Anne Potter (ap674) and Ashley Heckman (agh93)
Lab Report 1
ECE 4750
9/15/22

Lab Report 1

1. Introduction

In this lab, we implemented two different iterative multipliers: one fixed-latency implementation and one variable-latency implementation. Both designs iteratively calculate the multiplication by using a series of shift and add operations. The fixed-latency implementation, or the baseline design, performs one shift per cycle until we have shifted all of the original input out, adding as necessary. This shifting and adding takes 32 cycles, as there are 32 bits in our input operands. The latter implementation, or the alternative design, exploits properties of the input to decrease the number of cycles necessary to produce the output. It does this by taking advantage of strings of consecutive zeros; it performs a shift by the amount of zeros that we have in a row in the least significant bits, rather than always shifting the inputs once. This lab is intended to give us a strong foundation to complete the rest of the labs, as well as to familiarize us with and give us hands-on experience with the material we have covered in lecture. These topics include Verilog hardware description language, functional-level and register-transfer-level modeling, design principles and patterns such as encapsulation and FSM control, and incremental and test-driven development. The multiplier is the first step in the implementation of a complete multicore processor.

2. Alternative Design

The multiplication algorithm takes two inputs, a and b (derived from the single istream input), and shifts both inputs with an add when the least significant bit of b is one. It shifts without an add when the least significant bit of b is zero. We can decrease the number of cycles necessary by shifting through all the zeros adjacent to each other in one cycle. Though this decreases the number of cycles if we have multiple zeros in a row, it does increase the length of the critical path by adding an additional piece of hardware to each cycle. We can assume that this is an advantageous change to make because most inputs will have at least a couple of zeros in a row, so it should decrease the total run time on average. It takes less time to check how many zeros are in a row than it does to complete that many individual cycles, which include a mux, register access, and single shift. We do this modification in the alternative design.

The alternative implementation is very similar to the baseline design, but it differs from the original implementation by taking advantage of the input properties. The datapath for both designs is implemented as a series of modules that represent each component. Each module has inputs and outputs, and we can wire the modules together through shared variables. The alternative design's datapath is very similar to the baseline design's datapath, with the only notable difference being in the way we shift. In the alternative design, the output of the b register is fed into a pattern module. This pattern module determines how many times we can shift our operands based on how many zeros we have in a row in the least significant bits of the input. It will output this number, which is then passed to both shifter units and the control unit. The datapaths for both implementations are shown below in *Figure 1* and *Figure 2*.

