ECE 4750 Computer Architecture, Fall 2022
Lab 1: Iterative Integer Multiplier

School of Electrical and Computer Engineering
Cornell University

revision: 2022-09-01-23-26

The first lab assignment is a warmup lab 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. **The milestone for this lab is to complete the baseline multiplier design along with an initial set of directed tests. 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:

- the Verilog hardware description language;
- abstraction levels including functional- and register-transfer-level modeling;
- design principles including modularity, hierarchy, and encapsulation;
- design patterns including streaming 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.

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. You should have already used the `ece4750-lab-admin` script to create or join a GitHub group. To get started, login to an `ecelinux` server, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:cornell-ece4750/lab-groupXX
```

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}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% pytest ../lab1_imul
```

All of the tests should pass except for the tests related to the iterative 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
- `IntMulBase.v` – Verilog RTL fixed-latency multiplier
- `IntMulBase.py` – Python wrapper for testing
- `IntMulAlt.v` – Verilog RTL variable-latency multiplier
- `IntMulAlt.py` – Pythnon wrapper for testing
- `imul-sim` – Multiplier simulator for evalution
- `__init__.py` – Package setup
- `test/IntMulFL_test.py` – FL multiplier unit tests
- `test/IntMulBase_test.py` – RTL fixed-latency multiplier unit tests
- `test/IntMulAlt_test.py` – RTL variable-latency multiplier unit tests
- `test/imul_sim_test.py` – Test to make sure multiplier simulator works
- `test/__init__.py` – Test package setup

1. Introduction

You learned about basic digital logic design in your previous coursework, and in this warmup 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, such an implementation is likely to 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 functional-level (FL) model of an integer multiplier as shown in Figure 1. You can find this FL implementation in `IntMulFL.py` and the associated unit tests in `test/IntMulFL_test.py`. This implementation always takes a single-cycle and uses a higher-level modeling style compared to the register-transfer-level (RTL) modeling you will be using in your

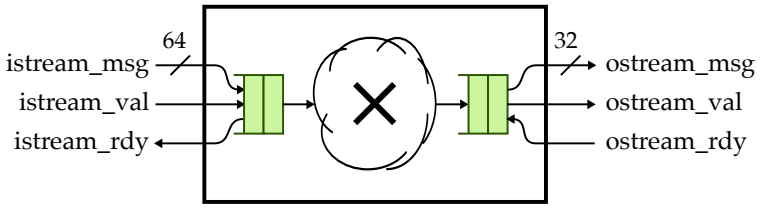

Figure 1: Functional-Level Implementation of Integer Multiplier – Input and output use latency-insensitive val/rdy interfaces. The input message includes two 32-bit operands; output message is a 32-bit result. Clock and reset signals are not shown.

```
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.

designs. These varying levels of modeling (i.e., FL versus RTL) are an example of *abstraction*. The interfaces for all three designs (i.e., FL 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 *stream interface design pattern*. We are hiding 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.

2. 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, as well as the parent module that connects the 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 the `vc` subdirectory.

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 which 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 is an example of 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

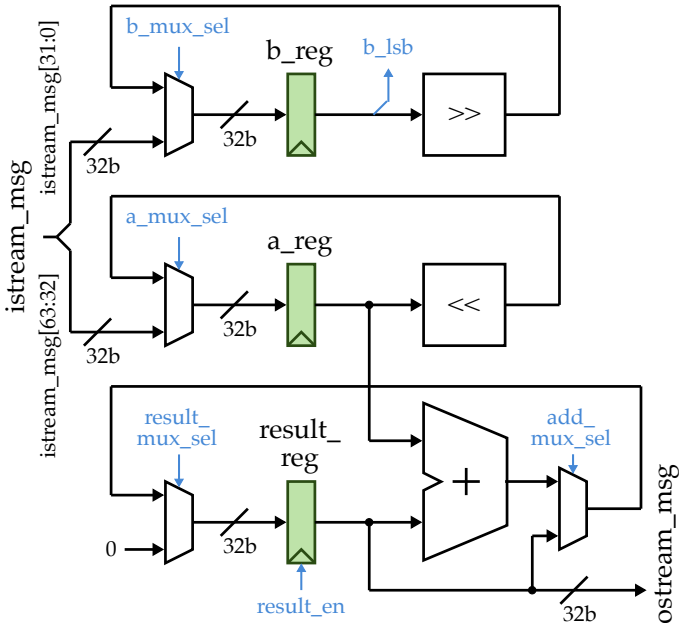

Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier – All datpath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

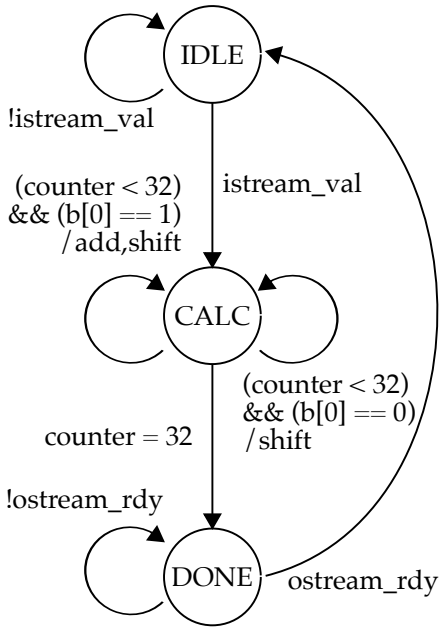

Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier – Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

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. It is relatively straight-forward 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 calcluation 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 src/sink delays. 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*.

3. Alternative Design

The fixed-latency iterative integer multiplier should 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. 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 lab report.

As with all of the alternative designs in this course, we simply sketch out the requirements and 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 `IntMulAltVRTL.v`. 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 so that you can unit test it.

4. Testing Strategy

We provide you one directed test for your multiplier, 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 FL model. Once these tests are working on the FL model, 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 the entire project, how to run just the tests for this lab, and how to run just the tests for the FL model, baseline design, and alternative design.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ..
% pytest ../lab1_imul
% pytest ../lab1_imul/test/IntMulFL_test.py
% pytest ../lab1_imul/test/IntMulBase_test.py
% pytest ../lab1_imul/test/IntMulAlt_test.py
```

You will add your directed tests to `IntMulFL_test.py`. Since this harness is shared across the FL model, the baseline design, and the alternative design you can write your tests once and reuse them to test all three models. You will be adding many more test cases. Do not just make the given test case larger. Some suggestions for what you might want to test are listed below. Each of these would probably be a separate test case.

- 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 source and sink delays

Once you have finished writing your directed tests you should move on to writing random tests. Each random test should be a separate test case. You will want to use both zero and non-zero src/sink 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 added any new datapath or control components you must write unit tests for these new components!

5. Evaluation

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

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab1_imul/imul-sim --impl base --input small
% ../lab1_imul/imul-sim --impl base --input small --stats
% ../lab1_imul/imul-sim --impl base --input small --trace --dump-vcd
```

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.

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 maybe six or so datasets for evaluation. Obviously these datasets need to be carefully chosen to highlight the differences between the baseline and alternative designs. We actually recommend you reuse some of the random datasets you already developed for verification.

Acknowledgments

This lab was created by Christopher Batten, Shreesha Srinath, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.