EECS 151/251A ASIC Lab 4: Floorplanning, Placement and Power

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

Overview

This lab consists of two parts. For the first part, you will be writing a GCD coprocessor that could be included alongside a general-purpose CPU (like your final project). You will then learn how the tools can create a floorplan, route power straps, place standard cells and perform timing optimizations on your design.

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

```
git clone /home/ff/eecs151/labs/lab4.git
cd lab4
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
```

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

Writing Your Coprocessor

Take a look at the `gcd_coprocessor.v` file in the `src` folder. You will see the following empty verilog module.

```
module gcd_coprocessor #( parameter W = 32 ) (
  input clk,
  input reset,

  input operands_val,
  input [W-1:0] operands_bits_A,
  input [W-1:0] operands_bits_B,
  output operands_rdy,

  output result_val,
  output [W-1:0] result_bits,
  input result_rdy

);

// You should be able to build this with only structural verilog!

// Define wires

// Instantiate gcd_datapath
// Instantiate gcd_control
// Instantiate request FIFO
// Instantiate response FIFO

endmodule
```

First notice the parameter W. W is the data width of your coprocessor; the input data and output data will all be this bitwidth. Be sure to pass this parameter on to any submodules that may use it! You should implement a coprocessor that can handle 4 outstanding requests at a time. For now, you will use a FIFO (First-In, First-Out) block to store requests (operands) and responses (results).

A FIFO is a sequential logic element which accepts (enqueues) valid data and outputs (dequeues) it in the same order when the next block is ready to accept. This is useful for buffering between the producer of data and its consumer. When the input data is valid (`enq_val`) and the FIFO is ready for data (`enq_rdy`), the input data is enqueued into the FIFO. There are similar signals for the output data. This interface is called a "decoupled" interface, and if implemented correctly it makes modular design easy (although sometimes with performance penalties).

This FIFO is implemented with a 2-dimensional array of data called `buffer`. There are two pointers: a read pointer `rptr` and a write pointer `wptr`. When data is enqueued, the write pointer is incremented. When data is dequeued, the read pointer is incremented. Because the FIFO depth is a power of 2, we can leverage the fact that addition rolls over and the FIFO will continue to work. However, once the read and write pointers are the same, we don't know if the FIFO is full or empty. We fix this by writing to the `full` register when they are the same and we just enqueued, and clearing the `full` register otherwise.

A partially written FIFO has been provided for you in `fifo.v`. Using the information above, complete the implementation so that it behaves as expected.

A testbench has been provided for you (`gcd_coprocessor_testbench.v`). You can run the testbench to test your code by typing `make sim-rtl` in the root directory as before.

Question 1: Design

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

Floorplanning, Placement, Pre-CTS Optimization and Post-CTS Optimization

We will first bring our design to the point we stopped in last lab. Type the following command to synthesize your design:

```
make syn
```

And now, to simulate the post-synthesis design to make sure it is working:

```
make sim-syn
```

If everything is working correctly, this means our design is ready to become a real integrated circuit. This involves a process called Place&Route(P&R or PAR). P&R itself is a rather big process, so we will look only at the first several steps and leave the last part to another lab. The steps that we will look at in this lab are floorplanning, placement, pre-CTS optimization and post-CTS optimization. Between pre-CTS optimization and post-CTS optimization, there is a step called CTS, which stands for Clock Tree Synthesis. While we won't be looking into the details of CTS this lab, it involves distribution of the clock signal to each synchronous element (flip-flop or latches) in a low-skew (low mismatch between arrival timing of clock to different elements)and power-efficient (as few and small clock buffers as possible) way.

