LAB 1 REPORT: ITERATIVE MULTIPLIER

Luke Puchalla [lcp77]
Eldor Bekpulatov [eb654]
Shivangi Gambhir [sg2439]

Luke Puchalla (RTL Design Engineer):

- I developed on the code framework written by my fellow RTL design engineer
- Together, we drafted pseudocode, sketched module representations and data flow charts, and peer coded
- I helped on developing a few more test cases that would give a good test to the design
- I designed the new module in the alternate design, the key difference between the baseline and alternate designs
- I finalized and formatted the working version of our alternate design, along with my RTL design partner
- I worked with my RTL design partner to debug the alternate design
- I completed the test strategy and evaluation section in the report

Eldor Bekpulatov (RTL Verification Engineer)

- I was responsible to carry out the test cases and evaluation on close discussion with the RTL design engineers
- I put in tables and charts depicting the performance of both the designs
- I also put in the introduction in the report that summarizes what the designs are and how they work

Shivangi Gambhir (RTL architect and Design Engineer):

- I initiated the group meetings and chalked out a roadmap for the implementation and testing of the design
- I coordinated within the group and allotted roles so that everyone could take some learning out of the lab
- I wrote the initial code along with the other RTL design engineer for the baseline and alternative design and provided a plan of test cases to the verification engineer so that the design is tested well
- Along with the other RTL design engineer, I sketched out the basic architecture and FSM diagram for the alternative design
- I debugged the module designed by the fellow RTL design engineer for alternative design
- I worked on the baseline and alternative design part of the report which describes the implementation, trade offs and working of our iterative multiplier
- In the end I was responsible in collaborating and connecting the report together and assessing the performance differences between the two implementations

INTRODUCTION

In the first lab, we designed and tested two implementations of an iterative multiplier. The first baseline implementation was analogous to a single-cycle CPU always requiring the same number of cycles to complete one instruction. The alternative(alt) design is a variable-latency implementation that helps to reduce the cycle time by increasing the cycles per instruction. Our implementation of alt design took advantage of the input operands and decided the number of shifts based on this. This replaces the need of a single shift every cycle to the required number of shifts in one cycle. This lab was designed to give students the opportunity to get familiar with the development environment of System Verilog and PyMTL and the concepts of processor performance using fixed latency and variable latency taught in class. This lab is part of a series of labs that will lead to a multi-core processor that is capable of executing parallel instructions. The implementation and evaluation of both the designs gave us a fair idea of how the cycle time got optimised and reduced. We saw an **improvement of about 63%** in the total number of cycles taken when a specific test case was run on both the base and alt design. The dataset for the test cases consisted of masking the lower order bits of the input operands and carrying out a multiplication operation to test the best case improvement. We also observed an increase in the cycles per instruction using the alt design as more operations were done in a single cycle. With only one additional mux and a synthesizable combinational block in the alt design and an almost 3x improvement in cycle time, it gives a good area-time tradeoff between the two. This increased performance comes at the cost of hardware (space) and energy usage.

BASELINE DESIGN

The baseline structure is designed on the basis of the conventional add and shift method to solve a multiplication problem. This algorithm is based on the concept that A adds itself B times. To multiply two numbers by paper and pencil, take the digits of the multiplier one at a time from right to left (shifting it right every time), multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the positions to the left of the earlier results (shifting it left every time). As an example, consider the multiplication of two 4-bit numbers (2x3) as depicted in Figure 1.

We can take this as a framework to implement binary multiplication where if the multiplier digit is 1, the copy of the multiplicand can be taken and if the multiplier is 0, a string of zeros equal to the number of digits of the multiplier can be placed in appropriate positions. The first implementation of the above algorithm is using our baseline design which we divide into two units, i.e. the datapath and control unit shown in Figure 2 and 3 respectively. The data path consists of all the logical and arithmetic operations carried out on the basis of control signals received by the control unit.

