C2S2 Digital RTL Training

Aidan McNay, acm289 and Will Salcedo, wrs225

C2S2

This tutorial should serve as an introduction to using the open-source OpenLane flow. By the end, you should have a working knowledge of how to use the flow, where different files are stored, and to be able to harden your own design within the Caravel Harness.

1 Setup

To setup your environment, first log into the C2S2 server using the SSH protocol. The server is <NetID>@c2s2-dev.ece.cornell.edu, and the password should be your NetID password. (If you've never logged in using SSH before, please see the Linux Development Environment Tutorial). From there, you should always begin by sourcing the setup script for C2S2:

```
% source setup-c2s2.sh
```

(Note that in this tutorial, a "%" preceeding a terminal command only indicates that it is such, and should not be included in the terminal command.)

Next, we'll create the space we'll be working in. This involves cloning the Caravel User Project and Digital RTL Tutorial repository from GitHub; if you haven't used GitHub or set up GitHub before, be sure to check out the GitHub tutorial. We will assume here that GitHub has been set up properly for you.

```
% mkdir -p ${HOME}/c2s2/digital_rtl_tutorial
% cd ${HOME}/c2s2/digital_rtl_tutorial
% TUTROOT=${PWD}
% git clone git@github.com:efabless/caravel_user_project.git caravel_user_project
% git clone git@github.com:cornellcustomsiliconsystems/DigitalRTL-Tut.git rtl_design
% cd rtl_design
% TOPDIR=${PWD}
```

Great! Now we're all set to start designing.

2 Registered Incrementer

For this portion of the tutorial we will walk through the process of designing, testing, and simulating a registered incrementer using Verilog and a PyMTL test bench. We will walk through the creation

of a test bench before moving onto Verilog RTL design.

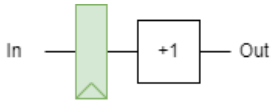

Figure 1: Block Diagram of Registered Incrementer

2.a Designing the Incrementer

The *rtl_design* repository contains a skeleton of modules and unit tests you could use to follow along with the tutorial. If you get tired of copy and pasting every line individually, you could use the command:

```
% alias %=""
```

Next, create a build folder in sim and navigate to your sim/block_test folder in the structure. A folder named "sim" will be our convention for the file structure which stores our test benches, simulators, and Verilog source files. We will use tools in the sim file strucure to generate pickled Verilog files to push through the OpenLANE flow.

```
% cd sim
% mkdir build
% cd reginc/block_test
```

Next you should use your preferred text editor to open the file RegInc_test.py.

2.b PyMTL Test Bench

We will now discuss how to create a PyMTL test bench. We use PyMTL to test and simuilate our RTL designs because it is easier to develop unit tests with Python rather than Verilog. However, it is preferable to develop our hardware with Verilog because that is the language we use in our intro classes. Thus, we develop wrappers for our Verilog code which interface with PyMTL test benches. Test benches are an important part of hardware design because they ensure our designs are functionally correct. Mistakes in code are much cheaper to fix than mistakes on silicon.

