Lab assignment 2 / Week 2

To be done during week 18/02/19-22/02/19.

Preparation for the lab

A couple of good references for verilog are http://www.asic-world.com/verilog/index.html and https://www.nandland.com/verilog/tutorials/index.html. You can/should consult them.

The Icoboard that I have shown you earlier is documented at http://icoboard.org/. The tools and examples are contained in https://github.com/cliffordwolf/icotools. Visit http://www.clifford.at/icestorm/ for the toolchain for the iCE40 FPGA. The raspbian image you have installed on the SD card already has these tools.

As mentioned in class, we will use yosys (http://www.clifford.at/yosys/) for synthesis from verilog code that you will write. Yosys (and "abc", which it uses internally) generate a circuit in terms of LUT4s and flipflops available on the FPGA chip used on the icoboard. It is then placed and routed using arachne-pnr (https://github.com/YosysHQ/arachne-pnr). The output generated by arachne-pnr is converted to a bit stream, which is used to program the FPGA SRAM bits. The generation and analysis of bitstream is taken care of by icestorm set of tools. The programming of FPGA using this bitstream is done through the icotools. Note that all these tools except icotools can be used for some other boards/FPGAs. You should go through the appnote http://www.clifford.at/yosys/files/yosys_appnote_011_design_investigation.pdf to see how an interactive session can be run with yosys.

You can update your repositories for this week's folder by typing

```
git pull origin master
```

When you are located inside the folder "intro_hdl". (or start afresh with git clone https://www.bitbucket.org/shishirk/intro-hdl).
You will find a folder named "synthesis" inside "intro-hdl" when either of these commands are successful.

Task 1.

A. Go to folder named "blif" and read through file named carry.blif, bram.blif, gb_io.blif etc. .blif format is described in https://www.cse.iitb.ac.in/~supratik/courses/cs226/spr16/blif.pdf. These files represent circuit to be placed and routed on the FPGA. To do that for carry.blif:

```
arachne-pnr -d 8k carry.blif -o carry.asc
```

As mentioned in class this will generate a file called carry.asc which is the description of which LUTs/DFF and wires the circuit will be using on iCE40 HX8k chip.
B. Open the website: https://knielsen.github.io/ice40_viewer/ice40_viewer.html. Upload carry.asc file generated in last step to see a visual representation of which circuits are being used in FPGA. You can double-click on the list given on the right hand side to zoom to the part

of circuit corresponding to that set of wires. Make sure to find the 3 LUT4s and 2 carry blocks used in the carry.blif file. To see what is the fastest this circuit can run:

```
icetime carry.asc
```

C. Now write a .blif file to make a 2 bit multiplier. You are given two inputs which are two bits wide (say a and b) and you have to produce a four bit output which is multiplication of the input. Generate .asc file and run the timing check on this design.
D. Now we will convert a verilog file to .blif. Go through fa_behav.v, fa_struct.v and fa_dataflow.v to look at implementations for full adder in different styles. Do you think the final implementation of these styles will differ? Use:

```
yosys -q -p “synth_ice40 -blif fa_behav.blif” fa_behav.v
arachne-pnr -d 8k -o fa_behav.asc fa_behav.blif
```

to analyze the circuit generated from fa_behav.v (you can look at .blif itself, it is not complicated).
Do the same with fa_struct.v and fa_dataflow.v. Can you explain how possibly yosys can reach at the differences and/or similarities in output?
Now change one of the “nand” gates used in fa_struct.v to “and” gate and comment on the generated .blif file.

Task 2.
This task gives you more familiarity with yosys. It’s manual is located at http://www.clifford.at/yosys/files/yosys_manual.pdf. Go to folder named “yosys” in the folder “synthesis”.
A. Look inside the file named “counter.ys” which processes the design named “counter.v” (a 2 bit counter is described in that file). Run it using:

```
yosys counter.ys
```

It will generate a file “synth.v” which uses the internal library of yosys to represent the final logical network. Add following line after read_verilog line in counter.ys and run yosys again:

```
synth_ice40 -blif counter.blif
```

Look at counter.blif.
B. You can look try to run and change other file pairs ex1.v / ex1.ys to see how yosys can be used to run yosys interactively. The appnote I have mentioned earlier should be consulted for this part.

Task 3.
In this task we are going to see how the above tools come together to make actual designs work on FPGA. Go to the folder named “icoboard” in the folder “synthesis”.
A. We will use a Makefile to automate the execution of tools in the tool chain. I have described the program “make” in the classroom, a brief tutorial is found here: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/.

Look inside the Makefile given. At line 22, a series of targets and their dependencies/rules are described. Try to make sense of it.
To run this make file, type in the terminal:

```
make p=blinky bin
```

This sets the variable “p” (stands for project) to “blinky” (look at line 22 again) and runs the target named “bin”. It will generate a FPGA bitstream “blinky.bin” corresponding to design described in “blinky.v”. The latter turns on-off 3 LEDs on the FPGA board in a gradual manner.
You can clean the generated files (if you don’t need them):

```
make p=blinky clean
```

B. [This part is not yet covered in class, you can skip it at present if you want]. Use of verilator (https://www.veripool.org/wiki/verilator) as an alternative simulation engine to iverilog. You can install verilator by:

```
sudo apt install verilator
```

Run the sample simulation

```
make p=direct sim
./direct
```

Here the project “direct” corresponds to design in “direct.v”, which is just wire connecting input to output.
Verilator directly constructs an executable for the simulation, so it is ~10X faster that iverilog’s approach. This is necessary for big designs, which take a lot of time to simulate.
Look inside the file named “direct.cpp” to see how verilator is used.