LAB #7: State Machines and FPGAs

This lab is due at 5pm in the EE locker for CS/EE 3700 on Thursday, April 5, 2001. The hardware should be demo’ed to a TA before 5pm on Thursday, April 12, 2001. NO LATE HOMEWORK WILL BE ACCEPTED.

1 Overview

This lab will involve quite a few new things. This lab will require you to:

1. Design a state diagram for a controller based on a written description of the operation of the system.

2. Demonstrate your ability to turn your state diagram into a circuit using both paper and pencil methods and VHDL synthesis.

3. Simulate a sequential circuit in PowerView.

4. Synthesize your circuit using Synopsys to target a Xilinx FPGA as the implementation technology.

5. Implement and test your state machine using a Xilinx FPGA.

The state machine you will be designing will simulate the turn signals of a Ford Thunderbird. The implementation technology will be a Xilinx FPGA.

2 FPGA Board Checkout

In this lab, you will be using an FPGA board. This board has a Xilinx FPGA, a socket for an EPROM, and just enough space to wire your input toggle switches with resistor pack, a push button for your clock (remember to add a chip to debounce your clock), and two output LED packs with a buffer chip. For each lab kit, you will check out one board and a packet of EPROMs from DSL. This means, if you have a partner, you will need to share the Xilinx board. Note that you do not need the board to do most of the design. Please checkout your board and chips as soon as possible.

THIS IS NOT A GROUP PROJECT. You are responsible to do the entire design yourself except the I/O circuitry, if you share a board. This means you should have your own state machine design, PowerView schematic, and programmed EPROM. If you are in a partnership, both partners must show up for the demo at the same time.

3 Thunderbird Turn Signals

The Thunderbird (or at least a simplified model of a Thunderbird) has three tail lights on each side of the rear of the car. When you engage the turn signal, the lights flash in sequence to indicate the direction you are going to turn. For example, a very crude drawing of the back of the car is shown in Figure 1. You can see three lights to indicate a left turn numbered L0, L1, and L2, and three lights to indicate a right turn numbered R0, R1, and R2. These lights operate in sequence to indicate a turn. An example sequence of lights for a left turn is shown in Figure 2. In this figure, the open boxes are lights that are off, and the filled-in boxes are lights that are on. Thus, the filled-in boxes imply asserted signals that turn on the lights. Basically, the lights light up in a sequence that points to the direction that you will be turning the car. The other use for these
lights is a hazard indicator. If a hazard is being indicated, all six lights flash on and off at the same time.

![Figure 1: 1965 Ford Thunderbird Tail Lights.](image1.png)

![Figure 2: Tail Light Sequence for Left Turn](image2.png)

Your job is to design a finite-state controller for those lights. There are five inputs, Clk (P13), Reset (P14), HAZ (P15), RIGHT (P16), and LEFT (P17). There are six outputs, R2 (P18), R1 (P19), R0 (P20), L2 (P23), L1 (P24), L0 (P25), one for each of the six tail lights. The numbers in parentheses are the Xilinx FPGA pin assignments. The operation of the machine is as follows: assume the clock controlling the machine is running at the same frequency as the desired flash of the lights. Anytime the HAZ input is high, all six lights should flash on and off at the same time on successive rising clock edges. If there is no hazard, and LEFT is asserted, the lights should sequence through the left-turn sequence. The lights should go through a complete left-turn sequence even if LEFT is brought low sometime in the middle of the sequence. If HAZ is asserted in the middle of a left-turn sequence, you should immediately start hazard-flashing. In other words, HAZ has precedence over LEFT. RIGHT causes a similar sequence of actions indicating a right turn. You may assume that LEFT and RIGHT will never be asserted simultaneously.

## 4 Design Procedure

The procedure for designing a state machine is described in Chapter 8 of your text. Basically, you need to design an abstract representation of the state machine in the form of a state diagram. This is the only really tricky part of the problem (if there is a tricky part). You need to understand what the sequence of events is that you want the machine to go through, which inputs it is using to direct those state transitions, and what the outputs will be in each state.
Once you have a state diagram, the procedure is largely turn-the-crank (although there are a few choices along the way related to which crank you turn). At this point you need to take the state diagram and translate it into a state transition table. This table lists all the current states, and from each current state and input combination what the next states will be.

