ECE 5745 Lab 1

Brandon Wood
bpw42
Alain Antón
aa2282

February 2023

1 Introduction

Lab 1 serves the purpose of building a foundation in using the ASIC flow, both front-end and back-end, with the ulterior motive of gaining insight and experience with fast iterative design exploration. This ties to the overall theme of the course of abstracting away from detailed and custom design methodologies, which focuses on "micro-improvements" in a given implementation, and instead focusing on multiple design implementations that could perhaps have a bigger impact on performance. This will be accomplished by first implementing an alternative design, a parameterized pipelined multiplier, and evaluating it against 3 baseline designs: a fixed-latency iterative multiplier, a variable-latency iterative multiplier, and a single-cycle multiplier. As the different implementations are taken through the flow, we will quantitatively examine key performance metrics such as area, energy and cycle time to then conduct a detailed design space exploration. To demonstrate the power of this approach, we present an alternative design, a pipelined multiplier, that is parameterized by the number of pipeline stages. While writing one piece of verilog, we can effectively explore many possible designs. After pushing these through the ASIC flow, we compare and contrast the performance of each design at the architectural level.

2 Alternative Design

We decompose the pipelined multiplier into sub-modules, shown below, that each perform "one bit" of multiplication. We take in two operands, A and B as well as a partial result as input (similar to a partial adder). We then return "A prime," "B prime," and "Result prime" after performing the necessary operations.

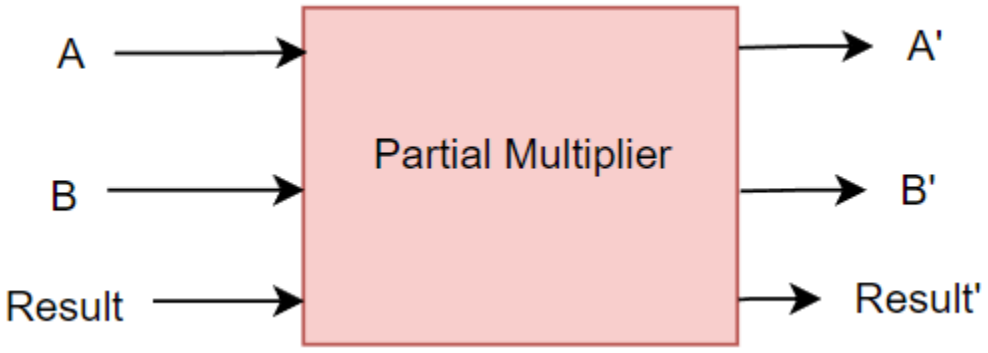

Figure 1: Interface of Partial Multiplier Sub-Unit

This partial multiplication can be best understood as a single cycle of the variable latency multiplier. We'll take in both operands, add A to the result, then mux that sum and the original result based on B's

LSB. We'll then bit-shift A and B (left and right, respectively), and push them and the partial sum out. The diagram in Figure 2 shows this schematically.

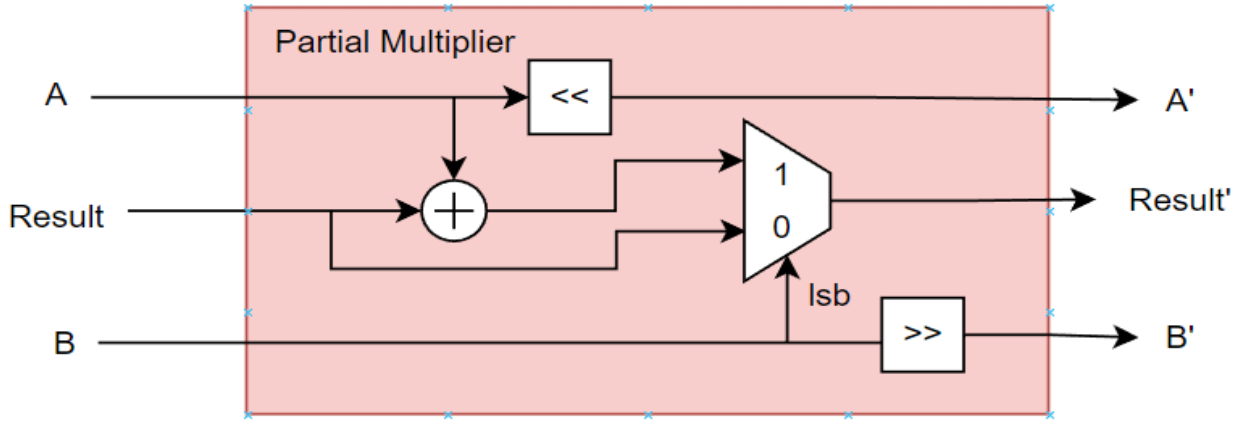

Figure 2: Internal Schematic of Partial Multiplier

We want to implement this sub-circuit in the first place because it gives us a convenient way of performing multiplications that can be broken up into sub-stages, allowing us to pipeline the processing.

That's what's shown in Figure 3 in the system-level view of our N-stage pipelined multiplier. We need to perform 32 multiplications, so we'll always have 32 partial multipliers (red boxes). However, we can insert registers in between any two stages following whatever algorithm you'd like. We chose to keep them evenly spaced, and to parameterize them by the number of stages. This is because we're looking to reduce the clock period by shortening the critical path (by inserting pipeline registers) while playing the regular pipelining trick of achieving approximately an instruction/cycle throughput once the pipeline is filled. Shown below, we immediately start with at least one pipeline register to catch the messages from the val/ready μprotocol. We then insert $\frac{32}{N}$ partial multipliers (giving an even distribution) then break the chain with another register. By doing this N times, we'll end up with 32 partial multipliers. Again, the idea is that we'll end up with a clock cycle that roughly an Nth of the single cycle multiplier while maintaining the same throughput.

