CS 207 Digital Logic - Spring 2019
Lab 1 - Hello Verilog

Monday, Feb. 18, 2019

1 Introduction

Field Programmable Gate Arrays (FPGAs) are this kind of "k" chips, that can be configured in order to create our own digital circuits. With FPGAs we can create actual hardware.

1.1 Background

Every digital circuit can be split in it's basic elements: logic gates that perform boolean operations with the bits and flip-flops to store the results. As a first approximation, we can think of an FPGA as a chip that has unconnected arrays of these elements. When you configure it, they connect to each other in an specific way and that's how we get our circuit.

This configuration is achieved by downloading to the FPGA a binary file, called a bitstream, which contains all the information necessary to establish the connections between the internal elements of the FPGA.

1.2 Bitstream Generation

The magic of FPGAs is in the software tools that allow one to generate the bitstream from the description of the circuit in an HDL language.

Circuits are designed using a hardware description language (HDL), such as Verilog or VHDL. These are the source files. The generation of the bitstream is done in two phases, from the sources:

Synthesis The synthesis tool infers the basic hardware elements from its description, and obtains a file netlist that describes the connections between them. This phase does not depend on the FPGA to use.

Place and route The components of the netlist are mapped to the physical elements of the FPGA, their placement is determined and the routing is performed. All FPGA configuration information is condensed into the bitstream. This phase does depend on the particular FPGA model that is being targetted.

Note on terminology
Although technically the synthesis phase in only part of the generation of the bitstream, colloquially when speaking of synthesis we usually refer to the complete process. Thus, if we say that "we have synthesized this circuit for the FPGA", we are referring to the fact that all phases have been performed: synthesis, place and routing, bitstream generation and loading into the FPGA.

1.3 Captive Audience

FPGAs have been around for 30 years. They are tremendously useful tools with a lot of potential. They allow you to design your own chip! However, FPGA manufacturers have never released either the software source or the specifications of their bitstreams' formats.

This has meant that no one can create software to work with FPGAs, but can only use that of the manufacturer. And it can only be used on computers that the manufacturer allows you. You can only design what the manufacturer has thought can be designed with their tools. If you can think of something new, not supported by your software, you will not be able to do anything. All this is quite frustrating. And in the end, many people have stopped using FPGAs.

2 Free tools for working with FPGAs

In May 2015 a historic milestone occurred: for the first time all the necessary tools were created to generate the bitstream from code in Verilog using only free software, thanks to the icestorm project, led by Clifford Wolf. From that moment, we already have tools that belong to humanities technological heritage to work with FPGAs, and to be able to develop hardware using only tools of this heritage.

2.1 Advantages of using free tools

- **Autonomy**: Hardware developers can develop their systems independently of the manufacturer. Design no longer depends on the whims of each manufacturer, or their tastes. With the free tools we become independent. Designers decide which operating system to use, or what environment to use. We are no longer obliged to do what the manufacturer tells us.
- **Access to knowledge**: These tools can be used normally, just like the closed source ones. However, if we are curious, we have access to the knowledge of how they are programmed, what algorithms are used, how the synthesis is implemented. This encourages the scientific spirit to understand how things work and then improve them. It is now possible for researchers around the world to analyze, understand and improve the algorithms. Before, only the manufacturers could do this.

- **New applications**: It opens the way to test new uses of the FPGA not foreseen by the manufacturers. From the beginning, there have been ideas to use FPGAs as hardware on demand, co-designing hw / hw, operating systems that use hw tasks, etc. Although many theses have been written about it, the actual implementations were very specific to a particular manufacturer with little reproducible by the community. Now it is viable to make implementations that run for example on a raspberry pi, and to synthesize the hardware on demand. With the closed tools it was impossible, because it was not part of the manufacturers plan.

- **Community Involvement** Now all can participate in the evolution of FPGAs, not only by using them, but by growing and improving the tools themselves.

- **Reconfigurable Free Hardware Repositories**: The time has come to "reinvent the free wheel". It is already possible to create repositories of free hardware designs that belong to us all and we can make them evolve over time. Share them. Improve them. These designs can be synthesized with free tools. And it is a knowledge that will last in time.

2.2 Limitations

No newborn tools have everything we want. But by being free, potentially any feature can be implemented. That's why all free software/ hardware systems evolve and mature over time. Einstein was also a baby, and at that age he could not create his theories. The important thing is the potential.

