# Lab 1: Iterative Integer Multiplier

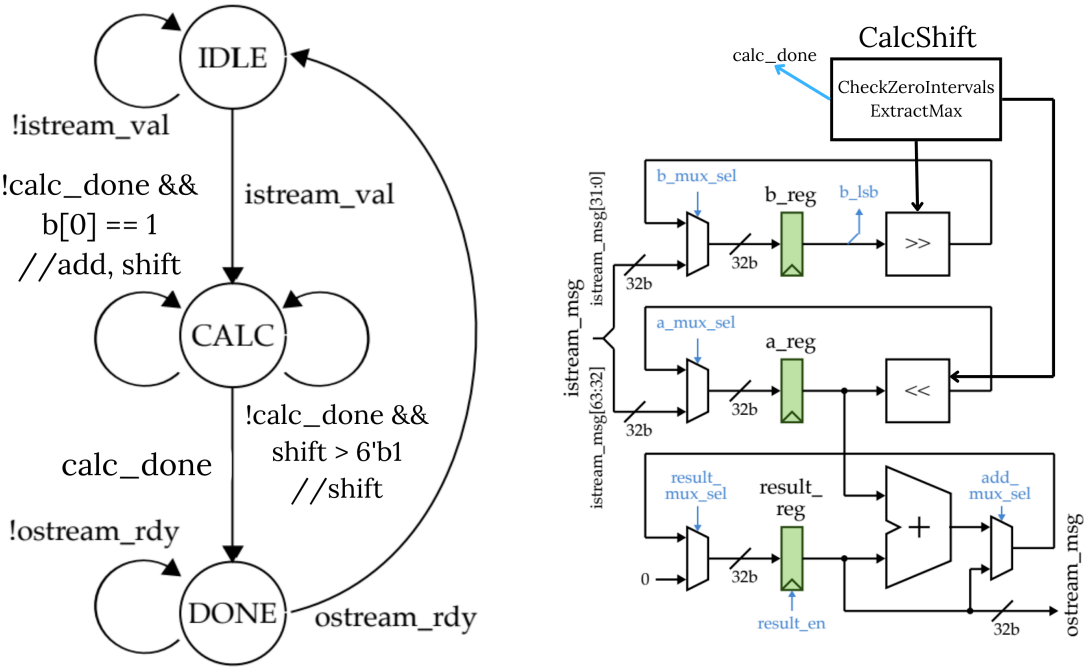
Asma Ansari (ara89) & Nathan Vogt (ncv7)

## I. Introduction

This lab's objective is to familiarize students with fundamental course concepts, such as hardware description languages, design hierarchies and tradeoffs, and testing principles. Students were expected to implement two versions of the iterative integer multiplier: fixed and variable latency. Our group utilized the data path and control unit diagrams from the lab handout to develop our base design. From there, we adjusted the data path and control unit slightly to accelerate multiplication in our alternative design.

## II. Alternative Design

Our alternative design built upon a working base design with nearly identical signals communicated between the datapath and control unit. The alternative design includes one additional module encapsulating logic calculating the bit shift amount within one clock cycle. Furthermore, we updated the control signals within the control unit such that the counter's value no longer determines when the CALC stage ends.




**Figures 1 & 2:** Updated control unit diagram with new signals from datapath (left); updated data path diagram including encapsulated modules producing new signals to communicate with the control unit.
*Credit: Figures 3 & 4 – ECE 4750, Fall 2024, Lab 1: Iterative Integer Multiplier*

Figure 1 and 2 illustrate the minor adjustments made to control unit and data path diagrams from the lab handout document. As mentioned above, the *CalcShift* module utilized encapsulation and modular design so that testing individual components would be easier (see more in "Testing Strategy"). Internally, the *CalcShift* module contains two modules called *CheckZeroIntervals* and *ExtractMax* which determine the bit shift amount. Previously, the values in *b_reg* and *a_reg* were shifted by 1 bit per clock cycle which locked the latency to at least 32 cycles, accounting for each bit in the register.

In the alternative design, *CheckZeroIntervals* takes in *b_reg*'s value and generates 32 *ZeroComparator* modules which take in subsets of the value contained in *b_reg* and outputs a boolean value. This boolean is true if the subset contains a string of zeros and false if not. We stored these boolean values in a 32-bit intermediary register called *zero_interval_encoding* where the most significant bit contains the boolean associated with the 32-bit subset of *b_reg*'s value while the least significant bit contains the subset of *b_reg*'s least significant bit.

