your next state logic. (This will be covered in the lecture.) What this means is that you will be manually and explicitly defining every element in the above diagram for every FSM you create in this class. So get comfortable with this diagram!

The master_control FSM

For the `master_control` FSM the Next State Combinational Logic is very simple. In pseudo-code we have:

```
case (state) {
      STOP:
            go = false;
            if (button)
                  next_state = GO;
            else
                  next_state = STOP;
      GO:
            go = true;
            if (button)
                  next_state = STOP;
            else
                  next_state = GO;
            }
```

This code above is very close to the real verilog and has everything except the FFs for storing the state in it. So all we need to make this FSM is a `module` definition, some `wires` to connect the parts together, a flip flop to store the state, and some definition of what `STOP` and `GO` mean, and we're all set.

I've already put the fully functional verilog for this FSM together for you. Go ahead and download the `master_control.v` file from the EE183 web page and save it in the same directory as your tutorial project. Then right-click in the "Sources" pane and choose "Add Source…" to add the file. You will notice that the file references a module called `dffre` (not surprisingly this is a DFF with reset and enable) which shows up as a ? because it doesn't exist. This module is defined in the `183lib.v` file, which you can (and should) download from the class home page and put in the same directory. If you now add that source file to your project you will see that the file changes to a known file and two other modules (`dff` and `dffr`) appear in your project.

PLEASE go through and read the entire `master_control.v` module and make sure you understand the contents. You haven't seen registers (`reg`) before, and this file should do a pretty good job of explaining them, particularly with regards to the output `go`. A `reg` is not a storage element as we use them in EE183. They are just a type of connection which can take on a combinational value assigned by a procedural block (i.e., with an `always @()`). Unlike a `wire`, you do not explicitly define the logic for a `reg` but you define the behavior using `if/else` and `case` statements and let the tools determine the logic. You should also note how the `` `define `` statements are used and the structure for the combinational logic. You will be responsible for implementing this part of the `color_control` FSM to complete the tutorial, so make sure you have at least a good idea of what's going on here.

Go and read through the master_control.v file carefully.

Now that you've seen how a FSM is implemented it's time to test it in the simulator. We will be using ModelSim for this class, which is a functional simulator. This means you don't have synthesize your projects before simulating them and you can use various verilog commands to make your simulations more intelligent. There is a specific ModelSim tutorial linked off the handouts page, which you should go through after this tutorial if you feel you need more practice. (It also has pictures.)

Our use of the simulator here will be purely for testing the `master_control.v` module and you will be on your own for testing your `color_control.v` module. You will be expected to turn in simulations of both with your tutorial writeup. (In EE183 you will be expected to turn in simulations of all your key FSMs and data-path elements for every lab.)

To simulate the `master_control` FSM we need to create a test-bench verilog script which instantiates the `master_control` module and defines the inputs we wish to use to test the module. This test-bench is a standard verilog file, but unlike your project files it will use a lot of verilog constructs which are not synthesizable. You can get a full list of these in the verilog handout on the handouts section of the web page if you're so inclined. The ones we will be using are `initial` blocks, `delays`, and `monitor` commands. Others can be used for more sophisticated testing, but it is critical that you remember that **these directives are purely for the simulator; they are NOT synthesizable!**

You should now download the `master_control_test.v` script from the web page into your project directory, but you do not need to add it to your project. Instead just open it and read through it. The key things are that this is a test module which instantiates the module(s) we wish to test and defines the inputs. In this case we have a clock which goes high and low every 10 simulation steps (so the clock period is 20 simulation steps) and we reset the device at the beginning, enable it, and then put in some pulses on the button input. At the end we display the inputs and the resulting `go` output.

