Lab1 Pipelined Integer Multiplier Report

Yibang Xiao (yx455), Yuxiang Long (yl3377)

Introduction:

In this lab, we have experienced the procedure of analyzing the area, energy, and performance of the design we made through the standard-cell-based ASIC tool flow quantitatively, which we didn’t do in detail during the Computer Architecture Course last semester. It is important to make a successful hardware design, but another crucial thing is to test the performance of the design. As we all know, it has been a trend to make electronic devices smaller and smaller. So it is crucial to optimize the design according to the test report. And this lab has taught us a reliable method to test the design and optimize it. Besides, this lab connected tightly with the lecture by introducing us to the detailed procedure of the ASIC automated design methodology, which is one of the key points of the lecture. We have completed the Alternative design which implements several stags into the multiplier and made it a pipelined multiplier successfully. And the alternative design has passed all the test cases. Then we finished pushing the FixedLatency, VariedLatency, Single-cycle and NstagesMultiplier designs through the ASIC tool flow, and successfully generated the analysis report which told us the performance of the designs and the place-route layout of the designs. In order to realize the pipelined multiplication, we build a partial unit which can do (32/ number of stages) bits shift and add, and then connect several units together with registers between each of them. For example, if we need to build a 4-stages multiplier, the number of stages will be set to 4, then we need to connect 4 partial units together, each unit performs 8 bits shift and add, then the temporal result and shifted inputs will be transferred to the next unit until the final result is generated. From our evaluation results, we can see that if we take out the NstagesMultiplier, the Fixed-Latency Multiplier has the smallest Area, which is only 870 um2, and the Single-Cycle Multiplier has the largest area, which is 2201 um2. In terms of the energy, the multiplier consumes different energy with different test set (small, large, sparse), but in general, the Single-Cycle multiplier consumes the least energy (vary from 0.2178 nJ to 0.6754nJ), and the Fixed-Latency multiplier consumes energy the most (vary from3.754 nJ to 4.431 nJ). As for the execution time, the FixedLat multiplier is always 3406 cycles, and the Varlat multiplier and the Single-cycle multiplier all less than it, one is 787 cycles, and the other is 107 cycles. All in all, we think the Varied-Latency multiplier performs the best, the reason is that it can detect the 0 in the multiplicand and then skip the add operation, which can save energy. Besides, the area and execution time of the Varied-Latency multiplier is also proper. Although the Single-Cycle multiplier has the lowest execution time and energy consumption, the area is too big so that it might not be proper to be used in reality. During this lab, Yibang Xiao designed the Alternative RTL and made the schematic of how to realize the control of the datapath. Yuxiang Logn performed the test and verification of the Alternative RTL design and the baseline Design. And both of us successfully finished the evaluation.

Baseline Design:

The Baseline Design consists of three parts, which are Fixed-Latency Design, Varied-Latency Design and the Single-Cycle Design. The datapath and the control logic of the Fixed-Latency Design and the Varied-Latency Design are described in below. The Fixed-Latency Design will shift the multiplicands one bit per cycle and then perform add operation. To be more detailed, every cycle, the least significant bit of multiplicand A will be detected, then A will be shifted right. If the LSB(A) is 0, 0 will be added to the result, if it is 1, then multiplicand B will be added to the result. After that, multiplicand B will also be shifted left. Then comes the next cycle. The main execution time of this multiplier will be fixed at 32 cycles because the multiplicands are 32 bits. The datapath and the control logic are shown below in figure 1 & 2. In order to improve the calculation efficiency, we come up with the Varied-Latency design, which can detect the number of consecutive “0”s in the multiplicand. If there are 0 in the multiplicand, then the multiplier will not perform add operatio

n. For example, if there are 3 consecutive “0”s in the multiplicand A, then the multiplier can shift A right and shift B left 3 bits in 1 cycle without performing add operation. And this function can greatly save cycles if there are “0”s in the multiplicand. The datapath and the control logic are shown below in figure 3 & 4. Another one is the Single-Cycle multiplier, which can perform all the 32 bits shifting and adding in one cycle. Because all the operation will be done in one cycle, the critical path will be large and the combinational logic in the datapath will be complex. In this lab, the RTL design of the Single-Cycle multiplier will be made by the ASIC design tool, because we implement a * operation symbol in the design file instead of doing the RTL Design. The reason that the Single-Cycle is proposed is that it can greatly reduce the execution time and help save energy greatly.

Alternative Design

In Alternative Design, we used a modified fixed-latency iterative multiplier as the pipeline stage because we think that the fixed-latency multiplier can perform well with small time constraint due to its minimum combinational logic between its registers.. Each of them has a separate datapath and control module. For example, if the number of stages is four, we will have a block diagram shown in Figure 5. Each stage shifts a and b eight times and sends the shifted a, b, and sum to the next stage. Therefore, we count 8 cycles as one pipeline unit and pipeline those eight cycles. It is a little bit different from the processor pipeline because the processor pipeline takes 1 cycle as one pipeline unit. There is a register followed by each stage to store the shifted value of two inputs and the added result: sum. There is one additional register at the beginning of the stage to hold the input signals of the whole module for one cycle.

Each stage can receive the ‘done’ signal from the previous stage and generate ‘done’ signal for the next stage. For example, the first stage will enter the ‘calculation state’ if it receives the req_en signal. It is the ‘done’ signal from the test source. After eight cycles (assume # of the stage is 4), it finishes the calculation and enters the ‘Done State’ and in the meanwhile, it asserts the ‘done’ signal for the second stage. The second stage will receive that signal and start to continue the operation left by the first stage while the first stage takes the next inputs and starts the next round of computation. The ‘done’ signal from the last stage is connected to the resp_en to tell the test sink that the result is ready to transmit.

Diving into the small pipeline stage module shown in Figure 6, we separated this module into two parts: datapath and control module. The datapath is shown in Figure 8. It has two shifters which can shift input ‘a’ to left and input ‘b’ to right in each cycle. Depending on the least significant bit of input ‘b’, the result_mux will choose either previous sum result or the zero to add with ‘a’. This datapath has three outputs: shifted ‘a’, shifted ‘b’, and the resulting sum and each of them is 32 bits. We connected the wire “a_reg_out” and “b_reg_out” to the output ‘a’ and output ‘b’ respectively. We do not connect the “a_shift_out” and “b_shift_out” to the output otherwise the ‘a’ and ‘b’ number will be shifted one more time before passing to the next stage and it causes the wrong result. Another modification compared to the Baseline Design is that we add the enable pin on the a_reg and b_reg. This modification can prevent the ‘a’ and ‘b’ from keeping shifting if the current stage is in ‘Done State’ because of the lack of resp_rdy signal. If the multiplier stage does not receive the resp_rdy signal, it will stay in ‘Done Stage’ and turn off the a_reg and b_reg to stop shifting.

The alternative design meets both the modularity and hierarchy. Each pipeline stage, called the small multiplier, has the same interface: three inputs, three outputs, one req_begin input which can initiate the ‘Calculation State’ and one ‘done’ signal to indicate it finished computing. This small multiplier was tested by unit test before integrating them together. The same interface makes sure that we can initiate them in the parameterized pipeline without adding extra hardware modules or interfaces. The top-level of the Alternative Design is the pipeline stage, then each stage is built by a fixed-latency iterative multiplier datapath and a control module with a parameterized state machine. All interfaces between the datapath and control module are encapsulated by building a module, ‘M’, shown in Figure 7 to hide the details of implementation for robust composition. Then we encapsulated one further step by connecting a pipeline register to a M module, called ‘M-Reg’. In the top-level module in Figure 5, we only need to initiate the encapsulated ‘M-Reg’ four times (if the number of the stage is 4) and add an additional reg at the beginning to hold input a and b from the test source for one cycle. Due to the great modularity, our Alternative Design also meets the rule of structure regularity. All M-Reg module and control signals are in a repeated pattern so we can easily debug the tiny problems. The parameterized number of stages and clean input and output interface give the Alternative Design great extensibility. Users can customize the number of pipeline stages according to the design specification and place it to any other model which wants to do multiplication and has the handshake protocol.

