EECS 151/251A ASIC Lab 5: Parallelization and Routing

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer

Overview

Like last week, this lab has two parts. For the first part, we will continue to develop our GCD coprocessor by improving its performance. After that, we will continue the physical design flow by performing routing.

To begin this lab, get the project files and set up your environment by typing the following commands

```
git clone /home/ff/eecs151/labs/lab5.git
cd lab5
export HAMMER_HOME=$PWD/hammer
source $HAMMER_HOME/sourceme.sh
```

If you have not done so already you should add the following line to your bashrc file (in your home folder) so that every time you open a new terminal you have the paths for the tools setup properly.

```
source /home/ff/eecs151/tutorials/eecs151.bashrc
```

You will have to modify `design.yml`, `design_gl.yml` and `design_par.yml` files and replace `<YOUR_LAB_ROOT_DIRECTORY>` with the absolute path to your lab clone's root directory.

As a final note, please make sure you run `make clean` in all your previous lab directories to minimize the space they occupy on the disk.

Design

One way we can improve the performance of our GCD coprocessor is by parallelizing the compute. We can do this by including multiple GCD units in our design, and routing traffic to them as they become available.

You will find that the solution to last week's lab (`fifo.v` and `gcd_coprocessor.v`) is included. The test has been modified to check the total number of cycles taken by the coprocessor to complete the tests. Run `make sim-rtl` to run the new testbench on the solution code. Take note of the number of cycles that the tests take without modification, as you will need it to calculate your speedup.

Your task is to edit `gcd_coprocessor.v` to improve the perfomance below 225 cycles. We will do this by using two instances of GCD.

You will find RTL that connects the datapath and controller into one module in `gcd_unit.v`. You may find this useful when refactoring the `gcd_coprocessor`, since you will need fewer wires to place both GCD instances.

You will also find stub code for an arbiter, which you should complete. We will use the arbiter to route traffic to GCD units and preserve the response ordering. Most of your design can be implemented with combinational logic, but you will need some state to remember which GCD block contains the earliest data to preserve ordering.

Question 1: Design

a) Submit your code (`gcd_coprocessor.v` and `gcd_arbiter.v`) with your lab assignment.

b) How many cycles did your simulation take? What was the % speedup?

Automated Flow

In the last lab, we had only run PAR flow up until a point. In this lab, we will perform the full design. The only major step we didn't run through in the last lab was routing. Routing is the final major modification performed on our design. Up until routing, Innovus uses a quick and dirty routing engine with errors and shorts, but ignores these errors and simply tries to get an estimate of the distance between cells and parasitics that each route will see. Once post-CTS optimization is done, it switches to a different tool that performs careful routing and eliminates these shorts while maintaining timing performance of our design. Routing is one of the most computationally heavy tasks of digital IC design and can take days to complete for complicated designs. After routing is complete, a post-Route optimization is run to ensure no timing violations remain. Post-Route optimization typically has little freedom to move cells around, and it tries to meet the timing constraints mostly by tweaking the length of the routings. The HAMMER flow is not yet complete for this 7 nm technology, so some DRC (Design Rule Check) errors will remain after routing.

First synthesize the design:

```
make syn
```

Then, simulate the synthesized design to make sure it still works:

```
make sim-syn
```

Once your synthesized design passes the test, you can start the PAR flow:

```
make par
```

The PAR command will take a long time to complete, as it runs through all stages of PAR. Once it completes, take a look at the build directory as in the previous labs.

Question 2: Automated Flow

a) What are the critical paths of your post-PAR and post-Synthesis designs? Are they the same path? How does this critical path compare to your single-unit critical path?

b) Iterate on your design by modifying `design.yml` to find a rough estimate (no need to be too precise) for the clock period until you start running into setup errors. Given the number of cycles it takes to complete the testbench, what is the shortest time your design can finish the computation?

Innovus Commands

As in the previous lab, we will look at the contents of `par.tcl` that HAMMER generates and follow along using Innovus. Navigate to the directory `build/par-rundir` and type:

```
innovus -common_ui
```

Run through the commands by copying and pasting them to the terminal window, until you reach the command `write_netlist`. By the time you reach this command, you will have run through the entire PAR flow. This command writes out the verilog netlist that corresponds to the current state of the design, and at this point it corresponds to the fully completed design. You will see that it excludes a bunch of cells we don't want in the Verilog netlist. These cells serve electrical or process purposes, but are of no use in Verilog. Notice that this command also has a switch `-phys`. This switch makes the tool write out a "physical" version of the cells, including their VDD and VSS connections. We also do not want that since the Verilog models of our cells do not have these ports. Create a new version of this command by copying it and pasting it below. Modify the output name to be `gcd_coprocessor.sim.v` and remove the `-phys` switch. Now we can perform post-PAR simulation to verify nothing messed up the operation of our circuit. Do not execute the `quit` command, instead try to experiment with the reporting commands available to you. To see all the reporting commands, type `help report*` to innovus shell and read through the options available to you.

Question 3: Innovus Reports

a) How much power does your design consume? What is the distribution between internal, switching and leakage? Would we expect the switching power estimation to be accurate? How about leakage?

b) What is the area consumed by your design? What percentage of the total area does the arbiter occupy?

After you are done with the flow, it is time to simulate our newly printed post-PAR netlist. Type the following command:

```
make sim-par
```

This will use the same testbench, but will now use the post-PAR netlist of your design. Note that we are still using zero gate delays, meaning if there are any setup/hold issues in our design, our simulation will not capture that.

Question 4: Trade-Offs

a) Rerun the flow using your old design. Using the area and power values from Innovus, how does the performance improvement from the dual-unit design compare to area occupation and power consumption increase compared to your old design?

b) Modify your gcd_coprocessor.v to take an input parameter in terms of number of clock cycles we want our design to meet (`parameter TARGET_NUMBER_OF_CYCLES`) for this given testbench. Your code should generate a low area, low power design if the number is greater than that your simple gcd_coprocessor can achieve, and it should generate the dual-unit design if it is lower. Submit your code.

c) (Optional) Using a rough estimate of target number of cycles versus number of units in the design, write a code that will generate 1-8 cores depending on the performance demand. Do NOT do this by writing out every possible case explicitly. You can limit the number of units to powers of two (1,2,4,8) if it makes your life easier.