ECE 4750 - Lab 1 Report

Sabrina Herman (sh997), Yoon Jae Oh (yo82), Benjamin Xing (byx2)

Section 1: Introduction

The purpose of this lab was to provide an introductory exercise in RTL design using a hardware description language and to develop other skills including version control using Git, Linux/Unix development, and working as a team of design and verification engineers. The lab incorporates many important themes in computer architecture, and therefore relates to lecture materials as well as industry practices. We learned about functional level and register transfer level modeling. We designed a control unit and datapath with a split design pattern, requiring us to learn how to build a datapath and an FSM control unit as well as how the control unit and datapath communicate. We learned about encapsulation design principle and message interface design pattern through the ready/valid signal protocol. We practiced hierarchy design principles by defining and instantiating modules. We used an incremental development design methodology by beginning with a baseline design before moving on to an alternate design. All of these computer architecture themes are relevant not only to this class, but to how computer architecture, design, and verification are implemented in industry.

The task involves designing, implementing, and verifying a 32-bit iterative integer multiplier based on a finite state machine control system. The first design is a fixed-latency multiplier which always takes 35 cycles to run a single multiplication operation, while the second design is a variable-latency multiplier that improves the shortcomings of the first design, namely reducing the number of individual shift operations and removing excess cycles after the multiplication computation has completed. We implemented a functional baseline design after about three days of work, while it took another two days to successfully build the alternative design. The alternative design significantly reduced the average number of cycles required for each calculation, especially for operands that include many zeros in their binary forms. However, for small negative-negative calculations, not much improvement occurs. This is due to the fact that most of the binary bits in the operands are ones, which eliminates the effectiveness of the optimization. The alternative design has little impact on clock frequency, as the critical path does not increase to the point where a slower clock speed is required. Although the circuit will require a little more area than the baseline design in order to implement a 32-bit priority encoder, this is by no means a prohibitively large increase. Similarly, energy consumption should increase with the alternative design due to the more complicated logic that is required. However, this relatively small increase in energy use can be sacrificed for the significant increase in performance that the alternative design brings.

The results of the evaluation demonstrate the differences between a single-cycle processor and an FSM processor. A single cycle processor, much like our baseline design, has a fixed latency. Thus, we can view our baseline design as having a single overall cycle representing one multiplication operation that takes a large cycle time. This design misses the opportunity to utilize the processor's resources. Our alternative design embodies the improvements of an FSM processor over a single cycle processor in that a new multiplication operation can be started immediately after the previous operation has arrived at the final answer, which decreases the total clock cycles per execution.

Section 2: Baseline Design

The baseline design implements a fixed-latency iterative multiplier, meaning the same amount of clock cycles (35) are dedicated to each execution. Our baseline design was implemented in Verilog. The main components of this baseline design are the datapath and control unit modules. The baseline design also utilizes an instance of a counter module (provided) to keep track of the clock cycles. The datapath has paths for moving data through various arithmetic blocks, muxes, and registers. The control unit is responsible for managing the movement of data through the datapath by sending out enable and mux select signals. This decomposition is an example of the modular design principle, which is an approach that subdivides a system into smaller modules that can be independently created. This method is utilized because it allows the different parts to be independently created and used in different systems, which is effective as well as efficient as multiple copies of the same sub-system do not need to be created in different systems. The datapath/control split design is illustrated in Figure 1.

This datapath works by shifting the inputs each iteration and adding one operand to the result if the least significant bit of the second operand is one. The datapath, as illustrated in Figure 3, consists of 4 muxes, 3 registers, 2 shifters, and 1 adder. The datapath uses the original operands (req_msg_a and req_msg_b) in the first cycle, but then uses the shifted version of these operands for all subsequent iterations to perform the multiplication. The logic for determining which value to use comes from the control unit and is used by the muxes. Additionally, the first cycle will start with the result value being 0, and will increase that value based on signals from the control unit, which again sends a select value to a mux. The 3 registers are a_reg, b_reg, and result_reg. These registers hold the values of the mux outputs. The shifters are responsible for left or right shifting a zero into the outputs of the registers as part of the multiplication calculation. Instead of creating separate shifter modules, we chose to directly connect the shifted values to the mux inputs (a_mux and b_mux). For example, if the output value of b_reg is 1101…, one of the inputs to the b_mux will simply be (1101…. >> 1). This was done in order to optimize calculating shifts. A shifter module is not necessary in order to implement this design. The add_mux chooses and outputs either the output of the adder or the output of the result_reg. The control logic will send signals to the datapath on which one to output. The adder outputs the a_reg's output added together with the result_reg's output.

