Lab 1 Report
Elise Song (eys29)
Katherine Zhou (kz273)

Introduction

In this lab, we were tasked with implementing two integer iterative multipliers., one that has a fixed latency and one with a variable latency. In the first part of the assignment, we implemented the baseline multiplier design that was provided to us in the lab handout. This multiplier has a fixed latency, meaning that it takes the same number of cycles to perform the multiplication every time. In the second part of the lab, we designed and implemented a variable latency multiplier, which has improved cycle time compared to the baseline design. The fixed latency multiplier checks the least significant bit of the second argument every cycle, and if it is zero then it shifts the argument right, if it is one then it puts the bit through the adder and then shifts. As we can see, it goes through every single bit, regardless of how many zeros are adjacent to each other. The variable latency multiplier is instead able to keep track of adjacent zeros and shift by that number of zeros, cutting down on the total number of cycles needed.

Through this assignment we are able to gain experience with the SystemVerilog hardware description language. Additionally, we are able to use the design principles that we learn in the discussion sections, such as modular and hierarchical design. We get to implement and understand good code patterns, such as the control and datapath split, which allows for our code to be easily readable and understandable. Since we are also responsible for coming up with test cases, the assignment also tests our ability to write test cases with sufficient coverage to ensure complete functionality of our design using the testing strategies that we have discussed in class.

A multiplier is a component in a microprocessor, and it is used to implement the MUL instruction in the RISC-V ISA which we will be using in this class, and thus the multiplier that we implement in this lab is relevant to the larger picture of this class. Additionally, it allows us to better understand the way that the multiplier works, since it is not a component that is seen in previous classes. It also fits into our understanding of cycle time which we have recently touched on in class, since by shortening the number of cycles that the multiplier takes, we would potentially be able to shorten the critical path and allow for our processor to run at a higher clock frequency.

Alternative Design

Our alternative design uses the idea of shifting over consecutive 0 bits to decrease the number of cycles, since shift is the only operation when the LSB of b is 0 in the base design. Our strategy is shown in the following pseudocode:

```
b16 = b_reg_out[15:0]
if b16[0] == 1:
       shift by 1 and add
else if b16 == 0:
       shift by 16
else if b16[8:0] == 9'b100000000:
       shift by 8
else if b16[4:0] == 5'b10000:
       shift by 4
else if b16[2:0] == 3'b100:
       shift by 2
else:  shift by 1
```

The **b16** value is outputted by the datapath and the control unit assigns the shift amount, called **incr** that is used by the shifters in the datapath. The FSM for the Control Unit is shown in Figure 3. We chose the shift amounts to be powers of 2, so in the case there are a lot of consecutive 0's, we check half the length of b, and half of that length, and so on. If we checked the same number of bits, a long stretch of 0's would take more cycles than in our strategy. This strategy allows us to be fairly aggressive with our shifts without sacrificing efficiency.

In order to shift by a variable number of bits, we also had to adjust the counter that keeps track of the number of bits shifted. Thus, we wrote an Incremental Counter (incrCounter) that could increment the counter by the number of bits shifted. The incrCounter takes as input: a **clock**, **clear** signal, and **incr** (shift amount) signal from the Control Unit. The module outputs a signal for the Control Unit called **is_max** when the counter reaches a value greater than 32. Shown in Figure 1, the **incr** is added to the counter, and the module is tied to a clock using registers.