Once all possible subsets have been analyzed combinationally, *zero_interval_encoding* is passed into *ExtractMax* which uses its value as a selector for a **casez** statement. The **casez** statement acts as a priority encoder, starting from checking the most significant bit in zero_interval_encoding all the way until the least significant bit. Thus, once the maximum subset size has been selected, the **casez** statement ends, and the index becomes the output.

As mentioned, the *CalcShift* module connects these two modules and outputs the final bit shift amount. *CalcShift* adds one onto the output from *ExtractMax* since its values range from zero to 31, aligning with the index of the input register. This shift value is passed into the shifters seen in the diagrams after *b_reg* and *a_reg*. Additionally, the shift value's most significant bit determines whether or not a calculation is done which is a signal passed to the control unit. Once an interval of 32 consecutive zeros is identified, the shift value will be set to 32 which is represented in a 6-bit register; thus, the sixth bit going from 0 to 1 only occurs once *b_reg* equals 0.

These modules are entirely combinational logic, so they operate in parallel during the one cycle. The rest of the design is retained from the base design, so the IDLE and DONE states were not optimized out. We kept the FSM the same to simplify our evaluation of area, energy, and cycle time later on in the lab report.

## III. Testing Strategy

We developed our testing strategy using the provided test harness as well as the Verilog tutorial provided on the class's website. From the suggestions made in the lab handout as well as the milestone feedback, we implemented test cases particular to the base design: positive/negative number combinations, large/small numbers (including overflow), and src/sink delays. For the alternative design, we included test cases with "masked off" bits to ensure that our bit shift optimization was working. The existing test cases for the base design were included to verify that correct calculations were being made.

Our tests included traces that tracked the current finite state machine (FSM) state as well istream_msg (input message) and ostream_msg (output message). These traces informed us of any issues with the FSM's progression and the calculations' correctness without having to look in individually generated VCD files for each test case. Regardless, the VCD files were crucial for debugging since they illustrated internal signals, allowing us to find the cause of a particular bug.

While developing our alternative design, we utilized unit tests to verify the behavior of three modules we had instantiated: *CalcShift*, *ExtractMax*, and *CheckZeroIntervals*. Thus, unit tests were crucial to debugging since we had increased the area of our design. Furthermore, we also had a major issue with our FSM's progression, so without these unit tests, we would not have been able to rule out whether or not our new modules were causing issues with the state transitions.

On the other hand, integration testing allowed us to ensure that our modules were instantiated and wired correctly. As mentioned above, our FSM did not progress from CALC to DONE due to a signal not being set correctly in the DataPath module. Integration testing allowed us to view all the signals alongside each other since the signal that determined the FSM's progression had several dependencies in the DataPath module. The counter we used for our base design also served as a testing method when looking at waveforms since we no longer had a fixed latency for our calculations, so using the counter allowed us to track average cycle time manually.

Ad-hoc, stream source/sink, and FL models were also part of our testing strategy. These were a part of the provided test cases that we built upon. The FL model allowed us to understand what the correct behavior of our program should look like. Ad-hoc and stream source/sink testing allowed us to implement corner cases into our testing design to provide coverage to our implementation's correctness. Since these strategies were provided to us, we were able to build more complete test vectors and learn how to create our test files, as mentioned earlier regarding the unit tests.

## IV. Optimization

The most crucial optimization we made to the base design was including modules that could provide a variable bit shift amount. In the base design, the messages in *a_reg* and *b_reg* were only allowed to shift by one bit per clock cycle even if *b_reg* had several consecutive zeros in its message. The importance of this is that if *b_reg*'s least significant bit is zero, then both *a_reg* and *b_reg* would shift but *a_reg*'s message would not be added to the *result_reg*'s value. Thus, shifting every consecutive zero in one clock cycle eliminates the additional clock cycles required for the program to go through each of *b_reg*'s least significant bits as the message gets shifted over.

An additional optimization was to remove the IDLE and DONE states which would eliminate the FSM within the program. Due to time constraints, we chose to focus on the bit shift optimization since it had the most significant effect on the total cycle time. Saving these two extra cycles were not as important as the amount of cycles that could be saved by shifting multiple bits in one cycle. Furthermore, the evaluation of these programs qualitatively becomes simpler the more similar the designs are.