The control unit, as illustrated in Figure 4, has three states: IDLE, CALC, and DONE. The state should only leave the IDLE state and enter the CALC state whenever the given request is valid (req_val = 1). The state should only move from the CALC state to the DONE state when the counter's count has reached 32, meaning 32 clock cycles have passed and the calculation is done. Otherwise, the state stays in CALC. In the CALC state, the control unit sends out mux select signals to the datapath. In every cycle of the CALC state, the a_mux_sel, b_mux_sel, and result_mux signals are always set to 1 because the datapath should never select the initial operand values or 0 (for result_mux) after the IDLE state. The add_mux_sel signal's value depends on b_lsb (1 if b_lsb = 1, 0 if b_lsb = 0). The state should go from the DONE state back to the IDLE state when the response is ready (resp_rdy = 1) because the final response message has been read. In the DONE state, the control unit sends out the resp_val signal, which is connected to the result_en signal in the datapath. This signal indicates that the final response is available. The control path also clears the counter every time it reaches the IDLE state in order to begin counting the clock cycles for the next execution. Each execution should take a total of 35 clock cycles to finish, with the IDLE state taking 1 cycle, CALC state taking 31 cycles, and DONE taking 3 cycles.

This baseline design is analogous to a single cycle processor due to it having a fixed number of cycles being based on the worst case scenario. This analogy makes it a great baseline design. Being able to compare our design to a single cycle processor, allows us to easily evaluate this design and draw conclusions about the cycles per instruction and the cycle time.

Section 3: Alternative Design

The alternative design, illustrated in Figure 7, implements a variable-latency iterative multiplier, which takes advantage of the operand values in order to optimize the number of cycles required to complete the multiplication. The alternative design is essentially an optimized version of the baseline multiplier, and this optimization occurs in two steps: the first step is to shift the values stored in the two operand registers more than one bit at a time in a single cycle if no addition operation would have occurred during those shifts in the baseline implementation; the second step is to terminate the multiplication sequence as soon as the final answer is reached instead of waiting 32 cycles every time. These optimizations make it analogous to an FSM processor with a variable cycle time. The changed design from the baseline to the alternative design can be seen in Figure 2.

In the baseline implementation, the datapath would shift the values stored in the operand registers, b_reg and a_reg, by one bit every cycle. The control unit would then check the least-significant bit in b_reg (called b_lsb) and would allow the datapath to add the value in a_reg with the value in the result register, result_reg, if and only if b_lsb is equal to one. This is inefficient, however, because each shift operation that does not result in adding a_reg and result_reg uses up cycles without bringing the value in result_reg closer to the final answer.

This problem can be solved by shifting the values of the inputs to b_reg and a_reg by the number of consecutive zeros at once, resulting in an addition occuring every cycle (unless operands are zero). The shift occurs before storing them in their respective registers. For example, if the rightmost bits of the value in b_reg are ...1010000, then the datapath will right-shift this number by 4 such that the new least-significant bit is now equal to 1. This way, every cycle the result_reg will make progress toward the final answer.

Our initial thought process was to iterate through the 32-bit value in order of least-significant bit to most-significant, stopping once the first 1 is reached; however, while-loops are not synthesizable in real hardware. Instead, we chose to implement a priority encoder, with priority increasing from least-significant bit to most significant. The encoder takes as input the 32 bits passed through the b_mux, and outputs the number of bits by which to shift the values in b_reg and a_reg. In the Verilog code, this encoder is implemented as an "if- elseif- elseif..." construct, checking the value of the next least-significant bit in each subsequent "elseif" block. The elseif becomes true once the least significant 1 has been found, and then that bit number determines the shift size required. We chose to use a priority encoder since it can handle inputs that have more than one bit of value 1, and its behavior uniquely defines the required shift while providing simple input and output interfaces. Moreover, since all the inputs simultaneously enter the circuit, the priority encoder will not increase the critical path length by a prohibitively large amount. The value stored into b_reg and a_reg is shifted by the value of the output of the encoder, and the rest of the datapath remains the same. That way, a single bit shift still occurs at the end of each cycle in addition to the shift given by the encoder. We chose to implement the design this way in order to preserve the principle of modularity, in that adding the priority encoder did not affect the design of the rest of the datapath - it was merely another "module" inserted along the existing datapath.

