# ECE 4750 Computer Architecture, Fall 2013

# Lab 1: Iterative Integer Multiplier

School of Electrical and Computer Engineering
Cornell University

revision: 2013-09-13-23-05

This 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 create an effective testing strategy for the alternative design, and to perform an evaluation comparing the two 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:

- the Verilog hardware modeling framework
- effective design principles including abstraction, modularity, hierarchy, and aencapsulation
- effective design patterns including message interfaces, control/datapath split, and FSM control
- effective design methodologies including test-driven development and incremental 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. To get started, you should access a course development machine and perform the following steps:

```
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-lab1-imul
% mkdir ${HOME}/ece4750/ece4750-lab1-imul/build
% cd ${HOME}/ece4750/ece4750-lab1-imul/build
% ../configure
% make
% make check
```

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 `plab1-imul` subproject which includes the following files:

- `plab1-imul-msgs.v` – multiplier message types
- `plab1-imul-msgs.t.v` – unit tests for multiplier message types
- `plab1-imul-IntMulFL.v` – functional-level multiplier implementation
- `plab1-imul-IntMulIterFixedLat.v` – fixed-latency multiplier implementation
- `plab1-imul-IntMulIterVarLat.v` – variable-latency multiplier implementation
- `plab1-imul-test-harness.v` – multiplier test harness shared across implementations
- `plab1-imul-IntMulFL.t.v` – functional-level multiplier unit tests
- `plab1-imul-IntMulIterFixedLat.t.v` – fixed-latency multiplier unit tests
- `plab1-imul-IntMulIterVarLat.t.v` – variable-latency multiplier unit tests

- `plab1-imul-input-gen.py` – Python script to generate input datasets
- `plab1-imul-sim-harness` – multiplier simulator harness shared across implementations
- `plab1-imul-sim-fixed-lat` – multiplier simulator for fixed-latency multiplier
- `plab1-imul-sim-var-lat` – multiplier simulator for variable-latency multiplier

You can use the `ece4750-lab-admin` script and the `git` command to add members to your lab group, use your own group repository to collaborate with your group members, and ultimately prepare your submission for uploading to CMS.

## 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 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 model of an integer multiplier as shown in Figure 1. You can find this functional-level implementation in `plab1-imul-IntMulFL.v` and the associated unit tests in `plab1-imul-test-harness.t.v` and `plab1-imul-IntMulFL.t.v`. 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 designs. These varying levels of modeling (i.e., functional-level versus RTL) are an example of the *abstraction design principle*. The interfaces for all three designs (i.e., functional-level model, fixed-latency RTL model, and variable-latency RTL model) are identical. Each multiplier should take as input two 32-bit operands packed into a 64-bit message and produce a 32-bit result. All implementations should treat the input operands and the result as a two's complement number 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

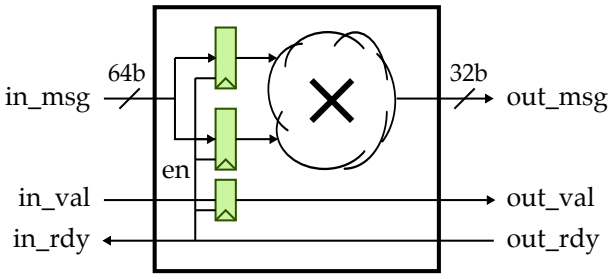


**Figure 1: Functional-Level Implementation of Integer Multplier –** Input and output use latency-insensitive val/rdy interfaces. The input message includes two 32-bit operands; output message is a 32-bit result. Clock and reset signals are not shown.

```
1  def imul( a, b ):
2
3    result = 0
4    for i in range(32):
5      if b & 0x1 == 1:
6        result += a
7      a = a << 1
8      b = b >> 1
9
10   return result
```

**Figure 2: Iterative Multiplication Algorithm –** Iteratively use shifts and subtractions to calculate the partial-products over time. This is executable Python code.

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 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-signficant 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 `plab1-imul-IntMulIterFixedLat.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, we will be using a structural design style in the datapath diagram. 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 library, the Verilog Component library. You can find out more about the VC library by looking in the `vc` subproject. The collections of modules that you might be particularly interested in for this lab are as follows:

- `vc-muxes.v` – collection of muxes
- `vc-regs.v` – collection of registers
- `vc-arithmetic.v` – collection of arithmetic units

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 shifts to calculate the multiplication; and the DONE state is responsible for sending the message out the output interface. 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 during 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. 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*.

As outlined in the course lab assignment assessment rubric, the baseline design section in your lab report will need to briefly summarize the baseline design and any changes you made to the provided design.

