ECE 5745 Complex Digital ASIC Design, Spring 2020

Lab 1: Pipelined Integer Multiplier

School of Electrical and Computer Engineering
Cornell University

revision: 2020-02-13-00-00

The first lab assignment is a warmup lab where you will use a standard-cell-based ASIC toolflow to quantitatively analyze the area, energy, and performance of various integer multiplier designs. You will push three given baseline designs through the ASIC toolflow: a fixed-latency iterative design that always takes the same number of cycles, a variable-latency iterative design that exploits properties of the input operands to reduce the execution time, and a simple single-cycle design. The alternative design is a pipelined integer multiplier that is parameterized by the number of pipeline stages. You are required to implement both a register-transfer-level (RTL) model of the alternative design, verify the design using an effective testing strategy, push all designs through the ASIC toolflow, and perform an evaluation comparing the various implementations. **As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with a state-of-the-art standard-cell-based ASIC toolflow for quantitatively analyzing the area, energy, and timing of a design. Your experience in this lab assignment will create a solid foundation for completing the rest of the lab assignments and the five-week design project.

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. To get started, login to an `ecelinux` machine, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece5745.sh
% mkdir -p ${HOME}/ece5745
% cd ${HOME}/ece5745
% git clone git@github.com:cornell-ece5745/lab1-groupXX.git
```

where `XX` is your group number. **You should never fork your lab group's remote repository! If you need to work in isolation then use a branch within your lab group's remote repository.** If you have already cloned your lab group's remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece5745/lab1-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% py.test ../lab1_imul
```

All of the tests should pass except for the tests related to the multipliers you will be implementing in this lab. For this lab you will be working in the `lab1_imul` subproject which includes the following files:

- `IntMulFL.py` – FL multiplier
- `IntMulFixedLatPRTL.py` – PyMTL RTL fixed-latency iterative multiplier
- `IntMulFixedLatVRTL.v` – Verilog RTL fixed-latency iterative multiplier
- `IntMulFixedLatRTL.py` – Wrapper to choose which RTL language
- `IntMulVarLatCalcShamtPRTL.py` – PyMTL RTL model to calculate shift amount
- `IntMulVarLatCalcShamtVRTL.v` – Verilog RTL model to calculate shift amount
- `IntMulVarLatCalcShamtRTL.py` – Wrapper to choose which RTL language
- `IntMulVarLatPRTL.py` – PyMTL RTL variable-latency iterative multiplier
- `IntMulVarLatVRTL.v` – Verilog RTL variable-latency iterative multiplier
- `IntMulVarLatRTL.py` – Wrapper to choose which RTL language
- `IntMulScyclePRTL.py` – PyMTL RTL single-cycle multiplier
- `IntMulScycleVRTL.v` – Verilog RTL single-cycle multiplier
- `IntMulScycleRTL.py` – Wrapper to choose which RTL language
- `IntMulNstagePRTL.py` – PyMTL RTL pipelined multiplier
- `IntMulNstageVRTL.v` – Verilog RTL pipelined multiplier
- `IntMulNstageRTL.py` – Wrapper to choose which RTL language
- `imul-sim` – Multiplier simulator for evalution
- `__init__.py` – Package setup
- `test/IntMulFL_test.py` – FL multiplier unit tests
- `test/IntMulScycleRTL_test.py` – RTL single-cycle multiplier unit tests
- `test/IntMulFixedLatRTL_test.py` – RTL fixed-latency iterative multiplier unit tests
- `test/IntMulVarLatCalcShamtRTL_test.py` – RTL calculate shift amount unit tests
- `test/IntMulVarLatRTL_test.py` – RTL variable-latency iterative multiplier unit tests
- `test/IntMulNstageRTL_test.py` – RTL pipelined multiplier unit tests
- `test/imul_sim_test.py` – Test to make sure multiplier simulator works
- `test/__init__.py` – Test package setup

1. Introduction

In ECE 4750, you learned how to evaluate baseline and alternative designs according to several key metrics: execution time (in cycles), area, energy, and cycle time. Using RTL modeling, we were able to quantitatively measure the execution time (in cycles) for a given input dataset for each design, but we relied on qualitative estimates to analyze the the area, energy, and cycle time of each design. In this lab, we will learn how to use a standard-cell-based ASIC toolflow to quantitatively measure the area, energy, and cycle time of a design; we can then combine these measurements with our understanding of the cycle-level performance of each design to conduct a much more detailed design-space exploration.