There is a trade off here between area, energy, and timing. We expect the area to go up because you're adding more components and more routing with each stage. Additionally, we expect to consume more energy per multiplication because it take energy to store it, while you're passing it through the same number of partial multipliers. It's not obvious how many stages gives the best trade off between overall processing time, area and energy. However, by parameterizing it, we can evaluate many designs all at once and really quickly get some interesting data, before maybe doing some deeper optimizations down the road on the architecture that we decide on. e note that these block diagrams are suggestive of the verilog implementation. By leveraging modularity recursively (hierarchically), we were able to break down a complicated design into a few sub-modules (ie; the shifters, adders, registers, and muxes).

This demonstrates the power of this approach, because we were able to implement a fairly complicated circuit by iteratively composing these hierarchical levels. For instance, we wrote one partial multiplier then composed 32 of those and N registers using essentially a single for loop. We then added some additional circuitry that completed the encapsulation of our top-level module (ie; allowed us to obey the val/ready μprotocol), and we able to leverage the testing already setup from ECE 4750.

3 Evaluation

A holistic approach is taken into evaluating the different micro-architectures to determine under which constraints a particular implementation will prove to be most beneficial. For example, it is not uncommon to have a competing optimization between performance and power consumption, however, another dimension; area and in turn cost, will also be factored into the evaluation of these micro-architectures. Additionally,

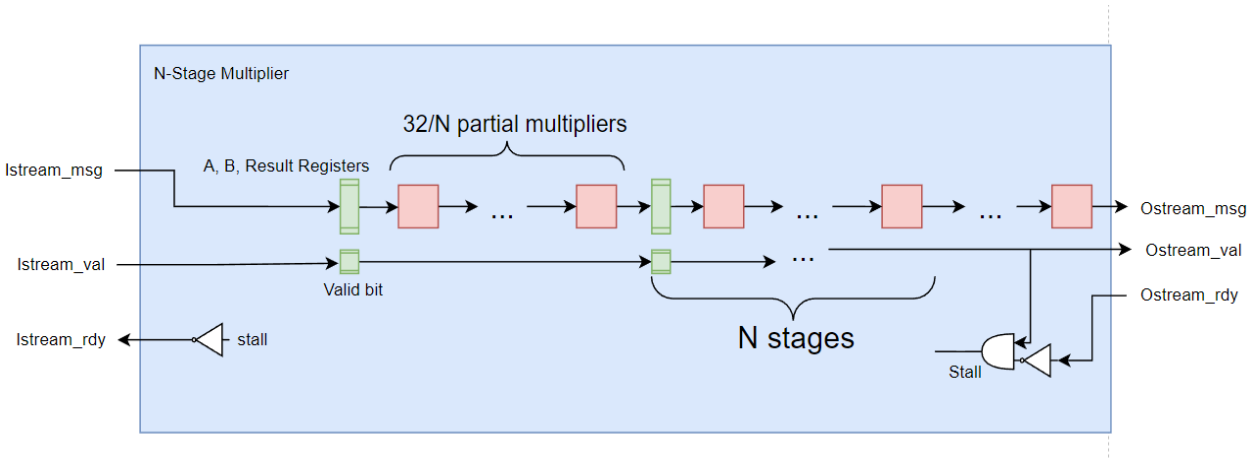

Figure 3: System Level Overview of N-Stage Multiplier

we will comment on how we believe the context of these implementations could effect their performance. Namely, how differing input data sets, and also physical context impact performance. We summarize key performance metrics in Table 1 on page 3 for all the micro-architectures. With further in depth-analysis being performed on subsequent subsections.

Multiplier Micro-Architecture Comparison Table				
Micro-Architecture	Area (μm^2)	Cycle Time (ns)	Tot. Execution Time (Cycles)	Tot. Energy (nJ)
Single-Cycle	2542.162	1.490	60 / 60 / 60	0.13 / 0.25 / 0.22
Variable-Latency Iterative	1685.908	0.597	377 / 1375 / 865	0.52 / 2.27 / 1.30
Fixed-Latency Iterative	1032.878	0.597	1759 / 1759 / 1759	1.66 / 2.13 / 1.89
1 Stage Pipeline	3907.274	6.960	60 / 60 / 60	0.22 / 0.57 / 0.45
2 Stage Pipeline	4183.900	4.774	61 / 61 / 61	0.25 / 0.60 / 0.48
4 Stage Pipeline	6091.930	1.996	63 / 63 / 63	0.35 / 0.78 / 0.64
8 Stage Pipeline	7442.680	1.489	67 / 67 / 67	0.54 / 1.01 / 0.85

Table 1: Multiplier Micro-Architecture Comparison Table (small/large/sparse)

3.1 Single-Cycle

The Single-Cycle multiplier was simply left up to the synthesis tool to implement and optimize given a simple $*$ operation between the operands. We can note from Table 1 that it achieves outstanding performance and a multiplication per cycle at the expense of a larger area overhead. Interestingly, the critical path of this multiplier, as seen in Fig.4 on page 4 has extra routing to the boundary of the implementation. This means that this extra unneccessary routing directly affects the maximum achievable clock frequency. We believe if there was more physical context, in other words, another block receiving the output of this block, this implementation could potentially have pushed a higher clock frequency.

3.2 Fixed-Latency

The Fixed-Latency multiplier algorithm essentially sets the number of cycles in an input sequence to be $32N$, where N is the number of multiplications. This is due to the fact that the algorithm completes a partial product every clock cycle and thus the only knob for optimization is the clock frequency. As expected, it can be seen in Table 1 that the total execution time is constant across the three input data sets. Ultimately, in increasing clock frequency, and thus power consumption, there is a limit as to what the process can support and the Fixed-Latency multiplier converges to it's maximum performance point.

