HW1 Report: Iterative Integer Multiplier

Team members: Yuou Qiu (2256129636),
Trinath Harikrishna (2455815909),
Yuke Zhang (4838266596)

Sep. 1, 2023

1. **Introduction**.

In this assignment, we have designed a Baseline and an Alternative design for Integer Multiplier Unit, aiming to comprehensively assess its functionality under diverse scenarios. The primary objective is to rigorously validate the multiplier's correctness and performance characteristics across a spectrum of multiplication operations, including different operand magnitudes and binary patterns. By meticulously crafting this testing harness, we gain valuable insights into the unit's behavior and digital design in general, which can be applied to future projects involving digital circuit design and optimization. The baseline design represents a well-defined multiplier unit which takes a fixed number of cycles to execute, while the alternative design explores optimizations and enhancements to improve its runtime by efficiently cutting through a few corner cases for minimal cost. Through this assignment, we not only ensure the reliability of the multiplier unit but also cultivate essential skills in developing versatile testing environments and refining digital circuitry for future assignments and tapeout.

2. **Project Management**

Our group consists of three members. In order to complete the task efficiently, each group member takes on a different role. In HW1, and hopefully, in future projects as well, Yuou Qiu works as an architect, Trinath Harikrishna takes on the role of a design lead, and Yuke is the Verification Lead. Figure.1 is the recommended task division and timeline.

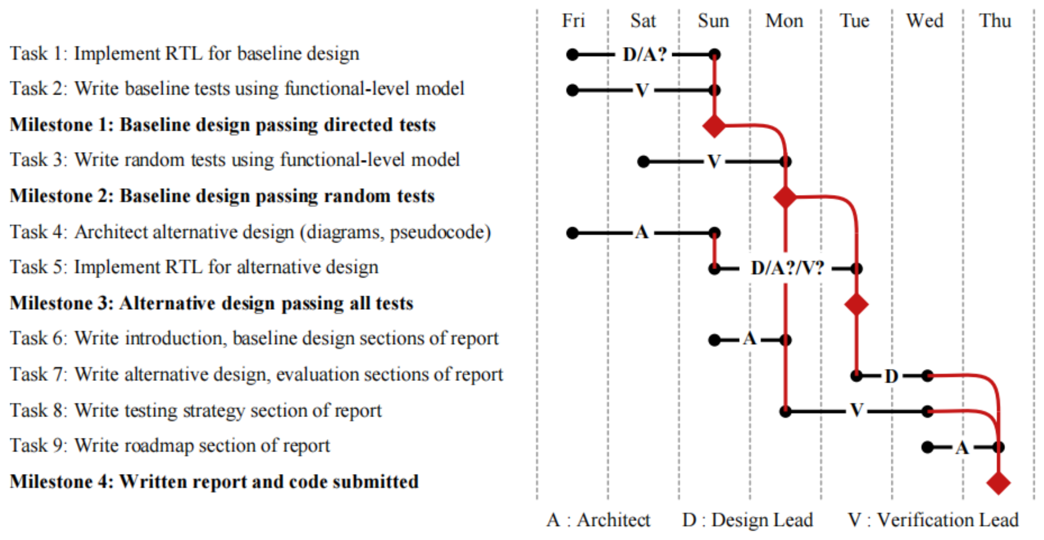
Fri
Sat
Sun
Mon
Tue
Wed
Thu
Task 1: Implement RTL for baseline design
Task 2: Write baseline tests using functional-level model
Milestone 1: Baseline design passing directed tests
Task 3: Write random tests using functional-level model
Milestone 2: Baseline design passing random tests
Task 4: Architect alternative design (diagrams, pseudocode)
Task 5: Implement RTL for alternative design
Milestone 3: Alternative design passing all tests
Task 6: Write introduction, baseline design sections of report
Task 7: Write alternative design, evaluation sections of report
Task 8: Write testing strategy section of report
Task 9: Write roadmap section of report
Milestone 4: Written report and code submitted
D/A?
V
V
A
D/A?/V?
A
D
V
A
A : Architect
D : Design Lead
V : Verification Lead