1. To run the simulator we need to start ModelSim.
2. When ModelSim starts up you need to create a new project (File->New->Project…) in your directory. Name the project something like "tutorial_simulation" and pick a nice default library name like "tutorial."
3. The "Add items to the Project" window now pops up and you should click "Add Existing Item" to add both the master_control.v and master_control_test.v files. Make sure "Reference from current location" is checked.
4. Then close the window and choose Compile->Compile All. Everything should go fine with no errors.
5. Now choose View->Signals, View->Source, and View->Waves to display your simulation data.
6. Now choose Simulate->Simulate and select tutorial->test_master_control. (The name is backwards because although the file is named "master_control_test.v" the actual module is named "test_master_control." Things are easier if you keep your file and module names consistent.) Then click Okay.
7. You should get an error saying that you don't have the dffre module loaded, so right-click on the file list and choose Add Existing File then add the 183lib.v file.

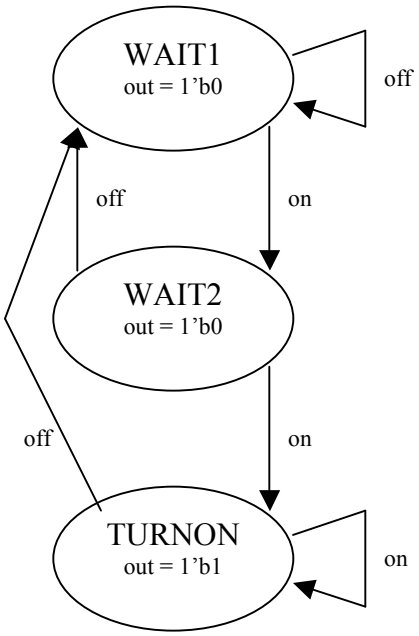
8. Re-compile all and choose simulate again.
9. Now you will see the signal names in the “signals” window. Select them all and drag them into the “waves” window.
10. Now choose Simulate->Run 100ns in the ModelSim window to run the simulation for 100ns.
11. Then choose Simulate-> Run –all in the ModelSim window. This will run the simulation through all the defined time in the master_control_test.v file, so you will get an extra 1000 simulation steps at the end and you will see that the simulation has stopped in the “source” window at the #1000 $stop; line.

Notice that there is something strange going on in this simulation. The first single button press results in three changes of the go output. Why is this? What do you need to do to make sure this doesn’t happen when the user presses the button? We’ll get back to this later, but keep it in mind.

That’s the extent of the simulation tutorial you’ll be getting. It should be clear from this how you can set up test-bench verilog files to test your code and how to run it. You have standard break-points and stepping through code functions in ModelSim as well. The one thing you need to keep in mind is that what you are simulating is a bunch of parallel processes, so if you instantiate two modules they can both be doing things at the same time!

Part IV: Designing a FSM on your own

Now it is time to design the `color_control` FSM. I'll give you the bubble diagram and a mostly complete verilog file and you'll have to fill in the next-state logic. Make sure you're comfortable with how this FSM does what was specified in the project description.



Now go and download the `color_control.v` file and fill in the missing bits. You'll be turning in this file so make it nice and well documented! There are 6 questions in the source code which you must answer in the code and include it your write-up. You have seen all the answers except on which you will figure out when you simulate if you don't already know it.

The only new thing in this code is the use of parameters in the `dffre` module instantiation at the bottom. The `#(3)` tells the `dffre` module to use the value "3" as the first parameter in its definition. (Take a look at the `183lib.v` file if you're curious.) This simply gives you a 3-bit wide bank of FFs.

Introduction to Inferred Latches: The Big EE183 no-no

(We'll talk about it a lot in lecture, so just go through this to get the gist of it.) Say you have a state machine which goes like this:

```
case(state) {
       STATE1:
              out = false;
              yummy = false;
              if (mushroom)
                     next_state = STATE2;

       STATE2:
              out = true;
              if (mushroom)
                     next_state = STATE1;
              else
                     next_state = STATE2;
```

What is the value of `yummy` if we go from `STATE1` to `STATE2`? Think about this.
Is it false because it was false in `STATE1` and we just moved to `STATE2`? Or is it undefined?