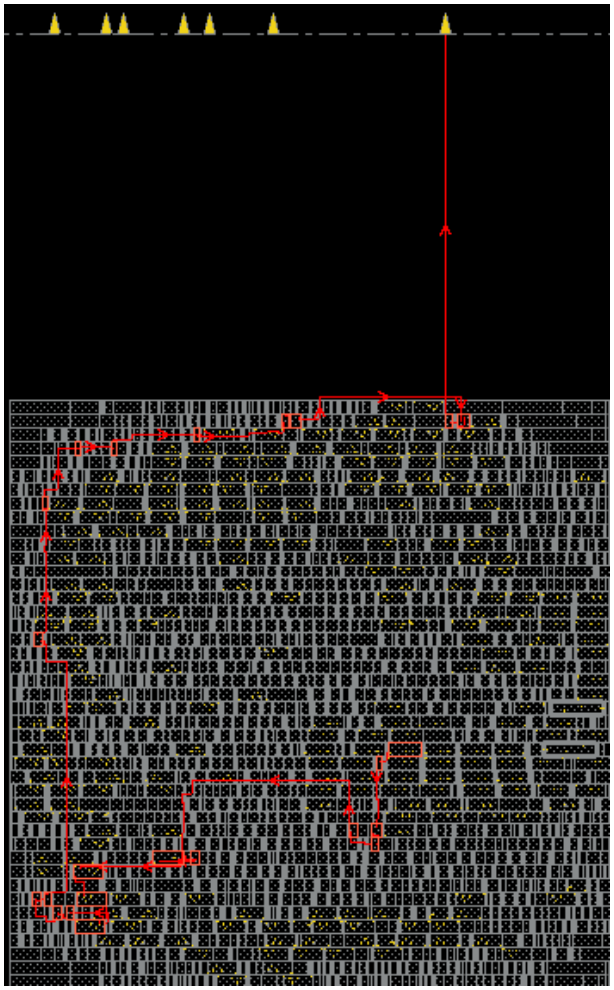

Figure 4: Single-Cycle Multiplier Amoeba Plot

3.3 Variable-Latency

The Variable-Latency multiplier has a similar structure to the Fixed-latency multiplier but it has a built in feature which takes advantage of the structure in some pairs of input operands to improve performance and energy efficiency. It has the ability to detect when there are consecutive zeros in one of the operands and shift by that same amount of consecutive zeros and thus saving those cycles of computation. This can easily be seen in Table 1 by noting the difference in total execution times between the different data sets. It takes this implementation 377 clock cycles for an input data set composed mainly of small numbers and around 1375 clock cycles for an input data set mainly composed of large numbers. It is interesting to note the number of cycles of the latter case approaches the number of cycles of the Fixed-Latency multiplier. This is expected as the "looking ahead" mechanism is less effective for larger numbers.

3.4 N-Stage

The N-stage multiplier was designed with the intention that it not really excel in one area or another, but that it allow us to explore a wide range of options and, by pipelining, allow us to get around 1 instruction per cycle throughput. We expect area and energy costs to increase with the number of pipeline stages, the number of cycles to remain the same once the pipeline is full(in "steady state"), and the overall cycle time to decrease. It's not immediately clear what should actually happen to each of the parameters, but we'd

expect, to the first order, for them to roughyl be linear with the number of stages.

That is not the case. While there are too few datapoints to fit a curve, we can see from Figure 6 that the area, while monotonically increasing with increasing stages, increases only marginally from 1 to 2 stages, hovering around $4000\mu m^2$ then starting to increase at a much faster rate, increasing at around $2000\mu m^2$ per stage after that. We can see an explanation for this from Figure 5, where we can see the "ameoba" plots highlighting the differences between the area consumed by registers (blue) vs the area consumed by partial multiplier units (red). As we'd expect, a larger proportion of the total area is consumed by the registers as we increase the number of stages. Additionally, we can see a larger proportion of the total area is lost to white space as we increase the number of stages, implying that the automatic tools are having significantly more difficulty managing it as the complexity increases only moderately (though, they're still doing a great job).

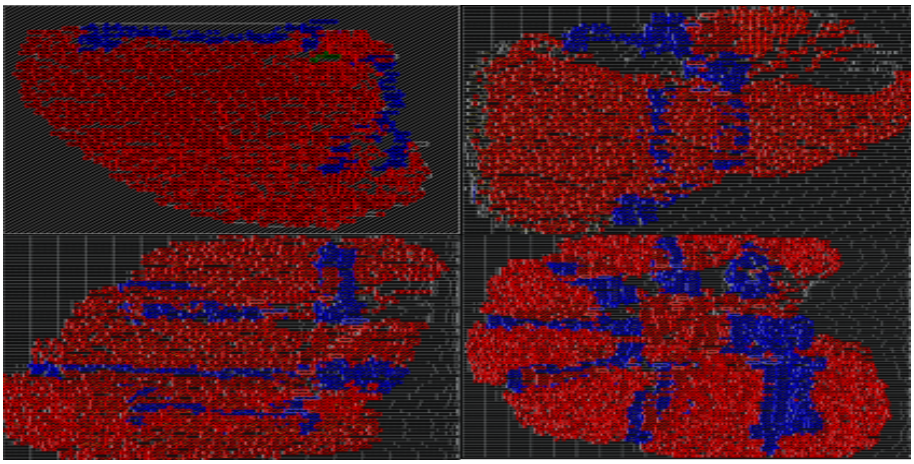

Figure 5: Comparison of amoeba plots. Clockwise, from top left: 1 stage, 2 stages, 4 stages, 8 stages. Shown is red are multipliers. Shown in blue are registers

