ECE/CS 6770 – Lab Assignment 2

Due 2 Feb 2017 via Canvas

1 Introduction

This lab assignment is the first of four labs that build on each other using the design of a relatively simple chip. The purpose of this assignment is to familiarize yourself with the first few steps in commercial VLSI design flows, including project organization, building a validation environment, behavioral HDL coding, and synthesis with *Design Compiler*. The next three labs will continue this process, taking you through floorplanning and partitioning this design for design automation, physical design (place and route) with *Silicon Encounter*, back annotation, Design Rule Checking (DRC) and Layout Versus Schematic (LVS) checks with *Caliber* in the *ICFB* layout tool, and finally timing closure and validation with PrimeTime.

Note that a significant portion of the effort involved in working through these labs requires you to independently *read the CAD tool manuals*, along with basic troubleshooting. Remember that CAD tools are like any other tool in that they help you reach your goal (designing a VLSI circuit), but may take considerable effort to initially learn and apply.

It is important while doing these lab assignments that you make a concentrated effort to think about how you are organizing your work, controlling and testing bugs, and increasing your productivity. Take this opportunity to familiarize yourself with the CAD tools and figure out how to best organize your work, your personal computing environment, the capabilities and limitations of the tools available, file formats and scripting languages that the tools support, etc. Also, if you are a little rusty with an HDL language, such as Verilog, you should spend some extra time writing several small example designs to familiarize yourself with the language. These labs also provide a good opportunity to learn about some of the several graphical Verilog debugging tools available (such as ModelSim, SignalScan, etc.).

2 Specification

In this lab you will build a pipelined function unit that executes the function $x^2 + 97y$. *The multiply must be a signed multiply, not the native unsigned multiply that you get with the multiply statement in HDL languages!* Your logic will compute the result in **four pipeline stages**. The design will take two 8-bit inputs `a` and `b`, and produce output `result` which is a 16-bit result. Your design will also take two control signals, `clk` and `reset`, which are the clock and active high **synchronous** reset respectively.

You are expected to write a Verilog or VHDL description of the design, validate its functionality, and synthesize it in Design Compiler, *targeting the highest frequency possible.* Indeed, the only real reason to pipeline this design is to achieve a higher frequency and throughput.

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

The first step in creating any design is to generate a reference standard that we use to validate the correctness of our designs. The reference standard should be in the form of a computer program that implements the specification and simulates the behavior of the chip component. Furthermore, the actual coding of the reference standard should be structured, and in a programming language that can be checked that it meets the intended specification. For this simple design we assume this can be done by a human inspecting the source code and its result.

Two programs written in C++ have been provided for the pipelined multiplier. These programs will need to be modified to create a signed multiplication, rather than the unsigned multiply currently designed. You can use this as a starting point for your own implementation of the reference standard. Please feel free to use a different programming language if desired, such as System-C. The C program named `goldenmodel.cpp` can be compiled and run using the following commands:

```
g++ -Wall -ggdb -O -o goldenmodel goldenmodel.cpp
goldenmodel > goldenresult
```

The output of this program is in the following format, where `a` and `b` are randomly generated 8-bit numbers and `result` is the signed result of the function. Note that the output is generated four cycles after the inputs due to the pipelining employed. The first four outputs will be zero if the circuit is properly reset.

```
a = 01101000, b = 10101010, result = 0000000000000000
a = 00111101, b = 10010100, result = 0000000000000000
a = 00100010, b = 01001100, result = 0000000000000000
a = 00010001, b = 11000000, result = 0000000000000000
a = 00000111, b = 10011101, result = 0000100110101010
a = 00100011, b = 10100011, result = 1110010110011101
a = 01001111, b = 11110111, result = 0010000101010000
a = 01111100, b = 10010111, result = 1110100011100001
a = 10011100, b = 01001010, result = 1101101010101110
a = 01101000, b = 00100100, result = 1110000110001100
a = 10111000, b = 00101000, result = 0001010011111000
a = 00111011, b = 01110111, result = 0001010001000111
a = 10000011, b = 00010000, result = 0100001100011010
a = 01011001, b = 00110111, result = 0011011111100100
```

4 Behavioral Model

After ensuring that the reference standard program matches our specification, the next step is to create an HDL behavioral model of the computational core of the component. Your top level module definition should match the structure of the specification, as follows:

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

Remember that you need to include the four pipeline registers in your behavioral model. Since this is only a behavioral model, it is not necessary to balance the logic in the pipeline stages. Synopsys' Design Compiler can perform an algorithm called *retiming* that will shuffle the logic and state elements so that the delay of each pipeline stage is balanced. While the algorithm used by Design Compiler has more limitations than some of the more recent research in this domain, it can still handle many common cases quite well, such as balancing the logic in the four pipeline stages of this simple chip.