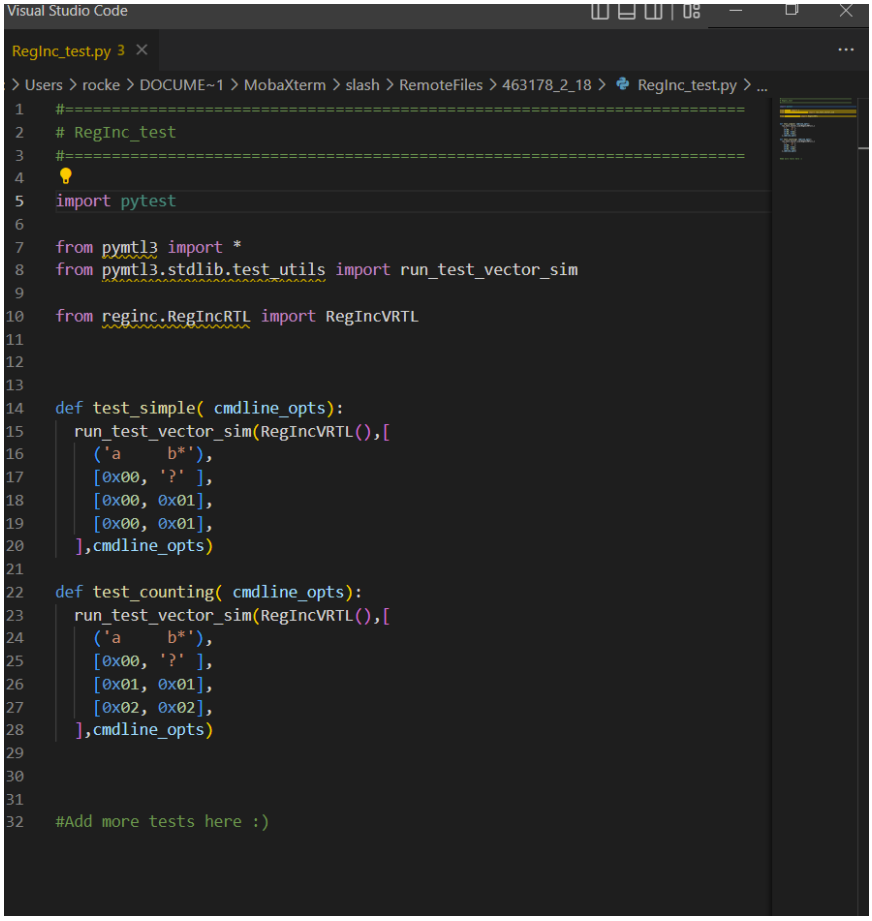

Figure 2: Image of PyMTL Test Bench Code for Registered Incrementer

To run our tests, we use the testing framework PyTest. Any file with "test_" at the start of the name, or "_test" at the end of its name will run all of the tests contained within when you run the command "pytest" in that directory.

Observe Figure 2. This is an image of a PyMTL test bench for the registered incrementer. Any method with "test_" in the name will be automatically ran by the PyTest framework. Within the method you must provide assertions to define the test cases. PyMTL provides tools to easily create hardware tests using input vectors.

Please make more tests for the registered incrementer. Note that the input takes one cycle to propagate to the output.

Run your unit tests by using the following commands.

```
% cd ${TOPDIR}/sim/build
% pytest ../reginc/block_test
```

2.c Python Verilog Wrapper

Now, navigate to to your Verilog source code folder and open the file RegIncRTL.py.

```
% cd ../reginc
```

```python
# This is the PyMTL wrapper for the corresponding Verilog RTL model RegIncVRTL.

from pymtl3 import *
from pymtl3.stdlib import stream
from pymtl3.passes.backends.verilog import *


class RegIncVRTL( VerilogPlaceholder, Component ):

  # Constructor

  def construct( s ):
    # If translated into Verilog, we use the explicit name

    s.set_metadata( VerilogTranslationPass.explicit_module_name, 'RegInc' )

    # Interface
    s.a = InPort(32)
    s.b = OutPort(32)


RegInc = RegIncVRTL

```

Figure 3: Image of Python Verilog Wrapper

Shown in Figure 3 is the Python verilog wrapper. This wrapper allows our test cases to interface with the hardware we specify in Verilog. The name of your class must be the same as your Verilog module. Your port names defined in the construct method must correspond with those in your Verilog module as well.

2.d Verilog

Finally, open the file titled RegIncVRTLv. This file contains the Verilog source code for your registered incrementer.

Observe that it only consists of a module named RegIncVRTL. In Verilog you specify modules with ports. Within modules you implement your hardware specification.

In Verilog you could design hardware using behavioral or structural specifications. Behavioral specification allows you to specify hardware outputs as a function of input. Structural specification requires you specify the hardware structure as discrete blocks or gates.

In Verilog, the logic keyword is used to define nets, variables, and ports. Instantiate a 32 bit wide bus called regout by copying the following code.

```
logic [31:0] regout;
```

Next, we will create a register using behavioral specification. A register can be designed using an always block that gets triggered by the positive edge of a clock signal. "Always@" blocks trigger upon a certain event. Shown below is an always block which triggers on the positive edge of a clock signal. "Always@(*)" defines a combinational behavioral block.

```
always @(posedge clk) begin
    //<code>
end
```

In an always block there are two types of assingments: "=" and "<=." The "=" assignment statements are blocking. They are assinged in the order they appear. "<=" statements are non-blocking, meaning they are assigned concurrently with one another.