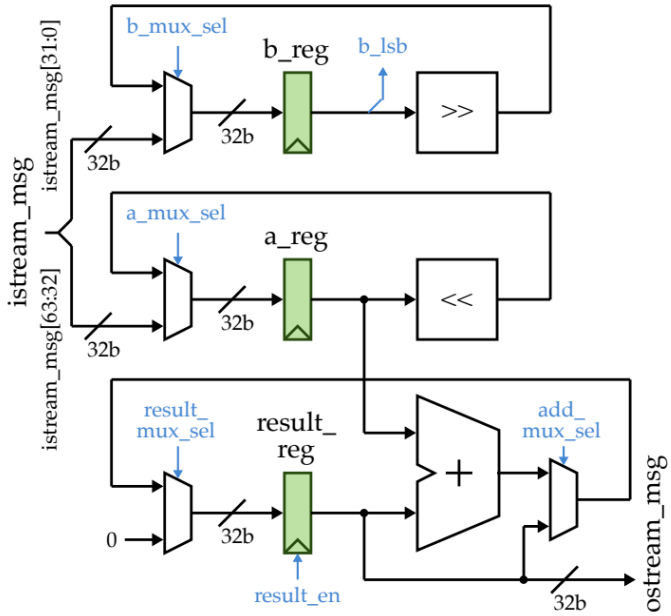

Figure 1: Fixed-latency Iterative Multiplier Datapath

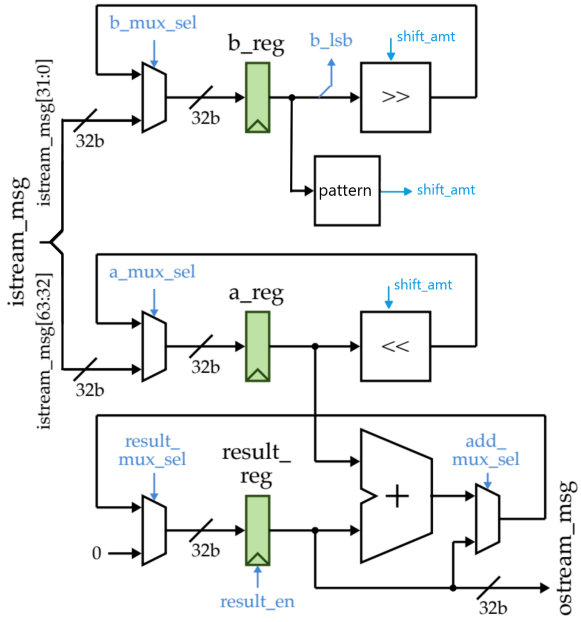

Figure 2: Variable-latency Iterative Multiplier Datapath

Just like with the datapath, the control unit between the two implementations is very similar. Both control paths have three stages: IDLE, CALC, and DONE. IDLE means the machine is ready to accept inputs. CALC means the machine is currently in the process of computing the multiplication and producing an output. DONE means that the calculation has been completed, but the number has not yet been received by the destination. The control paths between the two implementations differ in the transition from the CALC to the DONE state. The fixed-latency baseline multiplier transitions when the counter is exactly 32, while the variable-latency multiplier transitions when the counter is greater than or equal to 32. This is because of the way we implemented our shifting logic in the alternative design, as discussed in more detail below. See *Figure 3* and *Figure 4* for the finite state machines describing the control paths of both implementations.

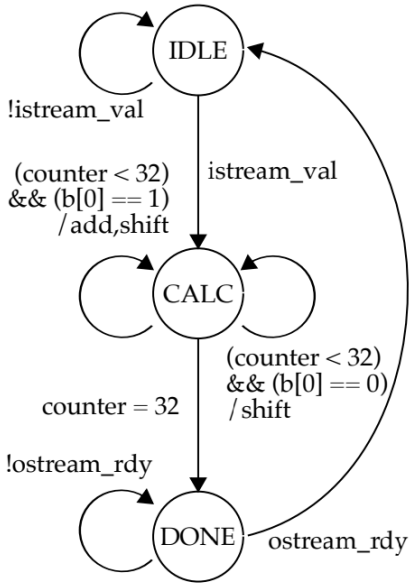

Figure 3: Fixed-Latency Iterative Multiplier Control FSM

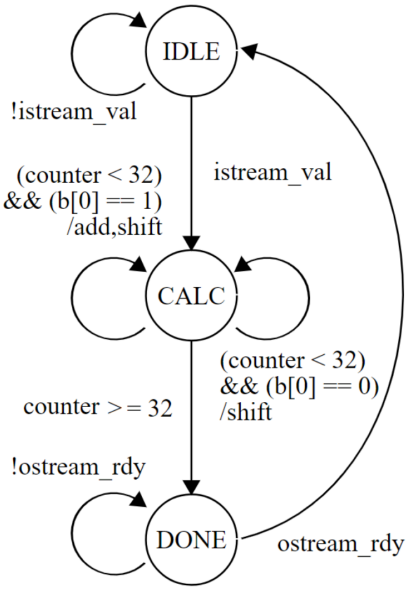

Figure 4: Variable-Latency Iterative Multiplier Control FSM

The transition between the CALC state and the DONE state needs to check for the counter being greater than or equal to 32 because it's possible that we pass over 32 without ever stopping at that value when we increment the counter. For example, imagine we have an a operand of four and a b operand of five. The b operand of five is passed into the pattern module. Five is represented as 000…0101 in binary. The pattern module ignores the least significant bit of the operand since that is what is going through the implementation in the current cycle, and it is only concerned with the next cycle. It looks at the input, sees that the second most significant bit is a 0 and the third least significant bit is a 1, and determines that we can shift the inputs two times. In the next cycle, the shifted b operand (which is 1, or 000…0001 in binary) is passed into the pattern module. The pattern module notes that all of the bits (with the exception of the least significant bit, which it doesn't care about) are zero, so it determines that we can shift the inputs 31 times. In each cycle, we increment the counter by the number of times that we shift in that cycle. Thus, the counter goes from 0 to 2 to 33, completely skipping over 32, making it necessary for the logic for switching from the CALC to DONE state to check for numbers both greater than and equal to 32. See *Table 1* for a line trace of the given example. We decided to keep the counter to check if the calculation is done rather than checking if one of the operands is equal to zero because we wanted our code to be easy to debug. With looking at line traces of the counter variable, we were able to fix a lot of problems with our code that otherwise would've been very difficult to find.