The project icestorm tools have just been born. And they have yet to mature and develop. Some limitations (at the time of writing) are:

- Only for Lattice FPGAs, models: HX1K-TQ144 and HX8K-CT256 (more have since been added).

- The tools only cover the low level: they are used in the command line. There is no graphical environment for managing projects. You have to do it based on makefiles.

- The tristate output support is still very limited.

- No post-routing time analysis support.

2.3 Free Simulation Tools

To design the circuits it is essential to have a Verilog simulator. The free tools we will use are:

- **Verilog Simulator**: Icarus Verilog `http://iverilog.icarus.com/`

- **Waveform Viewer**: GTKwave `http://gtkwave.sourceforge.net/`

Icarus Verilog creates an executable file from the Verilog code. When executed, the simulation is performed. The results are dumped into a .vcd file which is displayed with the GTKwave tool. This allows us to inspect the signals to verify their correct operation

2.4 Installation

2.4.1 Windows

Pablo Bleyer Kocik has created a Windows installer for Icarus Verilog and GTKwave at `http://bleyer.org/icarus/`. The latest installer is preferred (at the time of writing, `iverilog-10.1.1-x64_setup.exe`). After installing the software, the following PATH should be included in the system environment variables (replace `path-to-install-folder` with the actual folder to your `iverlog` installation):

- `path-to-install-folder\bin`
- `path-to-install-folder\gtkwave\bin`

2.4.2 Debian-based (e.g., Ubuntu)

```
sudo apt-get install iverilog gtkwave
```

2.4.3 Fedora

```
sudo dnf install iverilog gtkwave
```

3 Experiment A

Now we start to create a hardware, or at least simulate one. The simplest digital circuit is just a wire connected to a known logic level, "1", for example. This way, if you connect a LED to it, it will light up (1) or turn off (0).

3.1 `setbit.v`: Hardware description

In order to synthesize this circuit into an FPGA, we have to describe it first, using an HDL. In this course we use Verilog, given that we have all the free tools for its simulation/synthesis.

Nature of Verilog

Verilog is a language used for describing hardware. But beware, it is not a programming language. It's a description language. It allows us to describe the connections and the elements of a digital system.

The Verilog code for this "hello world" circuit implementation can be coded in a `setbit.v` file. It looks like this:

setbit.v

```
1  module setbit(output A);
2  wire A;
3
4  assign A = 1;
5
6  endmodule
```

3.2 Simulation with Testbenches

When we work with FPGAs we are actually making hardware and we always have to be very careful. We could write a design that contains a short circuit. And it could also happen that the tools don't warn us, (especially in these first versions, still in an alpha stage). If we upload that circuit into the FPGA we could break it.

Because of that, we always simulate the code that we write. Once we are sure enough that it works, (and it doesn't have a critical mistake) we upload it into the FPGA.

If we buy a chip and we want to test it, what do we do? Usually we solder it directly onto a PCB or we put it in a socket. But we can also plug it into a breadboard and place all the wires by ourselves.

In Verilog (and the rest of HDL languages) it's the same idea. You can't simulate a verilog component right away, you need to write a testbench that indicates which cables connect to which pins, which input signals to send the circuit, and check that the circuit outputs the right values. This testbench file is also written in Verilog.

How do we test the setbit component? It's a chip that has only one output pin, always high. In real life we would plug it onto a breadboard, we power it up, we would connect a cable to the output pin, and we would check it's high in voltage with a multimeter. We will do exactly the same, but with Verilog. We would get something like this:

setbit_tb.v

```
1  //-- Test bench module
2  module setbit_tb;
3
4  //-- Cable for connecting the output pin to setbit
5  //-- We could give it ANY name, but we call it A (like the
       ↪ setbit pin)
6  wire A;
7
8  //-- Place the component (it's actually called "instance")
       ↪ and
9  //-- connect the A cable to the A pin.
```

```
setbit SB1 (
    .A (A)
);

//-- Begin the test (Checking block)
initial begin

    //-- Define a file to dump all the data.
    $dumpfile("setbit_tb.vcd");

    //-- Dump all the data into that file when simulation
        ↪ finishes
    $dumpvars(0, setbit_tb);

    //-- After 10 time units, check if the cable is high.
    //-- In case of not being high, report the problem but
    //-- don't stop the simulation.
    # 10 if (A != 1)
            $display("---->ERROR! Output is not 1");
            else
            $display("Component works!");

    //-- End simulation 10 time units after checking
    # 10 $finish;
end
endmodule
```

3.3 Simulate!

We use Icarus Verilog and GTKwave tools for simulation. We execute the following commands one by one:

```
iverilog -o setbit_tb.out setbit.v setbit_tb.v # Both the
    ↪ design and testbench are compiled
vvp setbit_tb.out # The result is a simulation file, which
    ↪ can be executed by vvp provided by Icarus Verilog
gtkwave setbit_tb.vcd # Then we use GTKwave to inspect the
    ↪ waveform
```

3.4 Have a Try

Try to edit the code and output a constant `0`.