## V. Evaluation

**Performance Analysis**

| Input Dataset | num_cycles_per_mul (alt) | num_cycles_per_mul (base) |
|---|---|---|
| small | 8.00 | 35.06 |
| big | 14.76 | 35.06 |
| posneg | 23.06 | 35.06 |
| ones | 31.90 | 35.06 |
| zeroes | 6.40 | 35.06 |

**Table 1:** Evaluation of average number of cycles per calculation between alternative and base designs.

In evaluating the performance of the base and alternative designs, the most prominent difference is the reduction in the number of cycles required for multiplication in the alternative design compared to the base design. The base design required a fixed 35 cycles per multiplication, regardless of the specific input values. This fixed latency was due to the one-bit-per-cycle shift mechanism in the base multiplier, which processes each bit sequentially in b_reg.

In contrast, the alternative design achieved significantly lower cycle counts, ranging from 6.4 to 31.9 cycles, depending on the input dataset. This improvement stems from the optimized bit-shift logic, which can handle multiple consecutive zero bits in one cycle, greatly accelerating the multiplication process when contiguous zeros are encountered in *b_reg*. For example, in the "zeroes" dataset, where *b_reg* contains many consecutive zeros, the number of cycles dropped to 6.4. Conversely, the "ones" dataset, where no zeros are present, required the highest number of cycles (31.9), but this is still better than the fixed 35 cycles in the base design.

The 'flip' dataset was specifically designed to explore the relationship between the number of contiguous zeroes in the input and the number of cycles required for multiplication. In this dataset, a specified number of bits are randomly flipped in both *a_reg* and *b_reg*, generating input values with varying numbers of contiguous zeroes. By introducing this variation, the dataset provides a spectrum of cases, from inputs with few zeroes to those with many consecutive zeroes.

This allows us to observe how effectively the alternative design's optimization—skipping over blocks of zeros in *b_reg*—reduces the cycle count. For inputs with large stretches of contiguous zeroes, the alternative design is expected to significantly reduce the number of cycles by performing larger bit shifts in one clock cycle. Conversely, when the bit flips break up these zeroes, the number of cycles required will increase, as fewer zeroes are

available to be skipped. This dataset highlights the dynamic adaptability of the alternative design in response to the bit patterns in *b_reg*, showing a clear relationship between the number of contiguous zeroes and the performance gains in cycle time.

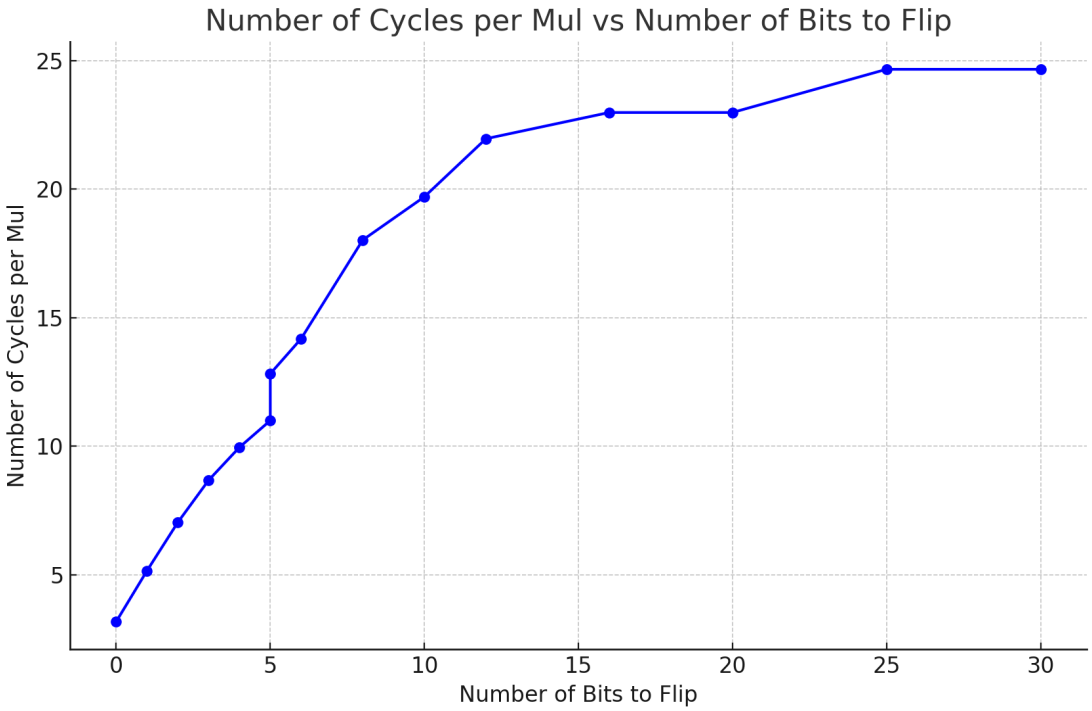



**Figure 3:** Illustration of average number of cycles for a calculation versus the number of bits flipped.

Overall, the alternative design demonstrated a significant performance boost, particularly in scenarios where zeros were prevalent in *b_reg*, leading to fewer clock cycles per multiplication. The variable-latency approach provided an adaptive mechanism that aligned the cycle count with the complexity of the input data, making the design more efficient for a range of input values.

