EECS 627, Lab Assignment 1

1 Introduction

In this lab assignment, you will begin the process of designing a trivial chip. The purpose of this lab assignment is for you to familiarize yourself with the first few steps in the standard VLSI design flow, including project organization, building a verification environment, Verilog coding and synthesis with *Design Compiler*. The next two labs will continue this process, walking you through floor-planning and partitioning a trivial design for design automation, physical design (place and route) with *Silicon Ensemble*, back-annotation/timing closure and final DRC/LVS checks with *Calibre* in *ICFB* 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 tools are like any other tool in that they help you reach your goal (designing a VLSI project), but may take considerable skill and effort to apply.

It is important that while doing these lab assignments, you make a concentrated effort to think about how you are organizing your work (and later, how you will organize your project), controlling for bugs, and increasing your productivity, both as an individual and as a group member. Please take this opportunity to play with the CAD tools and figure out how best to organize yourself and your work, your personal computing environment, the capabilities and limitations of the tools available, file formats and scripting languages the tools support. Also, if you are unfamiliar with an HDL language, such as Verilog HDL, we strongly suggest you spend some extra time writing several small example designs to familiarize yourself with the language. Also, now would be a good time to learn about one of the several graphical verilog debugging tools available on CAEN (modelsim, signalscan, etc).

2 Specification

The chip for this lab consists of an 8-bit multiplier that computes the result in **four** pipeline stages. *a* and *b* are the 8-bit inputs and *result* is the 16-bit product. *CLK* and *RESET* are the clock and active-high synchronous reset, respectively. You are expected to write a Verilog description of the design, verify its functionality and synthesize it in *Design Compiler* for the highest frequency possible.

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

The first step in designing our chip is a reference standard that we can use to check our designs against. The reference standard should be in the form of a computer program that simulates the behavior of the chip component that implements the specification. Furthermore, the actual coding of the reference standard should be structured and in a high enough level language that a human can verify by inspection that it does in fact meet the specification. A C code for the pipelined multiplier has been provided. You can use it as a starting point for your own implementation of the reference standard, and by all means, use a different programming language if desired.

The provided code (GoldenBrick.c) can be compiled and run using the following commands:

```
gcc -Wall -ggdb -o GoldenBrick GoldenBrick.c
GoldenBrick > GoldenResult
```

The output of this program (in GoldenResult file) will be in the following format:

```
a =   1, b =   1, result =     0
a =   1, b =   2, result =     0
a =   1, b =   3, result =     0
a =   1, b =   4, result =     0
a =   1, b =   5, result =     1
a =   1, b =   6, result =     2
a =   1, b =   7, result =     3
a =   1, b =   8, result =     4
a =   1, b =   9, result =     5
a =   1, b =  10, result =     6
a =   2, b =   1, result =     7
a =   2, b =   2, result =     8
```

where *a* and *b* are randomly generated 8-bit numbers and *result* is the product of unsigned multiplication of the two.

4 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
```

Remember to include the 4 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. 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 4 pipe stages into a combinational multiplier.

5 Testbench

The testbench is a piece of verilog code which is not part of the design, but rather instantiates a portion of the design (called the circuit under test, or CUT) along with some procedural code to test it. For the purposes of this lab assignment, it is sufficient to iterate each variable from 1 to 10, printing the output for each of the 100 iterations. In other situations, where the goldenbrick randomly generates input values (see goldenbrick_random.c), it might be convenient to parse the numbers *a* and *b* from the output of golden brick and form a testbench for the multiplier. The Perl scripting language is quite useful when building such a parser. Set up

your verilog test bench to output the values *a*, *b* and *result* in the same format as the golden brick's so that the results can be easily *diff*'d to verify the correctness of the verilog code. A sample *always* block for displaying the values of signals *a*, *b* and *result* in binary format is:

```
always @(negedge CLK)
begin
      $display(" a = %b",a);
      $display(" b = %b",b);
      $display(" result = %b",result);
end
```

Note that the golden brick code does not have a *RESET* signal. You can either add the *RESET* functionality in the golden brick or match the result of your behavioral code after the multiplier flip-flops have been reset.

6 Naming Conventions

You must document and rigorously adhere to a standard convention for naming your various design files, source code, scripts, makefiles, directory structure, variable names, signal names, etc. It doesn't matter what convention you decide to adopt, but it must be clear and consistent, and be 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. First there are those complete or partial files you edit by hand. Next you may write scripts that generate complete and/or partial pieces of verilog code. Finally, you may run various tools which read as input and write their output verilog code. Furthermore, as you will definitely need to iterate through your design flow several times as you fix bugs and/or add features, it is imperative that you have scripts or makefiles to automate running all of the commands needed to produce your final circuit netlist from a collection of input files, scripts, and tools. Also, some people prefer to have a single verilog module per source file, due to some restrictions in the synthesis tools. Other people prefer each verilog module to implement only a single pipe stage, or to not allow combinational logic to span module boundaries.

7 Synthesis

The synthesis process is controlled by a script file that the Synopsys tool **dc_shell** reads. To run synopsys type **dc_shell**. I usually pipe the results into a log file because synopsys is very verbose. For example:

```
dc_shell < yourscript.dc > yourscript.log
```

Make sure you have selected the newest synopsys synthesis package under *swselect*. 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 project. I recommend that you **do not use** the .synopsys dc.setup as this type of file is difficult to synchronize when you are doing a group project. For documentation, change your current directory to the *synopsys* install directory at **/usr/caen/synopsys-synth-2003.12/** and take a quick look at the documentation. You should, at the very least, look up each command in the above synthesis scripts. Synpopsys instantiates standard cells from the Artisan library. To simulate this file, you need to add the verilog models from the Artisan library. The file is at **/usr/caen/generic/artisan/tsmc18/aci/sc/verilog/tsmc18.v.** You also need to include the standard delay format or sdf file (generated as *.dc.sdf by the

provided script) in your structural verilog file by adding the following line after the i/o port definition:

```
initial $sdf_annotate("myfile.sdf");
```

The provided synthesis script generates the following reports:

- .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.dc file
- .paths: contains summarized information about *nworst* or *max_paths* timing constraint violaters

8 Deliverables

After working through this tutorial you should have the following items to submit to your GSI in an organized directory tree, which you should submit as a tar file (see the command "tar"). Detailed directions on submitting lab assignments will be posted on the course web page.

- 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
  - diff: Compares the result of the behavioral code with that of the golden brick
  - syn: Synthesizes the design
  - all: Does all of the above in the above sequence

- source files: all of the code needed to build and simulate your lab assignment including the following
  - reference standard program
  - mult.v: behavioral level code implementing the pipelined multiplier
  - test_mult.v: testbench

- mult.dc: synthesis script

- reports:
  - mult.constraints
  - mult.power
  - mult.paths
  - mult.fullpaths
  - mult.area

9 For further reference

- EECS ECAD web page (*http://www.eecs.umich.edu/dco/ecad*)
- Perl text scripting language (*http://www.perl.org*, *http://www.cpan.perl.org*)
- VLSI CAD tools (most have online documentation, poke around in installation directory, given on the ECAD web page).For Synopsys tools' documentation, type *sold* in shell window on any CAEN workstation.
- swselect (CAEN) enable/disable tools and versions of tools
- IEEE Verilog standard (*http://www.eecs.umich.edu/courses/eecs627/w06/pertinent_lit.html*)