gitECE 4750 Lab 1: Iterative Integer Multiplier

Carly Swetz (cms464), Raghav Kumar (rk524), Kyle Betts (kcb82)

1. Introduction

The purpose of this lab was to design two different implementations of an integer iterative multiplier. Our two designs use a series of add and shift operations to perform multiplication. Although a single-cycle integer multiplier is feasible to design, our iterative approach has improved cycle time. Nonetheless, the iterative approach does have reduced throughput compared to the single-cycle design. While it is possible to improve cycle time and maintain high throughput through pipelining, that was not the purpose of this lab. The high level functionality of our multiplier is to take as input a 64-bit message which consists of two 32-bit operands. Both implementations can handle signed and unsigned multiplication and utilize the val/rdy microprotocol to manage when new inputs are sent to the multiplier and when the multiplier is ready with the result. By using this protocol, one module is able to send messages to the multiplier without worrying about how many cycles the design takes to execute multiplication and return the result.

We first completed the baseline design that is fixed-latency and takes the same number of cycles each time. After implementing the baseline design, we moved onto our alternative design that is variable-latency and reduces the execution time by shifting the operands a variable amount depending on the number of trailing zeros in the B operand. To compare the two, the baseline design consistently took 35 cycles, while our alternative approach took less cycles for every dataset tested. To elaborate, our alternative design did substantially better on the base dataset and the dataset with many zeros, which makes sense given our implementation. Overall, our alternative design certainly performed better by completing in less clock cycles, but had a slightly longer clock cycle. Also, since our variable-latency design uses the same logic from the fixed-latency implementation plus some additional combinational logic, the variable-latency design will consume more power and space.

2. Baseline Design

We implemented our baseline design to match the one provided, which is a fixed-latency implementation that takes 35 cycles each execution. The baseline design is composed of two separate modules: the datapath and the control unit. The diagrams for each can be seen in **Figures 1** and **2** below, respectively. The two modules are connected in a parent module and communicate using control/status signals, which are shown in blue in **Figure 1**. This breakdown exemplifies the *modular design principle*, and more specifically the *control/datapath split design pattern*.

The datapath is made up of registers, muxes, one-bit left and right shifters, and an adder. For each of these blocks, we instantiated a child module using the ones provided in vclib. As a result, there is a one-to-one correspondence between the structural implementation in our code and the datapath diagram, which is an example of the *hierarchy design principle*.

The control unit is an example of the *finite-state-machine design pattern* and decides how the data moves through the datapath. The IDLE state receives the input message and separates the two operands into the input registers. When both the val/rdy signals are 1, we move into the CALC state where we check the least significant bit of the B operand. If it is a 1, we add A to the result, shift A one-bit to the left and shift B one-bit to the right. If it is a 0, we just shift both operands. We are in the CALC state for 32 cycles in order to check each of the 32 bits. We use a counter to keep track of the number of cycles and when it reaches 32, we move to the DONE state that sends the result message out when the response val/rdy signals are 1. In total, the multiplication will take 35 cycles: 1 cycle for the IDLE state, 32 cycles for the CALC state, 1 cycle to recognize the calculation is done, and 1 final cycle for the DONE state.

To fully grasp how the baseline design works, it is helpful to go through an example calculation of A=2 multiplied by B=3. Although both operands are actually represented by 32 bits in our design, let's just look at the last four for simplicity reasons: A=4'b0010 and of B=4'b0011. Once in the CALC state, we will check if the least significant bit of B is a 1. Since it is, we will add A to the result (result=2). Then, we shift both operands as previously described, making A=4'b0100 and B=4'b0001. We repeat the same process to end up with result=6, A=4'b1000, and B=4'b0000. This cycle will repeat 30 more times, until we have gone through all 32 bits. Our final result is 6, which is the correct answer.

Overall, this design is a good baseline for comparison because it is fixed-latency and always takes 35 cycles to complete. Therefore, any design with variable-latency will be interesting to compare with our baseline design. While the baseline design may have more cycles and lower throughput, the cycle time is shorter.

3. Alternative Design

The alternative implementation of the multiplier intended to take advantage of the structure of the input operands, shifting the operands by a variable amount depending on the number of trailing zeros. With this approach, we intended to create a design which would take less cycles on average per multiplication while potentially having a longer clock cycle.

