EECS 627
Lab Assignment 1

1. Introduction

In this lab assignment, you will begin the process of designing a synthesized chip. The purpose of this lab is to familiarize you with the first few steps in an example VLSI design flow, including project organization, building a verification environment, Verilog coding and synthesis with *Synospys Design Compiler*. The next two labs will continue this process, walking you through floor-planning and partitioning a small design by doing physical design (place and route) with *Cadence Encounter*, back-annotation (timing closure), and final verification DRC/LVS checks with *Mentor Graphics Calibre* in Cadence's *Virtuoso* layout tool.

Note that a significant portion of the effort involved in working through these labs involves you independently reading the CAD tool manuals, along with basic troubleshooting. Remember CAD programs are tools and not intelligent. It is tempting to think of a CAD flow as a black box that takes Verilog as input and outputs layout at the push of a button, but flows are very temperamental and require many custom scripts. Automated place and route flows may require as much effort to use as creating custom layout, except it is much easier to rerun a flow if your design changes than it is to modify custom layout. All projects require modification of the flow so reading and understanding the commands used in these labs will save you time when you use them for your project. Note that there is no one "right" flow to use and the commands given in these scripts are one basic example of a flow but many others exist with different advantages and drawbacks. Additionally, CAD tools include many more commands and features not covered in these labs that your project may depend on, so you are highly encouraged to skim through CAD documentation even if you do not initially need or understand all features.

2. Assignment

The goal for this lab is to design an 8-bit multiplier that computes the product in four pipeline stages. The ports *a* and *b* are 8-bit inputs and *result* is the 16-bit product $a \times b$. *clk* and *reset* are the clock and active-high synchronous reset, respectively. You are expected to write a behavioral Verilog description of the design, verify its functionality and synthesize it in *Design Compiler*. When synthesizing the multiplier, you need to use the **retiming** feature of *Design Complier*, which is used to move combinational gates across flip-flop boundaries to create a balanced 4-stage pipeline multiplier.

3. Tool setup

In this lab you will use *Synopsys Design Compiler* (synthesis), *Synopsys VCS* (Verilog simulation), and *Cadence SimVision* (waveform viewing). To setup your shell environment for the correct tool versions, run the following command each time you login to a new machine or start a new shell session (alternatively, add it to the bottom of your ~/.bashrc file to automatically load):

```
module load eecs627/w16
```

Also note that starting in 2014 CAEN requires you to be on-campus to access most of the tools. Remote access is possible through two-factor authentication. Please contact CAEN for more information if you receive a "permission denied" when running the tools even after running eecstokens. At the end of this lab, you will deliver a series of scripts that can **simulate a behavioral Verilog netlist, synthesize the same netlist, and then simulate the synthesized design**. When I grade

this lab, I should be able to run this sequence using a `Makefile`. A sample `Makefile` has been included along with a set of other scripts in a file located at:

`/afs/umich.edu/class/eecs627/w16/lab_resource/lab1/lab1.tgz`.

The required targets for the `Makefile` are described at the end of this document. The tarball can be expanded with the command: `tar -zxvf lab1.tgz`

4. Reference Standard (a.k.a. Golden Brick)

The first step in the design process is to create a reference standard that you can use to check your design. The reference standard should be in the form of a computer program that simulates the behavior of the chip component (block) that implements the specification. Furthermore, the actual coding of the reference standard should be structured in such a way that a person can verify by inspection that it does meet the specification. A C-program for the pipelined multiplier has been provided. The provided code in `goldenbrick/goldenbrick.c` can be compiled and run using the following two commands:

```
gcc -Wall -ggdb -o goldenbrick goldenbrick.c
goldenbrick > goldenbrick.txt
```

or by using the Makefile with `make c_compile`. The output of this program (in goldenbrick.txt file) will be in the following format:

```
A = 00000001      B = 00000001      RESULT = 00000000
A = 00000001      B = 00000010      RESULT = 00000000
A = 00000001      B = 00000011      RESULT = 00000000
A = 00000001      B = 00000100      RESULT = 00000000
A = 00000001      B = 00000101      RESULT = 00000001
A = 00000001      B = 00000110      RESULT = 00000010
A = 00000001      B = 00000111      RESULT = 00000011
A = 00000001      B = 00001000      RESULT = 00000100
A = 00000001      B = 00001001      RESULT = 00000101
A = 00000001      B = 00001010      RESULT = 00000110
A = 00000010      B = 00000001      RESULT = 00000111
A = 00000010      B = 00000010      RESULT = 00001000
```

where A and B are randomly generated 8-bit numbers and RESULT is the product of unsigned multiplication of the two. Note that the first result comes out after four cycles because you are designing a 4-stage pipelined multiplier.