4 Experiment B

Now, instead of just one bit we will output four of them. It's a fixed value, wired "within the hardware". If we wanted to use the circuit for another value, we will need to synthesize again. We will name this component "fport" (Fixed

port). It has a 4-bit output bus, labeled as "data", wired to the binary value "1010".

4.1 `fport.v`: Hardware description

This circuit is similar to the one in the last tutorial, but instead of having just 1 output bits, it has 4 bits. It's described like this:

fport.v

```
1   module fport(output [3:0] data);
2
3   //-- Module output is a 4 wire bus.
4   wire [3:0] data;
5
6   //-- Output the value through that 4-bit bus.
7   assign data = 4'b1010; //-- 4'hA
8
9   endmodule
```

The output is now a 4 wire array. You can express that by writing `[3:0]` before the wire name. In order to assign a value to that wire, we write the value using Verilog's notation: First, the number of bits, then the ' character, after that, the base of the number (in this case, "b" for "binary") and finally, the 4 binary digits. This number could be expressed like a single hexadecimal digit, by writing `4'hA`. We could also write it in decimal as `4'd10`.

4.2 Simulation with Testbenches

This testbench is similar to the one in the last chapter. But now, instead of checking just one bit, we check the whole 4-bit bus. If it's not exacly as expected, it throws an error. The testbench code is:

fport_tb.v

```
1  module fport_tb;
2
3  //-- 4-wire bus, to connect it to the Fport component output
       ↪ .
4  wire [3:0] DATA;
5
6  //--Instantiating the component. Connect output to DATA.
7  fport FP1 (
8      .data (DATA)
9  );
10
11 //-- Begin the test
12 initial begin
13
14     //-- File in which store the results.
15     $dumpfile("fport_tb.vcd");
```

```
16      $dumpvars(0, fport_tb);
17
18      //-- After 10 time units we check
19      //-- whether the cable has the previously given pattern
            ↪ or not.
20      # 10 if (DATA != 4'b1010)
21              $display("---->ERROR!");
22          else
23              $display("Component works!");
24
25      //-- Finish the simulation 10 time units after that.
26      # 10 $finish;
27  end
28
29  endmodule
```

4.3 Simulate!

We execute the following commands one by one:

```
1  iverilog -o fport_tb.out fport.v fport_tb.v # Both the
       ↪ design and testbench are compiled
2  vvp fport_tb.out # The result is a simulation file, which
       ↪ can be executed by vvp provided by Icarus Verilog
3  gtkwave fport_tb.vcd # Then we use GTKwave to inspect the
       ↪ waveform
```

4.4 Have a Try

Try to edit the code and output a constant (decimal) 12 with octal number with a 5-bit bus instead of 4.

5 Experiment C

Let's model our first binary logic: an inverter (NOT Gate).

5.1 `inv.v`: Hardware description

Not logic circuits operate on the input bits, and then output the result. They don't store bits, they just modify them. The simplest of them all is the NOT gate, which has an input bit and an output bit. The output is always the inverse of the input. "0" turns into "1", and "1" turns into "0":

inv.v

```
1  //-- Component has an input (A) and an output (B)
2  module inv(input A, output B);
3
```

```
//-- Both the input and the output are "wires"
wire A;
wire B;

//-- Use a primitive gate of Verilog, NOT gate
//-- Such gates start from the output to all inputs
not not_gate_1(B, A);

endmodule
```

This time we use the gate-level modeling technique, which is virtually the lowest-level of abstraction. In general, gate-level modeling is used for implementing lowest level modules in a design like, full-adder, multiplexers, etc. Verilog HDL has gate primitives for all basic gates. Gate primitives are predefined in Verilog, which are ready to use. They are instantiated like modules. There are two classes of gate primitives: Multiple input gate primitives and Single input gate primitives. Multiple input gate primitives include and, nand, or, nor, xor, and xnor. These can have multiple inputs and a single output. They are instantiated as follows:

```
// Two input AND gate.
and and_1 (out, in0, in1);
// Three input NAND gate.
nand nand_1 (out, in0, in1, in2);
// Two input OR gate.
or or_1 (out, in0, in1);
// Four input NOR gate.
nor nor_1 (out, in0, in1, in2, in3);
// Five input XOR gate.
xor xor_1 (out, in0, in1, in2, in3, in4);
// Two input XNOR gate.
xnor and_1 (out, in0, in1);
```