The datapath receives a 64 bit request message from the test source which it breaks it into multiplier and multiplicand (each 32 bit). Based on the current state, the control unit sends a signal as b_mux_sel and a_mux_sel to select between the original message or the shifted message After the message is selected, a and b registers are used to clock the input to the register output. The result is initially stored as 0 which is controlled by result_mux_sel controlled by the current state. Similar to a and b,the result is then registered out. Based on the

lowest significant bit of b, the datapath sends a status signal to the control unit to do a shift or add+shift operation. The message is then repeatedly passed to the shifter (right shift for b and left shift for a) and adder (based on LSB of b) till it is in the calculation state where it will remain for 32 cycles (we assume a 32 bit number requires 32 shifts). After completed iterations, until the counter reaches a maximum of 32, a 32 bit response message is taken to the test sink. This baseline design takes a fixed number of cycles i.e. equal to the size of the input irrespective of its value and that is why it stands a worst case scenario and our next design is enhanced on the basis of this.

This design gives us a good opportunity to apply the concepts of fixed latency processors in designing a baseline structure for an iterative multiplier. We also got an idea about the different styles of system verilog coding i.e. functional and RTL level, structuring our code into two different blocks i.e. data and control and connecting it under a single module, using encapsulation to use only the high level blocks without exposing the logic behind them, preserving the modularity of the code which makes it increases the debugging and reading capability.

ALTERNATIVE DESIGN

The alternative design uses the concept of variable latency by making use of the value of the operands and not only the size like in baseline design. The logic of add and shift remains the same but instead of a single bit shift in each cycle, we make use of multiple shifts in a single cycle. An implementation of our alt design can be seen in Figure 4 and 5.

In addition to the baseline design, we make use of a synthesizable combinational unit to compute the number of shifts required for the A and B registers and an additional mux that lets us decide between shift and add-shift operations based on the LSB of the B register. It can also be seen that we have added an additional status signal(“done”) from the data path to the control unit which is set as soon as the B register has totally shifted out i.e. is equal to zero. The done signal is used to control the calculation state and comes out of it as soon as it is set. This prevents us from using a preset counter that goes upto 32 cycles for every multiplication operation.

This works very well for sparse and masked numbers where the occurrence of zeros is more than ones and we can shift multiple zeros in one cycle. For example, taking the multiplier as 0x0000000042, the total number of cycles required by baseline design in the CALC state is 32 whereas alternative design will just take 4 cycles to go to the DONE state.

This experiment helped us analyse and fool proof our concepts of variable latency processors from class. Variable latency multipliers are expected to give lesser number multiplication, especially for special cases like sparse and masked numbers. The results can be seen in Table 1. Without affecting the critical path, with only 1 additional mux, we are able to achieve a significant impact.

TESTING STRATEGY

Our testing strategy consisted of directed testing and random testing. Besides the interface control signals (e.g. req_val) and delays for such signals (e.g. src_delay), the only influence the user has on the multiplier are the two

integer inputs that we give it, so our testing strategy inherently must revolve around manipulating the inputs, testing edge cases and different combinations of inputs. Following directly testing the capability of handling different inputs, we tested our multiplier's fidelity with varied source and sink delay times. Finally, we created test cases that generated random integer inputs of various forms and under different constraints, as well as randomized delay times, to fully ensure the correctness of our implementation. We wrote all of our tests in PyMTL, first checking the verity of each test against the FL model, and then applying our tests to our baseline or alternate implementations. We made use of the trace capability to track certain registers and confirm their status.

As delineated in the handout (and also in **Table 2**), we have a sequence of directed test cases that demonstrate the multiplier's functionality in certain situations. In order to conclude that our implementations work correctly in all cases, we need to actually define and test those different situations. For example, in an alternate implementation, the multiplier behaves differently depending on the actual inputs (or, in our case, depending on what b is). If b is sparse with ones, the multiplier will behave much differently than if it had many consecutive ones; in the sparse-ones instance, the multiplier will shift more bits at a time, as described in the last section. Our directed tests include testing different combinations and sizes of positive and negative numbers, basic and edge cases (like multiplying by 1, 0, or -1), and masking sections of bits on the inputs, to name a few. Again, all of our directed tests are summarized in **Table 2**. For each directed case, we wrote on the scale of 5-10 individual tests (meaning, for the case of "negative times positive", we wrote 5-10 tests with different negative and positive factor combinations).