We have provided you with a functional-level model of an integer multiplier. You can find this functional-level (FL) implementation in `IntMulFL.py` and the associated unit tests in `test/IntMulFL_test.py`. This implementation verifies the functionality and uses a higher-level modeling style compared to register-transfer-level (RTL) modeling. The interface for the FL model and indeed for all of the integer multiplier designs is similar in that they accept two 32-bit numbers and produce a single 32-bit multiplication result. All implementations should treat the input operands and the result as two's complement numbers and thus should be able to handle both signed and unsigned multiplication. The FL model is a simple Python function and the RTL model uses port-based interfaces and the en/rdy micro-protocol. By leveraging the en/rdy micro-protocol, another module would be able to

```
1  def imul_fixed_algo( a, b ):
2
3    result = Bits( 32, 0 )
4    for i in range(32):
5      if b[0] == 1:
6        result += a
7      a = a << 1
8      b = b >> 1
9
10   return result
```

Figure 1: Fixed-Latency Iterative Multiplication Algorithm – Assumes `a` and `b` are 32-bit `Bits` objects. Always uses 32 shifts and subtractions to calculate the partial-products over time. This is executable Python code.

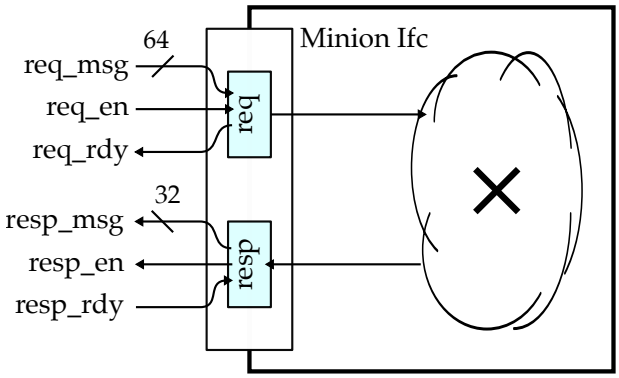

Figure 2: Cloud Diagram for Fixed-Latency Iterative Integer Multiplier RTL Model – RTL model uses port-based interfaces leveraging en/rdy micro-protocol.

send request messages to the multiplier and never explicitly be concerned with how many cycles the implementation takes to execute a multiply transaction and return the response message.

2. Baseline Design

There are three baseline designs for this lab assignment: a fixed-latency iterative integer multiplier, a variable-latency iterative multiplier, and a simple single-cycle multiplier. The RTL models of each design are provided for you.

2.1. Fixed-Latency Iterative Integer Multiplier

The fixed-latency iterative multiplier will always take approximately 32 cycles. Figure 1 illustrates the fixed-latency iterative multiplication algorithm using "pseudocode" which is really executable Python code. Each iteration we check the least significant bit of the `b` operand; if this bit is zero then we shift the `b` operand to right and the `a` operand to the left, but if this bit is one then we add `a` to the result before shifting. Each iteration is essentially calculating a partial product for the multiplication. Note that we say "approximately" because there may be a few extra cycles of overhead in handling the en/rdy micro-protocol (although a more optimized design can remove this overhead).

The interface for the fixed-latency iterative multiplier is show in Figure 2. The fixed-latency iterative multiplier RTL model uses a control/datapath split. The datapath for the fixed-latency design is shown in Figure 3. The blue signals represent control/status signals for communicating between the datapath and the control unit. The datapath model uses a structural design style by instantiating a child module for each of the blocks in the datapath diagram. The control unit for the fixed-latency design uses the simple finite-state-machine shown in Figure 4. The IDLE state is responsible for consuming the message from the input interface and placing the input operands into the input registers; the CALC state is responsible for iteratively using adds and shifts to calculate the multiplication; and the DONE state is responsible for sending the message out the output interface. Note that if you implement the FSM exactly as shown each multiply should take 35 cycles: one for the IDLE state, 32 for iterative calculation, one cycle to realize the calculation is done, and one cycle for the DONE state. The extra cycle to realize the calculation is done is because we are not using Mealy transitions from the CALC to DONE states. We need to wait for the counter to reach 32 and then we move into the idle state. The control unit is structured into three parts: a sequential concurrent block for just the