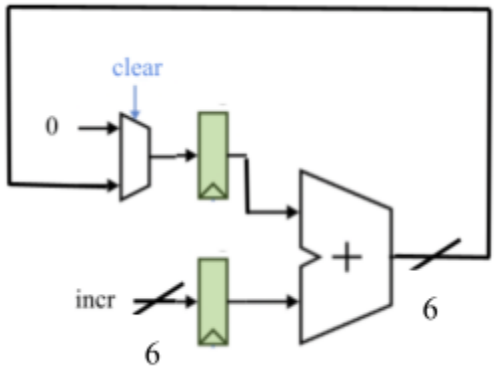

Figure 1: Incremental Counter

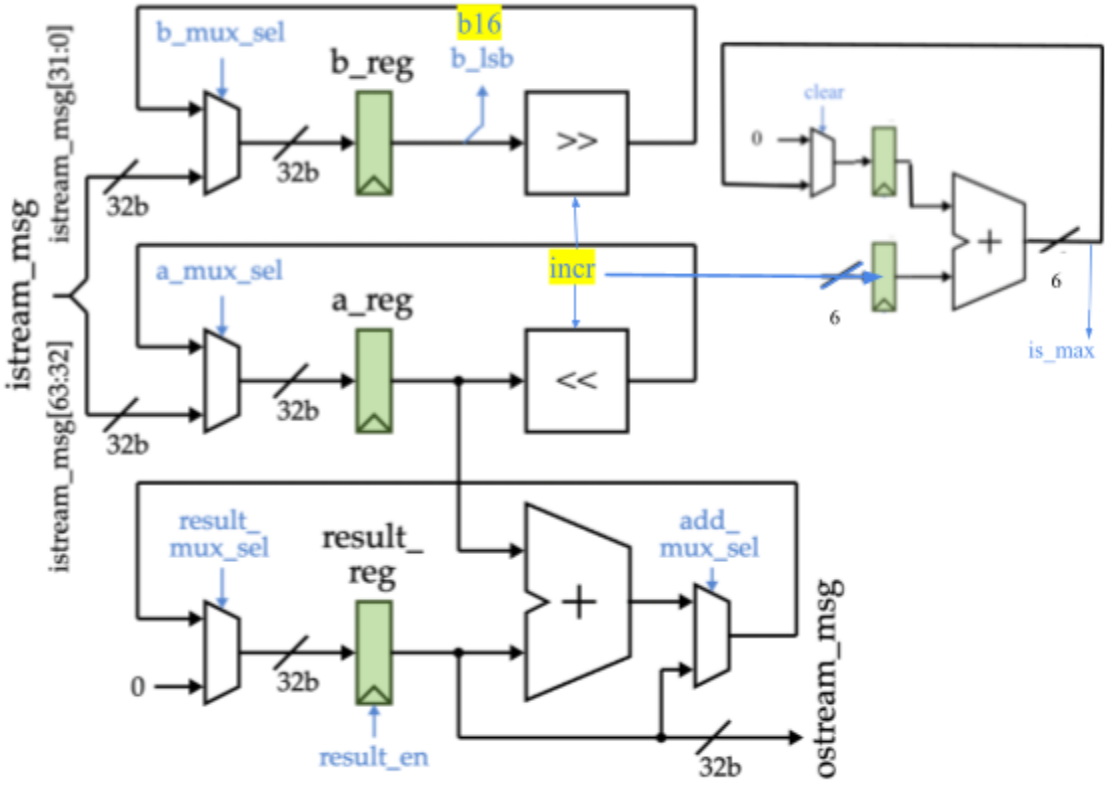

Figure 2: Incremental Counter in Base datapath

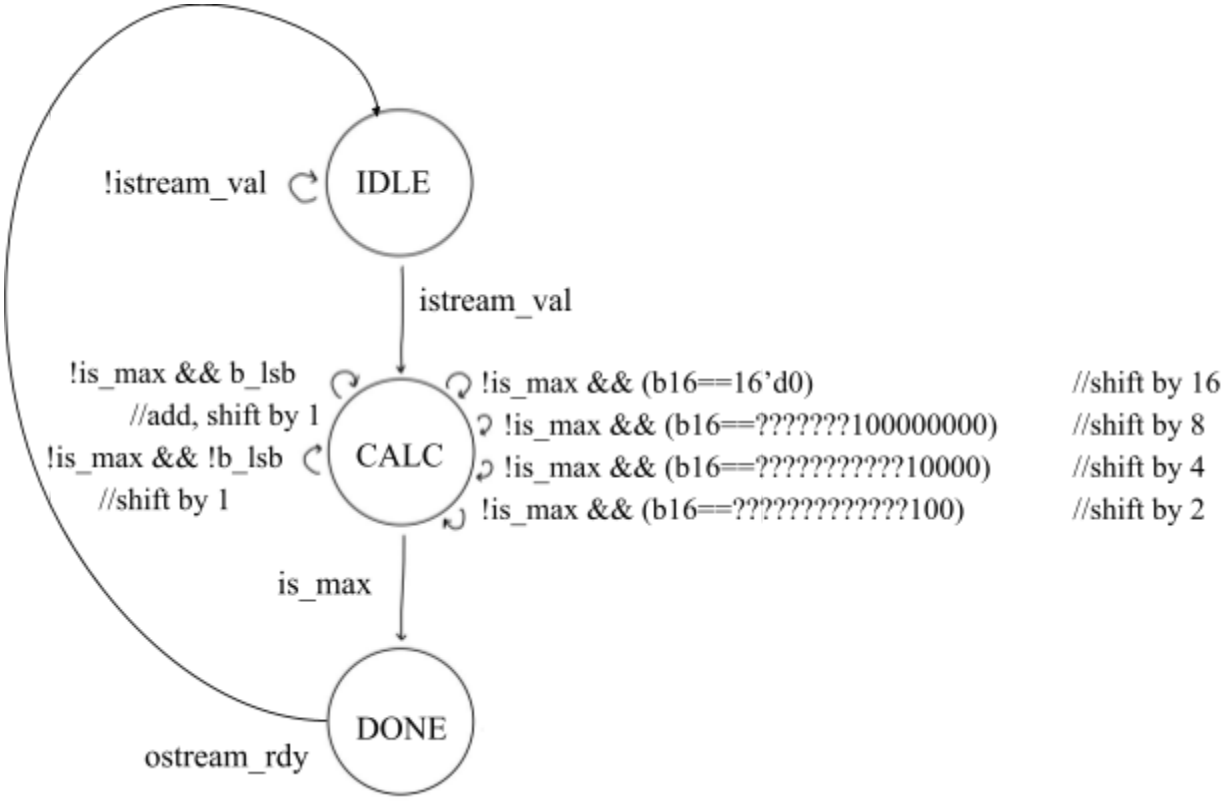

Figure 3: Finite State Machine

For example, given b = 0000 0000 0000 000 0000 0000 0010 0100, the **incr** set by the control unit and **counter** value is shown in Table 1.

Table 1: Step through of cycles using alternative design

Cycle	b_right_shift_out	incr (shift amount)	counter
0	0000 0000 0000 0000 0000 0000 0010 0100	2	2
1	0000 0000 0000 0000 0000 0000 0000 1001	1	3
2	0000 0000 0000 0000 0000 0000 0000 0100	2	5
3	0000 0000 0000 0000 0000 0000 0000 0001	1	6
4	0000 0000 0000 0000 0000 0000 0000 0000	16	22
5	0000 0000 0000 0000 0000 0000 0000 0000	16	38 → is_max = 1

We implemented modularity by separating the Control Unit and Datapath. When we implemented a new module, we inserted the Incremental Counter into the Datapath, which also demonstrates hierarchical design. The Datapath and Control Unit communicate through signals such as the selects. We only added two signals in the alternative design: **incr** and **b16** which demonstrates encapsulation.

Testing Strategy

We decided to mainly focus on black box testing in order to ensure the overall functionality of our design. Following this, we included directed tests to test any edge cases that we could think of as well as to implement some simple tests that were suggested in the lab handout. We made use of random testing to ensure that we did not miss anything in our design implementation and to guarantee broader coverage. Finally, we used delay testing to ensure that our design can handle arbitrary sink and source delays. A summary of some of our test cases is shown in the table below.

Table 2: Summary of Test Cases

a inputs	**b inputs**	**src delay**	**sink delay**
Large positive integers	Large positive integers	0	0
Small positive integers	Small negative integers	0	0
Zeros, ones and negative ones	Zeros, ones and negative ones	0	0
Random integers	Random integers	0	0
Masked integer	Random integer	0	0
Any integer	Dense integers	1	8
Any integer	Sparse integers	9	7
Random integers	Random integers	Random integers	Random integers

Our overall testing design methodology makes use of black box testing, since we are not basing any of our tests on the implementation.

After our implementations were able to pass the provided test case, we moved on to directed tests following the guidelines that were provided in the lab handout. The first test that we did was multiplying large positive integers together to ensure that the multiplier would be able to correctly do this. When doing this test, we did not take into account the sparseness or denseness of the zeros and ones in the b input. The “large” size was also determined relative to the provided “small” numbers.

After this, we moved on to multiplying a positive number with a negative number. This was done to ensure that we did not need to convert any inputs to or from two’s complement, and that the multiplier would be able to correctly handle negative numbers. We wanted to ensure that the output would correctly be a negative number. This test inadvertently also tests numbers with a dense number of oncs, since small negative numbers have many leading ones.

