Benjamin Hobbs | Frank Chang | Patrick Hanrahan
A113171489 | A08362337 | A53204304
WES 237C
10/11/2018

Project 1 : Optimizing FIR Filters in HLS

Introduction:

Without any optimizations, our initial performance is as shown in the table below. The goal of this assignment is to increase the performance of the FPGA using 6 different optimization methods.

FIR128 BASELINE					
Estimated Clock (ns)	Latency (clock cycles)	LUT	FF	DSP	BRAM (18K)
8.510	509	414	276	3	1

Figure 1 - Baseline performance.

Question 1-Variable Bitwidths: *The bitwidth of the variables provides a tradeoff between precision, resource usage, and performance. It is possible to specify the exact size of each variable to the tools. Make sure that you do not affect the results, i.e., the output still matches the golden output. Change the bitwidth of the variables inside the function. How does the bitwidth affect the performance? How many resources are used for the different data types? What is the minimum data size that you can use without losing accuracy (i.e., your results still match the golden output)?*

A: The smallest bitwidth you can use depends on the largest value in the output data. The output is the result of the multiply and add operations. If your largest output value is 500, you would need 9 bits, and make that 10 since we also need to consider the possibility of negative values. Larger bit widths mean more FPGA resources must be taken up to account for bit allocation. A 32-bit boolean operation may require more registers than an 8-bit boolean operation, and thus may cause a higher latency. Latency is more easily thought of as the number of registers between the input and the output.

For this assignment, our best bitwidth size was 16bit long. The default int variable type was already 16bit wide. Anything under 16bits long would truncate the data. Bitwidth sizes of 24 and 32 were implemented for comparison.

The testbench was also corrected to adjust for the type-casting required.

BITWIDTH						
Bitwidth	Estimated Clock (ns)	Latency (clock cycles)	LUT	FF	DSP	BRAM (18K)
16 bits	8.510	382	253	163	1	1
24 bits	6.5	382	301	203	1	1
32 bits	8.510	509	414	276	3	1

Figure 2 - Bitwidth performance.

Question 2-Pipelining: *This increases the throughput at the cost of additional resources (registers). The resource usage and performance can be varied by changing the initiation interval (II). Explicitly set the loop initiation interval (II) starting at 1 and increasing in increments of 1 cycle. How does increasing the II affect the loop latency? What are the trends? At some point setting the II to a larger value does not make sense. What is that value in this example? How would you calculate that value for a general for loop?*

A: With pipelining, you reduce cycle latency. Loop pipelining is overlapping iterations of a the for loop operations. This means the actions in the for loop will stagger and does not require one iteration to complete before starting. *Loop initiation interval (II) is the number of clock cycles before the next iteration can start.* Of course there is a point where more II value does not make sense because you may not need to wait, say 4 clock cycles before the next iteration finishes. Setting the value of II past 4 for our MAC results in a warning that indicates no pipelining is performed.

An interval (II) factor of 1 is the best. This will stagger the operations one after the other if possible. Finding the interval (II) factor for a general loop is dependant on what occurs in the loop. Perhaps there is an operation in the loop that prevents the next iteration to start, therefore the interval (II) factor can be greater. Since our case is very simple, an interval (II) factor of 1 is enough.

PIPELINING						
Pipeline Factor	Estimated Clock (ns)	Latency (clock cycles)	LUT	FF	DSP	BRAM (18K)
1	8.510	131	444	343	3	2
2	8.510	257	436	279	3	1
4	8.510	510	419	277	3	1

Figure 3 - Pipelining Performance

Question 3-Removing Conditional Statements: *If/else statements and other conditionals limit the possible parallelism and often require additional resources. If the code can be rewritten to remove them, it can make the resulting design smaller and faster. Compare designs with and without if/else condition. Is there a difference in performance and/or resource utilization? Does the presence of the conditional branch have any effect when the design is pipelined? If so, how and why?*

A: The code written for the baseline already removed the if/else statements outside of the for loop. The tests performed in this optimization will show how introducing if/else statements will increase our latency.

Pipelining without conditional “if” statements improves latency compared to pipelining with “if” statements. This makes sense because you’re able to wait less time (reduced latency) when you pipeline. Meaning you do not have to wait to check the conditionals before you can actually run the MAC. Overall, removing the “if/else” statements lowered the latency without a significant penalty on resources.

REMOVE CONDITIONALS							
if/else	Pipeline Factor	Est. Clock (ns)	Latency (clock cycles)	LUT	FF	DSP	BRAM (18K)
YES	0	8.510	257-513	456	388	3	1
YES	I	8.510	259	436	279	3	1
NO	I	8.510	131	444	343	3	1

Figure 4 - Remove Conditionals Performance

Question 4-Loop Partitioning: *Dividing the loop into two or more separate loops may allow for each of those loops to be executed in parallel. This may increase the performance and the resource usage. Compare your hardware designs before and after loop partitioning. What is the difference in performance? How do the number of resources change? Why?*

A: Loop Partitioning did not immediately increase our performance since all we did was separate it into two for loops instead of one. However, once we separated the loops and *then* unrolled the tapped delay line (TDL), and pipelined the MAC, we got increased performance. In terms of resources, loop partitioning alone increased resources by a small amount. However, when we *specialized* each for loop using unrolling and/or pipelining, our resource consumption increased dramatically.