Testing Strategy

In the testing part of this lab, we mainly use three kinds of tests, which are directed tests, unit tests, and random tests. And each kind of test consists of the small number test (the input numbers are small), large number test (the input numbers are large), lomask number test (the low half bits of the input numbers are 0), himask number test (the high half bits of the input numbers are 0), lohimask number test (the lowest part and the highest part of the input number are 0)and the sparse number (several bits of the input numbers are 0) test. And as for the alternative design testing, we test the functionality of the partial component before we test the whole RTL design because it can help us better figure out the bugs. The reason we choose this testing strategy is that it can help us take all the possible cases into consideration during the multiplication process and comprehensively evaluate the RTL design. The detail of the testing strategies are as followed:

The first test we write is the simple directed test. We need the simple tests to verify the functionality when the design of the partial unit is finished at the beginning. These tests can help us find the bugs quickly. The simple directed test just takes data from the test source to the instruction and then compares the calculated result with the reference result in the test sink. For example, the test of 2 * 3, we set the input value to be 2 and 3, the reference value to be 6. Then we start the test, calculate the result using the RTL design and send the result to the test sink for comparing.

The second test we used is the unit test. Unit tests can give us the comprehensive feedback of the functionality for when performing one kind of input numbers multiplication using our RTL design. We separate the test case into small numbers, large numbers, lomask numbers, himask numbers, lohimasks numbers and sparse numbers. And each of these kinds are then separated into positive & positive numbers, positive & negative numbers, negative numbers & positive numbers, negative numbers & negative numbers. The reason to create this kind of unit tests is that it can help us better evaluate the performance and characteristics of our RTL design. For example, the energy consumption of the multiplier is different when performing different numbers (energy of large number multiplication is larger than small number multiplication). After passing the unit test, it means that our RTL implementation is very robust.

The third test we designed is the random test with randomly generated input numbers. Also the random input numbers are also separated into small, large, … sparse numbers by setting the range of random generation. The reason we perform this is that we want to expose our RTL design in the real and more complex situation and detect the errors that we have not thought about. Therefore, we need the test program to generate some random cases.

The last test we implemented is the delay test. In the delay test, we delay the test source and tes t sink to check the correctness. The reason for this test is that we want to verify that our multip lier design is suitable for the real environment. Because in the real operation, the sending device and the receiver may have different speeds with our multiplier, the multiplier needs to wait until the device from the next level to be ready to receive the result. In this part, we set exact value to the source delay and test sink delay. Besides, we also generate random delay to take corner case s into consideration.

Evaluation

The area, energy, and cycles plots with different microarchitectures are in Chart 1, Chart2, and Ch art 3. The separated area of the datapath and control module for each RTL design is in Chart 7. We will discuss each Chart in each paragraph below. From Chart 1, we can see in Baseline Design, fixed -latency has the smallest area while single cycle has the largest one. The reason is that variable- latency has an additional hardware module to detect the continuous zero in the input and in the sin gle cycle, we do not specify how we want to implement the multiplier, it is the tool’s choice to f ind the best approach to implement that. In the Alternative Design, as the number of stages goes u p, the area becomes larger. Because our Alternative Design uses a fixed number of hardware to do on e multiplying step: two shifters, three registers, one adder, two muxes, and the control module, wh en we have more pipeline stages, we actually duplicated the modules several times. This causes a li near increase in the area when we add the pipeline stage from 1 to 8. From Chart 8, we plot the dif ferent areas for each microarchitecture with different time constraints. It is clearly to see that when we close the cycle time, the tool uses more area. The area increases a lot when the timing con straint shrinks from 1 to 0.5. This is because the tool might assert additional buffers in the path to improve timing and reduce the setup and hold timing violation, but the trade off is that the buf fer brings more area to the design.

From Chart 2, we can see all three designs in the Baseline consume more energy when the input data set is ‘large’ and less energy when the input is ‘small’. The reason is that the large data set contains more bits ‘1’ than the small data set. Therefore, it leads more switching activities in the computation to consume more dynamic energy. Unlike the ‘large’ data set, the ‘small’ data s et usually has 0 in their higher bits such as 0x32h00000008 so there is less dynamic switching and less energy consumed. In the Single Cycle Multiplier design, we used the behaviour RTL to describe the multiplication function and let the tool choose the most optimized implementation approach. The tool chooses a cell called ‘mult_x_1’ and it consumes the least energy and takes the fewest execu tion time. Although we used modified fix-latency iterative multipliers in our Alternative Design, t he average energy consumed by the multiplier with pipeline stage number 1 is a little bit higher th an the fix-latency iterative multiplier does because we added extra registers for the inputs and ou tputs to hold one cycle. There is a linear increase in the energy consumed when we increase the num ber of the stages. As we talked about before, adding more pipeline stages in our design means to du plicate the datapath and control module in ‘M’. Although it consumes twice energy (say # of stage is 2) when all pipeline stages are fully operating, it can take less cycles to finish all input cal culations. However, adding more pipelines brings more registers. So the overall energy goes up with the rising number of the pipeline stages.

From Chart 3 and Chart 4, we can see that the Fixed-Latency multiplier has the highest execution ti me among all the baseline designs. The Varied-Latency design has less execution time than the Fixed -Latency one, the reason is that the Varied-Latency design can detect the 0 in the multiplicand and then skip the adding operation which can save cycle number and execution time. Then we can notice t hat the Single-Cycle multiplier has the lowest execution time, the reason is that although the Sing le-Cycle design has stricter time constraint which means the cycle period is longer, the total numb er of cycles to execute of the Single-Cycle design is much lower than other designs because it can perform the whole multiplication in just one cycle. Then comes to the multistage RTL design which implements the pipeline. We can see that the execution time drops as the number of stages go up. The main reason is that the number of cycles is reduced as the number of stages increases. We can see from Chart 5 that the clock constraints of the 1-stage, 2-stages, 4-stages, and 8-stages designs are nearly the same. The reason that the number of cycles drops as the number of stages increases is that when executing many multiplications, deeper pipelines means that more cycles could be saved.

From Chart 5, we can see that the variable-latency iterative multiplier needs a little bit larger time constraint than the fixed-latency does. By diving into the DC report, we found that these two designs have different critical paths: from Figure 1, the yellow highlighted path is the critical path. It starts from the a_reg output, passes from the adder, add_mux, result_mux, and finally to the input of result_reg. The total required time for this critical path is 0.455 ns. The most time-consuming module in this path is the adder which uses 0.258 ns. From Figure 3, the DC timing report tells us that the variable-latency multiplier has a different critical path in the datapath. It starts from the b_reg output, passes through calc_shamt, right shifter, b_mux, and finally to the input of b_reg. The total required time is 0.559 which is higher than the clock constraint of the fixed-latency design. This is why the variable-latency needs larger constraint than the fixed-latency does. Both right shifter and calc_shamt modules in variable-latency design costs much time. In our alternative design, all of the four designs with different number of stages can almost access the same clock time: around 0.5. The critical path in the same path with the one in fixed-latency design because we used similar datapath in the design. The small minimum clock time is the biggest advantage of our alternative design because it is very small among all numbers of pipeline stages. We compared with other groups’ constraints of all the pipeline stages and found most of them have 5.0 for one pipeline stage and 1..0 for eight pipeline stages. Our parameterized pipeline multiplier has around 0.5 ns minimum clock time without violating timing and can be placed in a system with very fast speed (from 1GHz to 2GHz). The single cycle performs best in energy, area, and cycles while at this point, it has the largest clock constraint which is 1 ns. By checking the timing report, the critical path is from reg to out which contains some NAND, XNOR, XOR, AND gates, full adders, and inverters.