After our implementations passed all of our directed tests, we wrote several test cases that generate random inputs. The purpose of random testing is to check our implementation's functionality with an even wider range of inputs (outside of our directed test cases). Occasionally, random tests may unveil minute or odd errors in design. Fortunately, for our group, we encountered no such surprises during random testing; this is likely due to our thorough test bank of directed test cases. We generated random inputs for both the factors (integer inputs a and b) and the delay controls. Within our random tests, we were able to control the "random-ness" of the inputs by masking sections of the number. So, we could target a range of numbers and then "randomly" test that range. Though we could have run thousands, we produced (and passed) 40 to 50 tests per "random" test case.

Upon completion of all of our directed and random tests, we determine that both our baseline and alternate/optimized implementations of a 32-bit integer multiplier are fully functional and correct. We have written test cases for every reasonably-conceivable category of number combinations. This is not to say that we didn't fail tests at first. In fact, the failure of certain tests, and the subsequent correction of the code, only increased our confidence in our implementations. A quick example: after our alternate implementation failed our first round of testing, we traced our registers and found a specific number that our multiplier failed on. We noticed it failed to shift the register when it was supposed to, and discovered that the reason it did not recognize it that the number should be shifted is because we defined our case statements with hexadecimal 32-bit numbers, and not fully-binary 32-bit numbers. In summary, after thorough directed testing and targeted random testing, we can safely justify the functionality of our implementations.

EVALUATION

After reviewing the results in **Table 1** and **Figure 6**, we have realized several conclusions related to area use, energy consumption, and performance. The variable-latency multiplier performs the same operations as the fixed latency multiplier in less clock cycles, but this comes at the cost of more hardware, and thus more energy and area use.

Even in its worst case scenario, where it took the MOST clock cycles, the alternate design still completed in less cycles that the baseline implementation. Of course, the speed of the variable-latency multiplier varies with the inputs, but on average, this multiplier is 2-3 times as fast as the fixed-latency timer. Plus, the critical path of each multiplier is the same, so the variable latency multiplier uses less clock cycles, but doesn't sacrifice cycle time.

The variable-latency multiplier does use more hardware than the fixed latency timer, and thus more area and energy. The AND gates and mux used to implement the advanced shifting capability of the variable-latency multiplier take up more space (and also consume more energy, compared to the fixed-latency multiplier).

Thus, the tradeoff is clear; increased performance, at the cost of space and energy use. If energy use or space was a concern, the fixed-latency multiplier would be more suitable. If performance outweighed those factors, however, the variable-latency multiplier would be preferred.

Multiplicand				0	0	1	0	mul
Multiplier				0	0	1	1	
				0	0	1	0	
			0	0	1	0		
		0	0	0	0			
	0	0	0	0				add
				0	1	1	0	