LOOP PARTITIONING						
unroll / pipe?	Estimated Clock (ns)	Latency (clock cycles)	LUT	FF	DSP	BRAM (18K)
NO unroll / NO pipelining	8.510	764	463	285	3	1
unroll / pipelining	8.510	258	2181	4424	3	2

Figure 5 - Loop Partitioning Performance

Question 5-Memory Partitioning: *The storage of the arrays in memory plays an important role in area and performance. On one hand, you could put an array entirely in one memory. And this memory can have a different number of ports (e.g., one or two ports for FPGA block RAM). Or you can divide the array into two or more memories to increase the number of ports. Or you could instantiate each of the variables as its own register, which allows simultaneous access to all of the variables at every clock cycle, but has high resource usage. Compare the memory partitioning parameters: block, cyclic, and complete. What is the difference in performance and resource usage (particularly with respect to BRAMs and FFs)? Which one gives the best performance? Why?*

A: In our case, complete array partitioning had the best latency. Cyclic and Block were more or less the same. This makes sense because complete array partitioning stores each array index in its own register. In addition to being the best, compete array partitioning was of course the most resource intensive by a factor 4. In fact, complete used 0 BRAM, while cyclic and block both used 4 BRAM each. This makes sense since "complete" had the least amount of latency, and since BRAM's are very slow to access (limited access/write availability) To reiterate, the best memory partition was to increase our number of registers, while decreasing the amount of array block usage; registers can be easily accessed..

MEMORY PARTITIONING							
TYPE	Pipeline Factor	Est. Clock (ns)	LAT	LUT	FF	DSP	BRAM (18K)
CYCLIC	4	8.510	636	495	270	3	4
BLOCK	4	8.510	636	523	323	3	4
COMPLETE	NA	8.510	382	2124	4354	3	0

Figure 6 - Memory Partitioning Performance

Question 6-Best Design: *Combine any number of the above optimizations in order to get your best architecture. In what way is it the best? What optimizations did you use to obtain this result*?

A: We used a combination of pipelining, unrolling, and memory partitioning for the TDL and MAC loops to achieve our best design. Ultimately we were able to get down to a latency of 13 cycles - the least amount of cycles of any of our designs. Although this seems like an unrealistic result (Figure 11), it is our best simulated design based on latency and without regard for use of resources. The most resources were used to achieve the highest throughput. This design also had the best throughput, which is a function of latency. “HLS pipeline II” was implemented using a factor of 1 for both TDL and MAC. Factors of 2 and 4 were compared.

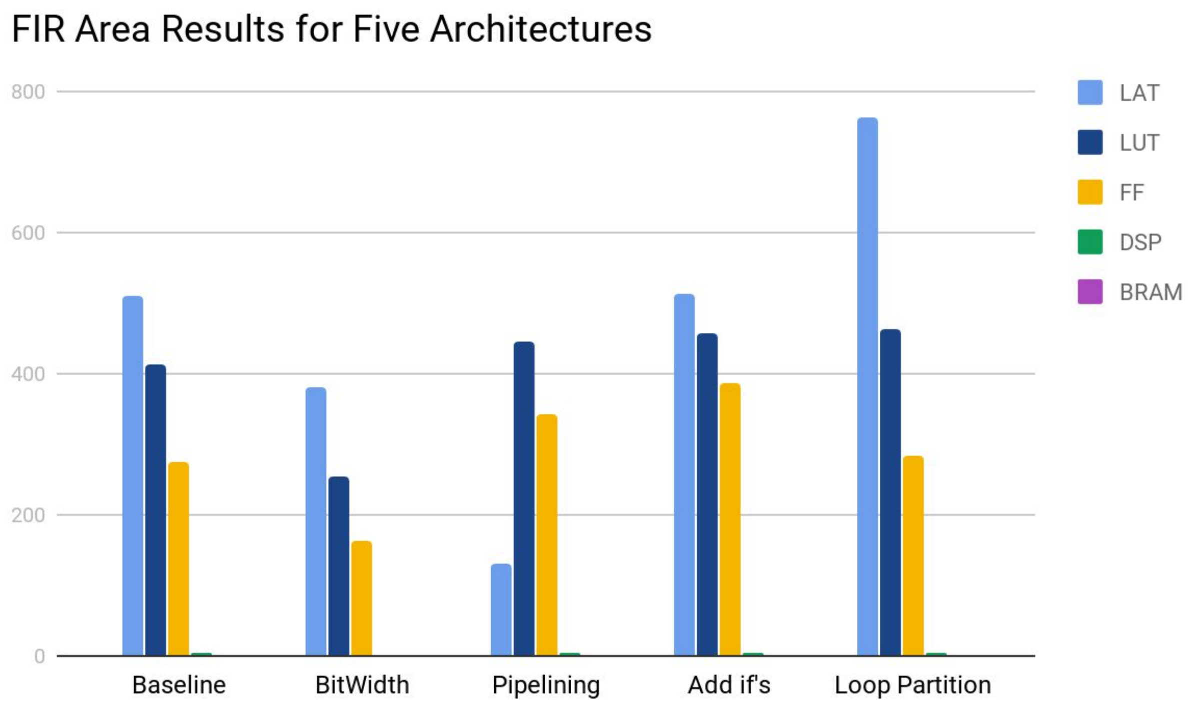


Figure 7: The five architectures without the Best implementation. This was done in an effort to make the data easier to visualize.