The answer is that it will be false, because the tools will realize that the only way to define that output (and all actual circuits have to have their outputs defined at all times) is to insert a FF, which will keep track of the value of `yummy`. When you're in `STATE1` it will get set to false, and otherwise it will just keep that value. This is called an **inferred latch** and is **a very bad thing in EE183**, and a hard thing to debug in general.

Can you see another inferred latch in the code above? Think about what happens if you are in `STATE1` and `mushroom` is false. What is the value of `next_state` in that case? You guessed it: the tools will infer a latch to store that value for you. Bad idea.

In EE183 we require that every single latch you use must be explicitly defined using the 183lib.v FFs. This makes our lives easier and yours, and doesn't slow down your hardware. There are two ways to make sure you don't get caught by this. The first is to make sure that every possible output for your combinational logic is defined and the second is to never use `@(posedge clk)`.

The first rule boils down to thinking about what logic you are instantiating. If one part of your logic defines an output for a given variable, then every possible case in your logic must do so as well. In this case it would mean defining the output for `yummy` in `STATE2` and putting an `else` in `STATE1`. **You can conclude that for every single `if` you must have a default `else`, and that it is a good idea to have a `default case` for every `case` statement. We will require both for this course.**

The second rule about no `@(posedge clk)` is simply stating that you are not allowed to gate the clock, which is how you make FFs. (Check out the 183lib.v if you're interested.) Again, **in EE183 you may not ever use `@(posedge clk)`.**

Once you have created your `color_control` FSM and a simulation script (probably based on the `master_control_test` script) you will discover that if you implement the `color_control` it will give you a warning saying it has inferred a latch if you did not specify outputs for the default case. This is bad. The lesson: always check all your warnings to make sure you're okay with them. (This, by the way, is the answer to question 6.)

At this point I'm going to assume you have a working and simulated `color_control` FSM. You will need to turn in the simulation results for this so make sure you've got them. (You can either use the Print Screen button to take a snap shot and then edit it in MS Paint or print to a PDF and add it in later with Acrobat.)

Part V: Putting it all together (or, "the last 10% which takes the other 90% of the time")

Now we've got all the FSMs for the project together and simulated and it's time to wire it up. (Make sure you've added both the `master_control` and `color_control` files to your project at this point.) We need to define all the inputs and outputs we will be using and set pin numbers for them. We do this through the `.UCF` constraints file (`vgaflag.ucf` in this case). You can edit this by double-clicking on it in the project window, but unless you have synthesized your project it won't let you. (Or at least I don't know how to make it let you.) So just open it up with Notepad.

Here's the original UCF file.

```
NET "vga_hsync" LOC = "P23";
NET "vga_vsync" LOC = "P26";
NET "vga_red0" LOC = "P12";
NET "vga_red1" LOC = "P13";
NET "vga_blue0" LOC = "P21";
NET "vga_blue1" LOC = "P22";
NET "vga_green0" LOC = "P19";
NET "vga_green1" LOC = "P20";
NET "clock" LOC = "P88";
NET "push_button" LOC = "P93"; #Inverted pushbutton
```

1. We want to comment out the `push_button` and add in the game pad inputs. (More information on the game pads can be found on the web page under Spring 2002 handouts, number 5.) For now I'll give you the information. You want to add the following NETs:

```
NET "ab_in" LOC = "P64";
NET "dot_led" LOC = "P44";
NET "gamepad_select" LOC = "P30";
NET "right_in" LOC = "P54";
NET "up_in" LOC = "P28";
NET "down_in" LOC = "P27";
NET "left_in" LOC = "P56";
```

And comment out the `push_button` net. The `gamepad_select` and the `dot_led` are outputs that you will use to choose between the A/start and B/C buttons on the gamepad (it's multiplexed internally) and to indicate to the user that the device is enabled (with the dot LED on the display).

2. Now that you've got that added to your `UCF` file you need to change your top-level tutorial file to include the correct inputs and outputs in the module definition. Go through and do this.

3. Now instantiate your `master_control` in your tutorial module and wire it up to the clock, the `ab_in` button input, and make a wire to connect its output to the two `color_control` FSMs. Don't forget to hard-wire the resets and enables to 0 and 1 on this FSM.