Figure 1: Multiplication of two 4-bit numbers

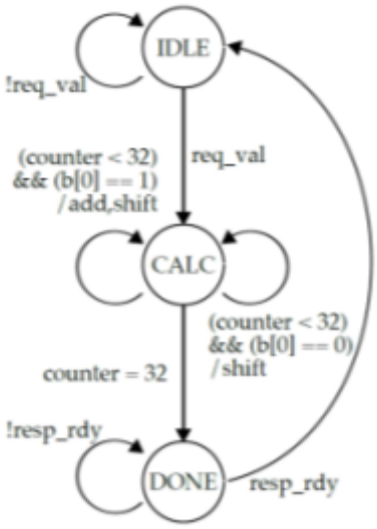

Figure 3: FSM of control unit of Baseline

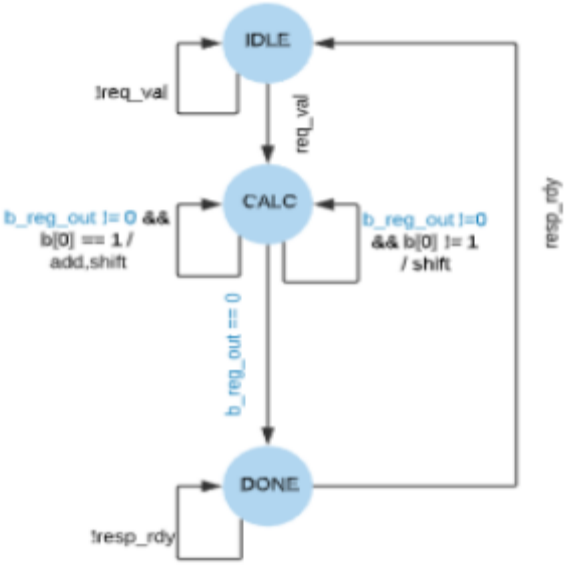

Figure 5: FSM of the control unit of alternative iterative multiplier

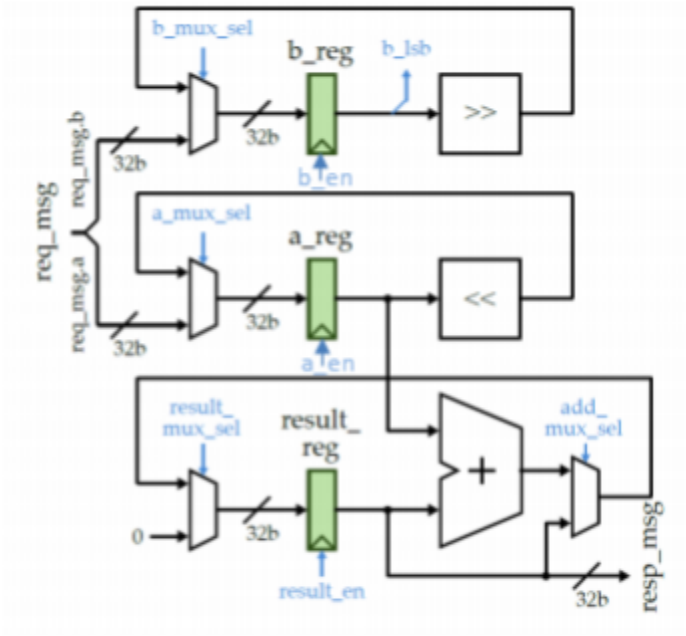

Figure 2: Datapath for the fixed latency iterative multiplier

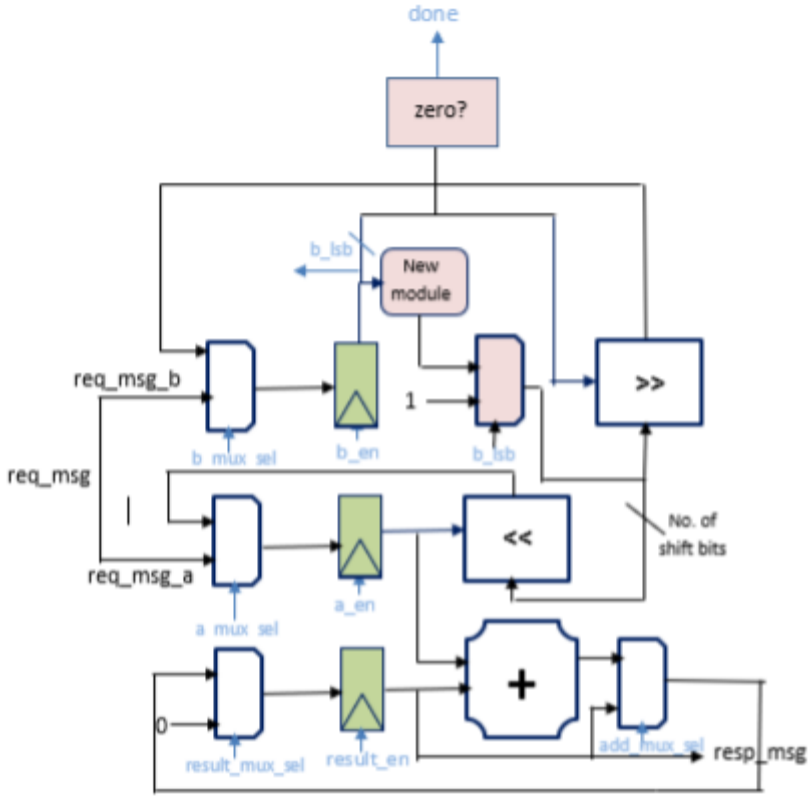

Figure 4: Datapath for Variable Latency Iterative Multiplier

Dataset	Baseline design	Alternative design
Random small positive	35.12	7.88
Large positive x Large positive	37	31
Lower bits mask off	37	13.67
Higher bits mask off	37	16.33
Low-High mask off	35.12	20.12
Sparse Numbers	37	19.67

Table 1: Representing the number of cycles per multiplication for various test cases

Table 2:

Summary of directed and random test cases:

Test Case Profiles	Tally
Combinations of multiplying zero, one, and negative one	9
Small negative numbers×small positive numbers	5