Figure.1 Recommended Initial Project Roadmap for HW1

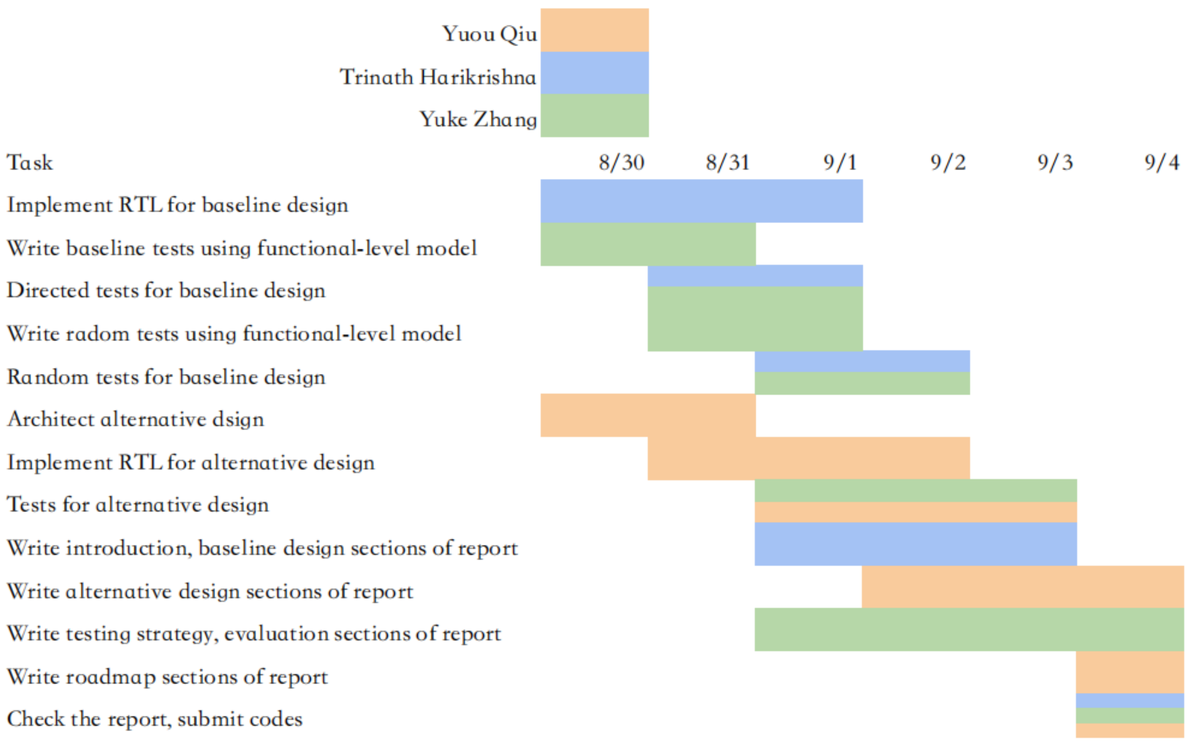

Figure.2 Actual Roadmap for HW1

However, due to the real situation, and our desire to work in a more parallel way, we have modified the recommended Gantt map. The real project roadmap for HW1 is shown in Figure.2.

3. Baseline Design

The objective of this baseline design is to create a fixed-latency iterative integer multiplier using Verilog. The baseline design provides a foundational framework for this multiplier and serves as a starting point for more advanced variations and enhancements.