Another inefficiency in the baseline design was the fixed latency, since most calculations do not require the full 35 cycles to complete. The second goal of the alternative design was to terminate the execution of the multiplication task as soon as the result register contains the correct answer. Our implementation took advantage of the fact that the value in b_reg would be equal to zero when the final answer is computed. We added a simple condition flag, called "doneflag", that has value 1 (true) when the value in b_reg is equal to zero and 0 (false) otherwise. The value of doneflag is an output from the datapath and input to the control unit. When the calculation completes and the value of doneflag is set to 1, the control unit will set the next state to DONE, thereby resetting the counter and sending out the proper resp_val signal to indicate that the multiplication has completed. We chose to check the value of b_reg using a condition flag since it was the simplest way to ensure that the computation has finished. Since the flag value has the same purpose and format as the logic tracking the value of the counter in the baseline design, it was trivial to replace the CALC -> DONE transition condition with the value of the condition flag. This is an example of extensibility and encapsulation, as the alternative design required no more than a change to the state-transition condition rather than an entire redesign of the state logic.

Section 4: Testing Strategy

The primary method of testing that we utilized was directed testing using line traces because it allowed us to easily track different values at each clock cycle, which helped us debug which part of the system was incorrect. Our initial testing stage consisted of getting our design to pass the given test in the IntMulBaseRTL_test.py file, which was multiplying two small positive numbers together. This was a good base test because it does not test for overflow or negative numbers. Our first few designs failed the given test case, so we used line tracing to follow several important values such as the resp_msg and b_lsb to determine if there was a problem with either the datapath or control unit. After tracking values using line tracing, we discovered there was a problem with some faulty connections in our datapath. One specific example was the connection to the resp_val output: while we had implemented the correct logic for setting resp_val, we did not realize that we had named the connection in the control module "result_en." Using line tracing, we determined that the value of resp_val remained at zero throughout the test cases and we were able to easily connect result_en to resp_val. After fixing the connections, our design passed the given test case. We chose to use line tracing for our verification because it clearly shows the inputs and outputs to the system. Not only did it allow us to verify manually that the multiplication operation produced the right result, it also gave us a lot of flexibility in choosing which variables we wanted to track. Moreover, line tracing worked perfectly with the functional-level test harness provided to us because we could compare the behavior of our design to the behavior of a model that we know must be correct; we could be confident that as long as any test case passes in our implementation it would pass in the functional level model as well.

We also performed unit testing because we wanted to verify the functionality of larger components like the priority encoder we implemented. These tests involved manually creating 32-bit input test cases and checking the output from the encoder. These tests were helpful for two primary reasons: they showed that the priority encoder would select the shift value based on the position of the least-significant 1, and they showed that any additional 1s to the left of the least-significant 1 would not affect the output of the encoder. We chose to use unit testing because it is much simpler to test the functionality of individual components before integrating them with the rest of the system - this is consistent with the design principles of modularity and encapsulation. After implementing and testing the priority encoder, it was nice to view it as a black box with input and output connections.

Table 1 shows the test cases we created in order to test the correctness and functionality of our designs. The test cases were a part of unit testing. Each test case was created in order to test a different functionality of the design. The first three sets of test cases ensure that the identity and zero properties of multiplication are working with our design. The 3 positive and negative small number test cases test that the multiplication is executing correctly without testing for overflow. The 3 positive and negative large number test cases test for multiplication correctness with overflow. Masking bits allows multiple bits to be either set on, off, or inverted from on to off in a single bitwise operation. The low and middle order bit masking test cases ensure that our design works with masking. The sparse numbers with many zeros but few ones test case was created primarily for testing the alternative design. If there are many consecutive zero bits, the alternative design should optimize the multiplication by shifting multiple times at once instead of one bit at a time. The dense numbers with many ones but few zeros test case was also created to primarily test the alternative design. If there are many consecutive ones and few zeros, the alternative design should only shift multiple times in one cycle if there are consecutive zeros. This means that if there are many ones and very few zeros, the baseline and alternative designs should take a relatively similar amount of clock cycles for execution. For example, -1 in binary means all 32 bits are 1s. Therefore, the alternative design could never shift more than once at each cycle, meaning the two should take the same amount of clock cycles for execution. Corner cases (such as truncation) and random sink and source delays were also tested for the test cases we created. Both our baseline and alternative designs passed all the provided and our own test cases. These diverse test cases were created because they tested all the functions and possible types outcomes of the multiplier and passing them allowed us to confirm the correctness of our designs.

