ELC 3338 Project Book

Steve Potter

August 20, 2018

Contents

Lab 1

Program Counter Register

During the course of this semester, we will build a 64-bit computer so that we can understand how it works. To do this, we will make a synthesizable machine in Verilog, a common hardware description language (HDL).

A computer runs a program by executing individual instructions in sequential order. The instructions are stored in memory and are accessed by their memory address. During each clock cycle, an instruction is fetched from memory and executed on the processor. The memory address of the next instruction to fetch is stored in a register called the Program Counter (PC). During Lab 1, we will build and test the Program Counter register. In Lab 2, we build an incrementer (to count to the next instruction) and a mux (to select between the incremented count or a new starting value).

1.1 Program Counter Register

In order to make the Program Counter, we are going to make a Verilog module that explains how to build a register (a D flip-flop). Let me unpack the previous sentence:

1. Verilog is a Hardware Description Language (HDL).

2. We write Verilog code to tell Vivado how we want our register module to behave.

3. Vivado reads our Verilog code and synthesizes a realizable digital hardware design that meets the behavior that we specified. Thank you Vivado!

4. Vivado also simulates the behavior of the hardware, allowing you to test your design without building/programming hardware.

Consider the Verilog code in Listing 1.1. It is made up of three sections: a header (which has the include command), a port list or interface (which specifies the signals coming in or going out of our module), and a body or implementation (which describes how to build it).

Listing 1.1: Verilog code to make a register.

```verilog
`include "definitions.vh"

module register(
    input wire clk,
    input wire reset,
    input  wire ['WORD-1:0] D,
    output reg ['WORD-1:0] Q='WORD'b0
    );

    always @(posedge(clk),posedge(reset)) begin
        if (reset==1'b1)
            Q<='WORD'b0;
        else
            Q <= D;
    end

endmodule
```

The first part is the header. We will use this same header each time. It tells the Verilog compiler to get all the data from a file called definitions.vh. The extension vh is a Verilog header. We use this to specify common pieces of data we will use across our design, so that all the components we build will be consistent. By putting them in one file, we make it easier to maintain, and prevent mistakes that can happen easily by having multiple copies of these basic pieces of data. For our first component the piece of data we will be using is WORD (set to 64), which is the size of the memory addresses our computer will use (how many bits). Note that if we build things based around WORD, rather than the number 64, we can just change the value of WORD in the file and get a computer with a different size with a couple key strokes.

The second part is the port list or interface. In this area we specify what signals are coming in (input), going out (output), or could go either direction (inout). Ports can be defined as either "wire" or "reg". This can be confusing to some students. Think of it this way:

1. Wire

   (a) A wire is just a conductor that connects one component or module to another.

   (b) The value on a wire can only be changed by using combinational logic.

   (c) It has no memory, meaning that the value on the wire is driven by the results of combinational logic at that particular moment.

   (d) Module inputs are always wires

   (e) Module outputs can be wires or regs.

Figure 1.1: Module Diagram.
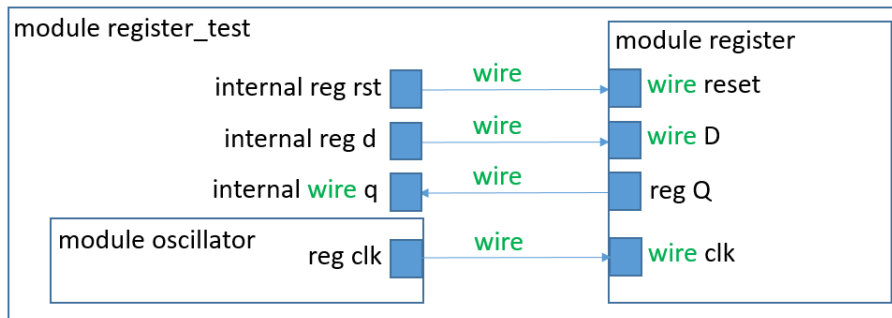
- Internal regs are set in the 'initial' block or an 'always' block

2. Reg

  (a) A reg more closely resembles a variable in software programming languages.

  (b) A value of a reg can only be set by using sequential logic.

  (c) A reg has memory, meaning that the value of the reg will remain the same until a sequential logic element updates it.

  (d) You can directly set a reg to a value using a procedural assignment.

  (e) Regs can be used internally in a module (neither input nor output), or they can be used as module outputs. They cannot be used as module inputs.

If you don't specify anything for the port type, you will get a wire - it is the default. In our case we have four signals: three inputs, and one output that is a register. The first two inputs are single-bit wires. One is the clock, which specifies the timing, and the other is reset, which clears the contents of the register (makes them zero). The final input is the value we want to store in memory, and I have called it D, following the convention of digital logic. D has multiple bits that are numbered from WORD-1 down to 0. Thus the leftmost bit is 63 in this case, and the rightmost bit is 0[1]. The output Q (also the digital logic conventional name) is a register (it will hold its value) and should also be of size WORD and follow the same order as the input D.