Once you have the state transition table, you need to encode each of the possible states into an encoding that gives each state a unique code. This simply means to give each state a name that you can store in a bank of flip-flops called the state register.

With a state transition table and a state encoding, you can then use K-maps, espresso, sis, or whatever you like to generate a combinational function for each of the state bits. The inputs to the combinational function are the inputs and the current-state bits, and the outputs of the function (called the next-state function) are the next-state bits. You may also need to generate a combinational function that takes the current-state bits and generates output signals (called the output decoder).

You only need to turn in this paper and pencil design, and you do not need to draw schematics for it in PowerView. Instead, you should translate your state diagram into behavioral VHDL code as described in Chapter 8, and synthesize from there. Turn in your VHDL code, command files, and simulation logs.

5 Synthesizing Your Circuit

Once you are confident that your design is working, you need to synthesize an FPGA implementation.

First, a few warnings:

- **WARNING:** Some of the tools down the line count on having the entity be named the same thing as the VHDL file. So, if you write an entity named my_cool_entity make sure it's in a file called my_cool_entity.vhd.

- **WARNING:** The names of entities must conform to VHDL naming conventions. VHDL identifiers can ONLY contain alphabetic characters, decimal digits, and the underline character "_". They must start with a letter, they may not end with an underline, and they can’t contain two underlines in a row.

- **WARNING:** VHDL has a lot of reserved words. If you try to name a signal the same as a reserved word, strange and bizarre things will happen. The following are reserved words in VHDL. You cannot use them as identifiers in your VHDL programs!

  abs, access, after, alias, all, and, architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, constant, disconnect, downto, else, elsif, end, entity, exit, file, for, function, generate, generic, group, guarded, if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod, nand, new, next, not, not, null, of, open, or, others, out, package, port, postpended, procedure, process, pure, range, record, register, reject, rem, report, return, rol, ror, select, severity, signal, shared, sla, sl, sra, srl, subtype, then, to, transport, type, unaffected, units, until, use, variable, wait, when, while, with, xnor, xor

- **WARNING:** Remember that the std_logic type (which is the basic type you should use for all signals) is an enumerated type with characters as the input. If you want to set a signal to a high logic value, you need to set it to '1' using appropriate signal syntax (i.e. signal <= '1').

The synthesis process involves new tools in Powerview, as well as a completely new tool from Synopsys. There are lots of little picky details along the way, so pay attention, and read error messages carefully if you get stuck. This document will take you through the process by way of a tutorial using the moore.s.vhd state machine as the example (this can be found in ~myers/ee3700/lab7 directory). I recommend you try the process on this VHDL file first before launching into your own code.
5.1 Integrating VHDL Code with Schematics

This is an optional step. If you like, you can go directly to synthesis and not integrate things until you have gates. With this option, you can mix a piece of VHDL code with the schematics you already have to see if the VHDL code works in the context of your schematics. The Fusion/Speedwave simulator is a multi-mode simulator that allows VHDL programs to run in the same simulation with schematics. If you want to try this option, here are the steps:

1. Take the VHDL code and automatically generate a symbol for that code. The vhdl sym program looks in the VHDL code for the entity description and generates a symbol from that. Since the entity describes the external interface, and that’s exactly what a symbol does for a schematic that’s pretty straightforward. It also puts a bunch of attributes on the new symbol to indicate to the Powerview system that the symbol represents VHDL code instead of a schematic full of gates. The most important two are the VHDL attribute that points to the code, and the fact that the “type” of the new symbol is “module.” Symbols of type “module” represent placeholders for symbolic simulation code. Symbols of type “composite” represent schematics in Viewdraw. At some point if you pop into the schematics far enough, you’ll run into the actual basic gate types that everything else is built upon. These basic gates are in the builtin library and are of type “module.”

   Anyway, in the vhdl sym dialog box, fill in the source name with the name of the vhdl file (i.e. moore_s.vhd), with the name of the entity in that file that you want to make a symbol for (i.e. moore_s) and everything else stays as the default.