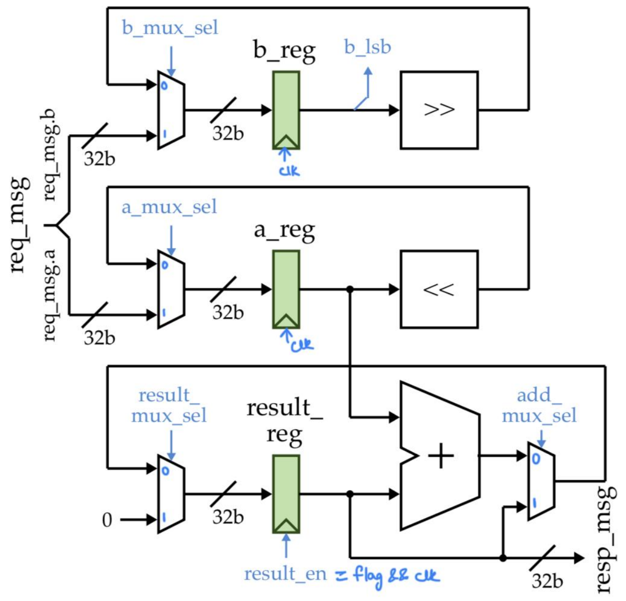

Figure.3. Datapath for Fixed-Latency Iterative Interger Multiplier

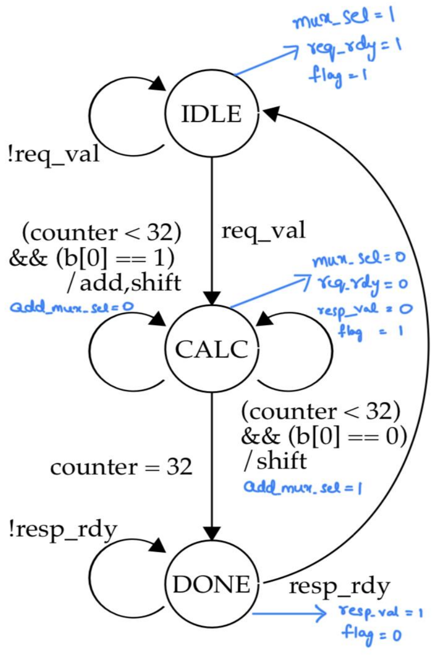

Figure 2 : Control FSM for Fixed-Latency Iterative Integer Multiplier

Figure.4. Control FSM for Fixed-Latency Iterative Interger Multiplier

- **Datapath Overview:**

The datapath is responsible for performing the actual multiplication operations. Figure 1 presents a visual representation of the datapath. It comprises various components such as registers, multiplexers, adders and shifters, all working on 32-bit data. One of the design's notable aspects is the structural composition of the datapath, which gives us more control over the optimization of the design. This structural design style ensures clarity and modularity in the code, making it easier to understand and maintain.

- **Control Unit Overview:**

The control unit plays a pivotal role in orchestrating the data flow within the datapath. It manages the progression of the multiplication operation through a simple finite-state machine (FSM) which generates the control signals as shown in Figure 2. This FSM consists of three states:

> **1. IDLE:** In this state, the control unit consumes input operands and places them into input registers. It prepares the multiplier for the multiplication process. Hence the ***a_mux_sel***, ***b_mux_sel***, ***result_mux_sel*** and ***req_rdy*** control signals must be set.(Reset of the control signals are dealt with accordingly)
>
> **2. CALC:** The CALC state is where the actual iterative multiplication takes place. It involves a series of additions and shifts to calculate the result iteratively. This state operates in a loop for 32 cycles to perform the multiplication. Hence the ***a_mux_sel***, ***b_mux_sel***, ***result_mux_sel*** control signals must be reset and depending on the ***b_lsb*** value ***add_mux_sel*** control signal must be set/reset depending on the shift/add&shift operation to be performed. The ***req_rdy*** and ***resp_val*** are both reset during CALC state.
>
> **3. DONE:** After completing the iterative calculation, the control unit transitions to the DONE state, where it sends the result out through the output interface. During this state the ***resp_val*** will be set indicating the readiness of the output. Also, since we want the output to be retained in the register we have to gate the clock to that particular register using another control signal i.e. ***result_en*** which is

nothing but an AND of a flag signal reset only in the DONE state and clock.

It's worth noting that the FSM design results in each multiplication operation taking 34 cycles on average: one cycle for IDLE, 32 cycles for iterative calculation, and one more cycle for the DONE state.

4. Alternative Design

4.1 Overall introduction of alternative design

In this section, we implemented a variable latency iterative multiplier, which takes advantage of the structure of input operands. We noticed that once the LSB of operand b is "0", the baseline design still takes one cycle for this meaningless add. As long as we could skip some of those cycles that are actually adding 0 to the final result, we could reduce the over runtime.
In order to save the cycles, this alternative design leverages the following two cases of operand b, bolded 0s are the parts that could possibly be skipped.