Then we plot the Energy vs. performance plot in Chart 6. The overall trend is that the microarchitecture with higher performance consumes more energy except the single cycle and Varied-Latency Multiplier. We calculate the performance by using the reciprocal of the total execution time. When we add more pipeline stages, we increase the throughput of the multiplier but more registers contribute the overall energy. When the designer creates a deeper pipeline and tries to increase the performance, in the meanwhile, the energy also rises. The single cycle has the lowest energy since the tool chooses the best approach to implement it. It is worth mentioning that the node with 1/1058.4 has a falling trend in energy because it is the variable-latency multiplier and its energy highly depends on the input data set. When the data set contains lots of zero, it will skip the operation and save energy. It is a good option to consider in further improvement when building the pipeline multiplier.

Chart 7 describes datapath & control logic area of our different RTL Design, we can see that both area of datapath and control logic increase from Fixed-Latency multiplier, Varied-Latency Design, 1-stage multiplier, 2-stages multiplier, 4-stages multiplier and 8-stage multiplier. We didn’ t take Single-Cycle multiplier into consideration because the RTL is not designed by us. The reason for this trend is that firstly, the Varied-Latency design is more complex than the Fixed-Latency one because it needed to add more control logic and datapath components to detect 0 and skip some operation to save execution time. This is an unavoidable trade-off. And as the stage of the pipeline multiplier does up, more partial components need to be created, so the area keeps going up.

In conclusion, we can see that everything comes with a trade-off. Although the execution time of Fixed-Latency design is high, its area is the smallest. And we can see that the area and energy consumption of the Varied-Latency Design are all low, but we cannot say it is the best, because its performance highly depends on the input number. The additional "calc shamt" module makes its path become the critical path and its minimum clock time is larger than the Fixed-Latency multiplier. From the energy and total execution_time side, the single-cycle mode seems the best choice, however its minimum time constraint could only reach 1ns while other designs almost reach 0.5 ns. Then come to our alternative design, we sacrificed the area and energy, in order to reduce the execution time by adding more pipeline stages. The 8-stages pipeline multiplier is even faster than the Varied-Latency multiplier. Another advantage of our alternative design is whatever how many pipeline stages we have, the minimum clock constraint will keep around 0.5 ns, which can be placed in a very fast system, with up to 2GHz.

Design Figures:

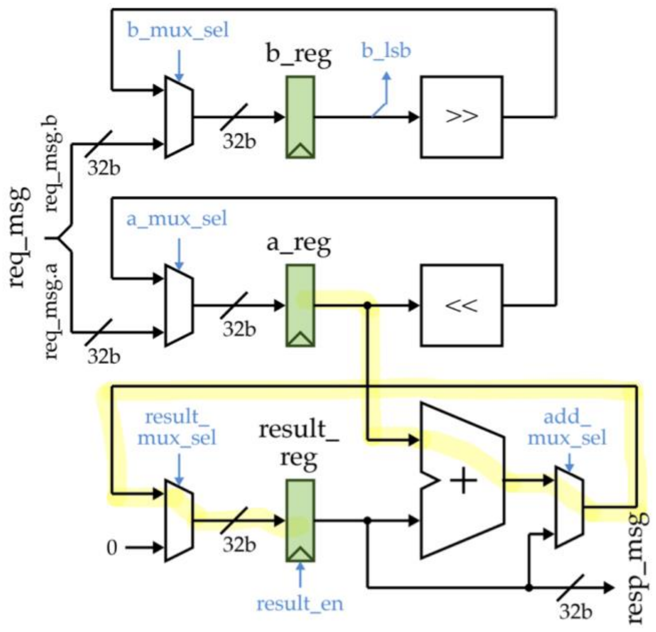


Figure 1: Datapath of the Fixed-Latency multiplier

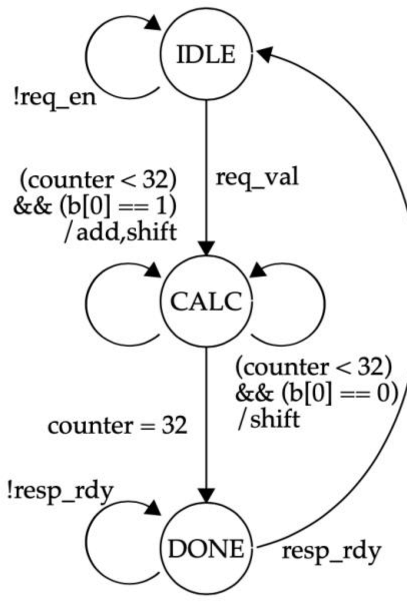


Figure 2: Control logic of the Fixed-Latency multiplier

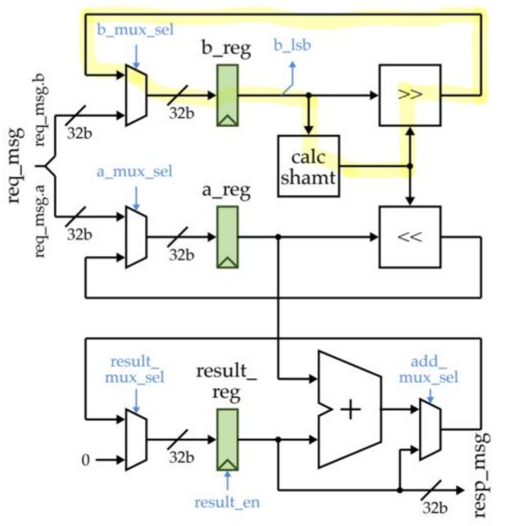


Figure 3: Datapath of the Varied-Latency multiplier

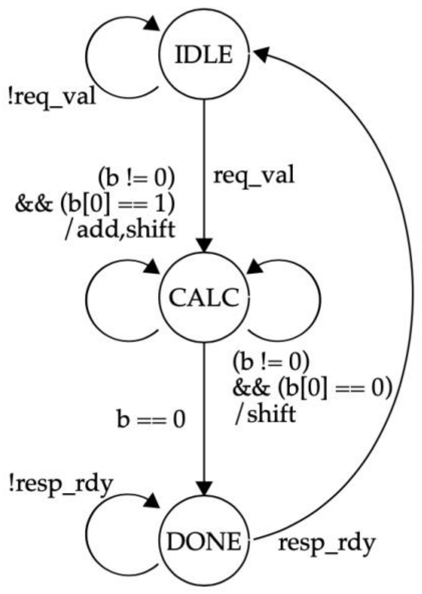


Figure 4:Control logic of the Varied-Latency multiplier

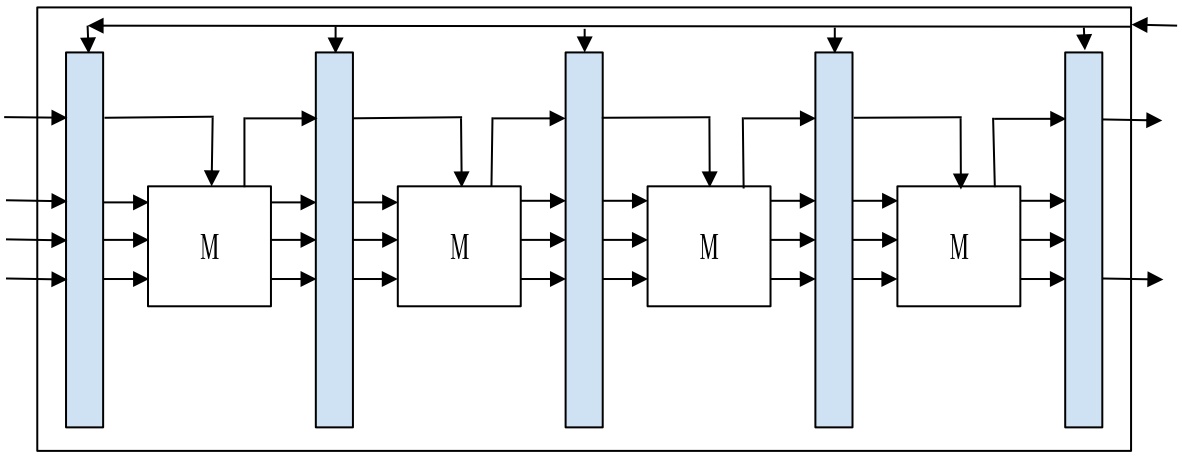


Figure 5: Block Diagram for the pipelined multiplier with stage 4

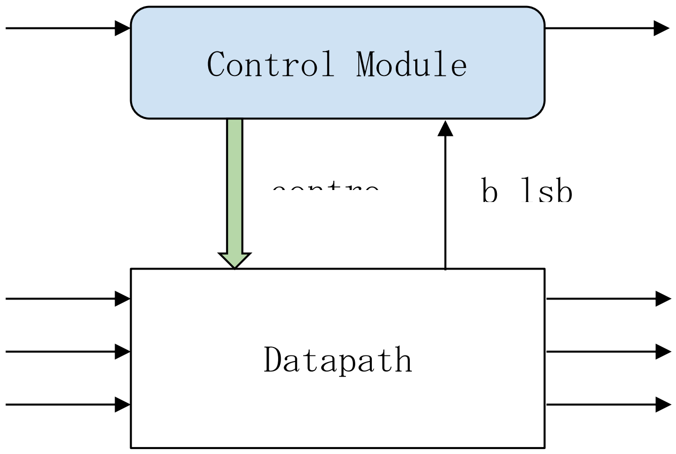


Figure 6: Block Diagram of M

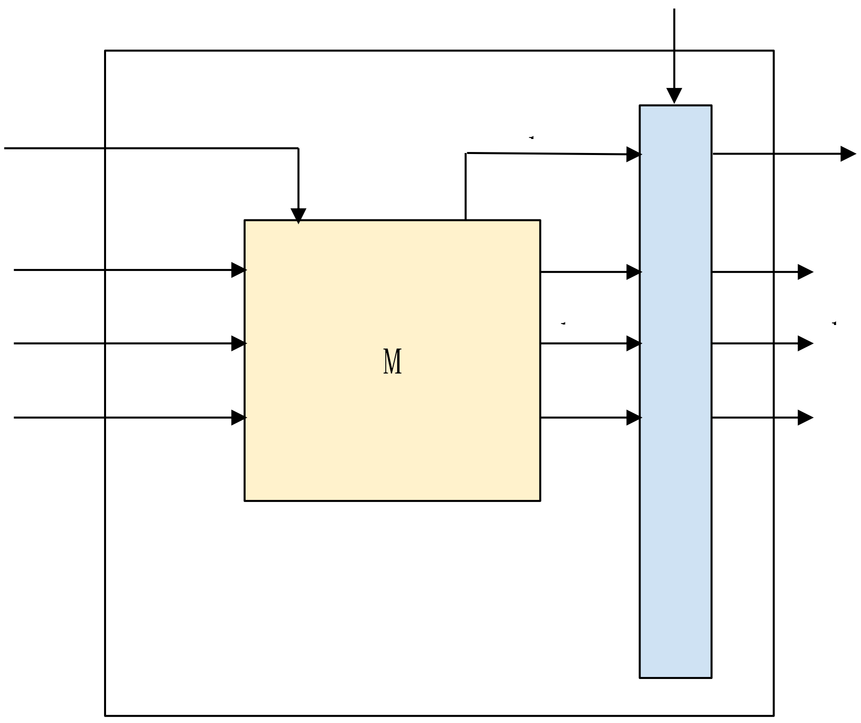


Figure 7: Encapsulating the multiplier stage with a reg

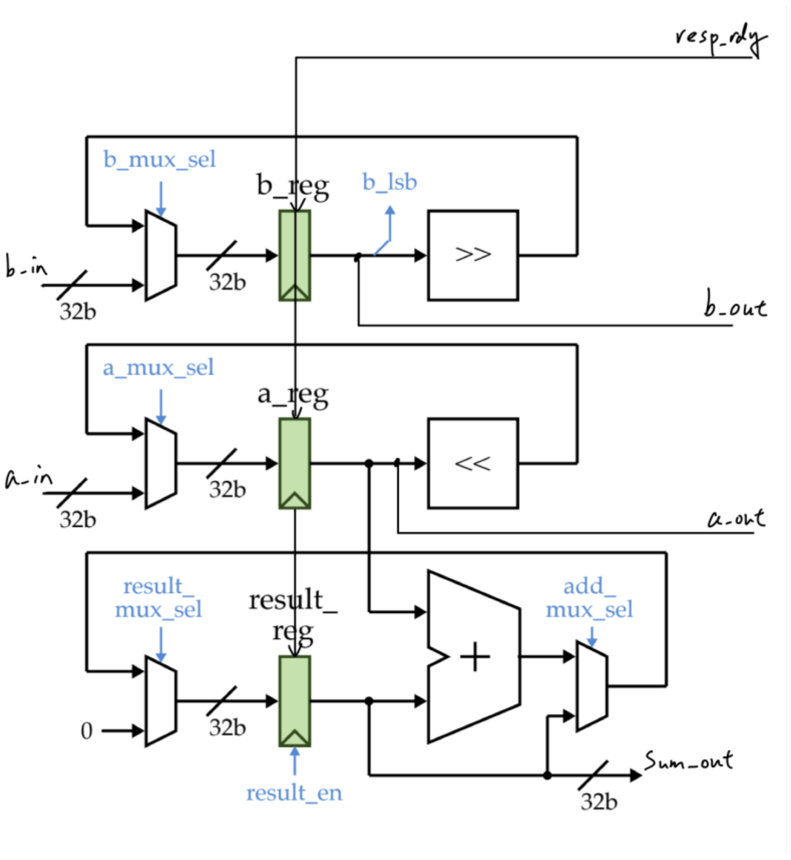


Figure 8: Datapath for each multiplier stage

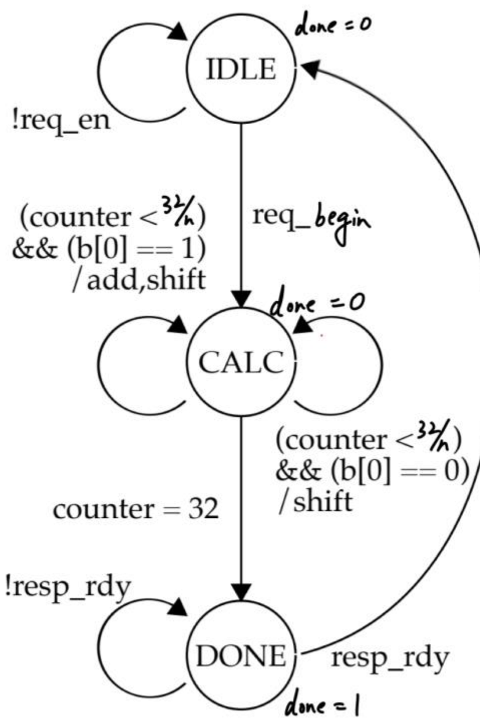


Figure 9: State Machine Diagram for the control part of the multiplier stage

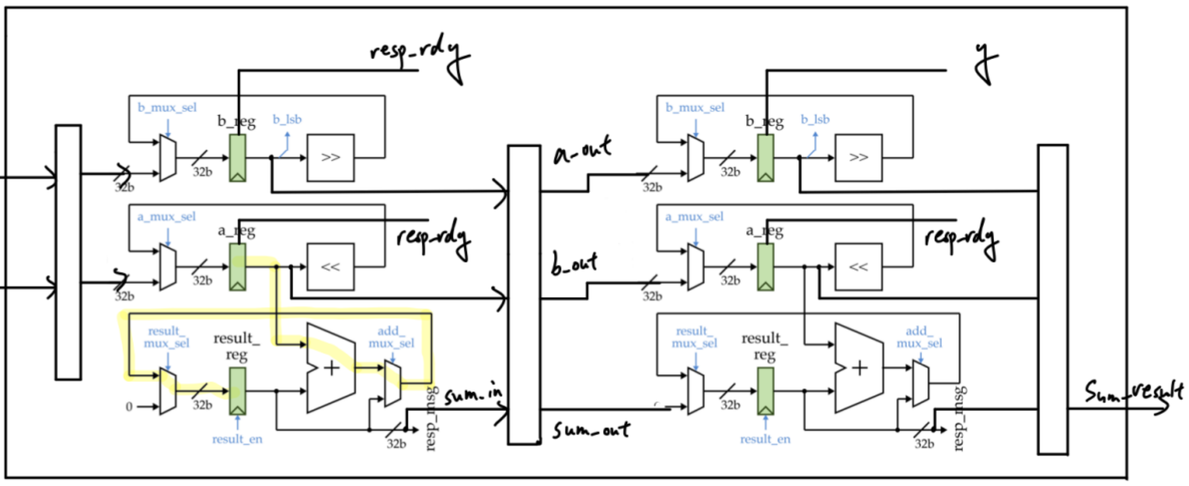


Figure 10: Datapath of pipeline with 2 pipeline stage and its critical path

Charts for the evaluation results:

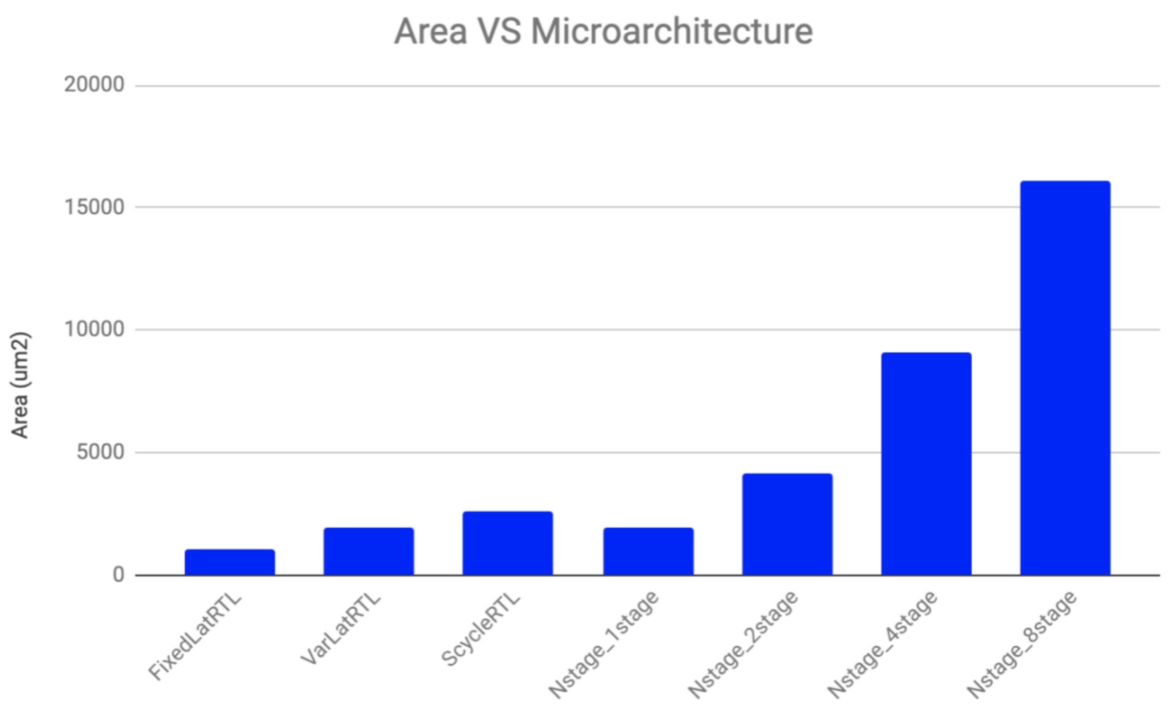


Chart 1: Area VS Microarchitecture (all of them are in minimum constraint)

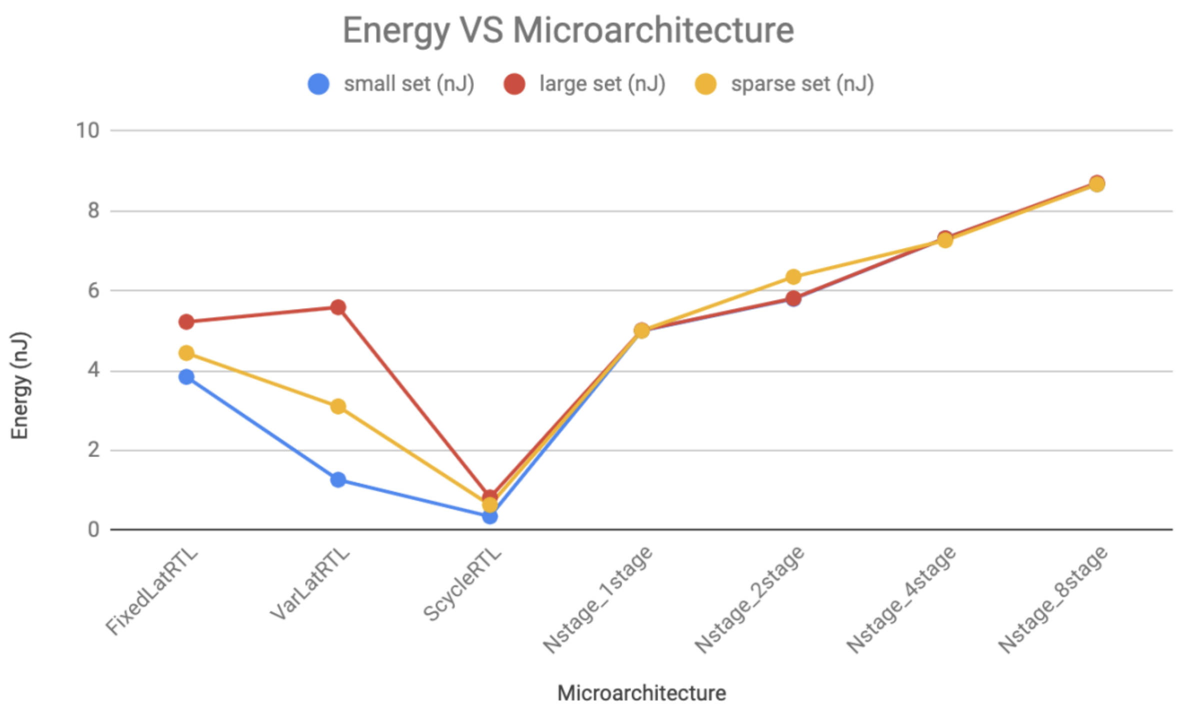


Chart 2: Energy VS Microarchitecture (all of them are in minimum constraint)

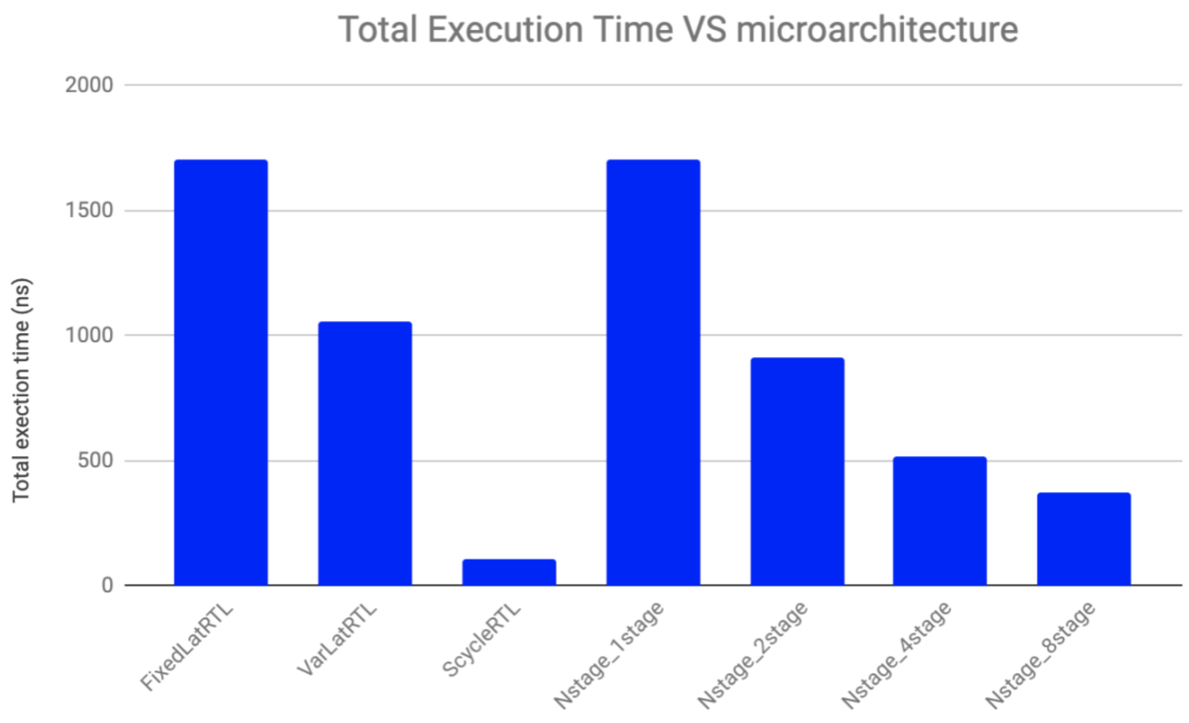


Chart 3: Execution time VS Microarchitecture (all of them are in minimum constraint)