2. Open viewdraw. You should see a new component available in the project library with the name of the entity you specified in the previous step. You can put this component in wherever you can use the other components, and everything should work. Note that if you double-click on the new component to see what’s inside it you get a warning message about trying to decend into a type “module” which isn’t allowed. However, if you use the menu to push into the symbol, you’ll notice there are two extra choices to push into: VHDL Entity, and VHDL Architecture. Pushing into these will open a text window showing you the entity and/or architectures pointed to by the symbol.

3. You can now save this schematic that includes your VHDL symbol as one of the components. You’ll notice something different when you save. The fact that there is a VHDL module in your design causes a Design Rule Check (DRC) procedure to be run that wasn’t part of writing a file before this. It will point out things about your design. For example, if you have nets that are connected to inpin and output symbols but that don’t have names, it will complain that pins are not named. If you have a dangling input (an input that is not connected to anything, and isn’t connected to an inpin) it will complain that you have an undriven input. If you have the output of the VHDL module connected directly to an obuf, it will complain about incompatible pin types. As far as I can tell, all of these are unimportant for our process. If the Viewdraw window says Check Complete, 0 errors and 0 warnings in project foo at the bottom of the screen, I believe you can ignore the warnings of the DRC window. You might still want to read them though and see if they make sense... (In fact, sometimes the DRC program doesn’t run. I haven’t figured out what triggers it yet...)

4. Once you have a schematic that you like, you can operate on it exactly like any other schematic. That is, you need to run VSM to make a simulation file, and then you can run either Viewsim or Fusion/Speedwave to simulate it (they are actually the same simulator with slightly different dialog boxes to start them up). Note that when you start up the simulator, it reports some extra things related to compiling the VHDL for simulation before starting.

   Note also that if you want to watch the values of internal signals in the VHDL code, you can reference them through the same hierarchical naming scheme that you’ve already used. The symbol that represents the VHDL code is just another component in the schematic. You can use it’s internally-generated name (i.e. $1123 or something like that), or label the component something else using the label command.
5.2 Synthesizing the VHDL with Synopsys

Now here's the fun part. You can take the VHDL code and let Synopsys turn it into gates for you so you can include it as part of your project and map it on the Xilinx part with the rest of the processor.

1. I’ve made a new script that starts up Synopsys for use with the Xilinx 3195 part. Believe me, this is not a trivial task to get set up! The script is called syn-cade. It will make a new directory in your home directory called 3700-syn in which to run Synopsys. This is for the same reason as PowerView making its own directory: you don’t want Synopsys splatting all its many intermediate files all over your home directory!

   Running this script will set up lots of things, make a $HOME/3700-syn directory if it doesn’t already exist, connect to that directory, and copy the synopsys_dc_setup file from the class directory to your new Synopsys directory. Finally, it will start up design_analyzer which is the main Synopsys synthesis engine.

2. Copy your VHDL file to the 3700-syn directory. For example, put a copy of moore.s.vhd in your 3700-syn directory.

3. Run syn-cade to start up design_analyzer. Remember to run this on a machine with as much memory as you can find. Synopsys chews up a lot of memory.

4. From the main menu, select File Analyze to analyze the VHDL code. You should see any VHDL files in your 3700-syn directory in the dialog box that you can choose. Make sure that after you choose a file (or type a file name) that the File Format is VHDL, and also make sure to check the box to Create New Library if it Doesn’t Exist. You should only have to do that the first time you run Synopsys, but it doesn’t hurt to do it after that too. Click OK to analyze the VHDL file.

   This will load a bunch of libraries for your Xilinx 3195 chip and analyze the file into the WORK directory which is the VHDL working library directory.

   Note – make sure to use Analyze instead of Read. The Read command reads in a Synopsys format db file. Later, you can save projects as <filename>.db files, and Read those in, but when you start from VHDL code, you need to Analyze.

5. Now from the menu select File Elaborate to expand the analyzed VHDL code into the first phase of synthesis. In the Elaborate dialog box you should select the DEFAULT library. When this is selected, you should see your entity/architecture pairs listed in the dialog box. You can now select the one you want to synthesize (i.e. moore.s(moore.s_arch) for the example file). Click on OK to start the elaboration process.