- consecutive/single 0s, (the 0 on LSB not included)
  e.g. b=32'b1**00**1_**0000_000**1_1111_1**00**1_1**000_00**11_1**0**10
- consecutive 0s in most significant bits,
  e.g. b=32'b**0000_0000_0**111_1111_1001_1111_1111_1111

Notice that in the following paragraphs, we will use $\{b_{31}...b_2b_1b_0\}$ to represent each bit of operand b.

4.2 Architecture of alternative design

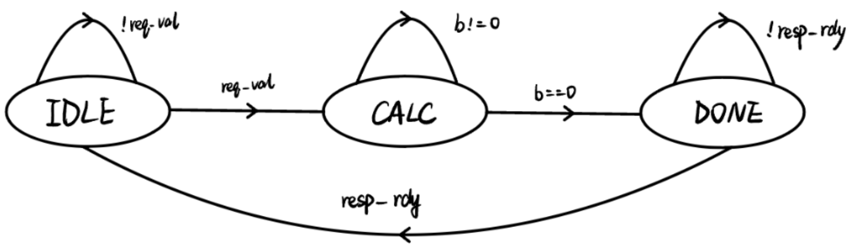

Figure.5. Control FSM for variable latency iterative multiplier

Figure.5 shows the FSM of our variable latency iterative multiplier. Similar to the baseline design, in the IDLE state, the input operands are consumed and stored in registers and the result register is reset. Once a pair of valid inputs appears (req_val=1), we start iteratively adding and shifting in the CALC stage. Either after computing all 32 bits, or after computing all valid additions, the remaining bits of b would be zero (beq0=1), so we can safely jump to the DONE state. Once the result is consumed (resp_rdy=1), it will come back to the IDLE state to wait for the next pair of valid inputs.

