Language Assignment #2: Verilog

Issued: Wednesday, February 12
Due: Wednesday, February 26

Purpose

This assignment allows you to program in a language introduced in lecture: Verilog.

Verilog was designed by Prabhu Goel, Phil Moorby, and Chi-Lai Huang in 1984. Its name comes from the words "verification" and "logic."

Why Verilog?

We study Verilog for two main reasons.

First, Verilog is a representative of the *declarative* paradigm of programming languages. In particular, it is a *dataflow* language. There aren't many dataflow languages. By far, most languages follow the *imperative* paradigm.

Second, Verilog is one of two programming languages (the other being VHDL), where a translated program can "become" semiconductor hardware. In other languages, the result of translating a source program is a target program, which can be executed by a fixed CPU. In Verilog, the result of translation can be written into an integrated circuit, effectively synthesizing a new CPU.

In lecture, I will demonstrate this hardware-programming process. However, this assignment simply asks you to use a (software) simulator, named `vvp`: the Verilog virtual processor.

Translator

In our lab, `onyx` is the home-directory file server for its nodes (e.g., `onyxnode01`). There is also a shared directory for "apps" at `/usr/local/apps`. Nodes share a translator for Verilog, named `iverilog`, which is installed below `/usr/local/apps`, which is a non-standard location.

Due to network constraints, `onyx` can be reached from the public Internet, but a node can only be reached from `onyx`. So, you can SSH and login to `onyx`, then SSH and login to a node.

An easy way to use `iverilog`, from a node, is to permanently add a line to the end of your `.bashrc` file. To do so, login to a random node, from `onyx`, by executing the script:

```
pub/bin/sshnode
```

Then, execute the script:

```
pub/bin/bashrc
```

Don't change your `$PATH`; just execute the script. Then, logout from the node and login to a node.

Documentation

Verilog lecture slides are at:

```
pub/slides/slides-verilog.pdf
```

Simulated Verilog is (poorly) demonstrated by:

```
pub/sum/verilog
```

Better Verilog examples, from lecture, are at:

```
pub/la2
```

As far as I know, Verilog is not described in any college-level programming-languages textbook. However, the Verilog book linked-to from the `pl.html` URL (see below), is an excellent resource for our dialect of the language.

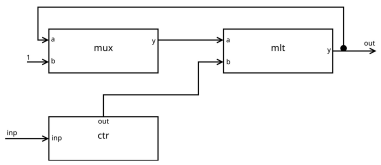
Links to programming-language documentation can be found at:

`http://csweb.boisestate.edu/~buff/pl.html`

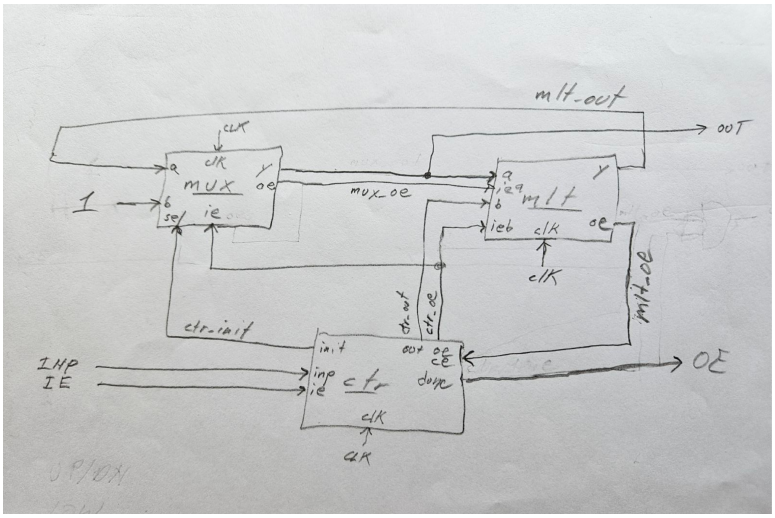
Assignment

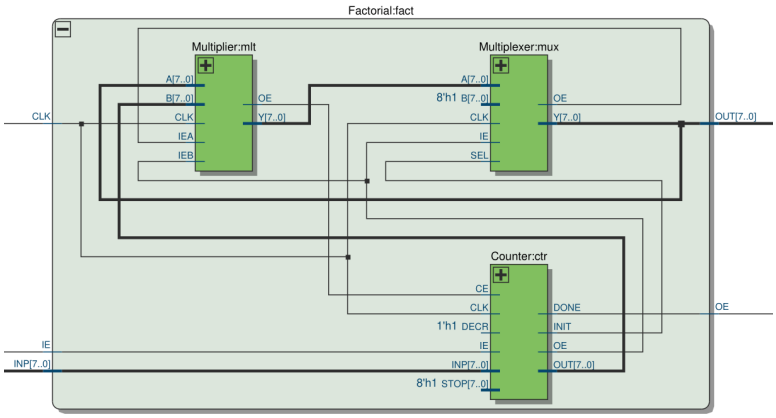
In lecture, we saw simple graphs describing dataflow programs for the slope-intercept and quadratic formulae.

We also saw a simple graph, and two detailed graphs, for a dataflow program that computes factorials. These three graphs are reproduced below. The simple graph only shows the main data flows:



The detailed graphs add control/status flows:





A Verilog implementation of this graph, from lecture is at:

```
pub/la2
```

A "testbench" and makefile is provided, to translate and simulate the implementation. I have also included a module named `Store`, which you may find to be useful.

Your assignment is to transform the factorial-computing program into a Verilog program that computes the n^{th} Fibonacci number. The Fibonacci numbers are the infinite sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \cdots$$

The next is the sum of the previous two. The 0^{th} Fibonacci number is zero.

Of course, you'll want to add numbers, rather than multiply them. Also, an instance of the `Store` module can store a previous number.

Hints and Advice

In lecture, I showed you the hand-drawn detailed graph for factorials, because you'll want to design your solution to this assignment in the same way. Draw a similar picture for your Fibonacci solution, rather than immediately trying to write Verilog source code for module interconnections. Include this picture with your submission.

In particular:

1. Draw an initial graph, using modules you have, and those you will need to create.
2. Then, *debug your graph*, by thinking through how control/status signals flow between output/input ports.
3. When you think your graph is bug-free, write the top-level module (e.g., `Fib.v`), and any new ones it needs (e.g., `Add.v`).
4. For each new module, write a testbench (e.g., `test.v`), which instantiates and exercises the module. Use `$monitor()` and `$display()` to debug it.
5. Finally, write a testbench (e.g., `test.v`), which instantiates and exercises your top-level module.