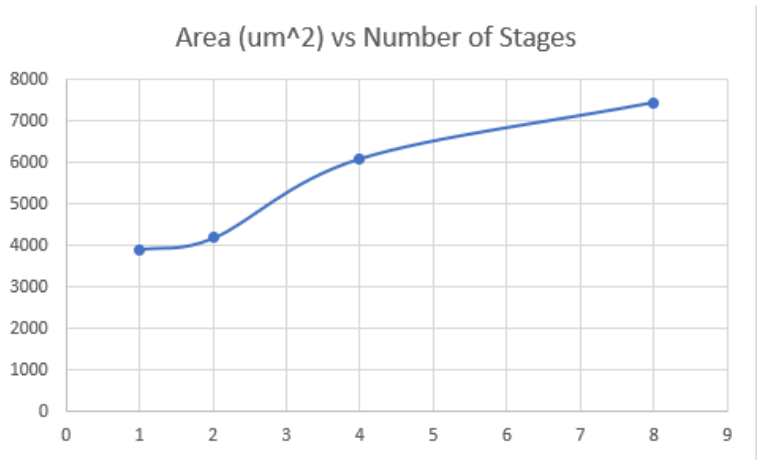

Figure 6: Design area consumption vs number of pipeline stages

That increase in area is not without benefit, as we see in Figure 8 that we get about the 1/N that we'd expect, reducing our clock time from 7ns with one stage to around 1.5 at 8 stages (I will mention anecdotally,

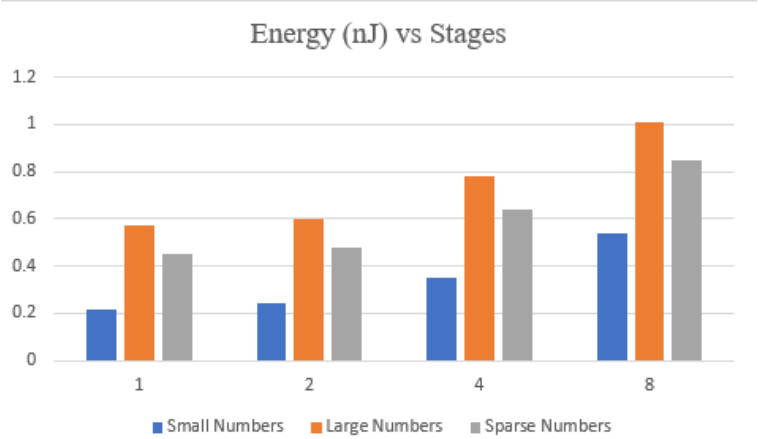

Figure 7: Energy consumption vs number of pipeline stages

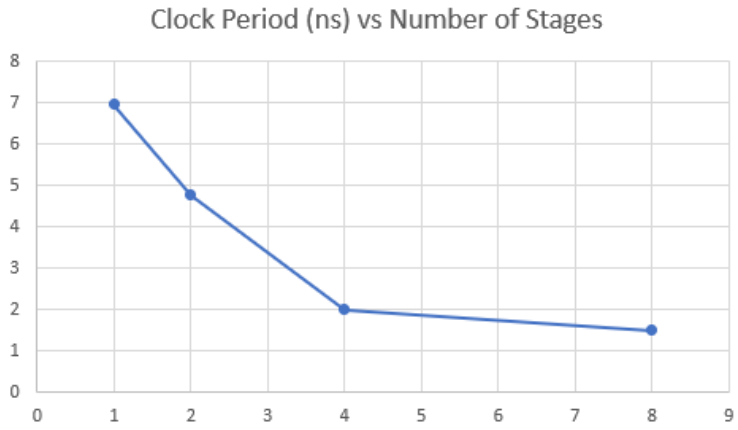

Figure 8: Clock period vs number of pipeline stages

however, that the time it took to lay this out was significantly increased, far faster than linearly). Analyzing the static timing analysis reveals that the critical path lay between the pipeline stages, though for different stages for each design, and after the only pipeline stage for the single stage design. Roughly, we can see this by comparing the critical paths in Figure 9 with the amoeba plots, as the critical paths end on registers except for the single stage, where is starts in a registers and leaves the design area.

Additionally, we can actually see from Figure 7 that energy does scale quite well with the number of stages, only increasing (roughly linearly) from $0.6nJ$ to 1 nj for the large numbers data set. Thus, while we doubled the number of pipelines, we used a bit less than double the amount of energy.

3.5 Timing Comparison

Assuming all things constant except the clock period and the total execution in cycles. It is clear the Single-Cycle multiplier is the fastest design. It computes a multiplication every 1.490ns (clock period) no matter which input data set is used. It is important to note, even though the clock period of the Variable and Fixed Latency multiplier is about 2 times lower, they still require between 5 to 30 times as many clock cycles to finish the sixty multiplications across the different input data sets. The designs in closest proximity, in terms of speed, to the Single-Cycle multiplier end up being the pipelined designs. It is evident that there is benefit in speed with pipelining the partial products, as we observe a tremendous decrease in clock period (6.960ns

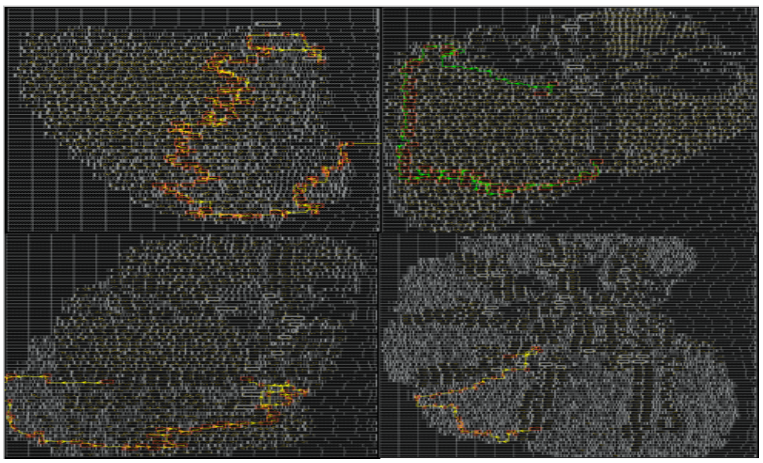

Figure 9: Comparison of critical paths. Clockwise, from top left: 1 stage, 2 stages, 4 stages, 8 stages.

to 1.996ns) for the first several increments in pipeline stages (1 to 4). However, we start to see diminishing returns as the we pipeline any further stages. This aligns with intuition because the utilized area for the implementation has now increased drastically and the routing is more complex. Additionally, the clock-to-Q delay of the registers start becoming more dominant as there is less combinational logic delay between the registers.