From a high level, our design looked at the last four bits of the B operand and shifted both A and B by a variable number between 1 to 4 depending upon the number of trailing 0s. So in the best case, the last four bits would be a 0000 and our design would shift 4, effectively optimizing 4 clock cycles of the base design into 1. In the worst case, the last four bits would be ???1 and we would only shift by 1. The cycle of evaluating the last four would then repeat till B became a 0 (which would always happen as we were using logical shifts). To realize this idea and implement it in hardware, we had to make changes to our datapath and control unit. The updated datapath diagram can be seen in **Figure 3** below. We added a 4 to 1 multiplexer (mux) in the datapath with fixed inputs - 1, 2, 3, 4 - that represented the shift amounts. To select one of these values, we needed a 2 bit select signal controlled by the FSM. In the control unit, we used ***casez*** statements to concisely set 4 flags - shift_1, shift_2, shift_3 and shift_4 - and then used these flags to set the appropriate shift mux select. It is important to note that we decoupled the addition from the shifting. In the first iteration of our design, we variably shifted and added in the same clock cycle but that didn't give reliable results presumably due to a timing issue wherein the system was adding and then shifting rather than the other way round. To fix such subtle timing specific issues, we decided to give the ADD its own dedicated clock cycle along with a single shift (shift needed to avoid looping forever). So the FSM first looked at whether the ***do_add*** signal was set. In that case it would do the addition and shift by 1. If not, then it would see which of the ***shift_**** signals were set and then shift the operands appropriately such that **the next clock cycle would do the add** (meaning that the set of shift signals made sure that post shifting, ***b_lsb*** was a 1).

Our design worked very well and gave us promising results as it considerably increased throughput. However, due to the increased combination logic in the control unit, the clock cycles were invariably longer in alternative design as compared to the base design. But the fact that our test suite took a full second less when run on the alternative vs the base design implied that the effect of the increased throughput outweighed the effect of the longer clock cycle. To further increase throughput, we could have used a 8 to 1 mux instead of a 4 to 1 but this would have increased the casez logic in the control unit that would have cycle time implications. So there is definitely a tradeoff between throughput and cycle time as a function of the aggressiveness of the optimization. Another interesting idea was to bypass the mux completely and send a signal directly from the FSM to the shifters. This would have gotten rid of the extra 'shifter' mux, saving space, energy and cost in the datapath but obviously wouldn't have increased throughput or reduced cycle time since the logic would have been the same.

4. Testing Strategy

In testing our design, we used the provided pclib framework to implement the tests. Using this framework enabled us to reduce the amount of code needed to write a large number of tests for both our designs. By writing tests for the functional level model, we were able to use the same tests to test both our fixed-latency design, as well as our variable-latency design. The overall testing strategy was to use blackbox testing for all the tests. We were able to use the message interface design pattern (val/rdy microprotocol) to completely abstract away the number of cycles the different designs of the multiplier required. This made the testing universal, and also allowed us to make changes to our designs without needing to change our tests. First we used directed testing with fixed delays to simply test if our multipliers were producing correct results. Next, we used random testing to verify the functionality of our val/rdy microprotocol. We used more random testing on the values for multiplication to test a large number of middle values without needing to type them all out. Finally, we used a test of random values and random delays to confirm the two design choices are correctly connected.

The first type of testing we did was directed testing which was used to test the multiplication functionality. The reason for this selection was to write a test for each type of test that we could think of. See **Figure 4** for the categories of directed testing. This allowed us to quickly identify error, and since the tests were split by type, it allowed us to debug our designs faster. We tested a handful of different types of multiplication, and for each type we had multiple tests to cover different cases in that type. For example, in the (large+)(large+) type, we tested every combination of 2-6 digits for both operands. For each of the tests, the src_delay and sink_delay were both fixed since the significance of these tests are primarily focused on the functionality of the actual multiplication, not the correctness of the val/rdy microprotocol.

The total number of directed tests is 228. We are confident that this many direct tests provides a strong confidence that our multiplier functions for not only normal input, but also the corner cases.

The next kind of testing we did was to confirm we correctly implemented the rdy/val protocol via random testing. We decided to do this test second, since our first test made us confident that our multiplier designs were working. This allowed us to believe if the tests were failing, it was due to our implementation of the rdy/val protocol, since we were already confident in our multiplier. Within this section of testing we used the simple test set of (small+)(small+), but set both delays to different random values between 0 and 5. We did four tests of the (small+)(small+) set with four different random pairs of delays. In passing these four tests, we are confident that our implementation of the rdy/val protocol is working.