5 Synthesis

5.1 Environment Setup

See lab 1 for setting up the search path in your shell to run the CAD tools from the CADE lab.

If you have a file `.synopsis_dc.setup` in your home directory from some previous class or project you *definitely* want to rename or remove it so that it doesn't interfere with this project. We recommend that you do not use global configuration files as they tend to change operations independently of any project setup and make things horribly difficult to debug if you forget about them. They also tend to be in files that are not normally visible when you list the files in a directory. Likewise, we do not recommend you use a `dc.setup` file as this approach is difficult to synchronize when you are doing a group project or maintain across different projects. Instead, for group projects, put all the initialization code in a group maintained file such as `common.dc` and ensure everyone in your project uses the same file.

5.2 Technology Files

Synthesis instantiates standard cells from the UofU, Arm's Artisan library, or the University of Washington's library. The mapping and performance are dependent on the technology and the library function units provided. We provide synthesis scripts that "include" either the UofU 500 nm library, a University of Washington or Artisan 130 nm library, or an Artisan 65 nm library. We suggest that you use the Artisan 65 nm libraries for this lab. You will need to have a signed NDA to have access to the 130 nm or 65 nm technology nodes.

5.3 Synthesis Engine

Synthesis is performed with Synopsys Design Compiler. Design compiler will always be run with a *script* to control its operations (unless you use the graphical front end called Design

Analyzer). If you write a stand-alone script, you can call design compiler directly with the command:
`syn-dc < yourscript.dc > yourscript.log`

You can read the documentation for design compiler by reading the pdf files in the following directory:
`/uusoc/facility/cad_common/Synopsys/SYN-S13/doc/`
The file dcug.pdf is the user guide, dvug.pdf is the Design Vision user guide, dcrrt.pdf talks about register retiming, dccli.pdf is the command line interface guide, and dcrmo.pdf is the optimization reference manual.

While testing and debugging, you may want to use *Design Analyzer*, which contains a graphical front end to design compiler. You run Design Vision with the command `syn-dv` in our environment.

I have provided two scripts that run design compiler. These are shell scripts which in turn run design compiler in two different modes. The first script is called mult.rtlopt.csh. This script uses Design Compiler to parse your Verilog design and ensure that it is well formed. It translates the design into a generic format that the synthesis and tech mapping algorithms employ. (You should look at the intermediate Verilog output of this tool – mult.rtlopt.v.) The second script, mult.dcopt.csh, performs the synthesis and tech mapping of the design.

Both the shell scripts use a control script with a .tcl extension – one being mult.rtlopt.tcl and the other mult.dcopt.tcl. This is where the technology files are identified, and where the scripted control for design compiler is specified. A third .tcl script, mult.cstr.tcl is necessary. This is a file that contains your *design constraints*. These constraints include timing constraints and design directives. For now the file largely just contains timing constraints such as the clock frequency and duty cycle. *Make sure you look through the entirety of all the .tcl scripts and understand all of the operations they perform.* This will be much easier if you have the Design Compiler manual to use as a script command reference as you walk through the tcl scripts. You should be able to use the script to generate a design, but you can improve the design by modifying the .tcl scripts. You certainly will need to at least modify the constraint file to optimize the performance of your design and to perform retiming.

The provided scripts generate 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 fanout constraints
- **.power**: Contains information about dynamic and leakage power.
- **.fullpaths**: Contains timing of the *nworst* or *max_paths* specified in the `mult.dcopt.tcl` file.
- **.paths**: Contains summarized information about *nworst* or *max_paths* timing constraint violators.

You will need to make sure that there are no errors in the logs (**.out** files) when you are done with the design, and that none of the constraints have been violated. If there are warnings, you need to be able to justify that they are not critical to the correctness of your design. You will lose points on the lab for errors or critical warnings unless they are justified.