3.6 Performance/Energy Trade-Off

The performance vs. energy plots as seen in Fig.10 to Fig.12 show the different trade-offs between all the designs across the different input data sets. The easiest way to analyze the plot is to look at the lower left corner, as this extremity gives both the highest performing implementations and the most energy efficient ones. The three plots are very similar, they all show the Fixed-Iterative multiplier having the worst performance and energy utilization. Aside from that, as discussed in Section 3.3, we observe that the Variable-Latency multiplier has the most variance in performance and energy utilization depending on the input data set. Of course, this is expected due to the nature of the "look-ahead" feature where the control unit looks to see if there are consecutive zeroes (small data set) in one of the operands. In this case, this implementation saves energy and does not compute a partial products and in turn skips cycles and is thus faster. In the other extreme, when there are no consecutive zeroes (large data set) the implementation must compute every single partial products resulting in higher energy consumption and a decrease in speed. Lastly, we can also observe the improvement in the pipelined implementation as we add more stages. Initially the improvement is more pronounced (1stg -¿ 2stg -¿ 4Stg), however when we increase the number of pipeline stages from 4 to 8, there is barely any performance improvement but a noticeable increase in energy consumption. This makes sense because as we add more pipelined stages the physical implementation gets more complicated and contributes significant delays and the register clock to Q delay become more significant compared to the combinational logic between pipeline stages.

3.7 Performance/Area Trade-Off

The registers tended to be the most area-intensive circuits, as one would expect. Similarly, as one would expect, the pipelined multipliers were drastically more area intensive than the other designs because they had a lot more registers. After that the single-cycle multiplier was the most area intensive, likely because of the fact that is has to have all of the hardware to implement the 32-bit multiply in a single step. The variable latency and fixed latency multipliers use a lot less area, as they get to reuse their adders. The variable latency has a bit more area because it needs to calculate the bit shifts. There's a clear performance vs area tradeoff, which can be seen from Figure **??**, and this effect is also seen in the pipelined processes in Figure 6 and Figure 7.

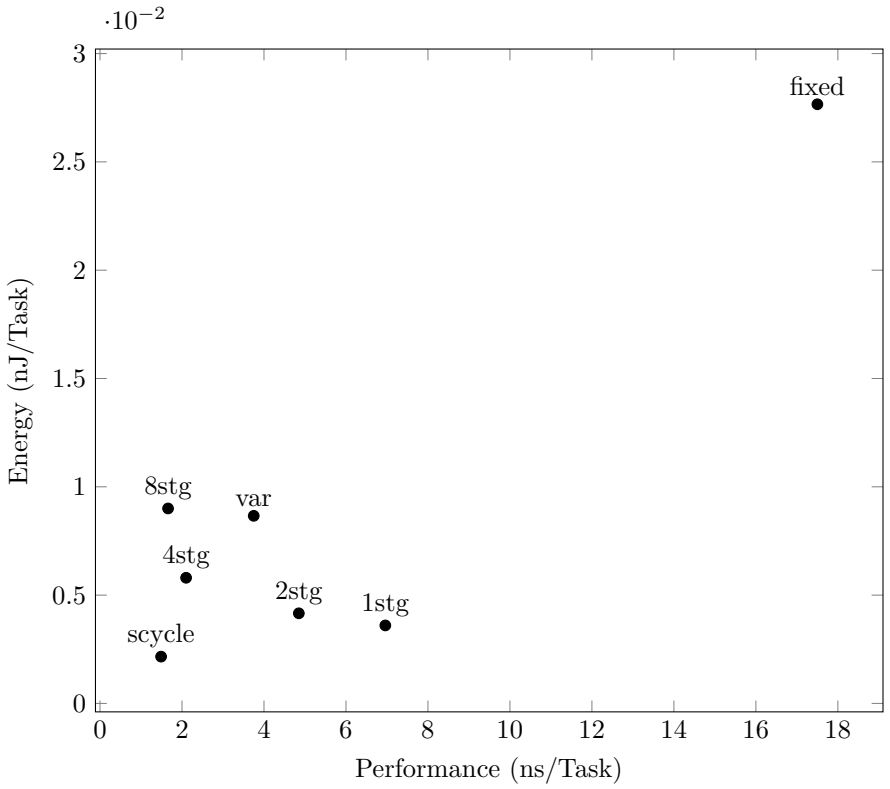

Figure 10: Performance of Multiplier Implementations with Small Input Dataset

In summary, we generally see all of the non-pipelined designs below around $2000\mu m^2$ while the pipelined designs all sit above $4000\mu m^2$, and adding area tends to get you performance, most efficiently when moving from one of the iterative processors to the single cycle processor.

4 Conclusion

This project allowed us, by using the automated ASIC design flow, to explore a rich design space very quickly. We were able to parameterize the pipelined design and get specific information on how the area, energy and clock frequency changes with the number of pipeline stages.