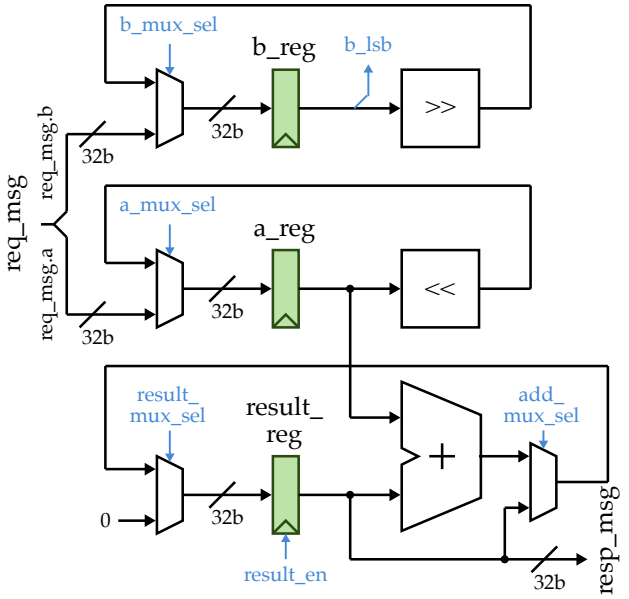

Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier – All datpath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

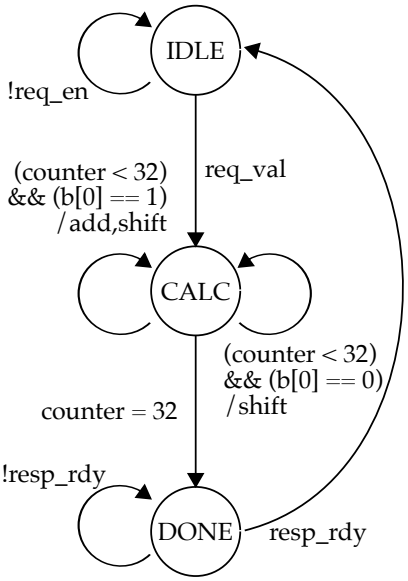

Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier – Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

state element, a combinational concurrent block for state transitions, and a combinational concurrent block for state outputs. The PyMTL code for the fixed-latency iterative multiplier RTL model is in `IntMulFixedLatPRTL.py` and Verilog code is in `IntMulFixedLatVRTL.v`.

2.2. Variable-Latency Iterative Integer Multiplier

The fixed-latency iterative integer multiplier will always take 35 cycles to compute the result. While it is possible to optimize away the cycle to realize the calculation is done and to eliminate the IDLE/-DONE states, fundamentally this algorithm is limited by the 32 cycles required for the iterative calculation. The variable-latency iterative multiplier takes advantage of the structure in some pairs of input operands to improve performance and energy efficiency. Figure 5 illustrates the variable-latency iterative multiplication algorithm using "pseudocode". If the `b` operand has many consecutive zeros we don't need to shift one bit per cycle; instead we can shift the B register multiple bits in one step and directly jump to the next required addition. The `calc_shamt` function calculates the shift amount based on the number of trailing zeros in the `b` operand. Various different implementations of `calc_shamt` are possible: considering more bits will improve the performance but likely increase area and energy.

The variable-latency iterative multiplier RTL model is very similar to the fixed-latency RTL model with a couple key differences. The datapath for the variable-latency design is shown in Figure 6. Notice that we have added a new model that takes the `b` operand and input and calculates the variable shift amount for both the left and right shifters. The PyMTL and Verilog code for this new model is in `IntMulVarLatCalcShamtPRTL.py` and `IntMulVarLatCalcShamtVRTL.v` respectively. Refactoring this logic into a separate model enables unit testing the logic before integrating it into the overall design. The control unit for the variable latency design uses the simple finite-state-machine shown in Figure 7. The only difference from the fixed-latency design is that we finish the calculation when

```
1  def imul_var_algo( a, b ):
2
3    result = Bits( 32, 0 )
4    while b != 0:
5      if b[0] == 1:
6        result += a
7      shamt = calc_shamt( b )
8      a = a << shamt
9      b = b >> shamt
10
11   return result
```

Figure 5: Variable-Latency Iterative Multiplication Algorithm – Assumes `a` and `b` are 32-bit `Bits` objects. Shifts by more than one to skip over sequences of zeros in the `b` operand. Various different `calc_shift` functions are possible. This is executable Python code.