6 Test Bench

The test bench is a piece of code that is not part of the design. This piece of code is typically written in Verilog, but can also be written in tcl code. The test bench controls the environment of a design module (called the *device under test*, or DUT) by driving the inputs and testing the outputs for correct operation. A test bench should never access internal signals of the DUT.

One common method of generating the test bench is to use the output from the reference model. This ensures that you are validating against your golden reference.

6.1 Validating against the Golden Model

For this lab, you will to parse the numbers `a` and `b` that are generated from the output of the golden model to create the test vectors for your synthesized logic design. Scripting languages, such as Perl or tcl, are commonly used for automating this task. Another approach is to have your test bench simulation create an output file of the input and output vectors in the same format as the golden model so that you can compare them directly against the reference standard program output. You can do this in Verilog with the following code snippet:

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

You can also do this with tcl, which is part of ModelSim's simulation interface, with the following snippet:

```
# show the results at this simulation point
set str "A = [exa A], B = [exa B], RESULT = [exa RESULT]"
puts $outfile $str
```

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

For this lab I have provided a testbench in tcl that you can use. You may use this test bench or create one in Verilog that has a similar behavior.

6.2 ModelSim

We will use ModelSim to perform simulations and validations of our designs in this course. One of the goals of this class is to teach you some of the best-of-breed tools. This lab will require that you use ModelSim to validate your design. The gui version is run with the command `modelsim`.

You can also run ModelSim in the background as part of a script. For ModelSim to simulate a circuit it must first compile the verilog design. The script call for the ModelSim

compiler is `vlog`. Once the design has been compiled it can be simulated. The script command for the ModelSim simulator is called `vsim`. A Makefile is included on canvas that includes scripting calls to run ModelSim in the Utah environment. Note that the make file contains calls to both vlog and vsim.

A script is required to run a simulation. For ModelSim, this is a script that uses tcl, the same language that design compiler uses. There are a number of custom simulation macros that can be called from the simulation script, such as `force` and `run`. For some strange reason ModelSim traditionally names its tcl scripts with a `.do` extension rather than .tcl. I have provided the file **mult.do** on canvas which is a sample ModelSim script for design validation using ModelSim and tcl.

You may want to refer to a tcl manual when writing your simulation scripts. A good source for learning tcl is <http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html>. The ModelSim test-bench uses tcl for input, output, and control flow, plus includes some extended commands to drive the signals (such as `force`).

6.3 Timing Simulation

If you simulate a design without timing information, most simulators will default to a *unit delay model*. In this mode each gate takes one time unit to perform the output action. In order to have a more accurate simulation, estimated or actual extracted delays must be employed.

Our synthesis scripts had Design Compiler generate estimated delays for the design based on the library gates, signal fanout, etc. These delays get saved out in an .sdf format file. The *Synopsys delay format*, or sdf file, can be imported into the ModelSim simulator to produce a more accurate simulation. This is included with the `-sdftyp` argument to `vsim`. The .sdf file passed is applied to the cell defined in the argument call. An example of the syntax is included in the Makefile.

One additional simulation parameter you might use when running ModelSim is to use the `-novopt` command. This will not optimize the signals in the simulation, which will allow you to observe any hierarchical node in your design.

6.4 Running ModelSim

We have had some problems in the past with environment variables in the makefile. Please contact the instructor or TA if you have this problem.

A quick tutorial is included to get you quickly up to speed on running ModelSim. For more detail read the user manuals. You will initially need to run ModelSim manually on your design to create the working directory and library files. To do this, run `modelsim` from a CADE machine.

- `File | New | Project`
  Enter name of project. This creates the "work" directory where all your compiled projects and simulations go. If you click on the library tab in ModelSim and expand the work directory, it will show all of the modules you have compiled.

- In the next pop-up window, add an existing file. This should be your `mult.dcopt.v` file.

You can now compile your multiplier by clicking on `Compile | All`. To simulate your design, you can click on `Simulate | Simulate` and then open up the work library, and select your multiplier module. (You can also do this by hand by typing `vsim work.mult` in the command window.) Note that you have lots of errors – the library modules are not found. You will need to compile a Verilog file that defines all the cells in the library. Once compiled, these will appear in the "work" directory, and the file will not need to be recompiled. For the Artisan 130 nm library, add the following file to the project,
`/uusoc/facility/cad_common/Artisan/IBM8RF-1.2v/aci/sc-x/verilog/ibm13svt.v`
For the Artisan 65 nm library, add the following file:
`/uusoc/facility/cad_common/Artisan/GF/cmos65g/aci/sc-adv12/verilog/cmos10sfrvt_a12_neg.v`
for the University of Washington 130 nm cells, add the following file:
`/uusoc/facility/cad_common/local/Cadence/lib/cg_lib13_v096/static184cg.func.v`
and for the University of Utah 500 nm library add the following file:
`/uusoc/facility/cad_common/local/Cadence/lib/UofU_Digital_v1_2/UofU_Digital_v1_2_behv.v`
These will define the behavior of all the library cells.