The qualitative results we expected from ECE 4750 were confirmed, where area increased, energy increased linearly, and clock period fell with 1/N. A surprising wrinkle was that the total area did not linearly follow the number of stages, but that makes sense as it becomes more difficult to manage white space. We were shocked by the speed and energy of the highly optimized single cycle multiplier, which was able to achieve a nearly four times improvement on the performance-energy product (a fun metric) over any other design on all datasets. We can see that the single cycle multiplier has a much higher area ($2600\mu m^2$) compared to the variable latency and the fixed latency designs at $1700\mu m^2$ and $1100\mu m^2$, respectively. All the baseline designs completely blew the N stage pipeline multiplier out of the water in terms of area, since there needs to be so many more registers. So, quantitatively, it's really hard to argue with the drastic speed up and decreased energy consumption of the single-cycle multiplier. The only situation we would recommend deviating from it is if area is the only concern, then the fixed multiplier is needed. You might want to use the N-stage multiplier in the situation where your design requirements aren't super well defined, and you don't expect them to be stenuous when resolved. Then, this is a great design because it's highly parameterized and ready to go and can give you a wide curve of performance vs energy. Figure 12, for instance shows an almost ideal 1/x shape for the energy vs performance curve over stages. The variable latency multiplier sits in the middle for everything, though (area, energy and performance) so maybe that's the best way to go

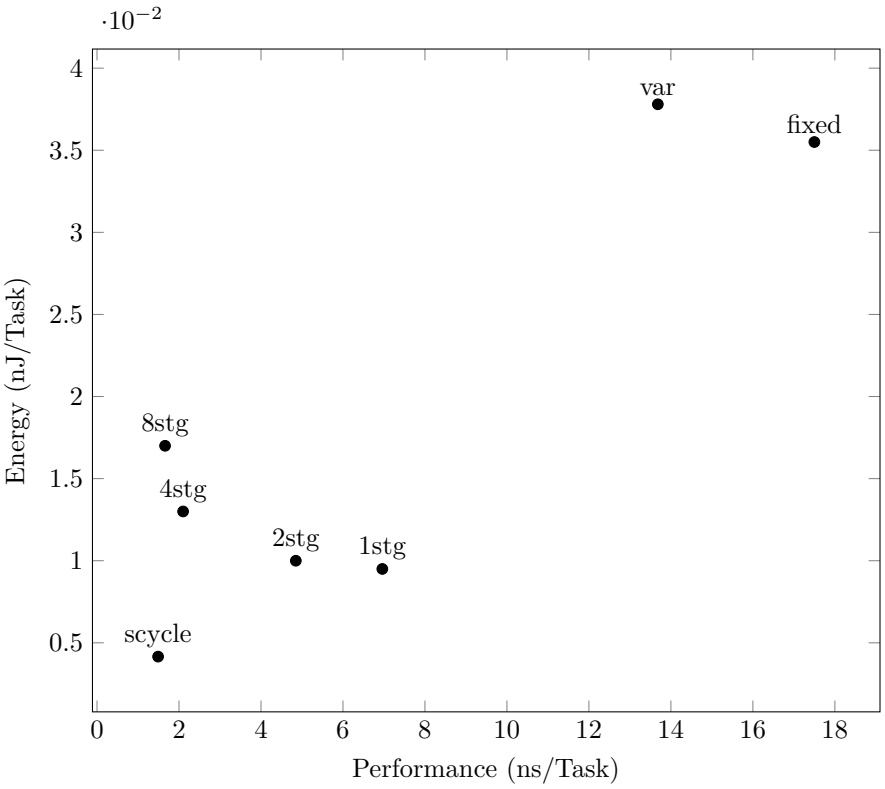

Figure 11: Performance of Multiplier Implementations with Large Input Dataset

if it's the only thing that closes on balanced set of design requirements. Regardless, it was really exciting to put numbers to the qualitative discussions we had in ECE 4750, generate some really cool amoeba plots and, in many ways, automate the exploration of an interesting design space.

5 Appendix

The appendix contains extra plots that provided extra insight but had redudndant information.