The next testing we did was using random values for the inputs to the multipliers. This allowed us to test a large number of both normal cases and corner cases, saving us a lot of time in typing them all out. We decided to test four types of random value test cases. The first test generated numbers between 0x00000000 and 0xffffffff, hence any possible value. The second test generated numbers like the first, but then masks the values by setting the 12 least significant bits to 0. The third test generated numbers like the first, but then masks the values by setting the 8 middle bits to 0. The final test generated numbers like the first, but then masks the values by setting the 12 most significant bits to 0. After passing these tests we were confident that we had tested a ton of random tests to be even more sure our multiplication implementations were correct.

The final test we used both random values and random delays. The reason for this test is to make sure that the connection between the val/rdy protocol and our multipliers are correct. We generate 20 pairs of numbers between 0x00000000 and 0xffffffff, and then run those tests with a pair of delays for the source and the sink. After passing all of these tests we were confident that the protocol and the functionality were properly incorporated together.

Through this combination of testing, we are confident that our implementation of both multipliers is correct. The first set of directed tests confirmed we could pass specific general and corner cases of inputs. The first random test confirmed our implementations correctly used the rdy/val protocol, a key to our black-box testing. Our tests of random values allowed us to test a massive amount of general and corner case inputs and further confirm our belief in the multiplier functionality. Finally, the random values and random delays testing confirm that the rdy/val protocol and the multiplier have been incorporated together correctly. Since we pass all of these tests, we are confident that our designs are functional.

5. Evaluation

In our design of the variable-latency we wanted to make sure we were getting a performance boost in cycles/calculation without increasing our cycle time too much. We increased cycle time minimally by only adding limited combinational logic to our control unit. We only check the 4 least significant bits and shift to the position of the next 1. If there are no 1s in the first four we only shift 4 bits. The reason we stopped at four is because checking more bits would have increased our cycle time more than we were willing to. In running tests for different datasets from our testing we see where our alternate design really excels.

As can be seen by the chart in **Figure 5** and the plot in **Figure 6**, our alternate design improves the number of cycles/multiplication for every data set we tested on. On the general completely random inputs, we saved nearly 8 cycles per multiplication. From the chart it is also easy to see that our alternate design performs extremely well on inputs with a large number of zeros, especially when they are trailing zeros. Our alternate multiplier does not perform as well when there is a large number of ones which is intuitive since the additional shifter logic will not be used as frequently. Overall we are extremely happy with our design, and we think the small additional cycle time is certainly worth the reduced cycles/multiplication. For programs where multiplication is common and values are expected to be small, our design is clearly the superior choice. For programs where multiplication is uncommon, or values being dealt with are very large, it might make more sense to use the fixed-latency design. Also, since our variable-latency design uses all the logic from the fixed-latency plus some additional combinational logic, the variable-latency design will consume more power and space for this additional logic. For embedded applications where power and space is a constraint, maybe the fixed-latency design would be the better choice.

