EECS 627, Lab Assignment 2

1 Introduction

In this lab assignment, you will extend the process of designing your multiplier chip. You will add two more blocks (a pseudo-random test pattern generator and a signature analyzer), constituting a built-in logic block observation (BILBO) system to the design. The pseudo-random test pattern generator will be used to feed inputs to the multiplier and the signature analyzer will verify the correct functionality of the multiplier. Behavioral code for the pattern generator (LFSR) and for the signature analyzer is provided and you will have to synthesize them to get their structural verilog netlists. You will perform automatic place-and-route (APR) on the multiplier, LFSR and signature analyzer separately. Finally, all the blocks will be assembled at the top level along with clock, power and ground routing. You will be using *Cadence Silicon Ensemble* for performing the block-level APR and top-level chip assembly.

While working through this lab, keep in mind the following: you should be looking at all of the files your tools are reading/writing on your behalf (most of which are plain text), especially in the beginning when you are still learning about how the tools work. You should read the manuals about each and every command you have in your scripts. Remember: **do not blindly copy and paste from the samples that have been provided,** but instead use them as a guide for figuring out how the tools work. Usually, a command-by-command execution order helps a lot!

2 Behavioral Verilog

The top-level design consists of a 4-stage pipelined multiplier, a 16-degree LFSR for generating pseudo-random inputs to the multiplier and a signature analyzer block for examining the output of the multiplier.

1. Pipelined Multiplier:

You will use the multiplier designed in lab 1, which has been verified against the golden brick and synthesized for maximum performance.

2. LFSR

A linear feedback shift register (LFSR) will be used for generating input vectors A and B for the pipelined multiplier. The LFSR implementing an *n*-degree polynomial goes through $2^n - 1$ states before repeating the states. The only state not covered is the all-zero state. The behavioral code for a 16 bit LFSR has been provided on course webpage.

3. Signature Analyzer

A variant of the LFSR can be used to compress the output of the pipelined multiplier. The behavioral code for the signature analyzer has been provided on the course webpage. All the modules have to be synthesized individually before place and route. The multiplier being used has already been synthesized in Lab1. You will have to synthesize the LFSR and signature analyzer modules to get a structural netlist before proceeding to the APR step of the design flow.

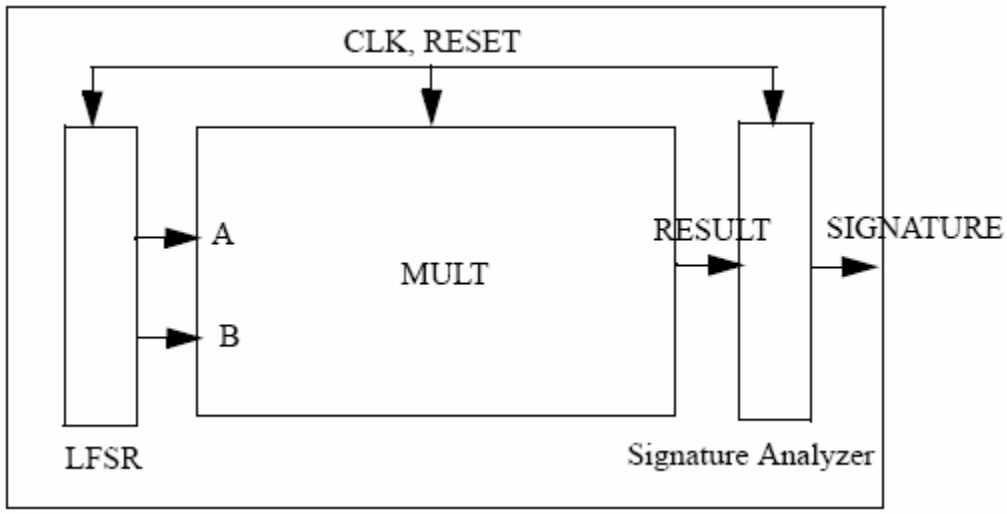

Figure 1: Sample Floorplan of the design

3 Floorplan and partitioning

Figure 1 gives a sample floorplan for the chip. The three blocks correspond to the three components instantiated at the top level. A rough guide for pin assignments is also annotated. This picture will guide us when writing our place and route scripts. Notice that we are planning to partition the design into three blocks, each of which will be synthesized and placed separately.

4 Place and route

The main tool for place and route is **sedsm**, ie. Silicon Ensemble. Make sure you have selected the latest version with *swselect*. While this tool has a graphical interface, it has a textual command driven interface. The graphical interface just presents forms for you to enter the needed information which is then printed out as commands. Thus, by looking at the *se.jnl* log file the tool generates, you can quickly learn the commands you need to script the tool.

4.1 mult.se

A quick and dirty place and route script for the multiplier module is provided on the web-page. Most of this is generic and can be split out into common files to be shared across all of your modules. The things that are specific are related to the floorplan command, the pin placement, and the power stripes. For the MULT module, I decided on an aspect ratio of 1 (ie. width/height) and a row utilization of 0.80. I decided this information by looking at the floorplan in Figure 1 as well as by trial and error (too high of a utilization causes the router to fail). The power stripes you need to decide based on the size and peak dissipation you expect from the module. I usually use the suffix. se for my silicon ensemble scripts, but the tool by default looks for files with the suffix .mac.