To help clarify this, please examine Figure 1.1, which shows the interconnection of the modules in this lab.

---

[1]If you want to be technical this is called little endian, since the little end (the least significant or unit bit) is going into the first memory location (bit 0). If you reversed the order by putting the 0 first and the WORD-1 last it would be big endian, since the big end (most significant bit) would go in the lowest addressed bit.

The final section is the body or implementation. It is composed of a single thread of code, that will keep running (hence always). It will run one time every time there is a positive edge (0 to 1 transition) for either the clock or reset. Reset has higher priority, so if reset is asserted the register is cleared (Q is set to zero), otherwise the value of D is stored it Q. That is it. A nice, simple module.

1.2   Testbench

We now want to test this. To test it, we need to tell the simulator to build a copy (instantiate) the module, and then we will need to supply the inputs and look at the outputs to verify that the module works correctly. Consider the testbench in Listing 1.2.

Listing 1.2: Verilog code to test a register.

```verilog
'include "definitions.vh"


module register_test;

wire clk;
reg rst;
reg['WORD − 1:0] d;
wire['WORD − 1:0] q;

oscillator clk_gen(clk);

register UUT(
    .clk(clk),
    .reset(rst),
    .D(d),
    .Q(q)
    );

initial
begin
    rst = 0;
    d<='WORD' d0; #'CYCLE;
    d<='WORD' d1; #'CYCLE;
    d<='WORD' d2; #'CYCLE;
    rst = 1;
    d<='WORD' d3; #'CYCLE;
    d<='WORD' d4; #('CYCLE/5);
    rst = 0;
    d<='WORD' d5; #('CYCLE*4/5);
```
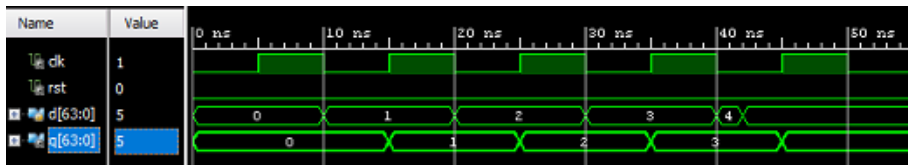
Figure 1.2: Timing diagram.

end

endmodule

Like our register it starts with our standard header, but this time there are no ports! A testbench is providing all the signals to simulate the inputs to the unit under test (UUT) and thus does not need them. This is how Verilog finds a top level simulation module - there are no ports. The clock signal will be driven by a module named oscillator, which will give us a nice square wave with period CYCLE, which is another constant defined in our definitions.vh file. The code thus makes an oscillator and a register, then runs the 'initial' section (it runs once at the start then never again). The initial section sets the value of the inputs then waits a CYCLE. The last couple of delays are not full cycles. I did this for two reasons:

1. To show you how to make Verilog do calculations for you.

2. To remind you that the input won't necessarily be nice and perfectly timed to your register. Unsynchronized signals happen, and is a frequent cause of problems, hence the need to test.

This is by no means an exhaustive testbench, but run it and look at the output. Does it do what you expect? What else might you want to test? Add this to your testbench and run it again to see if the register works.

1.3 Using LaTeX for Your Write-up

This section was originally written by Dr. Schubert, so any first person references are from Dr. Schubert.

All that is left is to write it up. I am going to have you use LaTeX to do your lab reports. Note how I include files, programs, and images. It is worth noting that LaTeX will automatically make the table of contents and bibliography for you also.

Why use LaTeX ? There are lots of reasons, but here are a few that matter in this course:

1. It typesets programs from the actual source, no need to copy the program and have spell checkers and grammar editors mess things up.

2. It quickly and correctly handles equations.

3. It automatically handles table of contents and bibliographies.

4. It is free, and generates high quality documents (book quality) - it is open source since before open source.

5. It is used in publication of research documents.

6. It is the only large program believed to be error free in its source code, and have no missing features (development is complete!)

1.3.1   Background

TEX refers to both a language for typesetting and the program (compiler actually) that does the typesetting. LATEX is a macro package which sits on top of TEX and provides additional functionality, and has become synonymous with the language variant (dialect) of TEX which it created. Since LATEX is hugely popular and really useful, TEX and LATEX have become synonymous to most people, and I will treat it so from now on. A note on pronunciation: TEX is in Greek letters - tau epsilon chi and hence is pronounced 'tek' not tex (similar for LATEX which is pronounced 'lay-tek' not latex).