Figure 1: Baseline Design Datapath

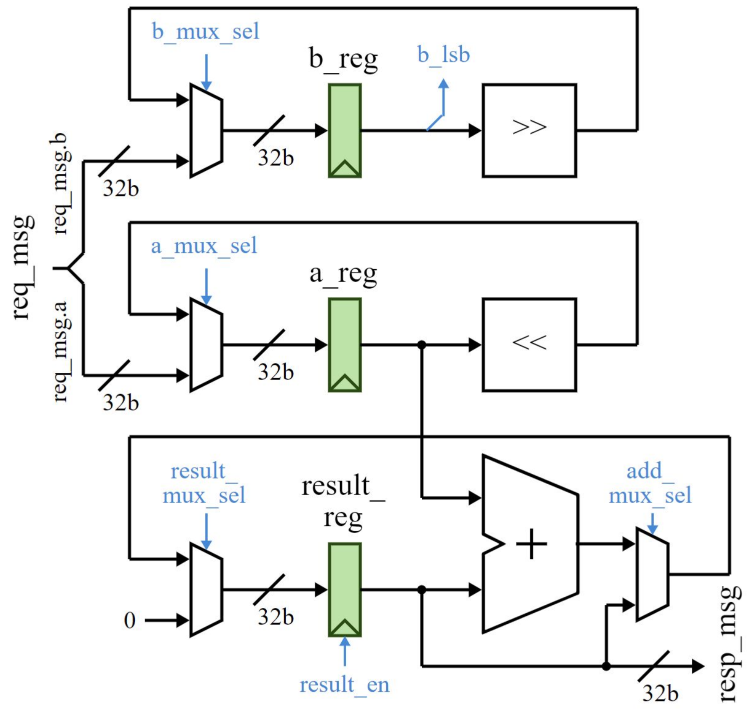


Figure 2: Baseline Design Control Unit

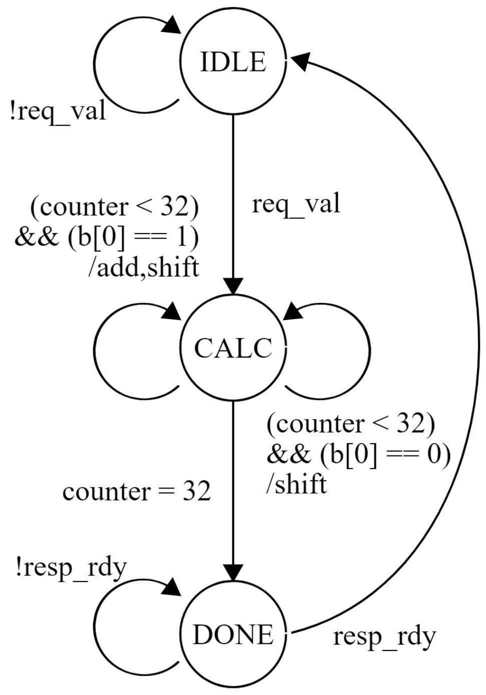


Figure 3: Alternative Design Datapath

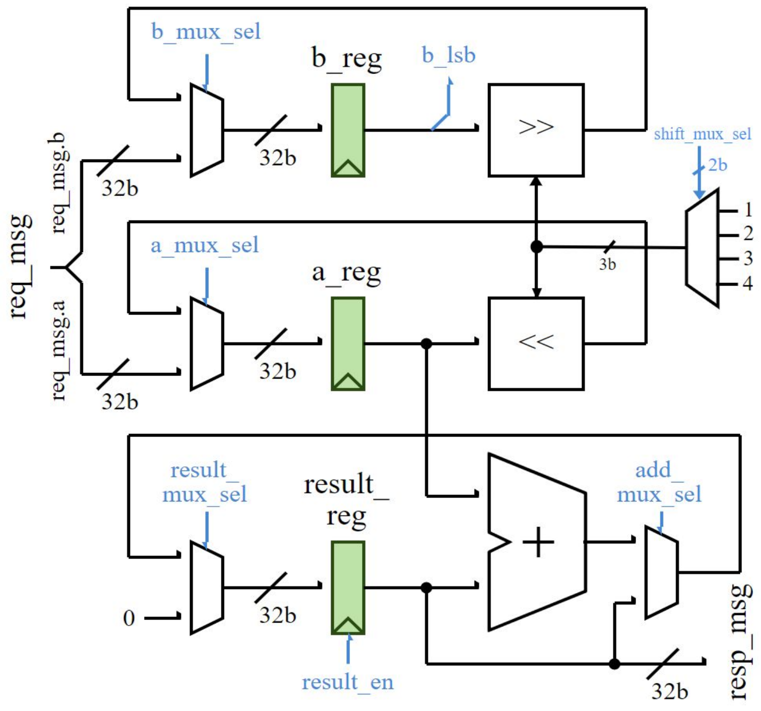


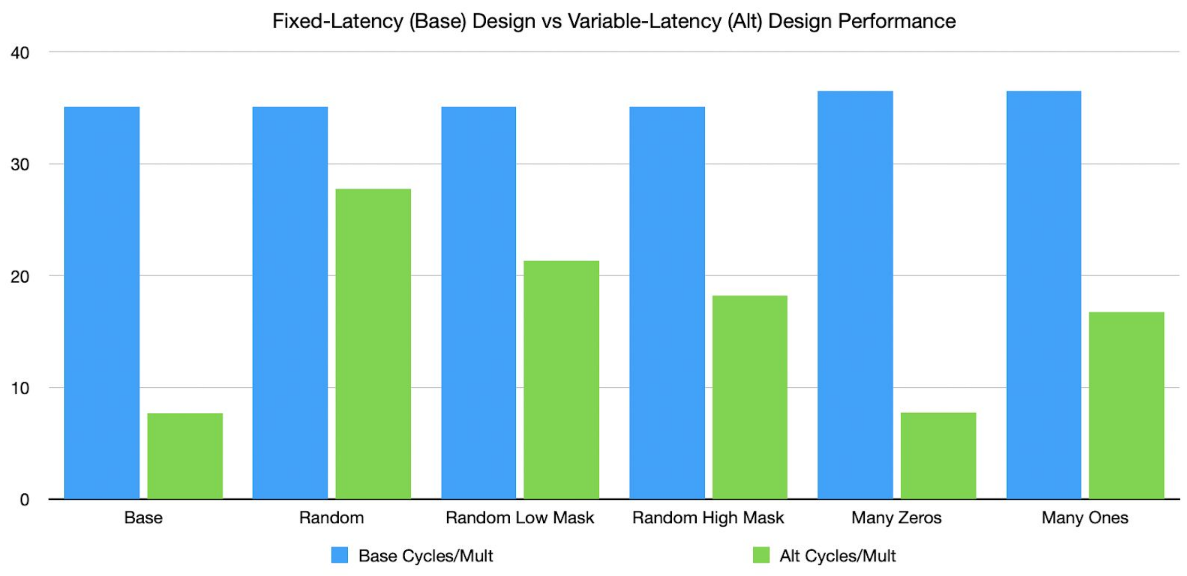
Figure 4: Directed Tests

Category	Example	Number of Tests
(Small+)(Small+)	(3)(2) = 6	9
Zeros and 1s	(-1)(0) = 0	9
(Small-)(Small+)	(-4)(9) = 36	9
(Small+)(Small-)	(5)(-5) = -25	9
(Small-)(Small-)	(-6)(-2) = 12	9
(Large+)(Large+)	(1758)(50674) = 89084892	24
(Large+)(Large-)	(91195)(-4091) = -373078745	24
(Large-)(Large+)	(-555)(181) = -100455	24
(Large-)(Large-)	(-55559)(-181) = 10056179	24
(Large+)(Small+)	(487946)(4) = 1951784	10
(Small+)(Large+)	(5)(98561) = 492805	10
(Large+)(Small-)	(8768768)(-6) = -52612608	8
(Small-)(Large+)	(-9)(3214) = -28926	8
(Large-)(Small+)	(-487946)(4) = -1951784	8
(Small+)(Large-)	(1)(-987321654) = -987321654	8
(Small-)(Large-)	(-7)(-465) = 3255	8
(Large-)(Small-)	(-567567567)(-2) = 1135135134	7
Low Order Bits Masked	(3)(0x000ff000) = 3133440	6
Middle Bits Masked	(0x00f0000f)(2) = 31457310	6
Many Zeros	(1000)(0x00000001) = 1000	4
Many Ones	(2)(0x00ffffff) = 33554430	4

Figure 5: Raw Performance Data

Test	Base # Cycle	Base Cycles/Mult	Alt # Cycle	Alt Cycles/Mult
Base	1756	35.12	384	7.68
Random	1756	35.12	1389	27.78
Random Low Mask	1756	35.12	1067	21.34
Random High Mask	1756	35.12	911	18.22
Many Zeros	146	36.50	31	7.75
Many Ones	146	36.50	67	16.76

Figure 6: Baseline vs. Alternative Design Performance



Role and Task Table

Carly Swetz	**Raghav Kumar**	**Kyle Betts**
RTL Design Engineer	RTL Design Engineer (architect)	RTL Verification Engineer