Next, we tested the multiplier’s ability to multiply by zero. We did this by multiplying together a combination of zeros, ones, and negative ones. Here, we see that the multiplier is able to successfully multiply a number by zero. This can also be considered an edge case, since having the b input be zero would complete the operation in the smallest number of cycles that our design can complete an operation in.

After these tests, we were sufficiently convinced that our multiplier would be able to handle negative numbers, large numbers, and zeros, so we moved on to random testing. For this, we created an array filled with random integers, ranging from -2^{30} to $2^{30}-1$. This ensures that our multiplier is able to handle any number that it encounters as long as it is within the allowed range, which is 32 bits, with one bit being used as a sign bit. Testing using random inputs also ensures that we are testing on unbiased numbers, and with combinations of inputs that we may not think of. Having the random tests pass helps us to increase our test coverage.

We also tested cases where the **a** input or the **b** input were masked off in either the beginning, middle, or end of the number. In order to limit redundancy in our table, only **a** input masking is listed.

Next, we tested our delays using the sink and source delays in the test file. We tested cases where the sink is greater than the source, the sink is less than the source, and a case where they are both random integers from 0 to 10. This ensures that our multiplier is able to handle random delays.

After performing all of these tests, we were confident that our design is functionally sound. We chose to test this way because it was a clear way to test the functionality of the multiplier. Having directed tests where we can choose the inputs allows us to think about the possible combinations of inputs, and the random test cases help us to get more coverage. Finally, the sink and source delay testing helps us to guarantee that the multiplier is able to handle latency, even if it is an arbitrary number.

Evaluation

Our alternative design was always able to perform better or at the level of the baseline design in terms of the number of cycles it took to execute each multiplication operation on our random test. A histogram of the number of cycles per multiply of the alternative design is shown in Figure 4.

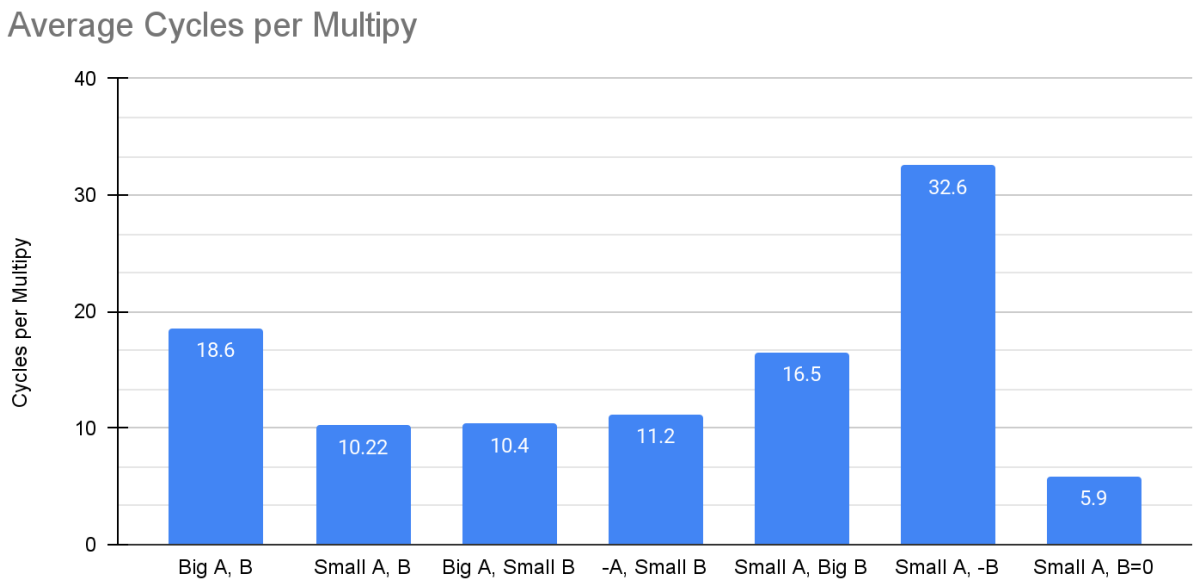

Figure 4: Cycles per Multiply on 6 datasets

The 6 datasets are each displayed as a bar in Figure 4. The name of the bar has a description of the two operands A and B, where a “Big” number is between 0 and 32000. The [Small A, B=0] bar has the least number of cycles per multiply since our alternative design would use two cycles and shift over 16 0-bits twice. Similarly, the bars with [Small B] have lower average cycles per multiply since small values of B will contain more 0’s that the alternative design can shift over. The bars with [Big B] have a greater average cycles per multiply, however, [-B] has an average close to 35.