Section 5: Evaluation

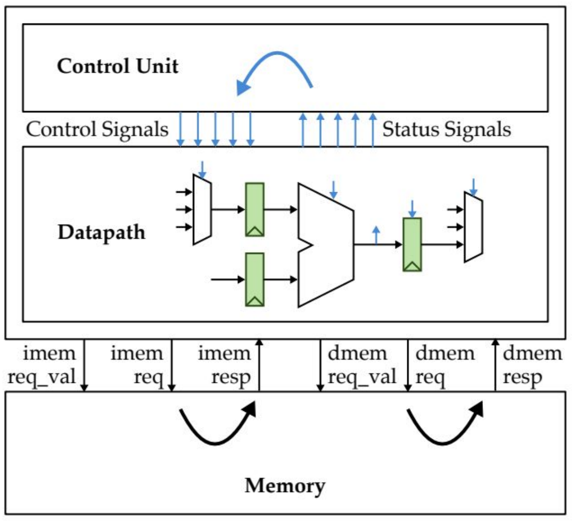
We chose to evaluate cases that would range in level of optimization, for example highly optimized, somewhat optimized, barely optimized, and not optimized. Our raw data for all these cases can be viewed in Table 2 and in Graph 1. That data demonstrates that our design was not optimized only for the specific case of -1 x -1. The number of cycles for this worst case in our alternative design was identical to the number of cycles the baseline design will always take. This behavior was what we predicted since our alternative design does not greatly optimize numbers that have dense 1s and few 0s. Our optimization process looks for zeros so that it can skip cycles that don't contribute to the final result, and -1x-1 has no zeros. So for very small negative numbers, the alternative design required less cycles, but was not very different from the baseline design. However, for cases of small positive numbers, and especially very small numbers (such as 1 through 5), the required cycles are 18% or less of the cycles the baseline design required. The most optimized is the case for a zero input, which only requires 3 cycles per multiplication (9% of what baseline design would take). Overall, for randomized inputs of all possible values, the number of cycles per multiplication was 19.40 (55% of baseline cycles). This shows significant improvement in the alternate design with respect to cycle time, decreasing the number of cycles to roughly half of what they were in baseline design. Reducing the cycles per instruction increases the performance.

The area that a particular feature occupies is an important aspect of architectural implementations and tradeoffs. Instruction sets that require more area are considered to be less valuable than those that require less unless they make up for area consumption with significant performance enhancements. The alternative design has more logic than the baseline design in order to optimize the number of clock cycles. Increased logic would mean increased area consumption with the alternative design. However, the level of optimization that the alternative design provides is worth the area consumption tradeoff. Therefore, although the alternative design may consume more area due to additional logic, the significant increase in performance that it produces makes the extra area consumption acceptable.

Energy consumption is another important factor of computer architecture. Reducing energy use is a crucial part of reducing operating costs. However, in some situations, prioritizing energy-use reduction is not the most important. For example, reducing energy consumption at the cost of performance degradation may not be acceptable. Comparing our baseline and alternative designs, the increase in energy use caused by the additional priority encoder logic is easily offset by the performance optimization that the alternative design provides. As can be seen in Graph 1, in certain situations, the alternative design can lead to a significant reduction in cycle time, which is more favorable than a small decrease in energy consumption. Therefore, although the alternative design may consume more energy due to additional logic, the significant increase in performance that it produces makes the tradeoff acceptable.

Overall, the results of the evaluation demonstrate the differences between a single-cycle processor and an FSM processor. A single cycle processor, much like our baseline design, has a fixed latency. Therefore, the cycle time and energy-consumption are not optimized because the baseline design misses the opportunity to take advantage of the processor's available resources. However, as previously discussed, the alternative design optimizes both the cycle time and energy-consumption through increased performance and efficiency. Although the area consumption was not decreased with the alternative design compared to the baseline design, the upgrades that the alternative design brings are far more important than the amount of area that it consumes. Therefore, we can conclude that the alternative design is superior to the baseline design.