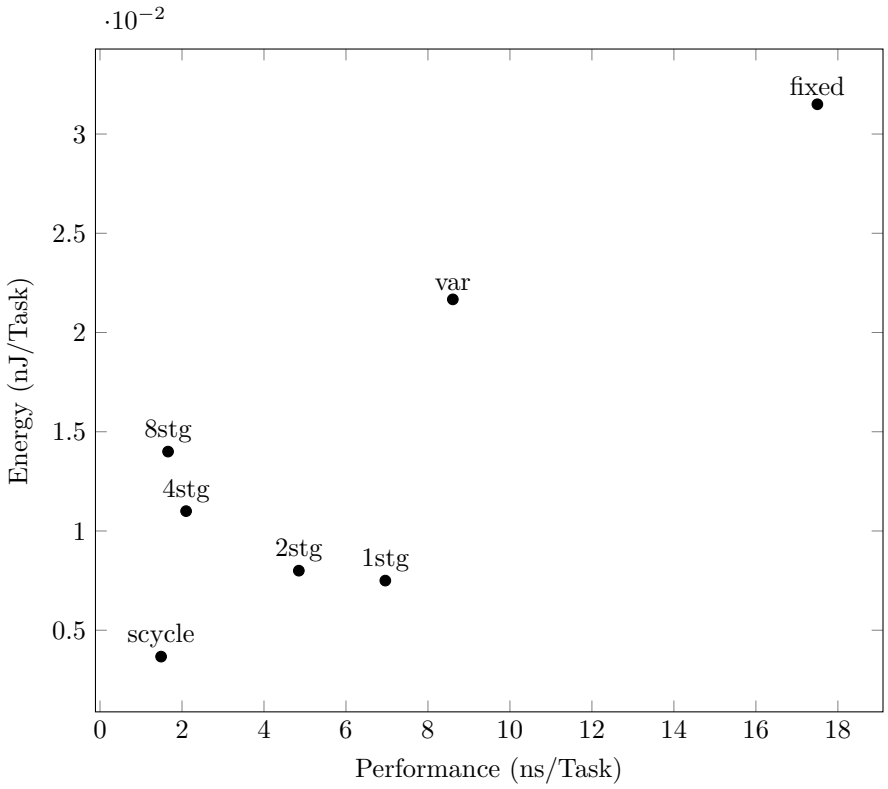

Figure 12: Performance of Multiplier Implementations with Sparse Input Dataset

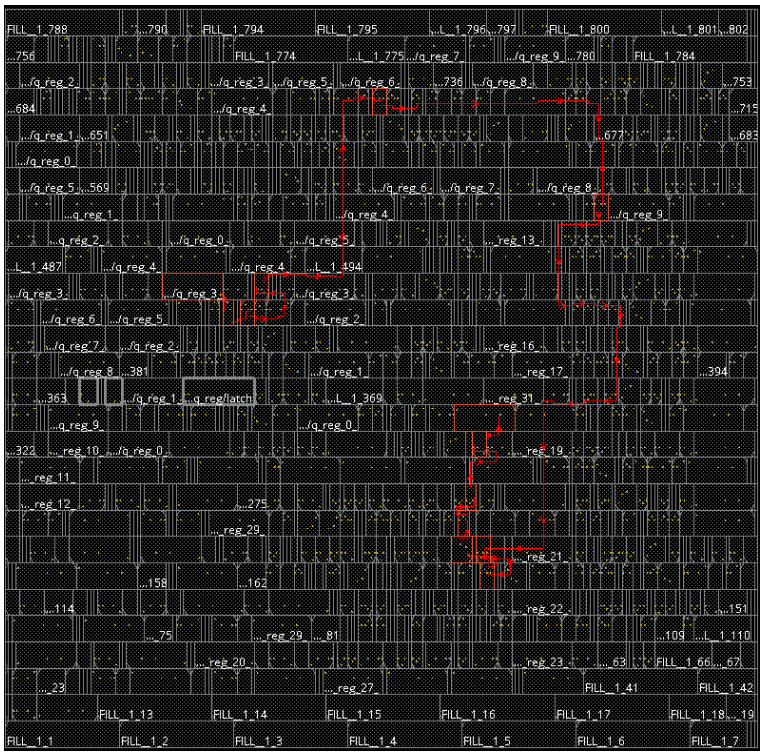

Figure 13: Fixed-Latency Iterative Amoeba Plot

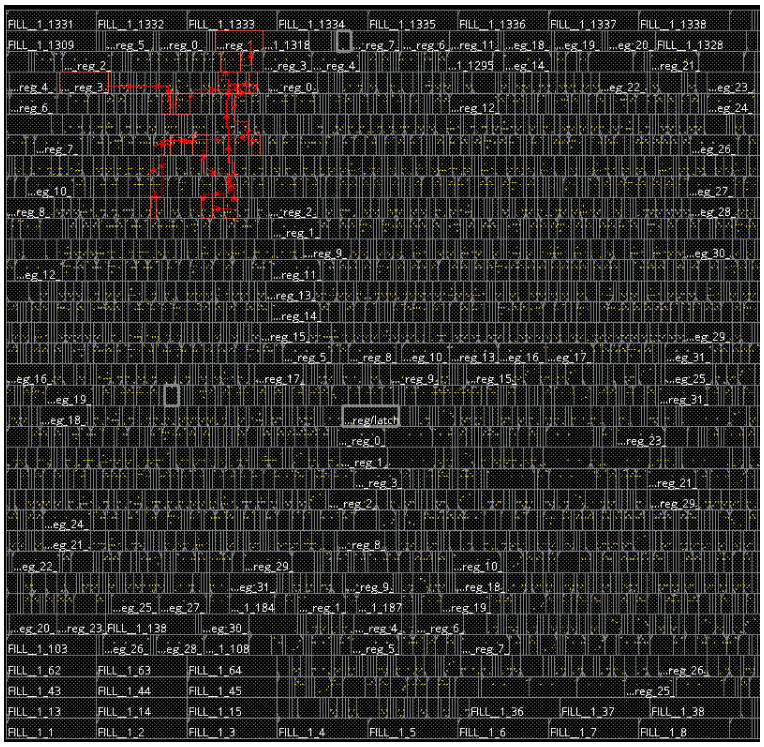

Figure 14: Variable-Latency Iterative Amoeba Plot

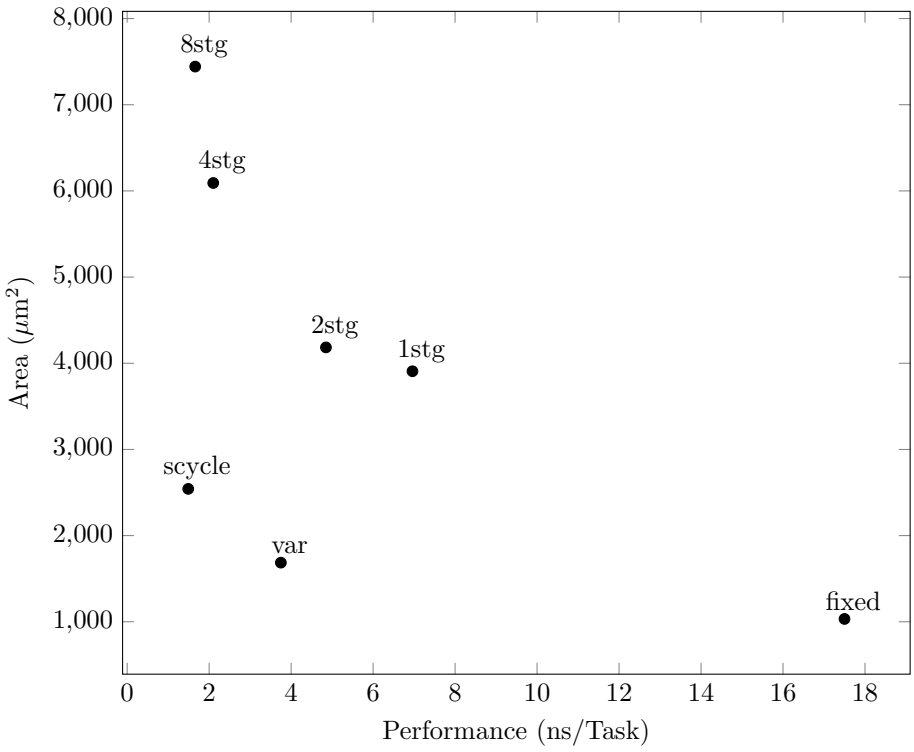