## 3. Alternative Design

The fixed-latency iterative integer multiplier should always takes 34 cycles to compute the result: one for the IDLE state, 32 for iterative calculation, and one cycle for the DONE state. While it is possible to optimize away the IDLE and DONE states with a more careful implementation, fundamentally this algorithm is limited by the 32 cycles required for the iterative calculation. For your alternative design

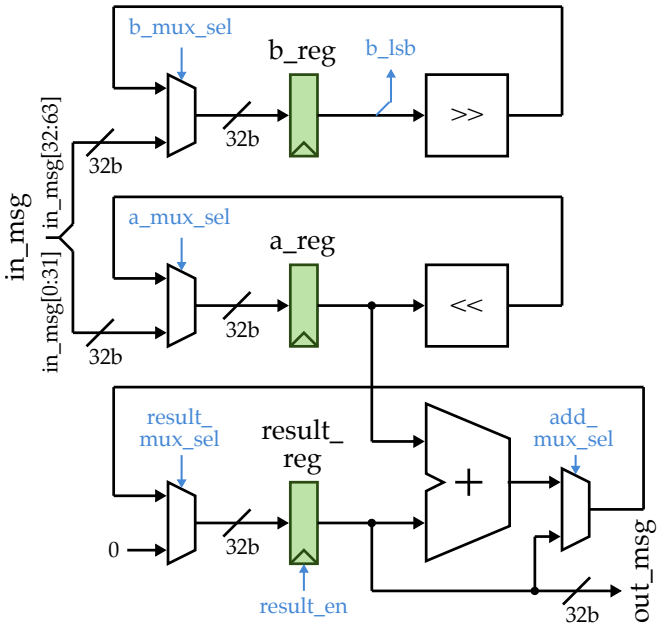


**Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier –** All datpath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

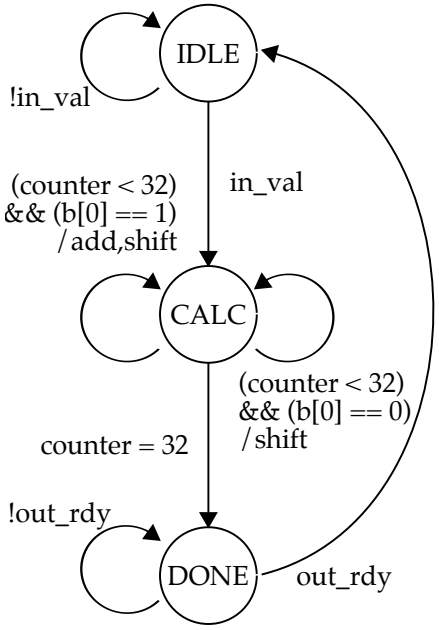


**Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier** – Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

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. 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 `plab1-imul-IntMulVarLat.v`. All of your code for the alternative design should be placed in this file.

As outlined in the course lab assignment assessment rubric, the alternative design section in your lab report will almost certainly need to include a datapath diagram, finite-state machine, and sufficient detail to accurately describe your design.

## 4. Testing Strategy

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. We have provided you with tests for the baseline design in `plab1-imul-IntMulIterFixedLat.t.v`; we have essentially reused the tests developed for the functional-level model. There are variety of directed tests along with a set of random tests, and tests which purposely delay the input messages and/or apply back pressure to the output message interface. Your baseline design should pass all of the provided tests. 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*.

We have provided an initial set of tests for your alternative design in `plab1-imul-IntMulIterVarLat.t.v`. Note that the tests that we provide for the alternative design are essentially the same tests we use for the functional-level and fixed-latency implementations. The tests we provide are not sufficient to effectively test your alternative design. You must develop a testing strategy that can enable you to create a compelling case for the functional verification of your alternative design. This will require you to add additional "gray box" tests that try and test various corner cases in your alternative design.

As outlined in the course lab assignment assessment rubric, the testing strategy section in your lab report should briefly summarize the approach used in testing the baseline design before focusing in detail on the tests you used to verify your alternative design.

## 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/ece4750-lab1-imul/build
% make
% ./plab1-imul-sim-var-lat   +input=small +stats
% ./plab1-imul-sim-fixed-lat +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 `+verbose=1` option) and possibly the waveforms (with the `+dump-vcd=<file-name>` option) to understand the reason why each design performs as it does on the various patterns. You can run simulations for all of the provided patterns like this:

```
% cd ${HOME}/ece4750/ece4750-lab1-imul/build
% make eval-plab1-imul
```

As outlined in the course lab assignment assessment rubric, the evaluation section in your lab report should present these results in a compelling way (e.g., nicely formatted table or bar blot), discuss the details behind these results, and then make a compelling case for one design or the other.

## 6. Extensions

For some of the lab assignments, we may allow students to pursue extensions as a way for students to demonstrate creativity and a deeper understanding of the course material. For those lab assignments, extensions will factor into the grading as described in the course lab assignment assessment rubric.

For this lab assignment, we will not be accepting any extensions. We will simply eliminate that criteria from the assessment rubric, and weight the other criteria as normal. You should exclusively focus on the baseline design and alternative design, your testing strategy, and a compelling evaluation.

## 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.