Typing `vsim work.mult` in the command window of ModelSim should now correctly load your multiplier for simulation, and you should see a hierarchical view of all the instances in the `sim` tab. Useful windows in the simulator include the waveform viewer (`View | Wave`) and signal viewer (`View | Signals`). You can drag signals to the wave viewer, and save them for re-loading as a "do" script.

The scripting files are executed in the simulator by the "do" command. You can execute the simulation script I provided by typing `do mult.do`. You can run all the ModelSim commands, including the compiler, in scripts with the `vlog` and `vsim` commands. You will need to call these in your Makefile.

The documentation for ModelSim can be accessed by running
`acroread /uusoc/facility/cad_tools/Mentor/fpgadv62/Modeltech/docs/se_docs.pdf`

7 Documentation and Naming Conventions

You must document your code 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 more complicated class project.

For example, you may end up with several different types of files containing Verilog code. First, there are those complete or partial files (such as behavioral HDL) that 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 Verilog as input and also write their results in Verilog. Note that some users prefer to have a single Verilog module per source file. Other users prefer each Verilog module to implement only a single pipe stage, or to not allow combinational logic to span module boundaries.

Furthermore, you will need to iterate through your design flow several times to fix bugs, add features, and improve the quality, performance, and power of your circuit. It will be 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.

8 Project Requirements

This is an individual lab, and you must design and develop your project yourself. It is cheating to copy or borrow code from another classmate. However, you are highly encouraged to discuss tool questions and design philosophy with your class mates. Bounce ideas across each other; see what others think. We want to develop the best design, debug, and evaluation style.

Many of the files that you need, such as the golden model, can be downloaded from canvas. Some of the files you can use as-is, but others will need to be modified to get your design to work. For example, you must add a call in the Makefile to execute mult.dcopt.csh in order to synthesize your circuit. Remember that **all** steps in the design process, including synthesizing and validating of your design, must be executed inside the makefile. You must use Mentor's ModelSim to simulate your project. I have provided a script file that does almost all the work for you.

In this lab you must produce a design that runs as fast as possible. The fastest design in the class gets extra credit, based on timing reported from Design Compiler. The timing that you use in Design Compiler will greatly affect the quality of your results. Make sure you don't have any negative slack in your design. This is equivalent to having a design that won't run at the specified frequency! Remember that the specification for the project requires that you use a synchronous reset. For full credit your simulation must run at the same frequency as synthesis clock.

9 Deliverables

After working through this tutorial you should have the following items to turn in as an organized directory tree, which you will submit via canvas as a tar file. Please only include the files requested in the lab; don't just make a tar of your entire directory tree.

Be sure to include all scripting files or other files that you use in the makefile to produce your results.

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

2. **Makefile** : The makefile should have the following targets:

   - clean: Removes unwanted files from the directory tree
   - c-compile: Compiles and generates the result of the golden model.
   - validate: Compares the result of the behavioral code with that of the golden model.
   - syn: Synthesizes the design.
   - sim: Compile and simulate the synthesized circuit.
   - all: Executes all of the above in the order listed.

3. **Source files** : All of the code needed to build and simulate your lab assignment including the following:

   - reference standard program (probably goldenmodel.cpp)
   - mult.v : behavioral level code implementing the pipelined multiplier
   - the test bench: mult.do or the verilog testbench.
   - Makefile, mult.dcopt.tcl, mult.cstr.tcl, etc.

4. **Data files** from your golden model and design compiler runs.

   - GoldenResult : the output of your golden model
   - VerilogResult : the output from design compiler
   - mult.dcopt.v : your synthesized design
   - The following DC report files: mult.dcopt.constraints, mult.dcopt.power, mult.dcopt.paths, mult.dcopts.fullpaths, and mult.dcopt.area
   - Runtime script report files: mult.dcopt.out and mult.rtlopt.out