"If" statements may also be used in always blocks. You can instantiate an "if" statement using the following structure.

```
if(<expression>) begin
    //<code>
end
```

We will now assign the output using combinational logic. Use this line to assign b to regout + 1.

```
assign b = regout + 1;
```

Finish implementing the registered incrementer (Hint: the rest of the work is in the always block). Ensure that you account for a high reset signal (regout = 0 when reset = 1). If you need help, reach out to your subteam lead. Ensure your test cases pass, and move onto the next section.

3 Moving to Caravel

Caravel is a open-source digital ASIC harness that expedites the design time/process for new users to make custom chips. It provides a harness for you to use (including a pad ring for all of your I/O's, as well as a tiny RISC-V processor and some memory), but leaves most of the space up to your design. While a user is free to experiment with the harness, it provides a clean way to serve as a base to build off of, so we will be using it for our designs.

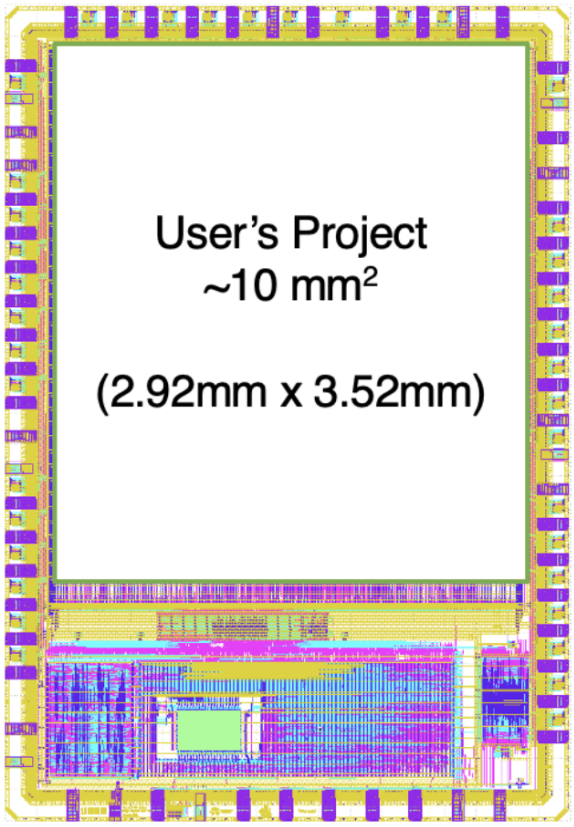

Figure 4: The Caravel harness

By here, you should have a fair amount of confidence in your design from your tests. We can now transfer the design over to the Caravel harness. We'll begin by copying your Verilog code over:

```
% cp ${TOPDIR}/sim/reginc/RegIncVRTL.v ${TUTROOT}/caravel_user_project/verilog/rtl
% cd ${TUTROOT}/caravel_user_project
% FLOW_DIR=${PWD}
% ls
```

As you can see from the output of *ls*, the Caravel directory is very complicated! Let's break it down:

- **def**: Contains all of your generated *.def* files. Def stands for **Design Exchange Format**, and is how the information about the physical design of your IC (such as the location of the cells) is stored
- **docs**: Some general documentation. However, I would recommend looking here instead (although for software design, there have been updates, so some of the memory-mapped IO documentation is incorrect)
- **gds**: Contains all of your *.gds* files. GDS stands for **Graphic Design System**, and is essentially a picture of all of the layers of your design. This is the final product, and is what

we send off to the foundry so they can make your design!