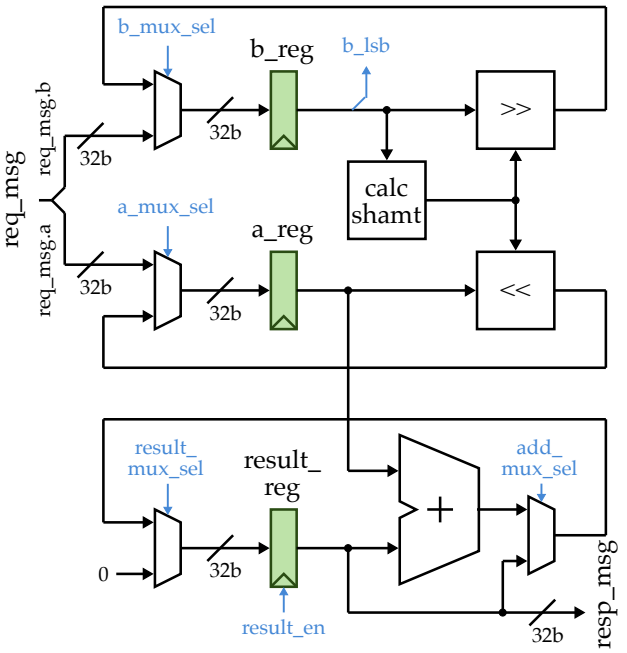

Figure 6: Datapath for Variable-Latency Iterative Integer Multiplier – All datpath components are 32-bits wide except for the shift amount signal to the variable shifters.

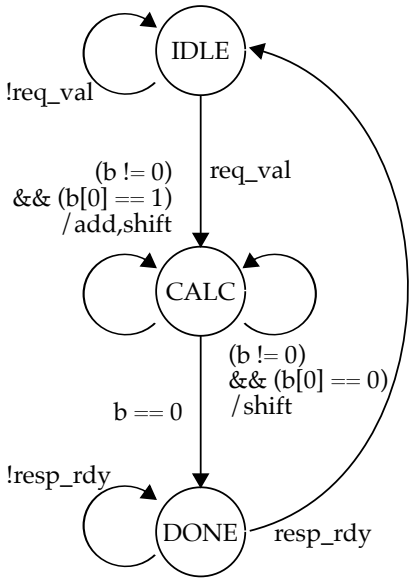

Figure 7: Control FSM for Variable-Latency Iterative Integer Multiplier – Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

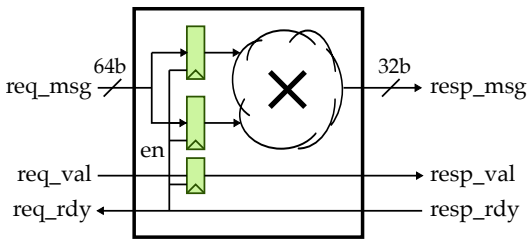

Figure 8: Single-Cycle Integer Multiplier – We use registered inputs and combintionally connect the response ready signal to the request ready and the input register enable signals.

the `b` operand is zero. The PyMTL and Verilog code for the variable-latency iterative multiplier RTL model is in `IntMulVarLatPRTL.py` and `IntMulVarLatVRTL.v` respectively.

2.3. Single-Cycle Integer Multiplier

While the iterative multipliers will likely require minimal area, they also require many cycles to calculate each result. As a point of comparison, we will also consider a simple single-cycle integer multiplier. Figure 8 shows the single-cycle RTL model. We use registered inputs for both the operands and the valid bit. If the response interface is not ready, then we stall the multiplier by disabling the register enable signals and combinationally propagating the response ready signal to the request ready signal. For the actual multiplier we simply use the `*` operator; we will rely on the ASIC toolflow to choose the most appropriate multiplication hardware. The PyMTL and Verilog code for the single-cycle multiplier RTL model is in `IntMulScyclePRTL.py` and `IntMulScycleVRTL.v` respectively.

