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

Prof. John Wawrzynek
TAs: Sean Huang, Tan Nguyen

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
```

As a final note, please make sure you run `make clean` and remove the `build` folders in all your previous lab directories to minimize the space they occupy on the file system.

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 flow. 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. You may see some DRC (Design Rule Check) errors caused by the 7nm technology library, after routing.

First synthesize the design:

```
make syn
```

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

```
make sim-gl-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. You might see additional files compare to the `syn-rundir`, and that's because the PAR flow incorporates the RC and parasitic delays, in addition to the cell delays. Open `build/par-rundir/gcd_coprocessor.setup.par.spef` and search for the first occurrence of `D_NET`. What does it say about the first net? You may find this wiki page helpful. (*thought experiment #1*: get a sense of the units at the top and orders of magnitude of the RC parasitics in the SPEF file. If we used a 5nm technology library, do you expect the resistance to generally increase or decrease? How about the capacitance?)

Question 2: Automated Flow

a) Check the post-Synthesis timing report (`syn_rundir/reports/final_time_PVT_0P63V_100C.setup_view.rpt`) and post-PAR timing report (`par_rundir/timingReports/gcd_coprocessor_postRoute_all.tarpt`). 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?

c) Open the post-CTS timing report(`hammer_cts_debug/hammer_cts_all.tarpt`) and the post-PAR timing report(`par_rundir/timingReports/gcd_coprocessor_postRoute_all.tarpt`). Find a common path (same start and end sequential elements). What differences do you notice within the paths?

Innovus Commands

As in the previous lab, we will look at the contents of `par.tcl` that HAMMER generates and follow along using Innovus. (*thought experiment #2* : open the `par.tcl` and search for the command `set_db add_fillers_cells`. Based on the names of the cells specified by this command, what do you think is the function of the filler cells?)

Navigate to the directory `build/par-rundir` and type:

```
innovus -common_ui
```

to open Innovus shell. Next, type `read_db gcd_coprocessor_FINAL` to load the current design database from the latest PAR flow. This will help us to avoid rerunning the entire flow. 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) What is the area consumed by your design? What percentage of the total area does the arbiter occupy?

b) Submit a screenshot of your setup slack histogram. Compared with the histogram you obtained in Lab 4, does your new slack distribution support the observed performance improvements you obtained in your coprocessor?

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

```
make sim-gl-par
```

This will use the same testbench, but will now use the post-PAR netlist of your design, back-annotated with delays and parasitics from PAR. Make sure to adjust the `CLOCK_PERIOD` variable in

`sim-gl-par.yml` to match the clock period you obtained from PAR. Note, however, that the exact clock period may not work and you may need to relax it slightly.

After running `make sim-gl-par` you can run power analysis using:

```
make power-par
```

Navigate to `power_rundir/activePowerReports` and open `PVT_0P63V_100C.setup_view.rpt`. Do the power estimation numbers match your expectation?

Question 4: Trade-Offs

a) Rerun the flow using your old design. You may prevent clobbering of files in `build` by setting the `OBJ_DIR` variable when typing `make` commands to something other than `build` for the 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.

Acknowledgments

This lab is the result of the work of many EECS151/251 GSIs over the years including:

Written By:

- Nathan Narevsky (2014, 2017)
- Brian Zimmer (2014)
- Cem Yalcin (2019)

Modified By:

- John Wright (2015,2016)
- Ali Moin (2018)
- Arya Reais-Parsi (2019)

- Tan Nguyen (2020)
- Harrison Liew, Jingyi Xu(2020)
- Sean Huang (2021)