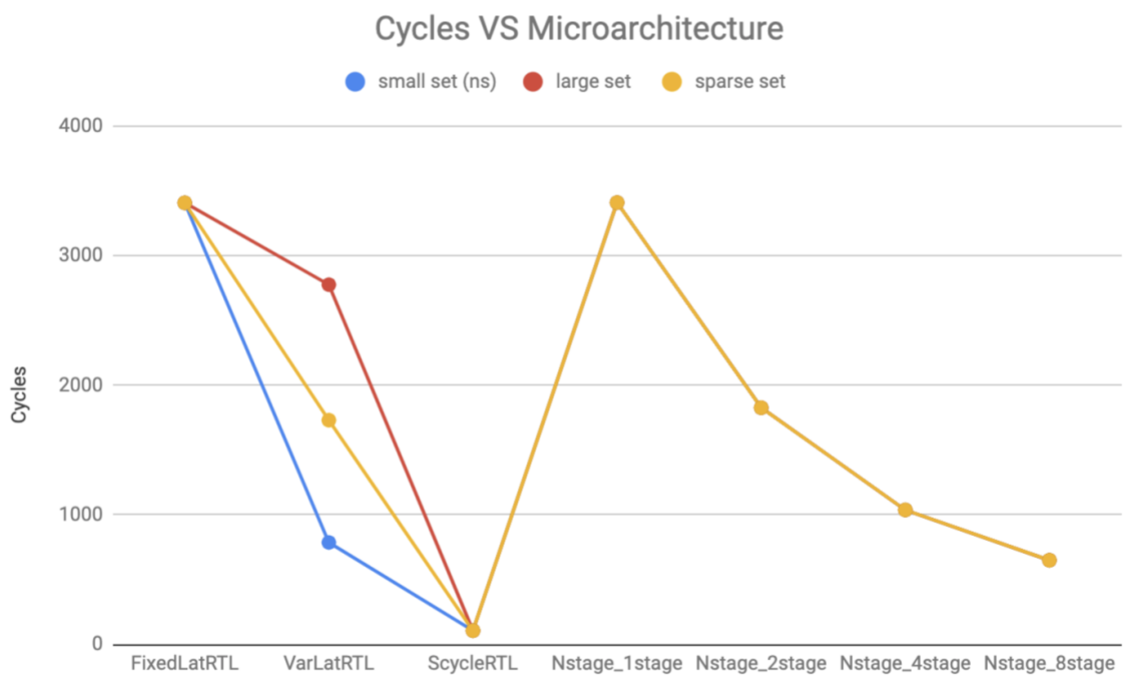


Chart 4: Cycles VS Microarchitecture (all of them are in minimum constraint)

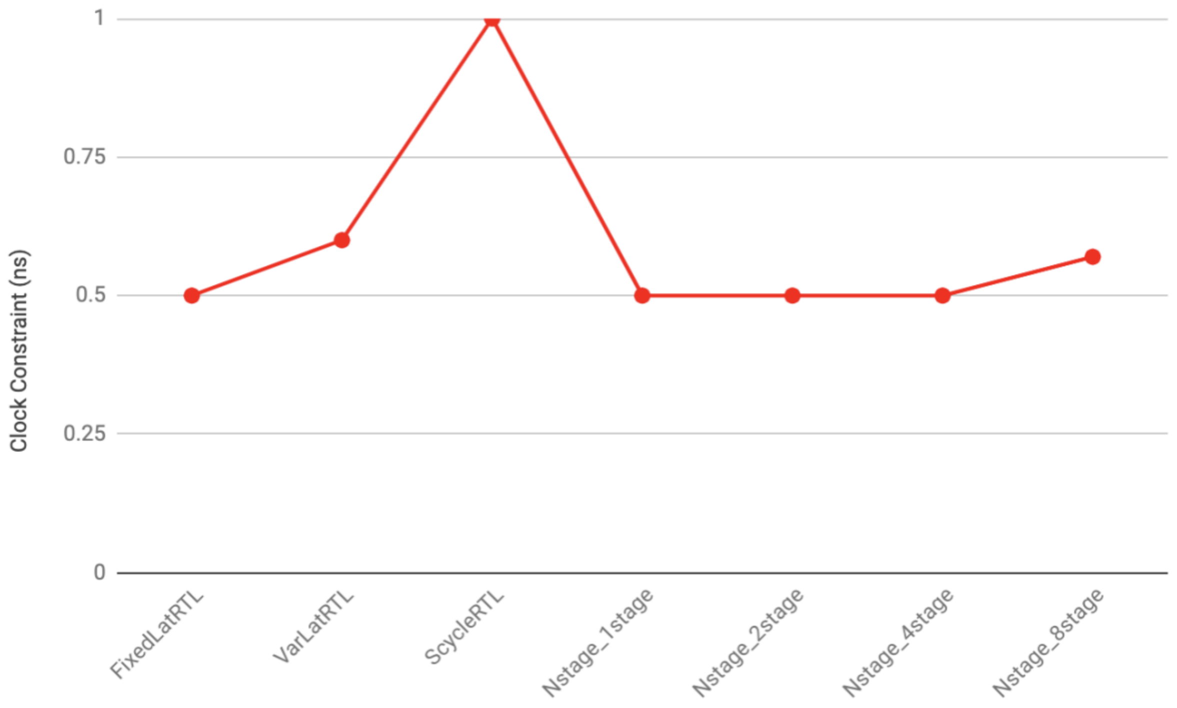


Chart 5: Slack VS Microarchitecture (all of them are in minimum constraint)