**Area Analysis**

The area of the alternative design increased due to the inclusion of additional logic modules, such as *CalcShift*, *CheckZeroIntervals*, and *ExtractMax*. The *CheckZeroIntervals* module simultaneously checks for 32 different interval lengths of zeros using *ZeroComparator* components. The *ZeroComparator* modules are fairly simple, so this *CheckZeroIntervals* module doesn't add a significant amount of complexity to the design.

The *ExtractMax* module acts as a mux that selects the highest possible index value from *CheckZeroIntervals* output which is only one additional component to the area. However, it is important to note that some area was reclaimed by removing the counter module, which was a necessary component in the base design to track the number of cycles.

The increase in area can be justified by the tradeoff between performance and resource usage. The alternative design, while more complex, provides substantial gains in cycle time, making it a more suitable option in scenarios where performance is critical and area constraints are less stringent.

We were able to encapsulate the variable shifting logic into a single module that gets instantiated in the data path. This helped keep our design and code clean. Most of the design from our base multiplier was left unchanged since the *CalcShift* module naturally fit into the preexisting logical flow of the system, which is a sign that our chosen abstraction for the *CalcShift* module was correct.

**Energy Analysis**

Energy consumption in both designs is influenced by the number of clock cycles and the area of the design. In the base design, the fixed 35-cycle latency results in consistent energy usage across all input datasets. Although the base design's area is smaller, the fact that each multiplication operation takes 35 cycles means that the design is not energy-efficient for datasets that could benefit from faster computation, such as those with many zeros in *b_reg*.

In contrast, the alternative design reduces the number of cycles, which can lead to lower dynamic power consumption, as fewer cycles mean less switching activity overall. However, the increase in area introduces more static power consumption due to the added logic. Thus, the energy savings from fewer cycles may be offset, to some extent, by the increased static power consumption. Nevertheless, for datasets like "zeroes" and "small", where the cycle count was dramatically reduced, the overall energy consumption is likely lower in the alternative design. For more complex datasets like "ones", the energy savings would be less pronounced, but the alternative design would still outperform the base design in terms of energy efficiency due to the variable cycle count.

**Cycle Time (Clock Frequency) Analysis**

The clock frequency is tied to the critical path of the design, which is the longest delay through the logic between clock cycles. In the base design, the critical path was relatively short because the one-bit shift operation was simple and occurred over multiple clock cycles. This allowed for a higher clock frequency but at the expense of a longer total execution time (more clock cycles).

The alternative design introduced more complex combinational logic, particularly in the *CalcShift* and *ExtractMax* modules, which likely increased the critical path delay. As a result, the cycle time (i.e., the period between clock edges) may have increased, reducing the clock frequency slightly. However, because the total number of cycles was significantly reduced in the alternative design, the overall execution time was still much shorter. This means that despite a potentially lower clock frequency, the alternative design achieved faster results because it required fewer cycles to complete each multiplication.

## VI. Conclusion

Although the base and alternative design achieved the same end result, there are important differences between them that illustrate a cost-benefit analysis in using one design versus the other. Although our alternative design significantly optimized the cycle time, the design increased the area due to the inclusion of several additional modules. Understanding these tradeoffs is important to utilizing available hardware effectively – programs are only as efficient as the hardware that processes them. Thus, both designs are valid but can be used in different contexts depending on hardware limitations or other specifications related to a design’s energy, area, and cycle time.

## VII. Work Distribution

Both group members contributed equally to the project. To successfully complete the base and alternative designs, we made an effort to collaborate as much as possible. In implementing the alternative design, we worked independently on developing the modules needed to perform multiple bit shifts in one cycle and reconvened if we ran into issues. Throughout the development of both the base and alternative designs, both of us created test cases and simulation datasets. Nathan focused on the unit tests to test the individual modules we had instantiated for the alternative whereas Asma focused on filling out the general test cases that ensured both designs were functioning correctly. Both of us provided insight into each section of the lab report.