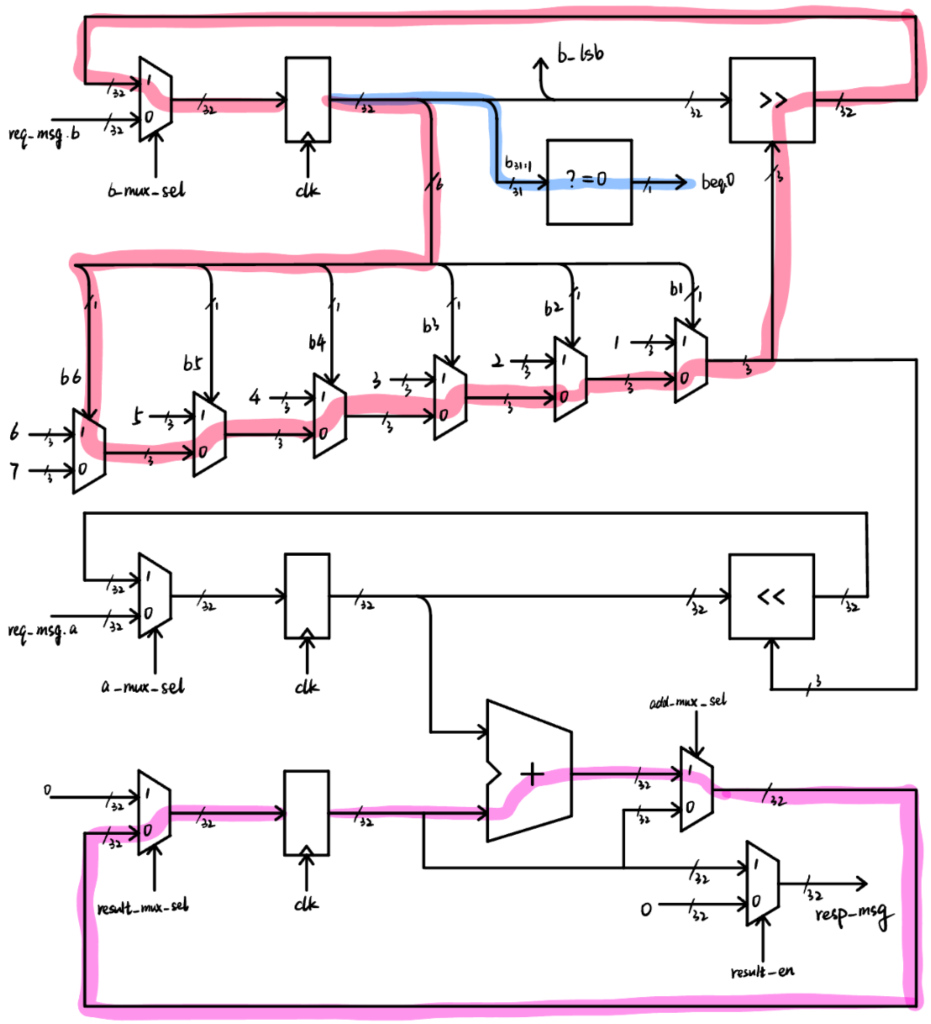

Figure.6. Datapath for variable latency iterative multiplier

- Leveraging consecutive/single 0s

As is shown in Figure.6, we use a chain of 6 muxes to detect how many consecutive 0s are in the next 6 bits of operand b ($\{b_6 \ldots b_3 b_2 b_1\}$). This allows us to shift multiple bits per cycle. So that in the next cycle, we can jump to the next required addition or b_7 instead of b_1. Moreover, we can increase the length of the mux chain to further decrease the number of required cycles. Using a N muxes chain, we can skip up to N consecutive 0s and compute up to N+1 bits in a single cycle.

Worth mentioning that, in order not to dramatically increase the critical path, instead of normal zero detectors, we use muxes chains to work as a zero detector, and we only detected the next 6 bits instead of all the remaining 31 bits.

- Leveraging consecutive 0s in most significant bits,

Either after computing all 32 bits, or after computing all valid additions, the remaining bits of b would be zero (beq0=1). While calculating the LSB of b, we are also checking whether the remaining bits are equal to 0. Once $\{b_{31}...b_3b_2b_1\}$ = 0. , we could realize that the current partial sum is already equal to the final answer and it's time to end this whole multiplication.

4.3 Theoretical Analysis

The critical path of the design might be a shifter plus 7 multiplexers (marked in red), or an adder plus 2 multiplexers (marked in pink), or a 31-bit zero detector (marked in blue).

The worst case would take 34 cycles to complete the whole multiplication, 1 for IDLE stage, 32 for CALC stage and 1 for DONE stage. This occurs when b=32'hFFFF or b=32'hFFFE.

The best case only need 3 cycles to complete the whole multiplication, 1 for IDLE stage, 1 for CALC stage and 1 for DONE stage. This occurs when b=32'h0000 or b=32'h0001.

5. Testing Strategy

The goal of our testing is to ensure that the multiplier design is functionally correct, efficient, and robust across various edge cases. Our test strategy is summarized in Table 1. Below is an in-depth overview of each test category for our multiplier.

Testing basic functionality. We begin by focusing on the fundamental capability of the multiplier to handle both small and large numbers across all possible sign combinations. This phase ensures that the basic multiplication operation is carried out flawlessly. For each combination, such as small positive numbers multiplied by small negative numbers, we provide five test stimuli. This granularity in testing aims to catch any inaccuracies or inconsistencies in the most straightforward operations, thereby establishing a baseline for the multiplier's functionality.

Special numbers test. The next phase involves testing with 'special' numbers that have specific bit patterns. These include tests with least significant bits (LSBs) masked off, middle bits masked off, sparse numbers with few '1' bits, and dense numbers with many '1' bits. This phase aims to uncover any hidden bugs related to specific bit-level operations and representations. We execute five test stimuli for each of these special number types to ensure that the design can handle these peculiar bit-level scenarios.

Random testing. In this stage, the objective is to validate the multiplier's general capabilities by subjecting it to a large set of random numbers. This includes three subsets: 50 random tests for small numbers, 50 for large numbers, and 50 for completely random numbers. The idea here is to confirm that the multiplier can consistently handle a wide variety of numbers, further emphasizing its robustness and reliability.

Random source and sink delay. Time-related characteristics are essential for any hardware design. In this phase, we introduce random source and sink delays to understand how well the multiplier performs under variable timing conditions. Three random delays are generated and applied in three sets: one for multiplications on small numbers, one for large numbers, and one for random numbers. Each set has 50 multiplication operations, ensuring that we have a good sample size for each timing condition.