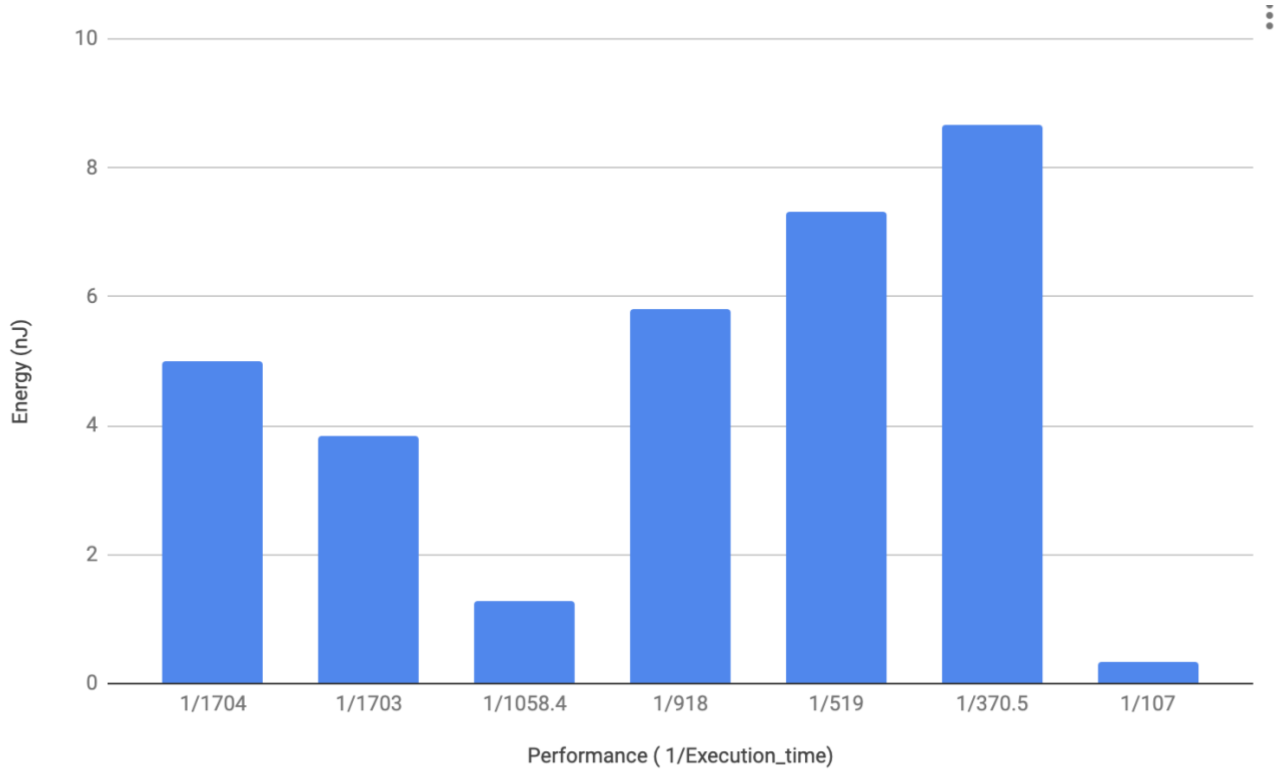


Chart 6: Energy VS Performance (We take the average for the three input data set (The sequence is, from left to right: 1 stage, fixed-latency, var-latency, 2 stage, 4 stage, 8stage, single cycle)