Section 6: Additional Diagrams



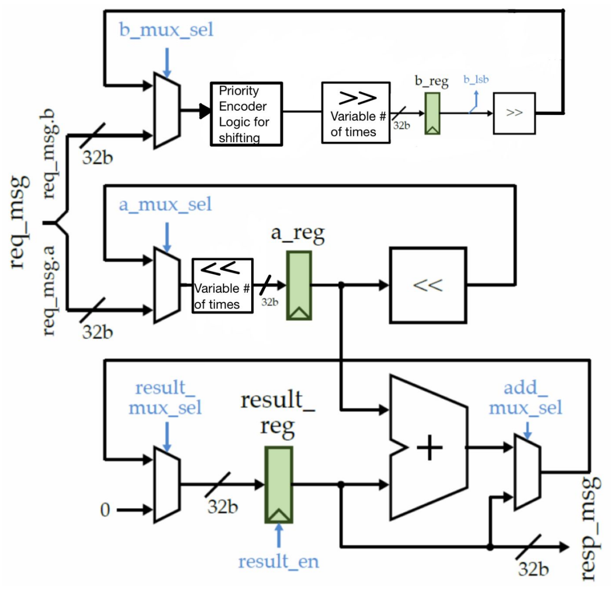
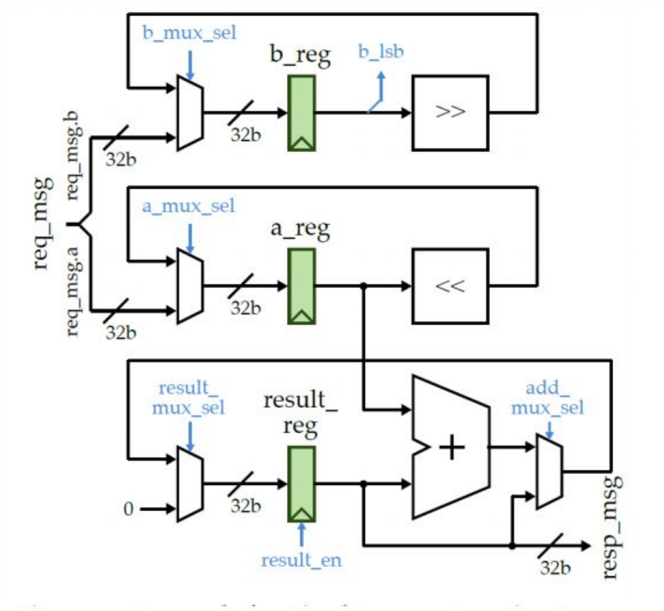


Figure 1: Microarchitecture: Control/Datapath Split Design

Figure 2: Full Datapath for Alternative



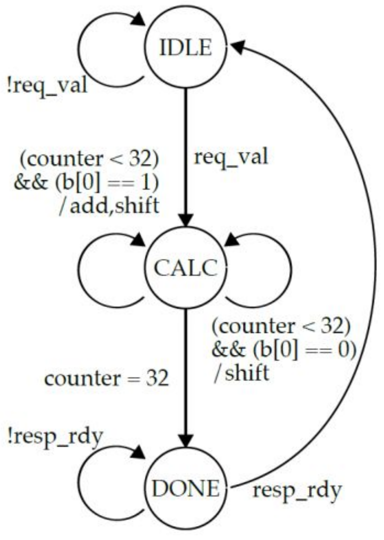

IDLE
!req_val
req_val
(counter < 32) && (b[0] == 1) / add,shift
CALC
(counter < 32) && (b[0] == 0) / shift
counter = 32
!resp_rdy
DONE
resp_rdy

Figure 3: Datapath for Fixed-Latency Iterative Integer Multiplier - All datapath components are 32-bits wide. Shifters are constant one-bit shifters. We use registered inputs with a minimal of logic before the registers.

Figure 4: Control FSM for Fixed-Latency Iterative Integer Multiplier - Hybrid Moore/Mealy FSM with Mealy transitions in the CALC state.

Table 1. Test Cases (in IntMulFL_test.py) - each set has three sub-tests numbered 1-3

Test Cases	#1	#2	#3
zero * number	0 * 3	-4 * 0	0 * 0
one * number	1 * 1	1 * 9	1 * -12

negative one * number	-1 * 1	-1 * 9	-1 * 12
small neg * small pos	-2 * 3	-5 * 12	-12 * 11
small pos * small neg	4 * -3	7 * -10	13 * -15
small neg * small neg	-4 * 3	-7 * -9	-10 * -2
large pos * large pos	131072 * 131072	262144 * 1048576	1024 * 8388608
large pos * large neg	131072 * -131072	262144 * -1048576	1024 * -8388608
large neg * large pos	-131072 * 131072	-262144 * 1048576	-1024 * 8388608
large neg* large neg	-131072 * -131072	-262144 * -1048576	-1024 * -8388608
low order bits masked	-4096 * -1	-4096 * -1048573	-4096 * 8123
middle order bits masked	-2097025 * -1	-2097025 * 5	-2097025 * 2000
sparse numbers with many zeros but few ones	5 * 9	73 * 69	4294967296 * 1
dense numbers with many ones but few zeros	-9 * -9	-513 * -129	-8193 * -1