4. Use an `assign` statement to have the output from the `master_control` also go directly to the `dot_led` so we can see what state it is in. Also assign the `gamepad_select` to a constant true value so we will activate the B/C buttons.

5. Now instantiate two `color_control` modules and call one `green_cc` and the other `red_cc`. Wire them up so that their enable comes from the output of the `master_control` and their output goes into the `enable_g` and `enable_r` on the `vgaflag` module. You will need to define two more wires to do this. Use the `left_in/right_in` and `up_in/down_in` to control their on/off inputs.

6. Now generate your .bit file, hook up a game pad according to the web instructions, and see what happens! (Make sure you have the top-level `tutorial` module selected when you re-generate your .bit file.) Are any of the warnings you got a problem? You'd better check!

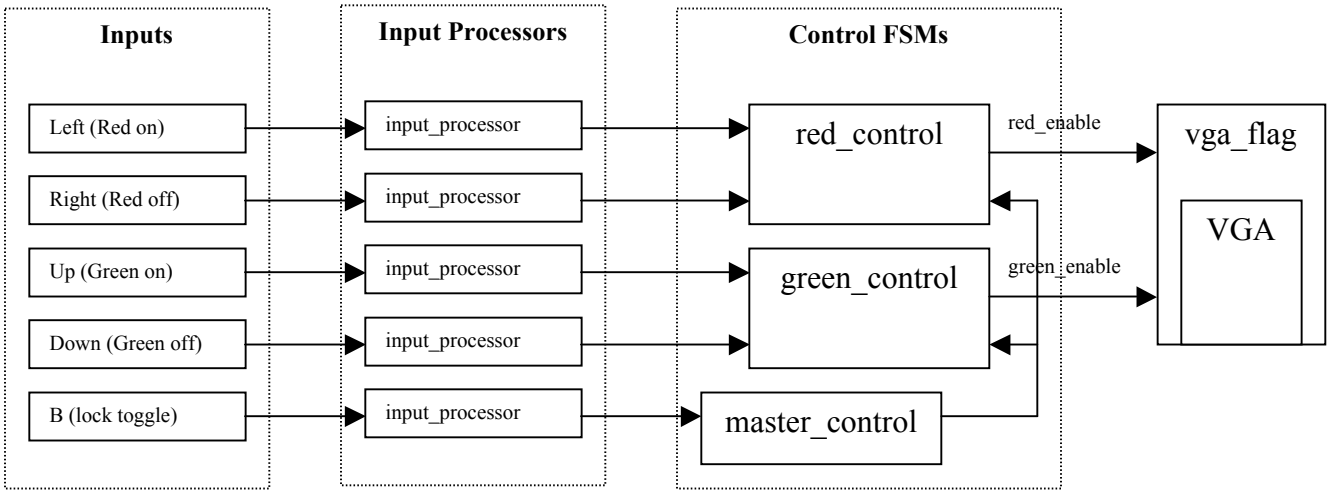
You will notice that this doesn't quite do what you want. The first problem is that you need the output of the buttons to come as a 1-clock-pulse rather than an on/off or your FSMs will keep changing over and over again at the rate of 50MHz. (Hence the small dots that appear to show up as on/off patterns in the color.) While we're fixing that we should also mention that it is very important to synchronize any and all inputs you have to your system to make sure they are in the same clock domain as the rest of your system to prevent metastability issues further on in your design. So now we will add a synchronizer and a one-pulse device.

Synchronizer module
This is simply a module where the input from the other clock domain goes into one flip-flop and the output goes into a second flip-flop. The output from that second flip-flop is then the "synchronized" input. That is, it is (with a phenomenally greater probability) synchronized to our clock domain. I've provided the framework for this module in `synchronizer.v` on the web page. You just need to instantiate the flip-flops and wire them up. This one is pretty simple so you don't have to simulate it if you don't want to.