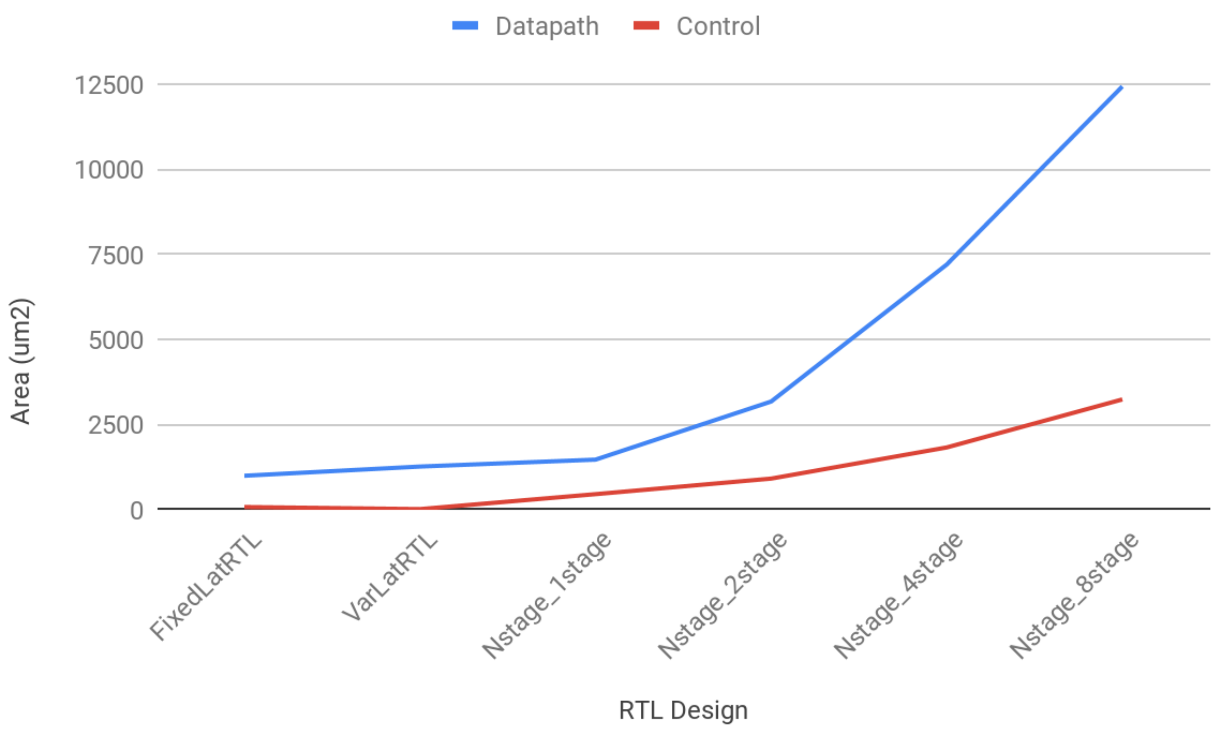


Chart 7: Component Area (datapath & control) VS RTL Design

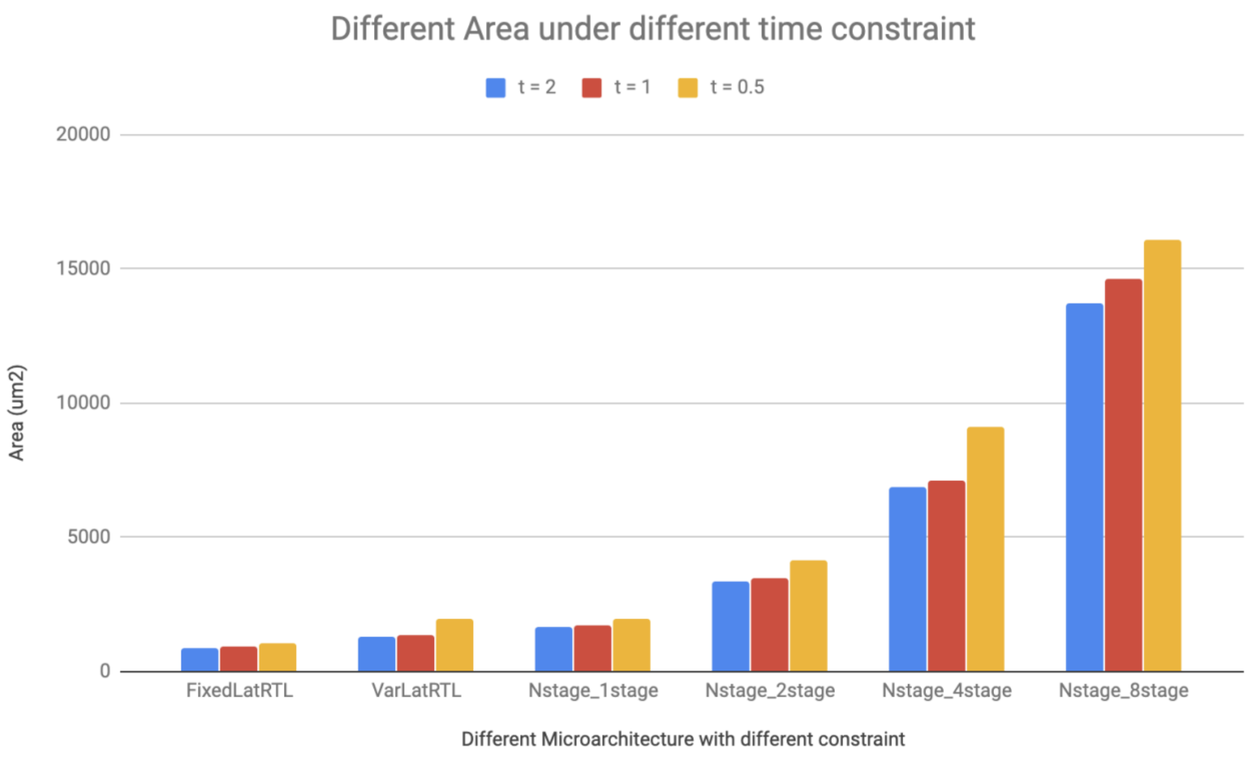
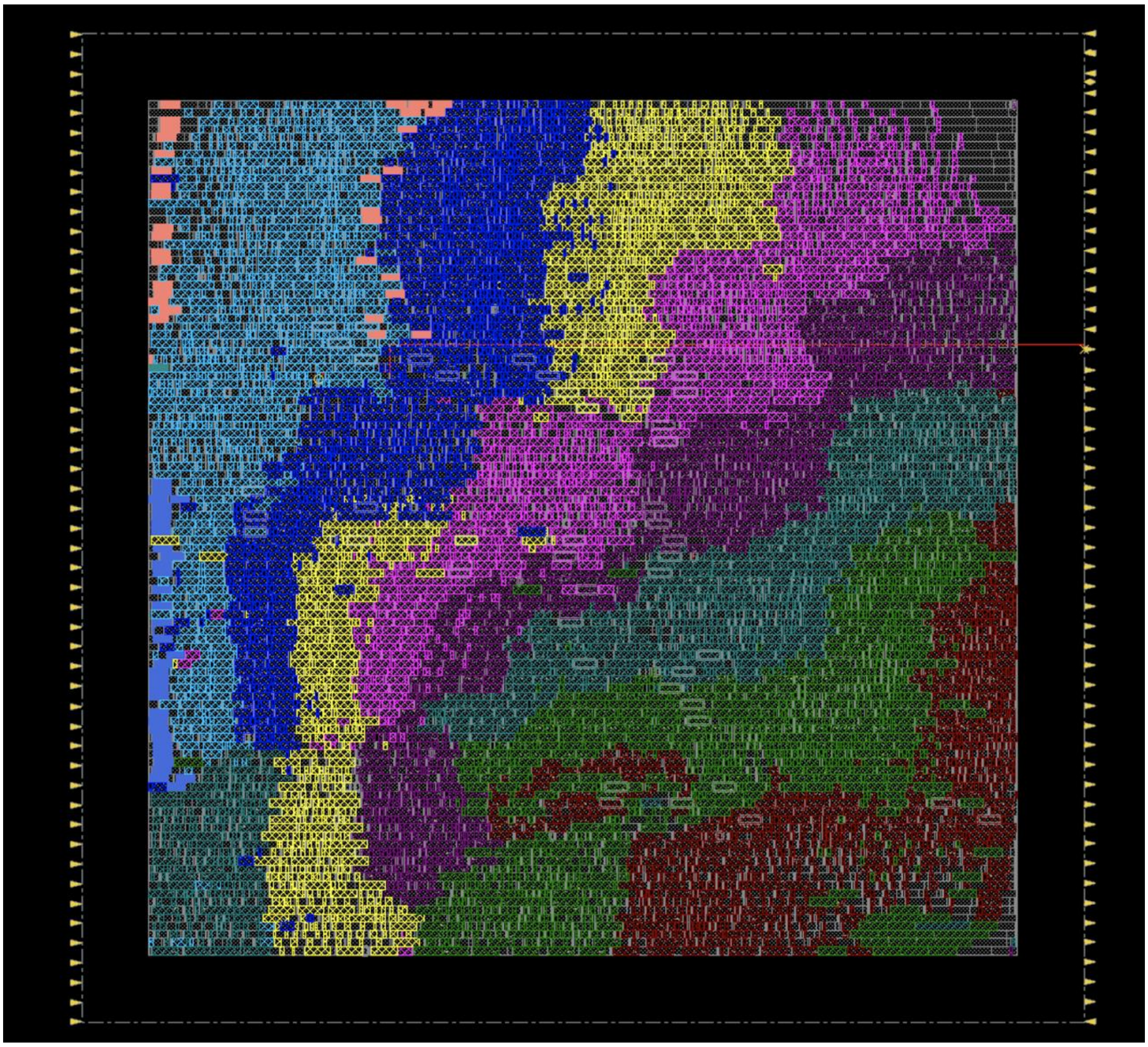
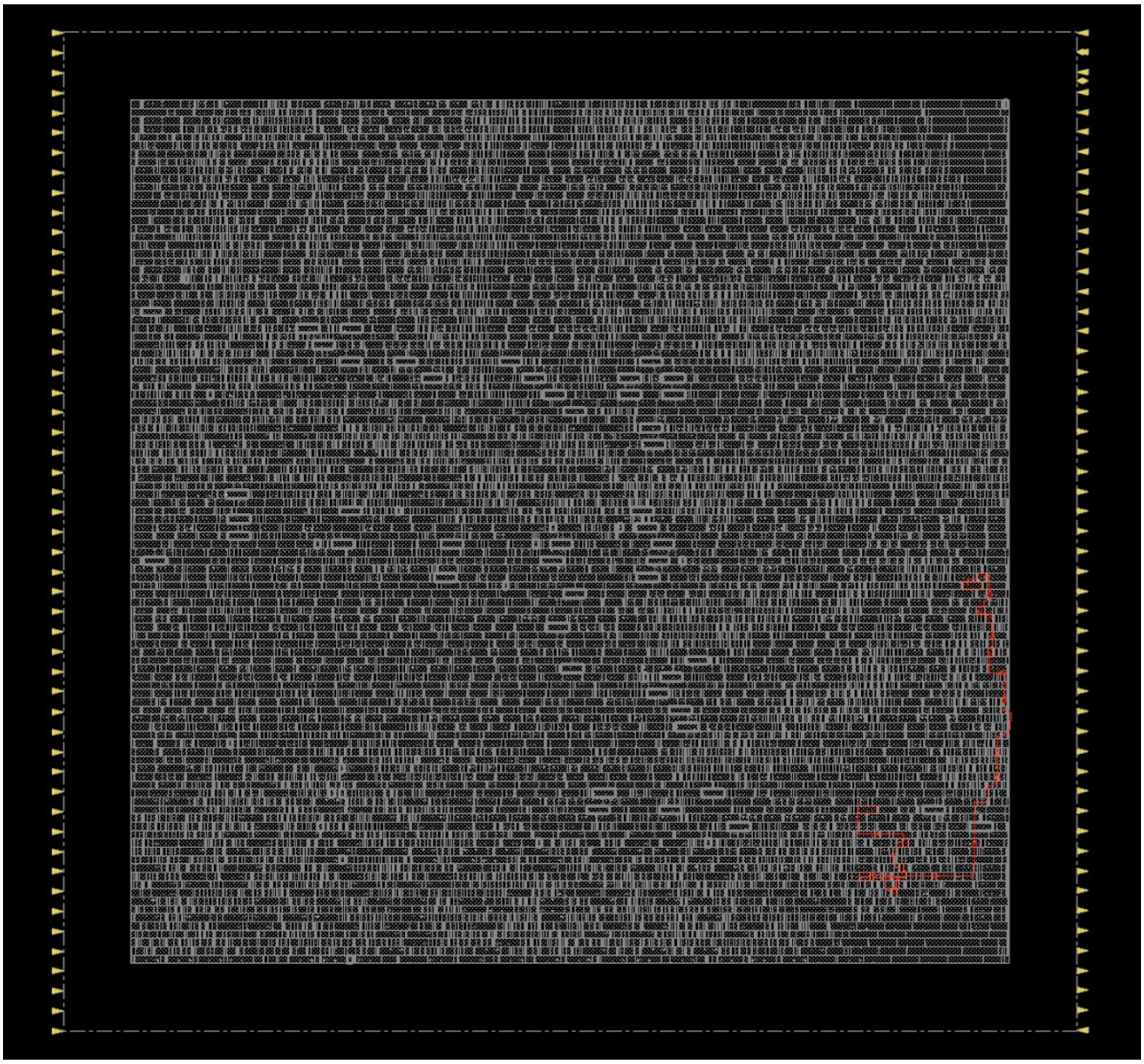


Chart 8: Different Area under different time constraint

Layout Figures:



Layout Figure 1: Amoeba plot of the multiplier with eight pipeline stage


Layout Figure 2: Critical Path from the layout from Cadence Innovus