Figure 15: Performance of Multiplier Implementations with Small Input Dataset

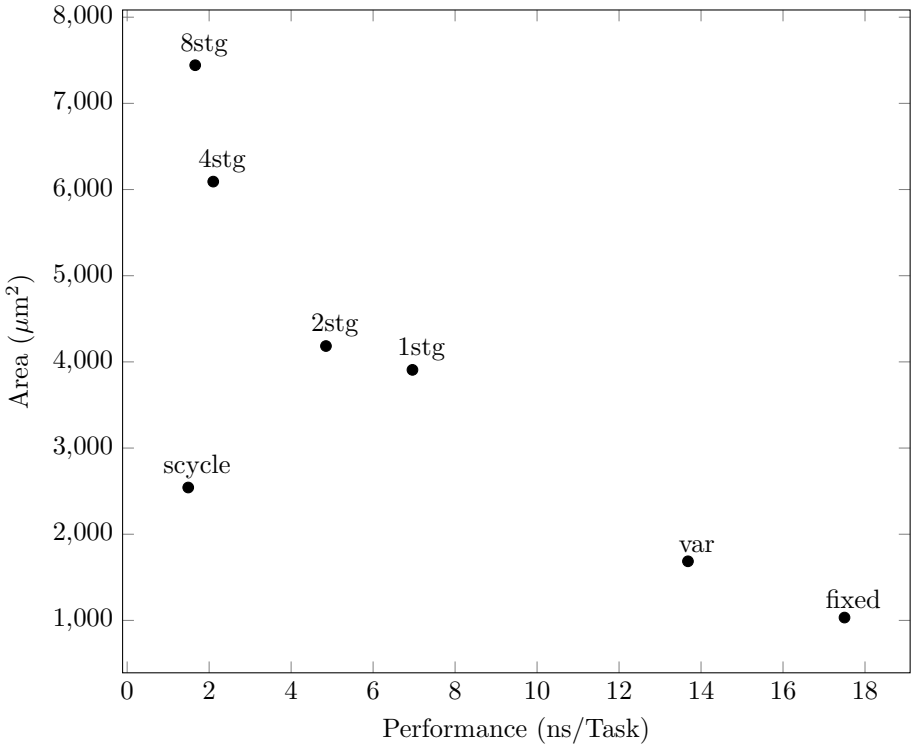

Figure 16: Performance of Multiplier Implementations with Large Input Dataset

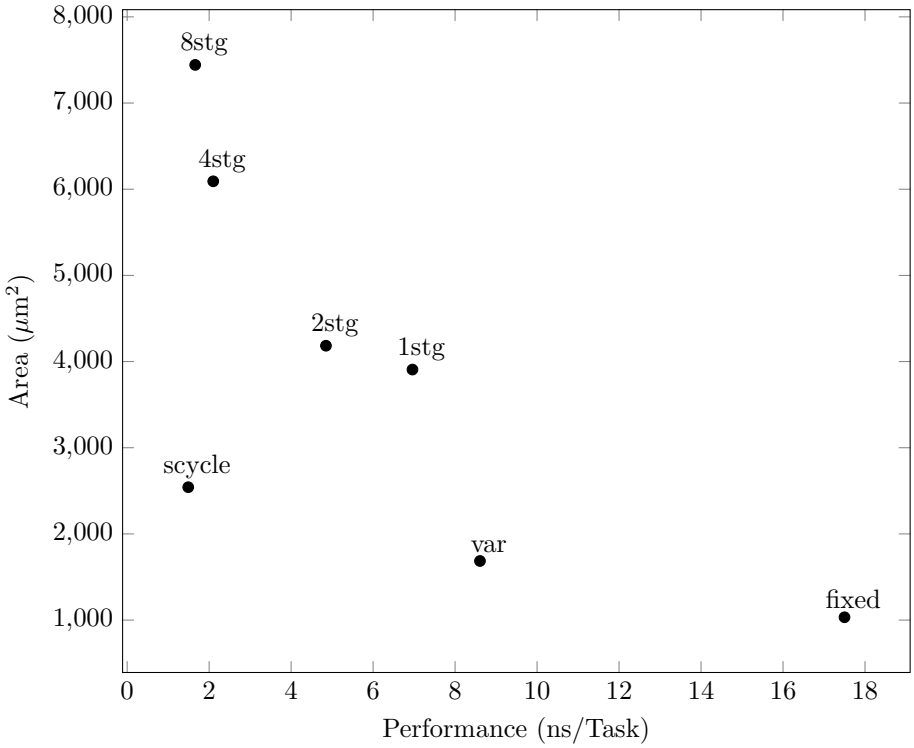

Figure 17: Performance and Area of Multiplier Implementations with Sparse Input Dataset