One-pulse module
The one-pulse is also very easy. The incoming signal goes into the first FF and then the output of the first FF goes into the second FF. The output is the AND of the output from the first FF and the inverted (~) value from the second FF. Make a `one_pulse.v` module and add it to your design.

Input Processor module
Since we are going to have to have a synchronizer and a one-pulse for every single one of our inputs it makes sense to make a module called `input_processor.v` which takes in one raw input, synchronizes it and then one-pulses it. This will save a lot of typing later on. Go ahead and make such a module and add it to your project. Remember you will need a wire to connect the synchronizer and one-pulse in the middle.

Here's what we've got by way of system design now:



Now we're really getting close! Modify your top tutorial module to pass each input through an `input_processor` (you'll need 5 of them) and then onto the state machines. Use names that make sense, such as `up_process` for the `input_processor` module that handles the up input and `clean_up` for the output from that processor. Don't forget to check any warnings and make sure tutorial is selected as your top-level file before you synthesize it.

Success! (Well, almost)

Well what do you know? It works! Well, it works most of the time. You'll notice that sometimes it only takes one button press to switch the colors on instead of the two you expect, and sometimes pressing the "lock-out" button doesn't change the state of the `master_control`. What is happening is that the mechanical switches are bouncing. That is, every time you press them they click on-and-off a few dozen (hundred?) times before they settle down. This means that every time you press them you are sending in a lot of pulses and depending on how fast and how many you get the effect may be different.

The only way around this is a debouncer. This is a circuit which looks for a change in the input and when it first sees one it ignores all other inputs for a certain amount of time after that first input to give the switch a chance to settle down. It has to do this for both the low-to-high and high-to-low transitions. I've provided you with a `bad_debouncer` on the web page that you should download and insert in the `input_processor` after the synchronizer and before the one-pulse. (why?) This debouncer is terribly inefficient in its use of FPGA resources since you will have five of them, so you won't be allowed to use it for subsequent labs.

Once you've got this in there it should all work (assuming the gamepad isn't too old and flaky).

Part VI: Documentation

Congratulations! You've finished your first EE183 project. This project is easier than the others because you've got all this explanation, but the concepts are the same. You'll be doing everything you did in this project on the real labs but the FSMs will be a bit more complicated and you'll have some other stuff like RAMs and multipliers to deal with.

What do you need to turn in?

Basically, a min-report for this lab. There's more info on this under the Tao of EE183 on the web page.

Title Page
Introduction – "I did the tutorial and made a system which did X" (We know what you did so keep this short unless you did something particularly clever.)
Results – How did it work out? (Should be along the lines of, "It worked as advertised but the flaky game pads were difficult to get working.")
Conclusions – What did you learn? Comment on the hierarchical design process used here and if you think that was a good way to do this project. What would you do differently if you did it over again? We want to see that you thought about what you were doing here.
Appendices:
A1. Include **annotated** simulations of your key modules. You can either annotate these in word with the drawing tools or elsewhere, but we will not read simulations unless they have arrows or markers **on them** telling us what exactly we are looking at.
A2. You should include any files you wrote or modified and they should be commented as to what you did.
A3. Performance:

From the Place & Route Report you should copy and include the Device Utilization data. The % slices is the key figure of merit. (I ended up with 12% used.)

Critical path speed and the critical path: Under "Implement Design->Place & Route-> Generate Post-Place & Route Static Timing->Analyze Post-Place & Route Static Timing" then "Analyze against auto-generated constraints…" This will list your minimum clock period and your critical path at the top of the document. Describe what your maximum speed is and what your critical path is and if you can think of any way to reduce that critical path. (mine was 9.548ns)

You also need to include floorplan and routing images so you can see how your design was spread out on the FPGA and what was using up most of the area. "View/Edit Placed Design (FloorPlanner)" and then print this to a PDF or take a screenshot of it and put it in your report. Also include a screenshot of the routing information from "View/Edit Routed Design (FPGA Editor)."

Finally please include any comments to the teaching staff on this lab.