ECE 4750 Computer Architecture, Fall 2023
Lab 1: Iterative Integer Multiplier

School of Electrical and Computer Engineering
Cornell University

revision: 2023-08-27-07-26

The first lab assignment is a warm-up, where you will design two implementations of an integer iterative multiplier: the baseline design is a fixed-latency implementation that always takes the same number of cycles, and the alternative design is a variable-latency implementation that exploits properties of the input operands to reduce the execution time. You are required to implement the baseline and alternative designs, verify the designs using an effective testing strategy, and perform an evaluation comparing the two implementations.

This lab is designed to give you experience with:

- the Verilog hardware modeling framework;
- abstraction levels, including functional- and register-transfer-level modeling;
- design principles, including modularity, hierarchy, and encapsulation;
- design patterns, including message interfaces, control/datapath split, and FSM control;
- agile design methodologies including incremental development and test-driven development.

Your experience in this lab assignment will create a solid foundation for completing the rest of the lab assignments, ultimately culminating in the implementation of a complete multicore processor.

1. Setup

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. To get started, log in to an `ecelinux` machine (you will need a graphical interface, such as MobaXTerm or X2Go, to initially download the files) and source the setup script using the command below:

```
% source setup-ece4750.sh
```

You should see your prompt change to a blue "ECE4750", indicating that the setup script was successfully sourced. **You will need to source this setup script each time that you log in when doing work for ECE 4750**. If you wish for the setup script to be automatically sourced for you on log-in, you can use the following command (note the additional flag):

```
% source setup-ece4750.sh --enable-auto-setup
```

Try logging out, and back in; you should see the modified prompt to indicate that the script was sourced for you. If at any point you want to disable this feature, run:

```
% source setup-ece4750.sh --disable-auto-setup
```

To begin the lab, download the starter files from the course website and extract it into a directory titled `sim` with the following commands:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750

# download the file to this folder from Canvas

% tar -xvzf lab1.tar.gz
% cd sim
% make setup
```

For this lab you will be working in the `lab1_imul` subproject which includes the following files in the `sim` directory:

- `Makefile` – Makefile to run the Verilog simulator
- `verilator.cpp` – C++ harness for the Verilog simulator
- `vc` – Common hardware components, for use in structural implementations
- `lab1_imul` – The main project files for Lab 1
- `- IntMulSimple.v` – Verilog RTL Single-Cycle multiplier
- `- IntMulBase.v` – Verilog RTL fixed-latency multiplier
- `- IntMulAlt.v` – Verilog RTL variable-latency multiplier
- `- tb_IntMul.v` – Simple Testbench for Verilog RTL
- `- ub_IntMul.v` – Simple Microbenchmark for Verilog RTL single-cycle multiplier
- `- default.config` – Config file for Makefile to allow testbench reuse.

2. Introduction

You learned about basic digital logic design in your previous coursework, and in this warm-up lab we will put this knowledge into practice by building multiple implementations of an integer multiplier. Although it is certainly possible to design a single-cycle combinational integer multiplier as shown in `tb_IntMulSimple.v`, such an implementation will likely be on the critical path in an aggressive design. Later in the course, we will learn about pipelining as a technique to improve cycle time (i.e., clock frequency) while maintaining high throughput, but in this lab, we take a different approach. Our designs will iteratively calculate the multiplication using a series of add and shift operations. The baseline design always uses a fixed number of steps, while the variable latency design is able to improve performance by exploiting structure in the input operands. The iterative approach will enable improved cycle time compared to a single-cycle design, but at reduced throughput (as measured in average cycles per transaction).