Input (hex)	A Register (hex)	B Register (hex)	Counter (dec)	Result Register (hex)	State	Output (hex)
0000000400000005	00000000	00000000	0	00000000	IDLE	
	00000004	00000005	0	00000000	CALC: shift & add	
	00000010	00000001	2	00000004	CALC: shift & add	
	00000000	00000000	33	00000014	CALC	
	00000000	00000000	0	00000014	DONE	00000014

Table 1: Line trace of the Alternative Variable-Latency Design, Given Inputs 4 and 5

When implementing our alternative design, we tried to keep as much in common with the baseline design as possible. We did this because the baseline design reflects the patterns and principles of modularity, hierarchy, and encapsulation. It exhibits modularity by decomposing the design into two modules: the datapath and the control unit. Within these modules, we are calling smaller modules. The design is broken down into very simple pieces. The design shows hierarchy with this decomposition into smaller modules. The top level calls the datapath and control unit modules, which then call their child modules, and so on. It's very easy to follow and see what module calls what, and to clearly see what the higher level modules are. The design displays encapsulation by hiding implementation details, such as the multiplier latency, from the interface by using the val/rdy signals. Our alternative design displays modularity, hierarchy, and encapsulation for the same reasons that our baseline design does.

3. Testing Strategy

We started by testing our implementations against the provided small_pos_pos test case. Once our code passed that simple test, we implemented a variety of test cases, with a mix of assertion, directed, random, value, and delay tests. These tests were all integration, black-box tests. We also did a very ad-hoc version of a unit test for the alternative design, which is white-box. We did this mix of testing to try to hit as many corner cases in our code as possible and to verify that our alternative design was working as expected.

At first, we were only concerned with whether our code was producing the correct output. We were confident that the muxes and registers were implemented correctly since they were provided to us, so we didn't feel a need to unit test them before moving on to the full integration testing. We began testing our designs with blackbox testing, which is only concerned about generating the correct output given the inputs, not with the implementation. This allowed us to verify the functionality of our code without worrying about the specifics of what's going on internally. All of the tests in *Table 2* are examples of this blackbox integration testing.

Once we determined that our alternative design was producing the correct outputs, we added line tracing, which is white-box, to ensure that we were progressing in the way we intended. We wanted to make sure that we were shifting the inputs in a way that we expected, and that we were changing states as expected. To do this, we ran the small_pos_pos test case with the -s flag. This allowed us to view the line trace, and see how the inputs, outputs, and shift amount were changing with each cycle. We repeated this with some of the other test cases, and felt that our alternative design progressed how we expected. This felt very much like an ad-hoc test since it relied on us observing what the data looked like. Since this was ad-hoc, it doesn't show up in our test case table. Doing this also allowed us to unit test our pattern module, as we could confirm that it was outputting the correct amount to shift by and that the operands were actually shifting by that amount.

We decided to use assertion testing because we wanted our test cases to be systematic and automatic; we wanted a test suite where the output is automatically checked for us. We implemented this by explicitly writing assertions that had to be true in order for the test to pass. All of the tests in *Table 2* are assertion test cases.

We wanted to use a mix of random and directed tests on our designs. We used directed testing to target specific input patterns and corner cases, and we used random testing to improve coverage and further ensure that our designs are functioning properly. With only directed cases, there's a chance that our implementation only works for those specific values, and random testing eases that fear. Random testing improves code coverage and helps bring to light any bugs we may have missed in our directed tests. We implemented directed testing by specifying specific inputs and expected outputs for those inputs. We implemented random testing by generating two random numbers for our inputs, then multiplying them together using functions that are known to work, and put that as our expected output. Rows 1, 2, 3, 4, and 6 of *Table 2* are directed tests, while row 5 is random tests.

We wanted to use value testing to ensure that our multipliers were producing the correct outputs given the inputs. If the value of the output is incorrect given the value of the inputs, then the multiplier is not functioning properly. We did this by using assertions tests to ensure that the expected value of the output is produced. All of the tests in *Table 2* are value tests.

We also wanted to have some delay tests. In addition to ensuring that the multiplier is producing the correct value, we also want to ensure that it can handle arbitrary source and sink delays, as we want to make sure that our multiplier is latency-insensitive. We implemented this by calling some of our previously created test cases with source and/or sink delays. None of the test cases in *Table 2* are explicitly delay tests, but any one of them can be by specifying a source or sink delay when calling the test case.

We created some simple positive and negative tests to test basic functionality, large positive and negative numbers to ensure functionality was consistent over all 32 bits, overflow tests to make sure that was working the way we expected, and masked, sparse and dense numbers to increase our test coverage, ensure our code is working with a variety of input types, and compare our basic and alternative implementations. The masked, sparse and dense test cases in particular ensure that our alternative implementation is improving the number of cycles wherever possible when we view the line traces. As our implementation shifts proportionally to the number of zeros in a row, we must test that it works when there are a lot of zeros in the beginning, middle, and end of our input. We feel that with all of these different testing strategies and test cases, we have ensured that our code works properly. We have a lot of code coverage with a lot of different types of inputs.