As discussed, the alternative design varies on how much the number of cycles per multiply decreases, depending on B. Based on our strategy and implementation of our alternative design, we know that the instruction that took the fewest number of cycles was most likely an instruction that had operands with a lot of consecutive zeros. Our design was fairly aggressive, with the potential to shift up to sixteen bits at once. A different implementation might show less variation in the number of cycles, for example if it only allowed to shift by up to four bits at once. Additionally, operands with a smaller number of consecutive zeros would not show as big of an improvement in the number of cycles. Below are the line traces for a few of the provided simple test cases, and we can clearly see that there is a large improvement in the number of cycles. This follows our expectations, since the operands are small and have a large number of consecutive leading zeros.

In addition to the number of cycles, our alternative design also differs from the baseline design in terms of the clock frequency, area, and energy that the design uses. Since we added an additional module, our incremental counter module, the alternative design uses more hardware than the baseline did, leading to it taking up more area. This additional hardware also causes the alternative design to use more energy than the baseline. However, the critical path of the total implementation does not change, since our additional module only contains a path through a mux and an adder, while the baseline implementation contains a path through two muxes and an adder.

Our alternative design adds a six bit two input adder, a 2 input mux, and two registers to the existing hardware from the baseline implementation. There are on average about 15 million transistors per square millimeter.

An adder, which is the largest component that we add, contains less than 100 transistors, which proportionally would not even take up 2000 square microns. When taking this into consideration, the area increase does not seem very significant, but when you put the increase in relation to how large (or really, how small) the baseline design is, it would probably be about a 15-25% increase in area. This number is purely an estimation based on the relative increase in the hardware that is added to the alternative design.

Since the alternative design uses additional hardware, it is reasonable to assume that it also consumes more energy, since each part uses energy. Since they are built at the gate level, they probably consume power on the order of micro watts. However, using the same argument as before, the baseline is also using very little energy, and so an increase in energy consumption in the alternative design would not be negligible when put in relation to the baseline.

As stated earlier, the critical path of our alternative design is the same as the critical path of the baseline design. This is because the incremental counter module that we added is essentially self contained, and only connects to the left and right shifters and the global clock. All the other outputs of the module loop back onto itself and the input is a value that we generate based on the number of consecutive zeros. The self loop path of the incremental counter contains only an adder and a mux, which is shorter than the critical path of the baseline design, which contains an adder and two muxes. Thus, the critical path of our alternative design and the baseline design are the same, and so the two designs are able to run at the same clock frequency.

Based on the above discussion, the alternative design is better than the baseline design because of its improvement in the number of cycles used per instruction coupled with the equal clock frequency. The decreased redundancy in the alternative design, which is what leads to the decreased number of cycles, is also preferred over the baseline design.

Conclusion

We found that our alternative design was always able to execute the operations in as many or fewer cycles than the baseline model, which was the expected behavior. Using a random test case as an example, the alternative design took fewer cycles on average as opposed to the 35 cycles of the baseline design, depending on the dataset which it was run on. In the cases where there were a lot of consecutive zeros, we were able to cut down on the number of cycles by a greater rate, since our design could potentially shift by 16 bits. In cases where there were very few consecutive zeros, the alternative design performed about the same as the baseline design, averaging around 32 cycles per instruction. From this, we can predict that as the number of consecutive zeros increases, the number of cycles it takes to perform the multiplication decreases. In the future, the alternative design should be used due to its decrease in cycle number and unchanging clock frequency when compared to the baseline.

Work Distribution

Overall, we were able to find time to meet up and work on the code together. We found that coding the assignment together was a good way to ensure that we both understood what was going on. We went to office hours, separately and together, when needed to ask any questions that we had. We split the lab report by section, although we agreed to work on the main sections (testing strategy and evaluation) together. Katherine did the introduction, conclusion, and work distribution, and Elise did the alternative design. We plan to switch the roles for each lab.