The steps that we will look at are as such:

Floorplanning & Placement

Floorplanning is the process of allocating area to the design as well as putting constraints on how this area is utilized. It can also involve pre-placement of "macro" designs, which can be anything from memory elements (SRAM arrays) to analog black boxes (like PLLs or LDOs). In this lab, we will just look at allocating a custom sized area to our design, specified in the `design.yml` file. Open up this file and locate the following text block:

```yaml
# Placement Constraints
vlsi.inputs.placement_constraints:
  - path: "gcd_coprocessor"
    type: "toplevel"
    x: 0
    y: 0
    width: 150
    height: 150
    margins:
      left: 10
      right: 10
      top: 10
      bottom: 10

# Pin Placement Constraints
vlsi.inputs.pin_mode: generated
vlsi.inputs.pin.generate_mode: semi_auto
vlsi.inputs.pin.assignments: [
  {pins: "*", layers: ["M5", "M7"], side: "bottom"}
]
```

This block specifies the origin (`x`, `y`), size (`width`, `height`) and border margins of the block `gcd_coprocessor`. For complicated designs, floorplans of major modules are often defined separately. Pin constraints are also shown here, all that we need to see is that all pins are located at the bottom boundary of the design, on Metal 5 and Metal 7 layers.

Placement is the process of placing the standard cells your design is synthesized into, to the specified floorplan. While there is a major placement step that involves placing all the synthesized gates, placement of minor cells (such as bulk connection cells, antenna-effect prevention cells, I/O buffers...) take place separately and in between various stages of design. But "placement" almost always refers to the initial placement of the standard cells. When the cells are placed, they are not "locked", they can be moved around by the tool during optimization steps. However, initial placement tries its best to place the cells optimally, obeying the floorplanning constraints and keeping in mind the connections between cells. Poor placement (as well as poor aspect ratio of the floorplan) can result in congestion of wires later on in the design, which may prevent successful routing.

Pre-CTS Optimization

Pre-CTS optimization is the first round of Static Timing Analysis (STA) and optimization performed on the design. It has a large freedom to move the cells around to optimize your design to meet setup checks. Hold errors are not checked during pre-CTS optimization. Because we do not have a clock tree in place yet, we do not know when the clocks will arrive to each sequential element, hence we don't know if there are hold violations. The tool therefore assumes that every sequential element receives the clock ideally at the same time, and tries to balance out the delays in data paths to ensure no setup violations occur. In the end, it generates a timing report, very similar to the ones we saw in the last lab.

Post-CTS Optimization

Post-CTS optimization is performed after CTS, and therefore clock is now a real signal that is being distributed unequally to different parts of the design. Therefore, setup and hold violations are fixed simultaneously. Often times, fixing one error may introduce one or multiple errors, so this process is iterative for the tool, but it is usually hidden from the user.

Compiling the Design with HAMMER

Now that we went over the flow (at least at a high level), it is time to actually perform these steps. Type the following commands to perform the above described operations:

```
make syn-to-par
make redo-par HAMMER_REDO_ARGS="--to_step clock_tree"
```

The first command here prepares the outputs of the synthesis step to comply with the inputs expected from the Place&Route steps. The second command is similar to the partial synthesis commands we used in the last lab. It tells HAMMER to run until it performs the Clock Tree Synthesis and post-CTS optimization and then stop. For this lab, HAMMER uses Cadence Innovus as the back-end tool to perform P&R. Wait until Innovus is done with the P&R steps until post-CTS optimization and exits. You will see that HAMMER again gives you an error - similar to last lab when HAMMER expected a synthesized output, this time HAMMER expects the full flow to be completed and gives an error whenever it can't find the finished IC.

Once done, look into the `build/par-rundir` folder. Similar to how all the synthesis files were placed under `build/syn-rundir` folder in the previous lab, this folder holds all the P&R files. Go ahead and open `par.tcl` file in a text editor. This is the file that HAMMER generated to communicate with Innovus, and these are Innovus Common UI commands. While we will be looking through some of these commands in a bit, first take a look at `timingReports`. Due to when HAMMER outputs timing reports, you should only see the Pre-CTS timing reports. `gcd_coprocessor_preCTS_all.tarpt.gz` contains the report in a g-zipped archive. Rest of the files also contain useful information regarding capacitances, length of wires etc. Unzip these using `gzip` or navigate with Caja, the file browser, and double click the archive you want to peek into. Going back a level, in `par-rundir`, the folder `hammer_cts_debug` has the post-CTS timing reports. The two important archives are `hammer_cts_all.tarpt.gz` and `hammer_cts_all_hold.tarpt.gz`. These contain the setup and hold timing analyses results after post-CTS optimization. Look into the hold report. You may actually see some violations! However, any violation should be small (<1 ps) and because we have a lot of margins during design (namely the `design.yml` file has "clock uncertainty" set to 100 ps), these small violations are not of concern, but should still be investigated in a real design.