● Assisted in implementing the datapath module in the baseline design by instantiating some of the child modules ● Created the parent module in the baseline design to connect the datapath and control unit modules ● Added the missing parts of the line tracing code in both designs that helped in testing and debugging code ● Brainstormed ideas for the alternative designs and what issues may arise when shifting by more than one bit ● Implemented the logic in the alternative design that checked when the B operand was zero ● Helped debug the alternative design by checking syntax and other compiling errors ● Added our additional mux to the baseline datapath diagram to create the datapath diagram for our alternative design ● Wrote the introduction and baseline design description in the report	● Wrote first few iterations of base code ● Fixed syntax errors to compile code before running unit tests ● Brainstormed alternative design and discussed various options with fellow design and verification engineers. ● Came up with the final alternative design ● Implemented alternative and base implementation ● Debugged code, finding issues in the FSM logic ● Commented code extensively so that my team could easily understand the structure of the code and the following logic. ● Wrote few unit tests ● Brainstormed directed test cases and potential edge cases ● Wrote the alternative design section on the wrote ● Gave pointers for major things to talk about in the 'evaluate' section	● Created the directed test cases for the designs ● Created the random test cases for the designs ● Helped in the brainstorming for the alternate design ● Created the data sets for the evaluation of the designs ● Collected data from the simulator on design performance ● Aided in the debugging of the designs through testing feedback and line tracing ● Completed the write up for the testing section ● Completed the write up for the evaluation section ● Drew the logic out for a potential variable-latency design ● Helper merge individual branches into the mater branch for final submission ● Debugged errors in the testing code ● Helped prepare the lab report for submission