6. When elaboration is complete, you should see a yellow box in the design_analyzer window the Y=A+B inside (which indicates that it’s still code, it hasn’t been synthesized yet), and the name of the entity underneath (i.e. moore.s). Select this design by clicking on it.

   You can now descend into this design to see what’s there if you want (this is strictly optional).

   Use the icon on the left of the window to descend. This will show you the entity (symbol) view of the elaborated component. If you select the little AND gate icon on the left, it will take you to the circuit view. Notice that this is a terrible circuit! It’s too big, and too complex. However, don’t panic, it’s just the intermediate form. It’s not the final circuit!

   If you’ve done this, pop back out with the button before the next step.

7. With the yellow design selected, use the menu and select Tools FPGA Compiler. This is where you’ll optimize that intermediate form into a circuit that you can use. Select Optimization... from the FPGA Compiler dialog box. Inside the Design Optimization box, you can choose how much effort you’d like the tool to spend on your behalf. You can also choose to let the tool verify that it has generated a correct design. To be honest, I have no idea what it’s doing to verify things, and if it’s at all useful. I also haven’t seen much difference in using Medium
mapping effort versus High effort. However, I’ve only tried tiny things through this path. Click on OK when you’ve got things set the way you want. This will optimize the circuit into something you can use.

This is also the part that’s going to take a long time if you’re on the wrong machine. If you’re on the right machine it should take about a minute. If you’re on the wrong machine, you might as well go to lunch, or take a nap, or something. If it’s taking more than a half hour or so, you might want to find another machine.

Cancel the dialog box when it’s finished. You’ll notice that the yellow box now has an AND gate in it to indicate that it’s been optimized.

8. When this is done, you can pop back in to the circuit (select the circuit and using the up and down arrows, AND gate, and other icons to select your view. You should see a vastly improved circuit!

9. Back in the FPGA Compiler box, you now select Save As... to save the circuit. You can save is as a Synopsys .db file in which case you can Read it back in at some point to work on it some more. However, what you really want to do is Save As... to an EDIF file. Select Save As... In that dialog box select EDIF as the File Format. Now, make sure that the File Name gets updated to <filename>.edn. It won’t do this automatically. You have to type it in. Then click OK to save the EDIF file.

You now have an EDIF file of the synthesized circuit that you can read back into Powerview. Note that the name of the EDIF file will be the same as the name of the entity from the VHDL. This is important. Powerview really likes to keep the name of the component the same as the name of the file that holds the information about the component. You should follow this convention unless you want lots of trouble from the tools. You can now quit Synopsys.

5.3 Integrating the Synthesized VHDL into Powerview

Now that you have an EDIF file, you can read that file back into Powerview and use it in your schematics. The basic process is to read the EDIF and convert it into a wire file that Powerview understands. You also need to create a symbol so you can use the new synthesized component. The steps are:

1. Start up Powerview with pv-cade or pv-cs depending on where you’re working.

2. Copy the EDIF file to your Powerview project directory. The EDIF file you ended up with from Synopsys will be in your 3700-syn directory. Copy it to your $HOME/3700-pv/<projectname> directory. (i.e. cp $HOME/3700-syn/moore_a.edn $HOME/3700-pv/<projectname>).

3. Read in the EDIF file using the EDIF netlist in program in Powerview (an icon in the cockpit). Put the name of the EDIF file in the EDIF netlist file space, and leave the other fields as they are. If everything is working properly, the tool will tell you that it created a wire file.

4. Now you need a schematic. Actually, you really don’t need a schematic. You can do everything without one, but it can be nice to have both to look at and as an aid for simulation, not to mention documentation. So, it’s a good idea to have a schematic. The program that generates the schematic from the wire file is ViewGen. Open up this tool from the Powerview cockpit. Now you have an interesting issue about symbols based on whether you previously made a symbol for this VHDL file or not.