- **lef**:Contains all of your generated *.lef* files. Lef stands for **Library Exchange Format**, and is used to contain an abstracted representation of an object. For example, all PDKs have a *lef* file containing enough information about the cells for placement (such as the size, where the pins are, etc.), but not their actual implementation (how the silicon is doped). It is useful to have a *lef* file of your design, in case you want to use it as a macro and place it inside of another design!
- **mag**: These contain all of the *.mag* files for the project. These are used by Magic, an open source circuit design tool. While it is primarily used for standard cell design and patterning, it can also be used to generate a GDS file. OpenLANE uses this ability to stream its final results through Magic to generate the final GDS
- **maglef**: Library files for Magic
- **openlane**: Contains all of the specifications needed to tell OpenLANE how you want to build your design (including any specifications about the physical design, where macros should be places, etc.)
- **signoff**: Generated from running the flow to indicate which versions of OpenLANE and the PDK's you were using (PDK stands for **Process Design Kit**, and contains the specifications for all of the standard cells used in your design)
- **spi**: Contains the Spice files for your design, which are used for the LVS check (Layout Versus Schematic - essentially checking whether the current layout/connection of the cells will have the same functionality as your design)
- **verilog**: ***THIS*** is where most of your work will be done. It contains:
  - **rtl**: All of your RTL design files (where you put your designs!). It comes with a wrapper (which interfaces your designs with the rest of Caravel), and an example project (an up-counter)
  - **gl**: Generated gate-level netlists of your designs
  - **dv**: All of your tests
  - **includes**: Contains the files that tell the simulator which designs to include when simulating

Not included here (but still necessary for running the flow) are the *pdk* directory (which contains all of the PDK's) and a separate *openlane* directory (which contains all of the OpenLANE tools and scripts). These are installed globally, and are already set up for you when you source the setup script. You can check their location with `echo ${OPENLANE_ROOT}` and `echo ${PDK_ROOT}`, respectively.

First, we need to set up the caravel directory. We already have OpenLANE and the PDK's set up, so all you need to do is run

```
% make install check-env install_mcw setup-timing-scripts
```

- **install** will pull and install the latest version of the Caravel harness. After running this, you should see a new *caravel* directory inside of your user project directory. This directory contains all of the source and build files for the harness
- **check-env** checks that Caravel was pulled successfully, and that there's nothing else to be done with it
- **install_mcw** installs the management core wrapper. While Caravel contains the files for the overall pad ring and some of the interconnects, the management SOC (System-on-Chip) is kept as a separate unit, so that the process can be updated separately from Caravel (...which has happened! The processor switched a little bit back from a Pico-based processor to a Litex VexRISCV, and this modular design allowed it to happen very easily). After running this, you should see a new *mgmt_core_wrapper* directory inside of your user project directory. This directory contains all of the source and build files for the management SOC
- **setup-timing-scripts** pulls the latest timing scripts into a *dependencies* folder. (Efabless ran into a few hold time issues previously, and after discovering that the tools for checking them weren't working, created their own timing scripts to check for timing violations)

Notice how the insides of both the *caravel* and *mgmt_core_wrapper* directories look just like your high-level design directory - that's because they were designed and build the same way! They can always serve as a reference for how things "should" be.

Great - now that the setup is done, we can move on to working on your design!

4 Modifying the Design for Caravel

4.a Goodbye, SystemVerilog

Unfortunately, SystemVerilog doesn't tend to play nicely with the synthesis tools. There is ongoing work to fix this (and a couple hacky work-arounds that we personally found), but for the large part, it works better if it's in Verilog. To achieve this, we will use SV2V, an open-source SystemVerilog-to-Verilog conversion tool. We can use it to convert all of our work from before into Verilog:

```
% cd ${FLOW_DIR}/verilog/rtl
% mv RegIncVRTL.v RegIncVRTL.sv
% sv2v -w adjacent RegIncVRTL.sv
% rm RegIncVRTL.sv
```

Take a look at the result (in *RegIncVRTL.v*) - you can see that, while it functions correctly, it's not quite what we're used to. SV2V is great at handling large files, but the flow tools still want all of the inputs and outputs declared in the interface, so let's modify it to have our old interface (keeping in mind that the I/O's are now considered wires):

```
module RegIncVRTL (
    input  wire        clk,
    input  wire        reset,
    input  wire [31:0] a,
    output wire [31:0] b
);

// Delete the other declarations of the wires as inputs and outputs
```

For small files like this, it might be easier to edit by hand, but for large files, it can prove to be a useful tool. The other alternative is to code in Verilog from the start, but that comes at the downside of not being able to use SystemVerilog constructs like *logic* - it's all up to you!