5. Behavioral Model

After verifying that the reference standard program matches our specification, it is time to think about an HDL behavioral implementation of the computational core component. Your module definition should be as follows:

```
module mult(clk, reset, a, b, result);
      input   [7:0] a, b;
      input         clk, reset;
      output [15:0] result;
      /* … */
endmodule
```

For those of you who have not worked much with Verilog, there are some excellent Verilog resources at http://www.asic-world.com/verilog/index.html. Your task is to create a behavior model for the multiplier in a file `mult.v`. Remember to include the four pipeline stage registers in your behavioral model. Since this is only a behavioral model, it is not necessary to evenly distribute the pipeline registers within the logic because that is what the retiming feature of design compiler is for.

6. Simulation of Behavioral design

The testbench is a piece of Verilog code which is not part of the design, but instead is used to instrument the design and verify its functional correctness. The design (called the Device Under Test, or DUT) is instantiated in the testbench along with some procedural code. For those of you unfamiliar with the subset of Verilog used in testbenches, please refer to the tutorial references in the last section. To verify a simple gate, it is often sufficient to sweep the inputs and observe the outputs. However, in this lab, we will explore the use of random input generation. We will use the randomly generated output of the provided golden brick code to drive a Verilog simulation in VCS. We will then compare the outputs of the Verilog simulation and the golden brick simulation to verify functionality.

Your task is to create a testbench that can take input generated by the provided golden brick simulator. You can assume that the randomly generated input/output combinations are in a file called `goldenbrick.txt` from Section 4 above. Your simulator must create a file called `testbench.txt` containing the same series of multiplications in the exact same format as `goldenbrick.txt` for easy comparison. You must automate the process of feeding the input test vectors (A and B) from the golden brick output into your Verilog testbench. This can be done in a number of ways. For example, a Perl script could be used to parse the output and generate a new testbench that toggles the inputs appropriately. Alternatively, the golden brick output could be fed directly into a static Verilog testbench using Verilog's built-in file I/O functions (e.g. $fopen). Once your testbench is modified to read the randomly generated inputs, you must be able dump the input/output pairs generated by the Verilog simulation to a file. This can be done using an `always` block as shown below:

```
integer handle;

handle=$fopen("testbench.txt");
always @(posedge clk)
begin
  $fdisplay(handle, "A = %b\t B = %b\t RESULT = %b", a, b, result);
end
```

There are several tools that you can use to perform Verilog simulations: *Synopsys VCS*, *Cadence NC-Verilog*, *Mentor Graphics ModelSim*. In this lab, we will use *Synopsys VCS*, which can be called using the following commands:

```
vcs +v2k +lint=all,noVCDE +warn=all -sverilog <Verilog files …>
./simv
```

The first command parses the Verilog files given and creates an executable called "simv", which can then be run to run the simulation. `+v2k` enables Verilog 1364-2000 syntax. `-sverilog` enables SystemVerilog extensions. `+warn=all` turns on all parser warnings. `+lint=all,noVCDE` enables 'linting' to warn you of potential problems and ambiguous portions in your Verilog code.

To debug your testbench and multiplier, you can dump waveforms from Verilog by adding the following statements to an `initial` block in your testbench:

```
$dumpfile( "filename.dump" );
$dumpvars( 0, mult_testbench );
```

After the simulation is complete you should have a `filename.dump` file. To view the simulation data, open the waveform viewer *Cadence SimVision* using the command `simvision` and load the dump file. *SimVision* will prompt to convert the dump file into its native waveform format, which you should accept.

7. Naming Conventions

You must document and rigorously adhere to a standard convention for naming your various design files, source code, scripts, Makefile, directory structure, variable names, signal names, etc... It does not matter what convention you decide to adopt, but it must be clear, consistent, used uniformly throughout your work. The structure you develop here will aid you in completing your final project. For example, you may end up with several different types of files containing Verilog code. The first type is source Verilog which you partially or completely edit by hand (for example, behavioral Verilog code). Next, you may write scripts to modify these netlists, generating intermediate Verilog netlists for your design. Finally, you will run various tools that modify the design, reading and writing Verilog code, such as synthesis, post-placement, and post-routing netlists. Because you will certainly need to iterate through your design flow at least several times, as you fix bugs and/or add features, it is imperative that you have scripts or a Makefile to automate running all of the commands needed to produce your final circuit netlist, as running the flow by hand will quickly become too tedious.

Lastly, some people prefer to have a single Verilog module per source file, due to some restrictions in the synthesis tools, while others prefer each Verilog module to implement only a single pipe stage, or to not allow combinational logic to span module boundaries.