(a) If you did not previously make a symbol for this VHDL component, then you should use ViewGen to make the symbol. Open ViewGen from the Powerview cockpit. Fill in the Design Name which is the name of the entity from the VHDL code (and should be the same as the file name). Make sure that both Make Schematic and Make Symbol are
selected (Make Schematic should be selected by default). Also select a sheet size for the generated schematic. If you select B the sheet will be the same size as all your other sheets, but ViewGen will split it up into multiple sheets if the circuit is too big. If you select C, D, or E the schematic sheet will be huge and your circuit might end up as a little speck in the middle of a huge sheet. If you select AUTO the tool will make a sheet just big enough to hold your schematic. It’s probably the choice you should make. Click on OK and ViewGen will generate a schematic and a symbol that you can use like any other schematic and symbol in your project.

(b) If you did previously make a symbol for this VHDL component (i.e. to simulate the VHDL code with another schematic you have a slightly more interesting choice. You can reuse the existing symbol and make some changes to indicate that it’s now a schematic, or you can let ViewGen write over the existing symbol. The tradeoffs are:

i. If you reuse the existing symbol you need to modify some attributes. Fire up ViewGen and run it, but make sure that the Make Symbol choice is OFF. This will generate a schematic but not a new symbol. Because your new schematic has the same name as the pre-existing symbol, Powerwill know they are connected. However, the existing symbol has lots of attributes related to its previous role as a wrapper around VHDL code. You need to change some of these. The most important is to change the type of the symbol to “composite.” Go into Viewdraw, and either open the VHDL symbol in the symbol editor, or put the symbol in a schematic and push into the symbol. Once you are editing the symbol, from the menu select Change Block Type,.... In that dialog box select composite and click OK. By changing the type you’ve also told the simulators to look for a schematic instead of a VHDL simulation. On the other hand, there are still lots of attributes associated with the symbol, so there will be some lingering effects of those other attributes. I don’t think any of them are harmful, but it they bother you you will have to go back into the symbol and remove all the attributes except Pinorder. You need to leave Pinorder in there!

ii. If you let ViewGen write over the symbol you will get a new symbol of the right type (“composite”) and with only the attributes you need. But, the Pinorder attribute will list the pins in a different order than the original symbol did. The symbols will look the same, but with the different Pinorder all your wires will be messed up in any schematic that contained the old symbol. To fix this you need to either delete the VHDL-derived component from all schematics, then run ViewGen, then put the component back, or you need to update the new Pinorder attribute to put the pins in the same order as in the old symbol (the one that was created by vhdl sym).

5. Once you’ve figured out the symbol mess, you now have a symbol that represents your VHDL-derived Synopsys-synthesized circuit. You can use this symbol in the same way as any other symbol and schematic in your design. You need to run VSM before simulation, just like before.

6. The last thing you should do is include this new symbol in a new schematic and connect buffers and pins to it. The buffers and pins can be found in the 3100 library and are named ibuf (input buffer), ipad (input pad), obuf (output buffer), and opad (output pad). For each input that you want taken to a pin, you should connect an ibuf and ipad to it. Similarly, for each output, connect an obuf and opad to it. This will tell the Xilinx tools that you want these pins taken to a pin on your Xilinx chip so you can wire to them. You should also add an attribute to each ipad and opad. For example, if you want that pin to be connected to pin 14 of the Xilinx chip, then that attribute should be “P=14”.

This process sounds pretty painful, but it’s really not that bad. I encourage you to try it once with the example VHDL files from the web page just to see what’s involved. There are lots of little things to keep straight, but if you stick to relatively straightforward VHDL code, you should not have too much trouble.
6 Running the Xilinx Tools

In order to map your design to the Xilinx part, you need to get the design into the Foundation software. We’re going to bypass the entire schematic portion of the Foundation software to try to avoid the subtle incompatibilities between Viewsim and Foundation Schematic. So, you need to output an EDIF file of your design to act as the file transfer mechanism. Once your design is completely simulated, use the EDIF Netlist Out tool from the Powerview cockpit. This tool takes a Viewlogic “wir” file (the netlist file that describes your circuit, this file is generated when you vsm the circuit as part of the process that generates the vsm simulation file), and converts it to EDIF format which is an interchange format common in the CAD world.