4.b Modifying RegIncVRTL.v

When considering how our design will be hardened into a chip, there's one set of connections that we're missing in the module declaration - the power connections! Normally, these are abstracted away when simulating, but when building a real chip, we absolutely need to remember them. When hardening the design, *USE_POWER_PINS* is defined, so we can use that to modify our module interface to include power pins. Navigate to *RegIncVRTL.v* inside *${FLOW_DIR}/verilog/rtl*, and use your favorite text editor to modify the module interface like so:

```
module RegIncVRTL(
`ifdef USE_POWER_PINS
    inout vccd1, // User area 1 1.8V supply
    inout vssd1, // User area 1 digital ground
`endif
    input  wire        clk,
    input  wire        reset,
    input  wire [31:0] a,
    output wire [31:0] b
);
```

4.c Modifying Caravel

We also need to be able to tell Caravel how to harden this new design. The design-specific instructions Caravel uses (including any specific parameters) are located in *${FLOW_DIR}/openlane*, in the design specific folder. First, we need to create a new folder for our design - we can use the example up-counter provided as a template

```
% cd ${FLOW_DIR}/openlane
% cp -r user_proj_example RegIncVRTL
% cd RegIncVRTL
% ls
```

Lets examine the contents of our new folder:

- **base_user_proj_example.sdc** contains the base constraints for our design. This tells us some specifications about the clock and delays that are part of the technology.
- **config.json** contains all of the configuration parameters that Caravel uses when synthesizing your design. This is where physical design comes into play - modifying these parameters is how we modify how Caravel hardens your design. A complete list of possible parameters can be found here - don't get too bogged down with all of them, the defaults are usually fine, but they're good to know about
- **pin_order.cfg** contains details about how the pins should be placed around your design. Your design is hardened into a rectangle, and this file tells us which side of the rectangle, as well as which order they should be in (if it isn't present, then OpenLANE just does whatever it wants). There was no good documentation for this, so I created my own here from things I learned by experimenting. You might consider the location of the final design and the location of the GPIO's when writing this file. Note that we don't handle power here - power rails extend across the entire chip, so we don't have to worry about them being "inputs" anywhere in particular

Alright - let's edit these files so that we can harden our design

4.c.1 Modifying *base_user_proj_example.sdc*

For this file, we don't need to change any of the specifications - we just need to rename it for our new design, instead of after the example project:

```
% mv base_user_proj_example.sdc base_RegIncVRTL.sdc
```

4.c.2 Modifying *pin_order.cfg*

You can edit this however you want to include all of the pins for your design. For this tutorial, I decided I wanted the *clk* and *reset* on the top of the chip, *a* on the left, and *b* on the right - see if you can modify the file to achieve this. Remember to include * if you want all items that match it (Hint: you might want to use *a.** and *b.** to connect all the bits of *a* and *b*)

4.c.3 Modifying *config.tcl*

Use your favorite code editor to examine this file and see how it defines the parameters it wants to use. We want to edit a couple of these:

- **DESIGN_NAME** - should be "RegIncVRTL"
- **VERILOG_FILES** - replace "user_proj_example.v" with "RegIncVRTL.v"
- **CLOCK_PORT** - should be "clk"
- **CLOCK_NET** - should be "clk"

- **BASE_SDC_FILE** - change from "base_user_proj_example.sdc" to our renamed "base_RegIncVRTL.sdc"

The rest of the configurations should be fine for now, although as you get more experienced and start making more designs, you may wish to play around with them

5 Hardening your Design

We should have all the configurations we need to harden our design now! Hardening is the process from going from an RTL design all the way to a final GDS file. A **flow** is the process of how you do this, involving many tools along the way. The OpenLANE flow is detailed below:

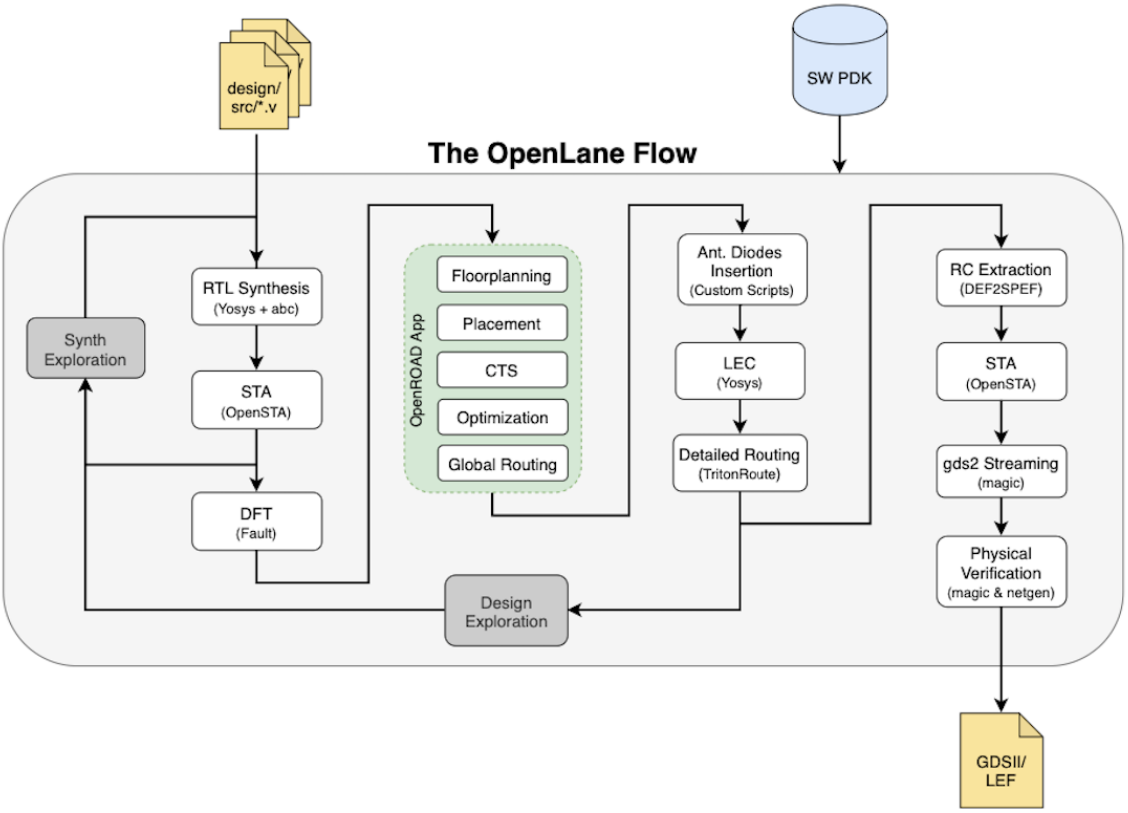

Figure 5: The OpenLANE flow

I won't go over all of these steps here (but will instead point you to this paper or the documentation if you're interested), as it's outside the scope of this tutorial, but I would be happy to go over it with you at some point if you wish. One could run all of the tools one-by-one, but luckily for us, it's all been automated in scripts:

```
% cd ${FLOW_DIR}
% make RegIncVRTL
```

This may take a bit, but at the end, it should be successful (I got some fanout warnings - while those aren't critical right now, it could be an area to improve upon with our physical design parameters!). All of the reports on the design are generated in *${FLOW_DIR}/openlane/RegIncVRTL/runs*, all labeled with when you ran them (although *RegIncVRTL* is a link to the most recent one). Here, you can go through the outputs to get detailed information on exactly how OpenLANE built your project.