8. Synthesis

The synthesis process is controlled by a script file that the Synopsys tool `dc_shell` reads. The newest version of `dc_shell` uses the TCL scripting language, as do most other CAD tools you will use in EECS 627 and in industry.

If you have any file called `.synopsys dc.setup` in your home directory from some previous class, please rename/remove it so it doesn't interfere with your 627 labs and project. For documentation, check in CAEN's Synopsys documentation directory (/usr/caen/synopsys-docs-2015.06/dc especially `dcug.pdf`, `synqr.pdf`, `dcrrt.pdf`, and `home.pdf`). You should, at the very least, look up each command in the above synthesis scripts. You can also look up commands within the tool itself by running `dc_shell` and then typing "man command_name".

To run `dc_shell` you must invoke the TCL mode of the tool. It is also very helpful to store the verbose output of the program into a file.

```
dc_shell -f <yourscript.tcl> | tee <log_file>
```

Synopsys can perform an operation called "retiming". This transformation can relocate state elements so that the delay of each pipeline stage is balanced. While the algorithm used by Synopsys is more limited than those in the literature, it can still handle many common cases fairly well, such as pushing four pipe stages into a combinational multiplier.

A set of simple synthesis scripts has been provided in `lab1.tgz`. The scripts will do synthesis but do not perform retiming of the registers. The multiplication is therefore performed before the first

register bank. Look in the Synopsys documentation for the retiming command. Also be sure to look up each of the commands in the various .tcl files. You will need to be familiar with these as you move forward on your projects this semester.

You will need to modify the synthesis script to meet timing. Do not change the clock period of the design in order to accomplish this. Some features to look into are the `retiming` and `compile_ultra`. These can be found in the documentation. Note that the provided synthesis script generates the following report:

- area: contains information about the sequential and combinational area of the design
- constraints: shows all paths which violate any of the specified timing or fan-out constraints. Ignore the clock net's slack violation in this file.
- power: contains information about dynamic and leakage power
- fullpaths: contains timing of the *nworst* or *max_paths* specified in `mult.syn.rpt` file

9. Simulation of Structural design

To simulate the structural version of the design, you need to add the Verilog models from the Artisan library of standard cells. The file is at:

```
/afs/umich.edu/class/eecs627/ibm13/artisan/2005q3v1/aci/sc-x/verilog/ibm13_neg.v
```

This is already included in the Makefile. You also need to include the standard delay format or sdf file (generated as *.syn.sdf by the provided script) in your Verilog testbench file by adding the following line for every synthesized module that is a part of your design.

```
initial $sdf_annotate("myfile.sdf",<module instance name>);
```

In the past the sdf annotation was done inside the structural Verilog file, but this method avoids the hassle of adding the line to the verilog file every time a new version is created.

You need to verify the functionality of your synthesized design (`mult.syn.v`) and use the simulation testbench and scripts created in Section 6. Ensure that the simulation output matches that of the golden brick.

10. Deliverables

After working through this lab you should submit the items listed below in an organized directory tree. Send the directory as a tar file called <uniquename>_lab1.tgz (tar -zcvf <uniqname>_lab1.tgz <your_lab1_directory>) onto *Canvas* by **11:59pm on Friday, January 15th.** Grading this lab will be done by first running `goldenbrick.c` to generate a set of inputs and then feeding these inputs to the synthesized multiplier in a Verilog simulation. In your .tgz file you should include:

- `README.txt`: text file containing a brief description of your design, the directory and file structure for your submission, and documentation of any problems you encountered.

- `Makefile`: The makefile should have the following targets:
  - clean: Removes unwanted files from the directory tree
  - c_compile: Compiles and generates result of the golden brick
  - sim: Parses the output of the golden brick simulation, simulates the behavioral verilog code and compares the result to the golden brick output

  - synth: Synthesizes the design
  - sim_synth: Parses the output of the golden brick simulation, simulates the structural verilog after synthesis and compares the result to the golden brick output
- Source files: all of the code needed to build and simulate your lab assignment including the following
  - `goldenbrick.c`: Reference standard program
  - `mult.v`: Behavioral level code implementing the pipelined multiplier
  - `mult_testbench.v`: Verilog testbench
- Any Perl or other scripting files used for post-processing
- `mult.syn.tcl`, `common.syn.tcl`, `namingrules.syn.tcl`: Synthesis scripts
- Synthesis reports:
  - `mult.syn.rpt`

Please use the same directory structure as given in `lab1.tgz`.

11. For further reference

- EECS ECAD web page (http://www.eecs.umich.edu/dco/docs/ecad/)
- Perl scripting language (http://www.perl.org/, http://www.cpan.org/)
- VLSI CAD documentation (generally in `/usr/caen` directories for CAD tools)
- `man` command within the tools