In the edifnoto dialog box, type the basename of the wir file that you’re converting (if your top-level design (the one with the pads) is called fsm, then the name of the wire file is also fsm). In “level” type xilinx. This will keep the netlister from trying to expand things below the 3195 level. The output filename can be anything you like, but I encourage you to give it a .edn extension. In this example, fsm.edn would be a good output file name. Leave the author name blank, and leave the config file with the default. Press OK and you should get fsm.edn as the EDIF file that describes your circuit.

Now you need to get that EDIF file over to the NT lab where the Foundation software lives. Use ftp to transfer the EDIF file to the CS filesystem, or use a disk to get it to one of the NT boxes.

Once you’re on NT, you need to fire up Foundation, but DON’T use the Foundation Project Manager shortcut that’s on the desktop! Instead, use the Start menu to get to the Xilinx Foundation Series menu, from there go to Accessories, and run Design Manager. This will get you right to the Xilinx mapping software and bypass all the schematic nonsense. When that is running you need to create a new project, and give it the name of your EDIF file as the input file for that project. Click “implement” and, if you don’t get any errors, you should end up with a .bit file. If you do get errors, look at them carefully to try to figure out what they mean. If you can’t figure it out, let a TA take a look and maybe we can help decipher the message.

The .bit file will live in the xproj/verN/revN directory underneath the project you created where the N is the number of the version and revision. If you get this right the first time, the .bit file will be in xproj/ver1/rev1/<filename>.bit.

If you want to generate the EPROM file to program the FPGA close the Flow engine with “Flow – > Close”. Now start up the PROM File Formatter program. This is the second tool icon from the top and looks like a funnel with bits going in the funnel and HEX coming out. Before generating the EPROM file select “File – > Prom Properties” from the menu. Make sure that you have MCS-86 as the Prom File Format, Byte Wide as the Type, Single PROM as the Prom File, and 16K*8[128k] as the Prom Device [size in Bytes]. Note that you need to unselect “Automatic selection” in order to chose the PROM size. You should probably save these as the defaults.

Now you should see a display in the PROM File Formatted window that shows a PROM with a Data Stream below that, and a .bit file included in that. If that’s true, select the disk shaped icon or select “File – > Save” from the menu. This will save all the PROM files including the <designname>.mcs file which is your EPROM file for programming the Xilinx part. This file will be in, assuming that you’ve used the default locations for the Xilinx files, in xproj/ver1/rev1/<designname>.mcs or in some variation of that if you have other versions or revisions of this design.

Before you program the EPROM and try to use your Xilinx part, there is one last thing you need to do to the .mcs file. It turns out that although our prom burning program uses .mcs format, there is a slight incompatibility between the Xilinx version of .mcs and the Shooter version. So, you need to use a text editor of some sort to remove the first line of your <designname>.mcs file. This first line should start with :0200 and you should just get rid of it completely so that the first line of the .mcs files starts with :1000. Now you have an EPROM file that you can program into a 128K EPROM using the Shooter in the DSL. Instructions for doing so are posted above the PC in the DSL lab (be sure to erase your EPROM first before programming it, there is a UV-EPROM eraser in the DSL lab for this purpose).
7 What to Turn In

By April 5th, you must turn in the following:

1. A complete, documented, design of the state machine, including state diagram, symbolic state transition table, state assignment, encoded state transition table, next state equations, Karnaugh maps (or espresso or sis input and output files) for the next state logic, and the output logic.

2. A behavioral VHDL model for your state machine. Print outs of any PowerView schematics, command files, and simulation log files. Your simulation should be comprehensive enough to show that your state machine works. Make sure you think about what it means to test a sequential circuit whose description is a state diagram. You need to run the machine through enough cycles to demonstrate that it really is doing the right thing. You should make sure to include some text that explains your testing strategy and what your waveform shows.


By April 12th, you must demo your design working on the FPGA board to a TA. NOTE: although you are possibly sharing a FPGA board, THIS IS NOT A GROUP PROJECT. You can share the wiring of the board, but you must each do your own design, your own entry into PowerView, and have burned your own EPROM. If you are in a partnership, both partners must show up for the demo at the same time.