However, these reports can be long and hard to navigate this. To combat this, a member of the open-source community named Matt Venn created a Python tool to parse these results for easier viewing, as well as view the generated GDS file (a picture of your design!). This tool has been installed and modified for C2S2, such that you need to export the top-level Caravel directory as *PROJ_ROOT* to use it.

```
% cd ${FLOW_DIR}
% export PROJ_ROOT=${PWD}
% summary.py --design RegIncVRTL --summary
```

If you're in a graphical viewer (such as X2GO or MobaXTerm), you can also use it it view the GDS in 2D (using klayout). You can also view it in 3D, using GDS3D (look at the README.txt inside the repository for instructions on how to use - it doesn't tend to play nicely with remote viewers, so you can either use a physical machine connected to the servers in Phillips 314, or download the .gds file and build GDS3D on your local machine, using the *sky130.txt* techfile to view your design)

```
% summary.py --design RegIncVRTL --gds
% summary.py --design RegIncVRTL --gds-3d
```

You can also view the GDS file with just klayout. The GDS files are generated both in your run directory, as well as in the top-leve *gds* directory. The GDS file can look a little bit weird if klayout doesn't know what it's looking at and tries to interpret it, so we can give it some information with a layer file. Move this file (*caravel.lyp*) to the same directory as your GDS, then run

```
% cd ${FLOW_DIR}/gds
% klayout RegIncVRTL.gds -l caravel.lyp
```

Notice from viewing that there is a *lot* of empty space - see if you can use what you've learned to build your design in a smaller amount of area!

6 Hardening Caravel

You've just hardened your design as a stand-alone block. The final step of design is instantiating this block inside of the Caravel User Space, so that we can compose the entire chip

6.a Modifying your Design

One last thing we need to do is to tell Caravel which pins are inputs and outputs, beyond just telling Verilog. This involves setting specific pins as 1 or 0, depending whether a connection is an input or output (respectively). However, we cannot do this at the top-level, as that only involves routing (for

those familiar with the different layers, it only routes on Metal 5), and can't place standard cells to drive signals. Instead, we need to modify our design to drive these signals appropriately. To do this, we'll create 4 new outputs from our design:

- *clk_en* should be 1 (*clk* is an input)
- *reset_en* should be 1 (*reset* is an input)
- *a_en* (32 bits) should be all 1 (*a* is an input)
- *b_en* (32 bits) should be all 0 (*b* is an output)

We'll re-harden our block with these new details (remember to include these new pins explicitly in your *pin_order.cfg* if they aren't covered by a regex expression)

```
% cd ${FLOW_DIR}
% make RegIncVRTL
```

6.b Instantiating within Caravel

Navigate to `$FLOW_DIR/verilog/rtl` and open up *user_project_wrapper.v*. This contains the entire instantiation of the user project space, so we want to declare our design module within here. First, comment out or delete the design already in there - this is a sample counter design that we won't use. Then, you can declare your module as normal in Verilog (naming the instantiation RegInc). For the connections, the GPIO's are controlled by either *io_in* or *io_out*, depending on the setting of *io_oeb* (1 for input, 0 for output). Caravel has 38 GPIO's (indexed 0 to 37). Therefore, if I wanted to hook up the *clk* signal to GPIO 10, I would instantiate it like this:

```
.clk (io_in[10]),

...

.clk_en (io_oeb[10]),
```

See if you can hook up the rest of the signals! Given the limited number of GPIO's, you might also choose to use the Logic Analyzer connections internal to the chip - there are 128 of them, with a similar syntax to the GPIO's (look at the interface for the user project wrapper - however, we don't need any enable lines for them). For this project, connect *clk* to GPIO 10 (like above), *reset* to GPIO 11, *a* to LA Probes [31:0], and *b* to LA Proves [63:32]. You should also hook up the power pins to their corresponding signals from the *user_project_wrapper* module, as done with the given example project.

6.c Modifying our build configurations

Lastly, we have to modify our build configurations for our overall user project wrapper! Navigate to `$FLOW_DIR/openlane/user_project_wrapper`. Here, we need to edit two files

6.c.1 Modifying *macro.cfg*

Here is the placement details of all of the macros, including the name that they were instantiated with and the distance of their bottom left corner from the bottom left corner of the user space (x distance, then y distance) in μm. We don't need to change the distances for right now, but modify the macro name from mprj to RegInc, as that's what we named our instantiation within the wrapper