We have provided you with a single-cycle algorithmic model of an integer multiplier, as shown in Figure 1. You can find this algorithmic implementation in `IntMulSimple.v` and the associated testbench in `tb_IntMul.v`. This implementation always takes a single-cycle and uses a higher-level algorithmic modeling style and is dependent on the compiler to convert it to logical gates. This will be different when compared to the register-transfer-level (RTL) modeling you will be using in your designs to divided the multiplier across multiple cycles. These varying levels of modeling (i.e., algorithmic versus RTL) are an example of *abstraction*. The interfaces for all three designs (i.e., single-cycle algorithmic model, fixed-latency RTL model, and variable-latency RTL model) are identical. Each multiplier should take as input a 64-bit message with two fields containing the 32-bit operands. 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. In addition, both the input and output interface use the val/rdy microprotocol to control when new inputs can be sent to the multiplier and when the multiplier has a new result ready to be consumed. This is an example of the *encapsulation design principle* in general, and more specifically, the latency-insensitive *message interface design pattern*. We hide implementation details (i.e., the multiplier latency) from the interface using the val/rdy microprotocol. Another module should be able to send messages to the multiplier and never explicitly be concerned with how many cycles the implementation takes to execute a multiply transaction and return the result message.

3. Baseline Design

The baseline design for this lab assignment is a fixed-latency iterative integer multiplier. As with all of the baseline designs in this course, we provide sufficient details in the lab handout such that your primary focus is simply on implementing the design. Figure 2 illustrates the iterative multiplication algorithm using "pseudocode" (which is really executable Python code). Try out this algorithm on your own and make sure you understand how it works before starting to implement the baseline design. Note that, while this Python code will work fine with positive integers, it will produce what looks like very large numbers when multiplying negative numbers. This is due to the fact that Python provides infinitely sized integers, with negative numbers being represented in two's complement with an infinite number of ones extending towards the most-significant bit. It is relatively straightforward to handle negative numbers in Python by explicitly checking the sign-bit and adding some additional masking and two's complement conversion logic. Since the real hardware will not need to do this, it is not shown in the algorithm.

We will be decomposing the baseline design into two separate modules: the datapath which has paths for moving data through various arithmetic blocks, muxes, and registers; and the control unit, which is in charge of managing the movement of data through the datapath. This decomposition is an example of the *modular design principle* in general, and more specifically the *control/datapath split design pattern*. Your datapath module, control unit module, and the parent module that connects datapath and control unit together should all be placed in a single file (i.e., `IntMulBase.v`).

The datapath for the baseline design is shown in Figure 3. The blue signals represent control/status signals for communicating between the datapath and the control unit. Your datapath module should instantiate a child module for each of the blocks in the datapath diagram; in other words, you must use a structural design style in the datapath. Although you are free to develop your own modules to use in the datapath, we recommend using the ones provided for you in `vc` folder.

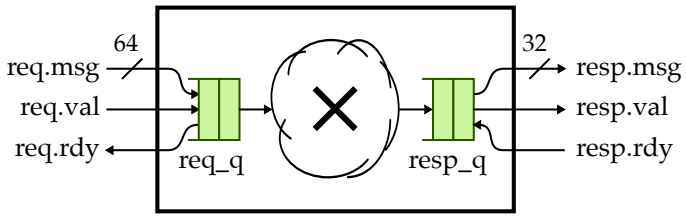

Figure 1: Functional-Level Implementation of Integer Multplier – Input and output use latency-insensitive val/rdy interfaces. The input message includes two 32-bit operands; output message is a 32-bit result. Clock and reset signals are not shown.