3. Alternative Design

For your alternative design, you should implement a pipelined integer multiplier with the same interfaces used in the baseline designs. You should implement an RTL model that is parameterized by the number of pipeline stages; you can assume that the number of stages will be either 1, 2, 4, or 8, although you can also support 16 and 32 stages if you like. As discussed in the tutorials, in this course we will adopt the general policy of using registered inputs for larger blocks, so your pipelined multiplier should start with a set of pipeline registers.

The pipelined integer multiplier RTL model should use a child model that implements a single "partial product step" of the fixed-latency iterative multiplication algorithm. Each partial product step would involve three inputs: one for `a`, `b`, and `result` and three outputs also for `a`, `b`, and `result`. Each partial product step will perform two shifts, an addition, and then mux the result of the addition conditionally based on the least-significant bit of the `b` input. As always, you should unit test the partial product step model, and potentially even build a composition of a few steps for integration testing. Once you are confident in your step model, implement the parameterized RTL model by using elaboration to insert the pipeline registers between the steps. Your pipelined multiplier should always include a total of 32 steps. What changes is just where you put the pipeline registers. If done correctly, your pipelined multiplier should be able to sustain one multiply transaction per cycle. Note that there are many optimizations you might want to try that may lead to higher quality hardware; you should feel free to explore different approaches if you like.

4. Testing Strategy

We have provided you with all of the unit tests for the integer multipliers, although you will need to add your own test scripts for any new models you create. You are also certainly welcome to add more tests for your alternative design if you would like. Most of the unit tests are defined in `test/IntMulFL_test.py`; the other test scripts reuse these unit tests as much as possible. The following commands illustrate how to run all of the tests for the entire project, how to run just the tests for this lab, and how to run just the tests for the FL model, and fixed-latency RTL model. You likely want a test that verifies your pipelined N-stage multiplier is in fact pipelining the multiplications with a cycle-accurate test. Note, you may have to create new testing modules to validate cycle accuracy within a test.

```
% cd ${HOME}/ece5745/lab1-groupXX/sim/build
% py.test ..
% py.test ../lab1_imul
% py.test ../lab1_imul/test/IntMulFL_test.py --verbose
```

```
% py.test ../lab1_imul/test/IntMulFixedLatRTL_test.py --verbose
```

Once your alternative design passes all of the provided tests, then you should verify that your RTL model can be successfully translated using the following command:

```
% cd ${HOME}/ece5745/lab1-groupXX/sim/build
% py.test ../lab1_imul/test/IntMulNstageRTL_test.py --test-verilog
```

5. Evaluation

Once you have verified the functionality of your alternative design, you should then use the provided simulator to evaluate the cycle-level performance of the various multiplier designs. You can run the simulator for the first baseline design like this:

```
% cd ${HOME}/ece5745/lab1-groupXX/sim/build
% ../lab1_imul/imul-sim --impl rtl-fixed --input small --stats
```

The simulator will display the total number of cycles to execute the specified input dataset. You should study the line traces (with the `--trace` option) and possibly the waveforms (with the `--dump-vcd` option) to understand the reason why each design performs as it does on the various patterns. For your pipelined multiplier, you can use the `--nstages` option to set the number of pipeline stages. Be careful to factor out the pipeline startup overhead for your pipelined multipliers for a fair comparison.

Once you have explored the cycle-level performance of the various multiplier designs, you should push the RTL models through the standard-cell-based ASIC toolflow to quantify the area, energy, and timing of each design. We would like you to push the following designs through the flow:

- Fixed-latency iterative multiplier
- Variable-latency iterative multiplier
- Single-cycle multiplier
- Pipelined multiplier with one stage
- Pipelined multiplier with two stages
- Pipelined multiplier with four stages
- Pipelined multiplier with eight stages

For each design we would like to analyze the power/energy when executing three specific input datasets: `small` (small random numbers), `large` (large random numbers), and `sparse` (higher probability of zero bits in the input operands). You can generate the Verilog and VCD files for a given design and input dataset using the simulator like this:

```
% cd ${HOME}/ece5745/lab1-groupXX/sim/build
% ../lab1_imul/imul-sim --impl rtl-fixed --input small --translate --dump-vcd
```