	Test name	Target Hardware	Number of tests	Passing?
1	small_pos_pos	Most basic functionality	5	yes
2	small_pos_neg, small_neg_neg	Basic negative number handling	5, 5	yes, yes
3	large_pos_pos, large_pos_neg, large_neg_neg	Overflow and underflow	4, 4, 4	yes, yes, yes
4	mid_ord_msk, sprse_num, dense_num	Alternative implementation, specifically the updated shifter	4, 6, 4	yes, yes, yes
5	random_no_delay, random_w/_src_delay, random_w/_sink_delay, random_w/_source_sink_delay	Improve coverage	30, 30, 30, 30	yes, yes, yes, yes
6	best_case, worst_case, ave_case	Best case and worst case are corner cases with all 0s and all 1s for the input. Also implemented these for evaluation purposes	1, 1, 1	yes, yes, yes

Table 2: Test Cases

4. Evaluation

Our alternative design is better than our baseline design. While the cycle time of the alternative design will be longer than that of the baseline design, the number of cycles required in the alternative design is, on average, much smaller. In the worst case scenario, where every bit of each input is one, the alternative design completes in the same number of cycles as the baseline design. In the average case, which we produced by thinking of a typical multiplication that we might complete in day-to-day life, the alternative design requires fewer cycles than the baseline design. In the best case scenario, where every bit of each input is zero, the alternative design completes in just three cycles, while the baseline still takes 35 cycles. While the alternative design does have more logic, making the cycle time a little bit longer, most inputs have at least some bits set to zero, allowing the multiplier to shift multiple times and reduce the required number of cycles, thus improving efficiency. *Figure 5* shows the cycle times of the best, worst, and average cases.

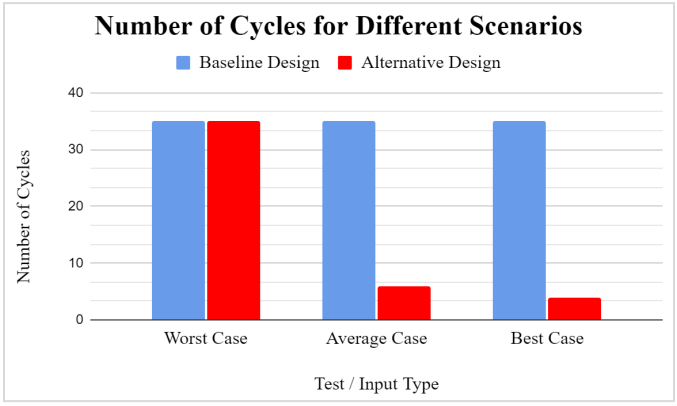

Figure 5: Number of Cycles for Best, Worst, Average Cases

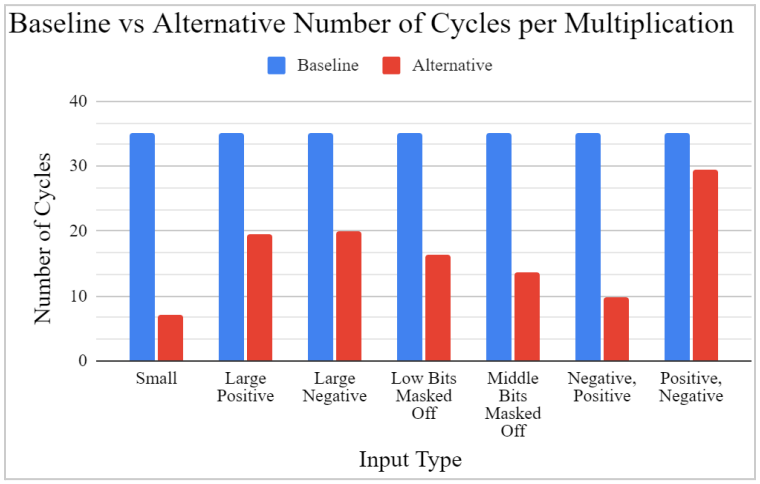

Figure 6: Number of Cycles for Different Input Types

If we were to add a lot more logic into our alternative design, the cycle time might increase so much that even though we are reducing the number of cycles, the total execution time is the same. Our current design would have better performance if we tell the user to list the number with biggest magnitude first and the smallest magnitude second, in which case the negative, positive and positive, negative cases in *Figure 6* would complete in the same amount of cycles on average. We thought about implementing our alternative design according to the datapath in *Figure 7* (load the input with the most zeros in the b register) in order to automatically do that for us, but we were concerned that it would increase our cycle time too much, and when we use it for later labs, we can order the inputs in the way described earlier. *Figure 7* shows how many cycles each implementation takes for a variety of input types.