Under the Hood: Innovus

While HAMMER obfuscates a lot from the end-user in terms of tool-based commands, most IC companies directly interface with Innovus and it is useful to know what tool-specific commands you are running in case you need to debug your circuit step-by-step. Therefore, we will now look into `par.tcl` and follow along using Innovus. Make sure you are in the directory `build/par-rundir` and type:

```
innovus -common_ui
```

After a few seconds a GUI should pop up and your terminal should have switched to innovus shell. GUI shows the current status of your design. Now, follow `par.tcl` command-by-command, copying and pasting the commands to the command prompt and looking at the GUI for any changes. You may skip the `puts` commands as they just tell the tool to print out what its doing. The steps that you will see significant changes are listed below. Once you reach the first step, click anywhere inside the black window at the center of the GUI and press "F" to zoom-to-fit. You should see your entire design (which is an empty canvas at this point). As you progress through the steps, feel free to zoom in to investigate what is going on with the design. One note of importance is, you should disable V8 M8 V9 M9 layers using the panel on the right side of the GUI. These layers are top-level metal layers and are used for dense power routing, but they obscure the lower levels of the design when enabled. Simply uncheck the first box adjacent to their names and they will now show up (or disappear if they are already on your screen). The steps design will undergo noticeable changes are:

1. After the command sourcing floorplan.tcl
2. After the command sourcing power_straps.tcl
3. After the command `edit_pin`
4. After the command place_opt_design

After the `ccopt` command is run, you may see a bunch of white X markers on your design. These are some Design Rule Violations (DRVs), indicating Innovus didn't quite comply with the technology's requirements. Ignore these for the purposes of this lab.

Question 2: Innovus Steps

a) Submit a snapshot of your design for each of the four steps described above. Make sure V8 M8 V9 M9 layers are not visible, and your design is zoomed-to-fit. Note the important changes from last step in the figure captions.

b) What is the critical path of your design post-CTS? Is it the same as the post-synthesis critical path?

Now zoom in to one of the cells and click the box next to "Cell" on the right panel of the GUI. This will show you the internal routings of the standard cells. While by default we have this off, it may prove useful when investigating DRVs in a design. You can now exit the application by closing the GUI window.

Project Preparation

Question 3: ALU

In this question, you will be designing and testing an ALU for later use in the semester. A header file containing define statements for operations (`ALUop.vh`) is provided inside the `src` directory of this lab. This file has already been included in an ALU template given to you in the same folder (`ALU.v`), but you may need to modify the `include` statement to match the correct path of the header file. Compare `ALUop` input of your ALU to the define statements inside the header file to select the function ALU is currently running. Definition of the functions is given below:

Op Code	Definition
ADD	Add A and B
SUB	Subtract B from A
AND	Bitwise `and` A and B
OR	Bitwise `or` A and B
XOR	Bitwise `xor` A and B
SLT	Perform a signed comparison, Out=1 if $A < B$
SLTU	Perform an unsigned comparison, Out=1 if $A < B$
SLL	Logical shift left A by an amount indicated by B[4:0]
SRA	Arithmetic shift right A by an amount indicated by B[4:0]
SRL	Logical shift right A by an amount indicated by B[4:0]
COPY_B	Output is equal to B
XXX	Output is 0

Given these definitions, complete `ALU.v` and write a testbench `tb_ALU.v` that checks all these operations with random inputs at least a 100 times per operation and outputs a PASS/FAIL indicator.