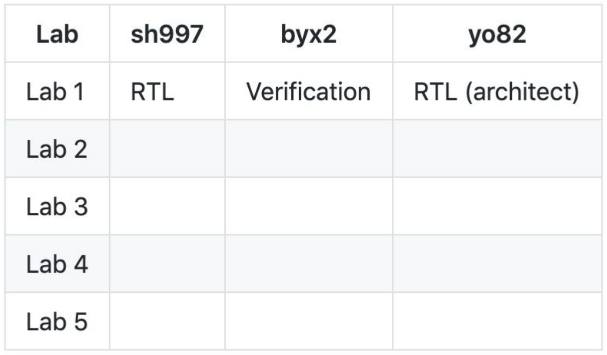
Table 2: Alternative Design Evaluation Table

Cases	Zero	Tiny	Small	All	Tiny Negative	Negative One
Random Inputs (both factors)	(0)	(0, 4)	(0,100)	(2147483647, -2147483647)	(-5, -1)	(-1)
Number of Cycles	156	205	322	970	1709	1756
Number of Cycles per Multiplication	3.12	4.10	6.44	19.40	34.18	35.12

Table 3: Baseline Design Evaluation Table

Cases	All Cases
Number of Cycles	1756
Number of Cycles per Multiplication	35.12

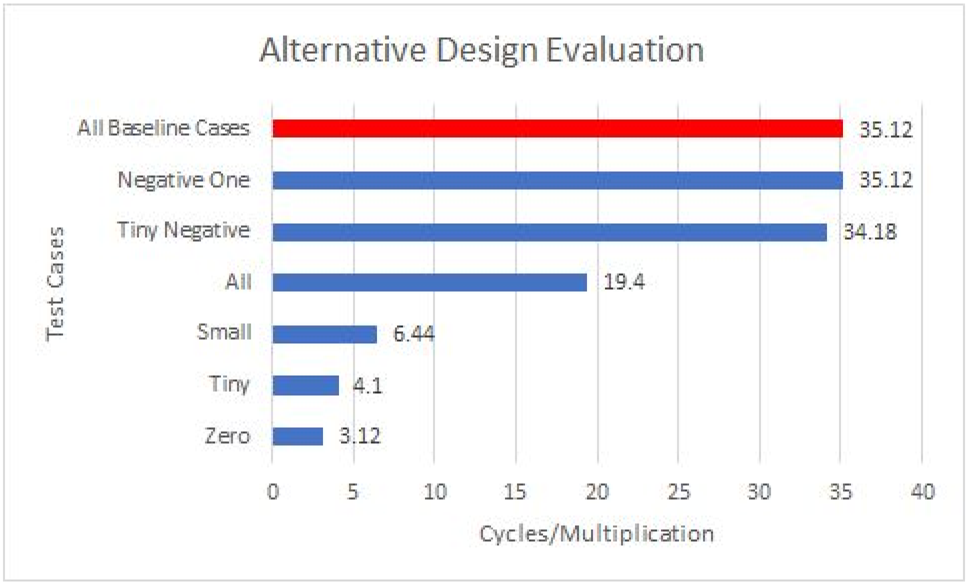
Table 4: Role and Task Table



Lab	sh997	byx2	yo82
Lab 1	RTL	Verification	RTL (architect)
Lab 2			
Lab 3			
Lab 4			
Lab 5			

	sh997	byx2	yo82
Tasks	Implement baseline and alternative designs, prepare pseudocode, prepare high-level descriptions	Implement baseline and alternative designs, unit testing, line tracing, debugging unexpected behavior	Prepare test cases, implement baseline design, sketch out a datapath diagram, FSM diagram, or block diagram, keep track of milestones

Graph 1: Alternative Design Evaluation Plot


Alternative Design Evaluation
Test Cases
All Baseline Cases
Negative One
Tiny Negative
All
Small
Tiny
Zero
Cycles/Multiplication