From *Figure 6,* we can see that our alternative design consistently requires fewer cycles than our baseline design. Our worst performance in that graph is with the positive, negative inputs. This is because we input a smaller magnitude negative number into our b register, which means that most bits are one, so we cannot shift multiple times very often. On average, the multiplier will be used on smaller numbers, or numbers with a lot of zeros in the most significant bits. The alternative implementation will be an improvement, because all the cycles that would be dedicated to shifting through the upper zeros can be handled in one cycle.

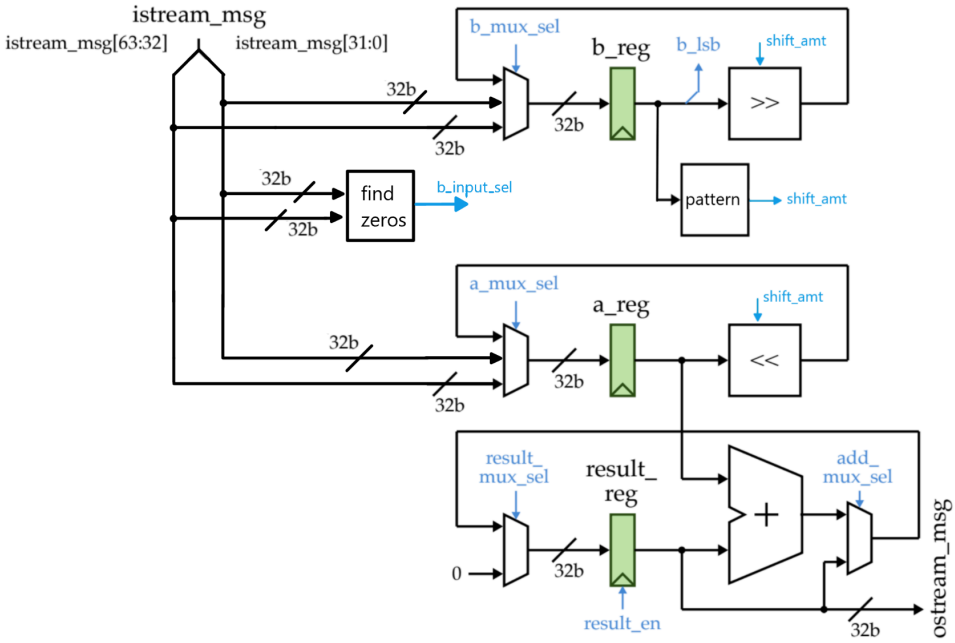

Figure 7: Different Alternative Design - Put Number with Most Zeros in B Register

The cycle time in our alternative design is longer than that in the baseline design. We believe that the critical path of our baseline design starts with the result register, moves through the adder, moves through the add mux, moves through the result mux, and ends back at the result register. We think the critical path of our alternative design starts at the b register, goes through the pattern module, goes into the control unit, goes into the right shifter, goes through the b mux, then goes back to the b register. The critical path of our alternative design seems much longer, with components that will take up more time, so the cycle time of our alternative implementation is longer than the cycle time of our baseline implementation.

Adding another piece of functionality to our implementation requires additional hardware. Our alternative design checks the b input to see how many times we should shift by using a cesez statement to implement a priority encoder. Since we check 32 conditions in this casez statement, we need quite a few extra gates from our baseline design. Thus, our alternative design uses more area than our baseline design.

The additional functionality also means that more energy will be required to process each cycle of the alternative implementation. However, because there will be, on average, fewer cycles, the overall energy consumption will be smaller in the alternative design than in the baseline design.

5. Conclusion

Our alternative design performed better than our baseline design with all inputs that contain at least one bit that is set to zero in the b operand. When there are a lot of ones in our b operand, the cycle difference is not large, but when there are a lot of zeros, there is a large difference between the two implementations. The small changes from the baseline design allow the alternative design to take advantage of the properties of the inputs. These changes were not very difficult to implement, and in the average case decreased the cycle count by a significant amount without increasing the cycle time very much. Though the worst case scenario is worse for the alternative algorithm, the average case is better, so this is the implementation we should use in the future.

6. Work Distribution

Ashley and Anne worked together on the original datapath and control system. Ashley completed the evaluations and the modifications to implement the alternative multiplier, and Anne created the tests. Both of us worked on the lab report, focusing on the sections that connected most directly to the parts that we each coded.