TEX is not a WYSIWYG (what you see is what you get) typesetting program like many editors you are familiar with, as it was designed to be a tagged language like the more recent html (yes, TEXis older). The idea is not to spend time thinking about how it should look, but rather to classify what it is and let the automated standards set the text by what the text is[2]. To provide flexibility and extension (and it was designed by one of the greatest computer scientists, Donald Knuth) it was set up as a programming language with a compiler. Since LATEX is a programming language, we have a comment character % that I had to escape by putting a \before it to make it print. Whitespace past the first space (word separation) is ignored, except for a blank line, which means start a new paragraph. More than one blank line is ignored. To get more space, you issue a command, such as \vspace{.25in}, which puts a quarter inch of vertical space. LATEX also knows pt (points), px (pixels), pc (pica), mm (millimeters), cm (centimeters), em (width of an 'm'), and many more. By default the space is not placed if it does not separate some object (i.e. at the top of a page), but you can force it by using \vspace*{.25in}. Starred commands are just versions of the main command.

There are many more commands than we could describe in this brief intro, including commands to let you define new commands and environments. We will not need too many fancy commands, we only need to describe the commands to include figures, code, and equations. If you want to learn more, then I have links to free manuals online at r2labs.org.

---

[2]For instance, note the chapter, section, and subsection commands in the tex files. LATEX assigns a number, records it, the title, and page so it can automatically put it in the table of contents for you.

1.3.2 Compile Process

One thing that will help you a lot in working with LATEX is how the compile process works. TEX is a two pass compiler, but it does only one pass each time it runs. Allow me a brief introduction to compilers, which is a great course if you can take it.

When you are compiling a file you have control statements (branches, loops, conditional execution statements like if or switch/case) that require you to know how many program lines ahead or behind something is in the assembled code, which you will not know at the start. While you are often just putting in a flag or label to be handled by the assembler later, you in truth don't even know if they actually put the destination of the transfer of control, and thus have an error. One easy way of handling this is to run through the process twice, collecting labels and such the first time and then doing the compile the second time through, which is what a two pass compiler does. TEX collects all the labels, notes all the chapter, section, and other structures, identifies all the bibliography references, and so on and puts them in a special auxiliary file for the next pass. It will also create a DVI file, which has most things right, but will lack table of contents, references, bibliography, and such. The second time through it already has the information before the file runs so it reads that first and uses it to create a fully correct output.

A logical question at this point is why not just have it run twice on its own? Well, in the 1980's computers were small and slow, so each run of TEX(we didn't even have LATEX at first) took an appreciable amount of time. If you know the compile process, there are times you only have to run things once, like small spelling changes not in a title, chapter, etc. Allowing people to do only one pass at a time was a big advantage (some TEX compiles I had to do could take 10 minutes even in the 1990's). Bibliographies are handled by an external program called BibTeX, which reads the .aux file to find the references (thus you need to run LATEX first), then pulls the data from the .bib files you specify in the calling command in your .tex file and creates a .bbl file. The .bbl file contains all the info formatted how the bibliography should look. LATEX reads this in the first pass and copies it over to the .aux file and resolves the links to the text references. The next run of LATEXreads all this in and places both the bibliography and the cross references. This means that to get a bibliography in you must run LATEX BibTeX, LATEX then LATEX once more. You only need to do this if you add new reference, which in the labs will be once, provided you don't delete those intermediary files.

1.3.3 Getting Started with LaTex

Now that you have some background knowledge, we need to learn how to build a document on your computer. There are many ways to do this, including text editors and command line tools. I prefer using a more user-friendly editing and buidling environment. While there are numerous options available, I choose to use TexStudio. It is installed on all ECS computers, and it is available for

free download at home. I would recommend opening LabN.tex and building it before making any changes. Then make some changes, rebuild, and view those changes in the PDF that is generated. Steps to build a document in TexStudio are:

1. Open TexStudio on your lab computer

2. Use the menus open LabN.tex

3. Click on the double green arrow icon near the top. If you hover over it, it says "Build & View".

4. This should produce a PDF document on the right side of the TexStudio window

This document should build properly as long as you don't modify it. Once you start editing, it is possible that you will get compile errors. These errors are listed in the bottom pane of TexStudio. Like many compilers, they are sometimes cryptic and don't lead you directly to the problem. The most common problem (by far) is using an underscore with using the escape character (backslash) first. For example, look at the fetch1.tex file to see how I made this example_of_how_to_use_underscores.

Note that all code and image references are relative to where the .tex document is located in the file system. It is important that you maintain the same file structure that I gave you so that these references are simple and consistent.

1.4   Your Assignment

You are to:

1. Finish the testbench in Listing 1.2 by testing several additional cases. For instance, what happens when D is set at different points during the clock cycle, or if D is set for longer than a single clock cycle. Also, reset is not currently being tested. Does reset work properly? Does the register work properly after reset has been cleared?

2. Run a simulation and generate a timing diagram.

3. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file.

4. Upload the pdf and all the Verilog files to Canvas.