6.c.2 Modifying *config.json*

Here, we have a lot of changes. We can first off change any reference from "user_proj_example" to "RecIncVRTL" (this should change a total of 3 things - the black-box Verilog files, the extra LEF's and the extra GDS's). In addition, change any reference from "mprj" to "RegInc" (this should change a total of 2 things - the name of the clock net, as well as the Macro power connections). Lastly, we need to change which port OpenLANE analyzes as our clock port. On the line where we set "CLOCK_PORT" (the pin that's used as the clock), change "user_clock2" to "io_in.10" (the name of GPIO 10 that Caravel recognizes)

With that, we should be good to go! You can harden your design similar to before

```
% cd ${FLOW_DIR}
% make user_project_wrapper
```

This should generate similar files to before - see if you can view the results of your overall user_project_area GDS! (This flow should be a fair amount quicker - the flow already knows everything it needs to about our *RegIncVRTL* block from the Verilog, LEF, and GDS file, so it doesn't need to push it through the flow again, giving us the advantages of incremental, hierarchal design)

7 Testing our Design

Having our design is good, but being able to test it is the key to having a functional product. In Caravel, we can do 4-state testing based on the Verilog, as well as gate-level testing based on the netlist that Yosys produces. OUr testing doesn't actually depend on the results of the flow (as it runs LVS to make sure that the result is equivalent to the input). All of our tests are kept in the folder `$FLOW_DIR/verilog/dv`. Here, we can see that they gave us a couple of sample tests - these are to test the counter, so we won't be using them. I've created an adhoc test directory here - let's clone it into this directory

```
% cd ${FLOW_DIR}/verilog/dv
% git clone git@github.com:cornell-c2s2/reginc_adhoc_test.git
```

We can see that this test directory (like all of the directories given) contains 3 files:

- **Makefile**: This specifies how Caravel will build and simulate our files - we don't need to touch this
- **reginc_adhoc_test.c**: This contains the C code that will be run on our chip!!

- **reginc_adhoc_test_tb.v**: This contains the Verilog code that will instantiate our Caravel module (a.k.a. our entire chip) and provides stimuli from outside the chip

I've done my best to try and comment the code to best explain it, so take a look through and see if you can understand what the test is doing, and let me know if you have any confusions! (Note that if you used different inputs than those I specified, you may have to modify the test a bit to account for it)

7.a Modifying test includes

The only other thing we need to do is modify what code that our test runs on - our test instantiates Caravel, but we may have hardened several versions, so which one do we use? These are contained in `$FLOW_DIR/verilog/includes`. Navigate here and view the files:

```
% cd ${FLOW_DIR}/verilog/includes
% ls
```

Here, we can see that there are a bunch of files that tell us what hardware to include, depending on which type of test we're running. We'll only use the 4-state RTL tests (with the includes in `includes.rtl.caravel_user_project`) and the gate-level tests (with the includes in `includes.gl.caravel_user_project`). For now, we'll just modify the includes for the RTL test - open up `includes.rtl.caravel_user_project`, and notice how the files are included. Change "user_proj_example" to "RegIncVRTL".

With that, we should be good to go for testing our design! Navigate to the top level:

```
% cd ${FLOW_DIR}
```

Our testing command is of the format `make verify-<test_name>-<test_type>`, where

- <**test_name**> is the name of the test directory that we're testing
- <**test_type**> is the type of test that we're running - either "rtl" or "gl"

With this, we can run our RTL test on our Verilog design

```
% make verify-reginc_adhoc_test-rtl
```

If everything was set up correctly, this test should pass. If you look back in the test directory, you can find a waveform of the test as it was run on your chip - feel free to open it up and explore it in GTKWave (assuming you are using a graphical viewer or are using X11 Forwarding):

```
% cd ${FLOW_DIR}/verilog/dv/reginc_adhoc_test/
% gtkwave RTL-reginc_adhoc_test.vcd
```

After hardning your design from before, we also have a gate-level netlist - see if you can test your design using a gate-level test, knowing our setup from before (don't forget to change the includes!). This will take a LOT longer (as we're simulating the signals propagating through the actual gates on the chip), but in the end, the result should be the same.

That wraps it up for this tutorial - hopefully you've learned a lot, including how to harden and test your designs in the OpenLANE flow. As always, feel free to reach out with any questions!