Single input gate primitives include not, buf, notif1, bufif1, notif0, and bufif0. These have a single input and one or more outputs. Gate primitives notif1, bufif1, notif0, and bufif0 have a control signal. The gates propagate if only control signal is asserted, else the output will be high impedance state (z). They are instantiated as follows:

```
// Inverting gate.
not not_1 (out, in);
// Two output buffer gate.
buf buf_1 (out0, out1, in);
// Single output Inverting gate with active-high control
    ↪ signal.
notif1 notif1_1 (out, in, ctrl);
// Double output buffer gate with active-high control signal
    ↪ .
bufif1 bufif1_1 (out0, out1, in, ctrl);
// Single output Inverting gate with active-low control
    ↪ signal.
```

```
notif0 notif0_1 (out, in, ctrl);
// Single output buffer gate with active-low control signal.
bufif0 bufif1_0 (out, in, ctrl);
```

This experiment can also be implemented with the previous data-flow model. Just use the following:

```
//-- Assign the inverse of the input, to the output
assign B = ~A;
```

We assign the negated input A, to the output B. We use the "~" operator in front of the A to invert the signal (this is the same operator as in the C programming language).

5.2 Simulation with Testbenches

In the testbench, we must instantiate the inverter, and connect the `dout` wire to the B output.

We'll input various values, and check what comes out from the output. To do so, we connect the register named `din`. Unlike cables, registers work as a variable, to which we can assign different values. First we'll enter a "0", and then check that the output is a "1" (the negative). Then we do the other case: we enter a "1" and the check if we get a "0".

In the testbench, we instantiate the inverter, connecting it's input A to the register `din`, and it's input B to the `dout` cable. From the main loop, we assign values to `din`, and check `dout`.

inv_tb.v

```
module inv_tb();

//-- 1-bit register connected to the inverter's input
reg din;

//-- Wire connected to the inverter's output
wire dout;

//-- Instantiate the inverter, connecting din to the input A
    ↪ , and dout to the output B
inv NOT1 (
    .A (din),
    .B (dout)
);

//-- Begin test
initial begin

    //-- File to store the results
    $dumpfile("inv_tb.vcd");
    $dumpvars(0, inv_tb);

```

```
22      //-- We put the input of the inverter to 0
23      #5 din = 0;
24
25      //-- After 5 time units, we check the output.
26      # 5 if (dout != 1)
27          $display("---->ERROR! Expected 1. Got: %d", dout);
28
29      //-- After another 5 time units, we change the input
30      # 5 din = 1;
31
32      //-- After 5 time units more, we check whether there is
            ↪ a 0 coming out the output
33      # 5 if (dout != 0)
34          $display("---> ERROR! Expected: 0. Got: %d", dout);
35
36      # 5 $display("Simulation finished!");
37      # 10 $finish;
38  end
39  endmodule
```

5.3 Simulate!

We execute the following commands one by one:

```
1  iverilog -o inv_tb.out inv.v inv_tb.v # Both the design and
       ↪ testbench are compiled
2  vvp inv_tb.out # The result is a simulation file, which can
       ↪ be executed by vvp provided by Icarus Verilog
3  gtkwave inv_tb.vcd # Then we use GTKwave to inspect the
       ↪ waveform
```

In the simulation we can see that `dout` is always the inverse of `din`. In the first 5 time units, `din` and `dout` have the X value (red). This means that its value is undefined. That happens because we didn't initialize `din` with any value until the 5th time unit. Before that, it had an undetermined value (X) and therefore the output is undetermined too. After the 5th time unit, `din` is 0, and hence `dout` is 1.

5.4 Have a Try

Make a component that has AND, OR, and NOT gates, which take two 1-bit inputs and develop the corresponding binary output as a 3-bit value. Test all different combinations of inputs.

Assignment

Save the source code in `gates.v` and the testbench in `gates_tb.v`. Save the waveform shown in GTKwave as `gates_tb.jpg` (or other common picture formats). Assignment 1 requires the source and waveform files.