```
1 def imul( a, b ):
2   result = 0
3   for i in range(32):
4     if b & 0x1 == 1:
5       result += a
6     a = a << 1
7     b = b >> 1
8   return result
```

Figure 2: Iterative Multiplication Algorithm – Iteratively use shifts and subtractions to calculate the partial-products over time.

Note that while you should implement the datapath by structurally composing child modules, those child modules themselves can simply use RTL modeling. For example, you do not need to explicitly model an adder with gates; simply instantiate an adder module that uses the + operator in a combinational concurrent block. Again, in your final design, there should be a one-to-one correspondence between the datapath diagram and the structural implementation in your code. This recursive modular decomposition exemplifies the *hierarchy design principle*.

The control unit for the baseline design should use 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 through 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. It is relatively straightforward to eliminate this extra bubble (although not required). Your control unit should be structured into three parts: a sequential concurrent block for just the state element, a combinational concurrent block for state transitions, and a combinational concurrent block for state outputs. You will probably want to use a counter to keep track of the 32 cycles required for the iterative calculation in the CALC state. The control unit for your iterative multiplier is an example of the *finite-state-machine design pattern*.

You may want to consider implementing a simpler version of the fixed-latency integer multiplier before trying to implement the fully featured design. For example, you could implement a version of the multiplier that does not handle the val/rdy signals correctly and test this without random delays.

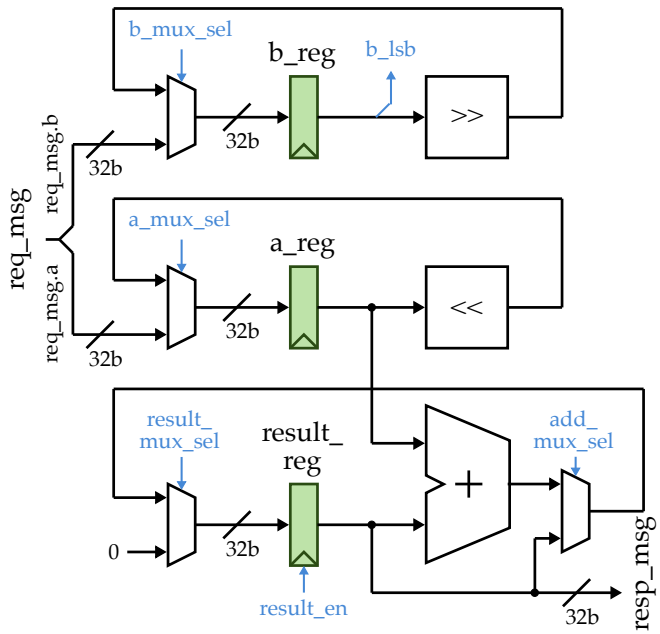

Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier – All datpath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

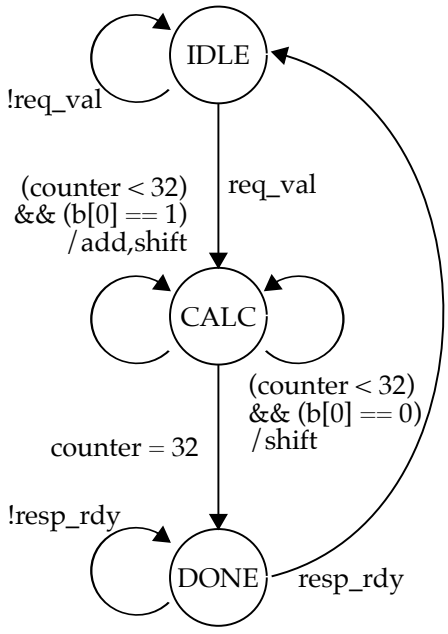

Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier – Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

Once you have the simpler version working and passing the tests, then you could add the additional complexity required to handle the val/rdy signals correctly. This is an example of an *incremental development design methodology*.

4. Alternative Design

The fixed-latency iterative integer multiplier should always take 34-35 cycles to compute the result. While it is possible to design more optimized variants, fundamentally, this algorithm is limited by the 32 cycles required for the iterative calculation. For your alternative design, you should implement a variable latency iterative multiplier, which takes advantage of the structure in some pairs of input operands. More specifically, if an input 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. You should try to be reasonably aggressive in terms of improving the performance, but do not try to squeeze too much logic into a single clock period. Your critical path should not be longer than an adder plus some muxing, or a shifter and some muxing. A critical path with an adder, a shifter, and some muxing might be acceptable, but this will certainly result in a longer cycle time (i.e., lower clock frequency), so you must discuss these trade-offs qualitatively in your signoffs and in your lab report.

As with all of the alternative designs in this course, we simply sketch out the requirements; you are responsible for both the design and the actual implementation. Your implementation should leverage the design principles, patterns, and methodologies you used for the baseline design. We have provided the top-level module interface for your variable latency multiplier design in the file in `IntMulAlt.v` (for Verilog). Most of your code for the alternative design should be placed in these files. We strongly suggest refactoring any logic to check for patterns in the inputs into its own file/-model and writing additional test benches so that you can unit test it.

5. Testing Strategy

For this lab, we provide you with a basic testbench. The testbench instantiates your design, and carefully manipulates the *val/rdy* interface to send message into and receive messages from the multiplier (which are then checked against the expected result). Because both our single-cycle, baseline and alternative designs use the same interface, our testbench can be re-used to test both designs, highlighting their *modularity*.

We provide you with a series of directed tests, and you are responsible for developing the rest of the verification. Writing tests is one of the most important and challenging parts of computer architecture. Designers often spend far more time designing tests than they do designing the actual hardware. You will want to initially write tests using the single-cycle design. Once these tests are working on the single-cycle design, you can move on to testing the baseline and alternative designs. Our emphasis on testing and, more specifically, on writing the tests first in order to motivate a specific implementation, is an example of the *test-driven design methodology*.

The following commands illustrate how to run the tests for this lab, including how to indicate to run the tests for the single-cycle design, baseline design, or alternative design.