This will generate a Verilog file named `lab1_imul_IntMulFixedLatRTL` along with a VCD file named `lab1_imul_IntMulFixedLatRTL.verilator1.vcd`. To push design down the ASIC flow, an appropriate configuration directory has to be created with correct values. Configuration directories live under the `designs` directory inside the `asic` directory. For your convenience, a configuration folder has been created for all possible design points for the first lab. Here is a mapping between the different configuration directories and corresponding designs:

```
% cd ${HOME}/ece5745/lab1-groupXX/asic/designs
```

```
% ls
 GcdUnit
 IntMulFixedLatRTL
 IntMulVarLatRTL
 IntMulScycleRTL
 IntMulNstagesRTL_1stages
 IntMulNstagesRTL_2stages
 IntMulNstagesRTL_4stages
 IntMulNstagesRTL_8stages
```

- **GcdUnit**: Example greater common divisor unit
- **IntMulFixedLatRTL**: Fixed-latency iterative multiplier
- **IntMulVarLatRTL**: Variable-latency iterative multiplier
- **IntMulScycleRTL**: Single-cycle multiplier
- **IntMulNstagesRTL_1stages**: Pipelined multiplier with one stage
- **IntMulNstagesRTL_2stages**: Pipelined multiplier with two stages
- **IntMulNstagesRTL_4stages**: Pipelined multiplier with four stages
- **IntMulNstagesRTL_8stages**: Pipelined multiplier with eight stages

To run the ASIC flow on a given design, create a build directory as follows:

```
% cd ${HOME}/ece5745/lab1-groupXX/asic
% rm -rf build-fixed
% mkdir build-fixed
% cd build-fixed
```

Then, you can configure the build directory to run the ASIC flow on a specific design:

```
% ../configure --design ../designs/IntMulFixedLatRTL
```

Note that the value passed as the "design" would be the directory that holds the configuration for the target design. **Remember that every design must meet timing after place-and-route!** For your convenience, you can create a build directory for different runs (i.e., different designs and different constraints). Having multiple runs on hand can help you explore the design space better more efficiently since you can refer to reports and design parameters when comparing with other runs.

Be sure to look at the reports from each run to analyze your results, especially the timing and area reports. Also confirm that your results seem reasonable in case part of the design was aborted by the standard cell ASIC flow since it can be difficult to see when using the automated scripts. In addition to tables and bar plots, you should include several energy vs. performance plots similar to what we use in lecture to compare the various designs. We recommend having one energy vs. performance plot per input dataset; so plot the results for all of the different microarchitectures executing this dataset on one plot, and then create another plot for a different input dataset. These energy vs. performance plots should enable you to provide deep insight into the various trade-offs in the evaluation section of your report.

By default, the ASIC flow is setup to preserve the design hierarchy. In other words, if you have a specific module A in your RTL, then module A will still be there in the final gate-level netlist. Preserving the design hierarchy makes it easier to interpret the area, energy, and timing reports, but it also prevents the tools from performing cross-module-boundary optimizations. After pushing your pipelined multiplier through the flow with the default setup, you should also modify the flow to *flatten* the design hierarchy. Flattening removes all of the module boundaries and allows the tools the

full freedom to do more wholistic optimization, but the trade-off is that the area, energy, and timing results can be much more difficult to interpret. After pushing their pipelined multiplier through the flow without flattening, students should also study the impact of flattening by declaring the following environment:

```
% export flatten_effort=3
```

Then do a clean build (i.e., create a new build directory and push the design through the flow from scratch). Include and discuss both the unflattened and flattened results in the evaluation section of your report. By default, the flattening effort is set to zero. The maximum effort for Synopsys Design Compiler is 3 (in integer increments). The meaning of each flattening effort is as follows:

- **Effort 0 —** No auto-ungrouping / boundary optimizations (strict hierarchy).
- **Effort 1 —** No auto-ungrouping / boundary optimizations, but DesignWare cells are ungrouped.
- **Effort 2 —** Enable auto-ungrouping / boundary optimizations.
- **Effort 3 —** Everything ungrouped + level param for how deep to ungroup.

Acknowledgments

This lab was created by Christopher Batten, Khalid Al-Hawaj, Jason Setter, and Derek Lockhart as part of the course ECE 5745 Complex Digital ASIC Design at Cornell University.