Corner case testing for alternative design. Finally, we delve into more specialized edge-case testing, focusing specifically on the alternative design's potential strong and weak spots. This stage involves 150 test stimuli: 50 for scenarios that produce the smallest computation cycles, 50 for scenarios that can save cycles by skipping certain bit computations, and 50 for scenarios that demand the largest computation cycles. This extensive corner-case testing aims to validate the performance and efficiency trade-offs of the alternative design under extreme conditions.

By adhering to this comprehensive testing strategy, we aim to validate that the multiplier design is not only functionally correct but also efficient and robust under a variety of scenarios.

Table 1: Test strategy

	Test index	Test cases	# Test stimulus
Basic functionality	1	(small) Positive x Positive	5
	2	(small) Positive x Negative	5
	3	(small) Negative x Positive	5
	4	(small) Negative x Negative	5
	5	(large) Positive x Positive	5
	6	(large) Positive x Negative	5
	7	(large) Negative x Positive	5
	8	(large) Negative x Negative	5
Special numbers	9	Mask off LSBs	5
	10	Mask off middle bits	5
	11	Sparse numbers	5
	12	Dense numbers	5
Random tests	13	Small numbers	50
	14	Large numbers	50
	15	Random numbers	50
Random source and sink delays	16-18	Random delays for rand. small numbers	3 rand. delays for 50 multiplications
	19-21	Random delays for rand. large numbers	3 rand. delays for 50 multiplications
	22-24	Random delays for rand. random numbers	3 rand. delays for 50 multiplications
Corners	25	Corner cases for the alt. design	150

6. Evaluation

Functionality. Both the baseline and the alternative designs successfully navigated through the exhaustive test scenarios outlined in our Testing Strategy section. This accomplishment serves as a robust indicator of the functional correctness inherent to both design approaches. Consequently, we have high confidence in the reliability and operational efficacy of these multipliers.

Cycle. In Table 2, we present a comparative analysis between the baseline and alternative designs, focusing on the number of cycles required for computation. The cycle counts are assessed across five scenarios: random small numbers, random large numbers, fully random numbers, and corner cases that result in both the maximum (Corner-L) and minimum (Corner-S) cycle counts for the alternative design. Notably, the alternative design requires the maximum number of cycles when the value of 'b' is set to 32’hFFFE and needs the minimum number of cycles when 'b' is either 32’b0 or 32’b1. According to the data in Table 2, our alternative design generally outperforms the baseline, requiring fewer cycles in most test scenarios. The only exception occurs in the worst-case condition (Corner-L), where the alternative design matches the baseline in cycle count. This enhanced efficiency can be attributed to the alternative design's ability to detect and concurrently shift multiple bits in a single operation when the operand 'b' contains a series of consecutive zeros.

Table 2: The number of cycles for the baseline and alternative design

		Small	Large	Random	Corner-L	Corner-S
Baseline	# cycles	1701	1701	1701	1701	1701
	Avg cycles	34.02	34.02	34.02	34.02	34.02
Alternative	# cycles	973	1457	958	1701	151
	Avg cycles	19.46	29.14	19.16	34.02	3.02

Area and Energy. In the absence of concrete synthesis and simulation data, a direct comparison between the area and energy consumption of the baseline and alternative designs remains speculative. Nevertheless, some preliminary analysis can be offered. In terms of physical area, the baseline design features a more compact datapath module but compensates with a larger controller. On the other hand, the alternative design exhibits a larger datapath, attributed to the additional logic required for operand bit-skipping, but benefits from a smaller controller that eliminates the need for a counter.

When it comes to energy consumption, the baseline design is likely to demonstrate greater stability across a variety of input conditions. In contrast, the energy efficiency of the alternative design appears to be more input-dependent. For example, when subjected to inputs that maximize latency—such as 32'hfffe—the alternative design is likely to be similar with the baseline. Conversely, for inputs that minimize latency—like 32'd0 or 32'd1—the alternative design is likely to be more energy-efficient than the baseline. Overall, we expect the average energy of the alternative design is lower than the baseline design.

7. Conclusion

Both our baseline and alternative designs have successfully passed our rigorous functional testing. By incorporating bit-skipping functionality into the alternative design, we were able to achieve significant reductions in computational cycles. Based on our test results, the alternative design demonstrates an impressive efficiency advantage, saving up to 91% of computational cycles when compared to the baseline design.