Note: Please note that the dimensions in the .se file are in db units (database units). 1 micron = 2000 db.

4.2 General Constraint File

The General Constraint File (design.gcf) specifies the ambient operating conditions for the chip. It is a guideline for the routing tool to choose which timing views to pick from those between slow, typical and fast. These are the process corners against which a design is tested against. In addition, the gcf file specifies the temperature conditions and voltage conditions. When you design a chip, you need to verify that it operates correctly under all conditions of process, voltage and temperature. Usually, when you have a single-sided timing constraint (critical path delay is all that matters), circuits are characterized at slow process corner for an ambient temperature of 100C and a 10% supply voltage drop.

For the current chip, we will however use the typical corners. The design.gcf file points to the appropriate timing file (.tlf) for the routing tool to use. I have uploaded the design.gcf file on the web-site for reference.

4.3 Pin Placement

In addition you will need to write by hand (potentially from the template file generated by the random pin-placer mode), a pin-placement constraint file. By looking at the floorplan diagracm, you have to decide the pin placement such that the congestion is minimized during global routing. The sample template .ioc file (pin placement file format is .ioc for *silicon ensemble*) “mult.ioc” is given on the web-site.

4.4 Running sedsm

I usually run sedsm in graphical mode, so I can visually inspect the results when things go wrong. Silicon Ensemble generates a huge volume of temporary files, so it is a good idea to run it from your alloted project directory in CAEN. Here is how you would invoke the executable:

```
cd $MY_APR_WORK_DIR
sedsm -m=500 &
```

The -m option is for allocating virtual memory to the process. When the program starts, you get an empty window. Go to File–Execute and type in the full pathname of the script **mult.se**.

Don’t forget to edit this file to change the pathnames to your own files. You may also have to change the aspect ratio, the size of your floorplan, the number of power stripes, etc. in order to get a compact layout. It is a bit annoying, but you should put the full directory names and paths of your files in your scripts so that you can run the tool from a different directory than your source code to avoid confusion. After a bit of running (during which the graphical display usually freezes, you should see your LFSR module placed and routed. Most likely you have to hit View–Redraw to see anything. Also, more than likely, something has gone horribly wrong with your scripts. The best place to debug this is from the *se.jnl* file that will be made in the working directory. This is a record of each command executed along with the results of execution (in comments). One typical mistake is to forget the trailing semicolon (**;**) after each command. Once you gain experience with the many limitations and quirks of the tool, things will go much more smoothly. Typically, I restart the tool periodically and remove the contents of the work directory to ensure correct operation. You should remove the unnecessary files and directories before invoking sedsm again because these files may cause the tool to crash.

Once everything is working you should get several output files, the main files of interest are **mult.APR.sdf** and **mult_APR.v**. The **mult.APR.sdf** is an extracted timing file that lets you backannotate your verilog simulations. Most of you would have noticed during synthesis in Lab1 that the sdf file generated by design compiler has no information about the wire delay and all the delay values corresponding to the wires is zero. **mult_APR.v** is the verilog description of the design after the placement. Note that your placed design will be different from the synthesized structural netlist because of the insertion of clock tree during APR and **mult_APR.v** will be used during LVS of the final layout in Lab3. Hence it is important to verify the functionality of the verilog file generated by *silicon ensemble* with proper back-annotation of the corresponding sdf file.

4.5 Clock Tree Generation

Silicon Ensemble has an inbuilt tool called **ctgen** for generating clock tree for specified constraints. You will need two files: **ctgen.cmd** and **ctgen.const** for invoking the **ctgen** tool. A sample **ctgen.cmd** containing the environment setup commands is given on the web-site.

ctgen.cont specifies the constraints on the clock skew, delay and transition time of the clock signal. A sample **ctgen.cmd** file is given on the web-site.

4.6 Using Silicon Ensemble for Block-Level APR and Global Routing

There are slight differences in .se files for the block-level APR and global APR. The block level APR can be automated and you can execute your .se script in one go. In global routing, the lef files containing the metal routing information for different instantiated modules (which have been placed using block level APR) need to be read in, along with the verilog netlists. You have to change the line 'CLASS MACRO” inside all the lef files to “CLASS BLOCK” before using them in global routing. After reading the files, you will find all the instantiated blocks at the bottom-left corner in your *silicon ensemble* GUI. These blocks have to be placed by hand inside the chip area before initiating the pin placement, power and signal routing. You will use the “move” command in the GUI for this purpose.

5 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. **Do not include any temp file in your tar ball**. Your submitted directory should have *only* the following files:

- README.txt: text file containing a brief description of any problems you encountered and how you solved them.
- detailed floorplan. A simple .ppt/.doc will do.
- synthesized structural verilog for mult, LFSR and signature modules
- place and route (.se) scripts for mult, LFSR, signature blocks and global routing.
- pin placement (.ioc) files for mult, LFSR, signature and top level.
- ctgen.cmd and design.gcf file.
- ctgen.const files for mult, LFSR and signature blocks.
- se.jnl for the three block-level APRs and the global APR

6 For further reference

- Cadence Online Documentation