Small positive numbers×small negative numbers	5
Small negative numbers×small negative numbers	5
Large positive numbers×large positive numbers	3
Large positive numbers×large negative numbers	3
Large negative numbers×large positive numbers	4
Large negative numbers×large negative numbers	4
Multiplying numbers with the low order bits masked off	3
Multiplying numbers with middle bits masked off	3
Multiplying sparse numbers with many zeros but few ones	3
Multiplying dense numbers with many ones but few zeros	3
Multiplying random small numbers	50
Multiplying random large numbers	50
Multiplying random numbers with low masking	50
Multiplying random numbers with high masking	50
Multiplying random numbers with hi & low masking	50
Multiplying random numbers random delays	40

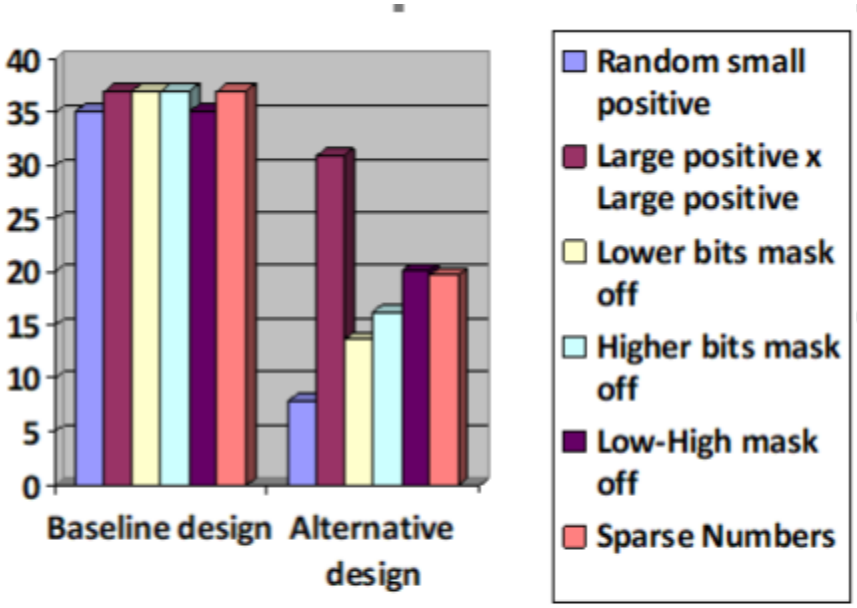

Figure 6: A pictorial representation showing significant improvement in number of cycles per multiplication