```
% cd ${HOME}/ece4750/sim/lab1_imul/
% make tb_IntMul.v DESIGN=${DESIGN} [RUN_ARG=--trace] [COVERAGE=${COVERAGE}]

# $DESIGN options: IntMulSimple IntMulBase IntMulAlt
# $COVERAGE options: --coverage --coverage-line --coverage-toggle
```

```
# [] indicate optional arguments
```

Writing testbenches are an essential part of the design process to ensure correctness in RTL design. As part of the assignment, you will add your directed tests to `tb_IntMul` to provide full statement and toggle coverage. While many other types of coverage are used in Verilog, such as path, expression, FSM transition, et al., for this class, we will focus mainly on **statement and toggle coverage**.

Statement coverage, also known as line coverage, is a metric on the percentage of statements being executed by your testbench over the total number in your DUT. We expect all student-written testbenches to reach 100% coverage in this category, and we consider it to be the bare minimum for RTL verification. For students who use default cases to express code paths that should never occur, we request that they be marked with `$stop` to exclude them from coverage computations. Conversely, reaching 100% coverage statement coverage doesn't imply you can stop testing, but rather this is your starting point, as this metric is neither exhaustive nor able to measure the different paths within a block and the interactions across multiple blocks.

Toggle coverage is another metric we focus on in this class; complete toggle coverage simply means all wires and regs went from low to high and high to low during the testing. It does not imply that every combination of values across the design has been tested, but simply that each individual wire or register has toggled both ways at least once.

Generating coverage report is an important step in accessing the quality of your testbench, and you can generate the coverage report with the following commands:

```
% cd ${HOME}/ece4750/sim/lab1_imul/
% make tb_IntMul.v DESIGN=${DESIGN} COVERAGE=${COVERAGE}
% make utb_IntMulBase.v COVERAGE=${COVERAGE}
# File not inculded in starter code, representing  a student added unit testbench
% make coverage-report
```

You are also able to generate coverage report across all your testbenches and unit testbenches by running the following commands:

```
% cd ${HOME}/ece4750/sim/lab1_imul/
% make clean
% make run-all COVERAGE=${COVERAGE}
```

After executing either of the commands, you will be able to view the html coverage reports in the `sim/lab1_imul/log/${COVERAGE}/html` folder.

You will be adding many more test cases. Do not just make the given test cases larger. Some suggestions for what you might want to test are listed below.

- Combinations of multiplying zero, one, and negative one
- Small negative numbers $\times$ small positive numbers
- Small positive numbers $\times$ small negative numbers
- Small negative numbers $\times$ small negative numbers
- Large positive numbers $\times$ large positive numbers
- Large positive numbers $\times$ large negative numbers
- Large negative numbers $\times$ large positive numbers
- Large negative numbers $\times$ large negative numbers
- Multiplying numbers with the low order bits masked off
- Multiplying numbers with middle bits masked off
- Multiplying sparse numbers with many zeros but few ones
- Multiplying dense numbers with many ones but few zeros

- Tests specifically designed to trigger corner cases in your alternative design
- Testing all or some of the above using random input and output delays

Once you have finished writing your directed tests, you should move on to expand on writing random tests. Each random test should be a separate test case. You will want to use both zero and non-zero random delays in the test source and sink with your random testing. The kinds of random testing should be similar in spirit to the directed testing discussed above. You might want to consider randomly generating 32b values and then masking off different bits to create different variations. For example, you can mask off the low bits, the high bits, the low and high bits, or just randomly force some bits to zero or one to create numbers with more or less zeros. In addition to testing the design as a whole, if you add any new datapath or control components, you must write unit tests for these as well!

6. Evaluation

Once you have verified the functionality of the baseline and alternate design, you should then use the provided simulator to evaluate these three designs. You can run the simulator for all the designs like this:

```
% cd ${HOME}/ece4750/sim/lab1_imul
% make run-all
```

The simulator will display the total number of cycles to execute the specified input dataset. You should study the line traces (with the `RUN_ARG=-trace`) and the waveforms (located in `waves` folder) to understand the reason why each design performs as it does on the various patterns and be able to articulate the differences at signoff.

We only provide a single random dataset for evaluating your design, but clearly, this is not sufficient. You will need to use more datasets to make a compelling comparative evaluation of how your baseline and alternative design perform, especially since the alternative design has data-dependent behavior. We recommend consulting with a member of the course staff on your testing strategy to ensure good code coverage and the ability to highlight the differences between the baseline and alternative designs. We actually recommend that you reuse some of the random datasets you already developed for verification.

7. Sign-offs and Lab Report

We recommend consulting with the members of the course staff as you work on the lab for both advice and to report on findings and design choices. To ensure you have made satisfactory progress in the lab, you must complete the sign-off sheet during TA office hours. In addition to showing your results to the TA, you must be ready to explain your design and choices to the TA and be prepared to answer questions. You will also need to explain the aforementioned items in your lab report.

8. Submission

The code submission for both milestone and final submission is the exact same process. For this lab you will be submitting a compressed tar file (with the file ending `.tar.gz`) to Canvas with the following files/folders:

- `Makefile` – Makefile to run the Verilog simulator
- `verilator.cpp` – C++ harness for the Verilog simulator

- `group$XX.txt` – XX is your group number. File contains all the netids of all of the group members (one per line).
- `lab1_imul/` – The lab1 sub-project folder with the required file listed below.

The `lab1_imul` sub-project folder should have the following files at minimum:

- `tb_IntMul.v` – Simple Testbench for Verilog RTL
- `ub_IntMul.v` – Simple Microbenchmark for Verilog RTL single-cycle multiplier
- `IntMulSimple.v` – Verilog RTL Single-Cycle multiplier
- `IntMulBase.v` – Verilog RTL fixed-latency multiplier
- `IntMulAlt.v` – Verilog RTL variable-latency multiplier
- `default.config` – Config file for Makefile to allow testbench reuse.

In additional to the following mandatory files as listed above, you should submit any additional modules, `.config` files, and testbenches (files with the `tb/ub/utb` prefix) you have created. Make sure you do not modify the provided module declarations, the Makefile, and `verilator.cpp`.

You can create a compressed tar file by the following command:

```
% tar -czvf lab1.tar.gz file1 [file2] [...]
```

For your convenience, we have a make rule to assist you in creating the tar file. However, you are ultimately responsible for the contents (and the lack of) of the tar file you submitted.

```
% source setup-ece4750.sh
% cd ${HOME}/ece4750/sim
% make build-tar
```

You should turn in your completed sign-off sheet to any member of the course staff in office hours.

Acknowledgments

Socrates Wong, Aidan McNay, and José Martínez prepared this version of Lab 1 as part of the course ECE 4750 Computer Architecture at Cornell University. Significant parts were adopted from earlier versions by Christopher Batten, Shreesha Srinath, and Ji Kim. We are grateful to Mike Woodson and the rest of the COECIS team for helping set up and maintain the computing infrastructure.

ECE 4750 Computer Architecture, Fall 2023 Lab 1 Sign off

Lab 1: Iterative Integer Multiplier

Student Names: _____ _____ _____
Netids: _____ _____ _____

TA Initals

1. Getting Started

Successfully executing the example testbench for Single Cycle Design _____
Successfully using line trace for the Single Cycle Design _____
Setting up signal and waves in GTKwave _____
Explaining the difference between synthesizable and non-synthesizable verilog _____

2. Diagrams

2.1. Alternative Multiplier

RTL Diagram for Alternative Design _____
FSM for Alternative Design _____

3. Testbench and RTL Design Checkoff

3.1. Single Cycle Design

Single Cycle Design Testbench Strategy _____
Single Cycle Design Testbench Line Coverage _____
Single Cycle Design Testbench Toggle Coverage _____
Single Cycle Design Considerations _____

3.2. Baseline Design

Baseline Design Testbench Strategy _____
Baseline Design Testbench Line Coverage _____
Baseline Design Testbench Toggle Coverage _____
Baseline Design Considerations _____
Baseline Design Code Quality _____

3.3. Alternative Multiplier

Alternative Design Testbench Strategy _____
Alternative Design Testbench Line Coverage _____
Alternative Design Testbench Toggle Coverage _____
Alternative Design Considerations _____
Alternative Design Code Quality _____

4. Work Distribution

Please fill in the name of the student leading the testbench and RTL module design. Team members are expected to take turns leading the testbench and RTL module.

All students are ultimately responsible for the all non-testbench code submitted is synthesizable. **Non-synthesizable code used to implement the hardware modules will incur a 20% minimum grade deduction.** Please consult with a member of the course staff or handouts if unsure if your code is synthesizable.

The usage of ChatGPT and any generative AI models to assist with the labs is prohibited. All students acknowledge that they are forbidden from sharing their code with anyone outside their group or the course staff. Sharing code will be considered a violation of the Code of Academic Integrity. A primary hearing will be held, and if found guilty, students will face a severe grade penalty for this course. More information about the Code of Academic Integrity can be found here: http://theuniversityfaculty.cornell.edu/academic-integrity

4.1. Single Cycle Design

Single Cycle Design Testbench _____

4.2. Baseline Design

Baseline Design RTL module _____

Baseline Design Testbench _____

4.3. Alternative Design

Alternative Design RTL module _____

Alternative Design Testbench _____

4.4. Acknowledgement and Collaborators

As per the syllabus, you can give or receive "consulting help" to or from other students; however, you must acknowledge any help you received below.

_____

_____

_____

_____

_____

_____