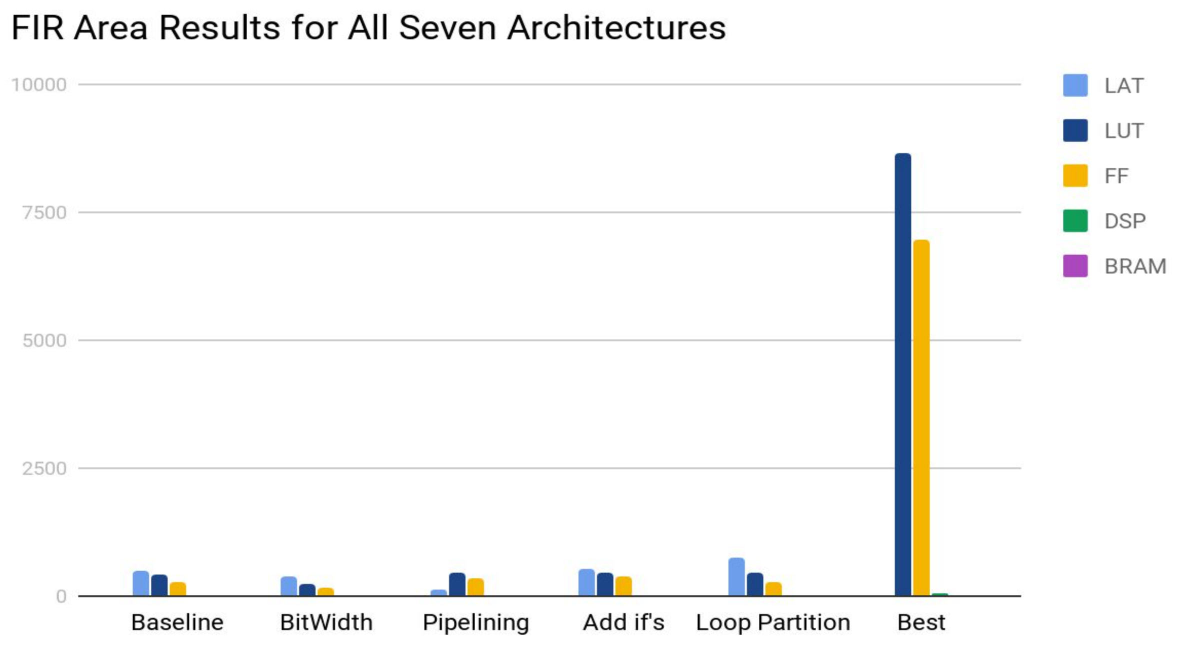


Figure 8: Area Architectures with the Best Architecture shown.

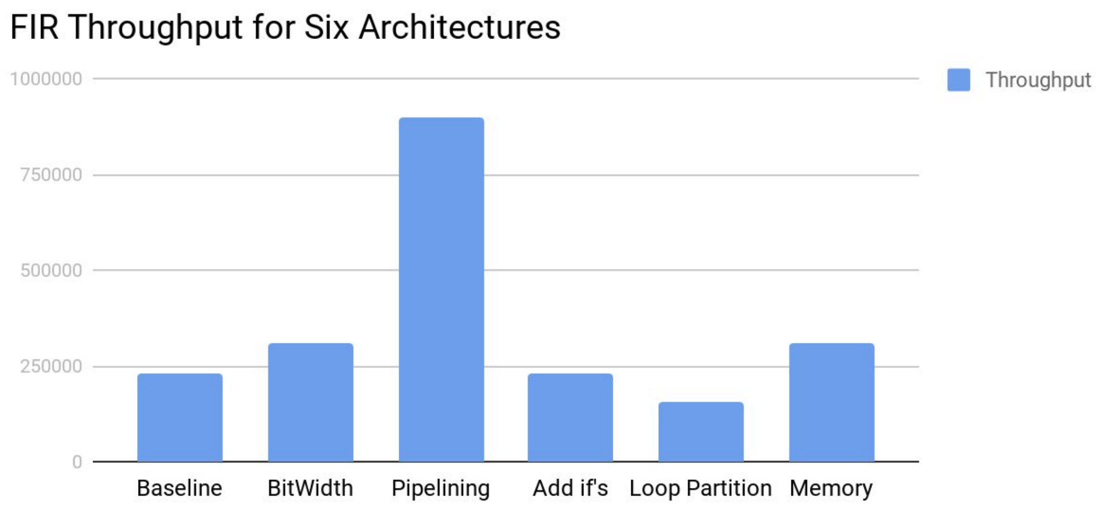


Figure 9: Throughput with "best" architecture taken out. Shows the Performance of each.

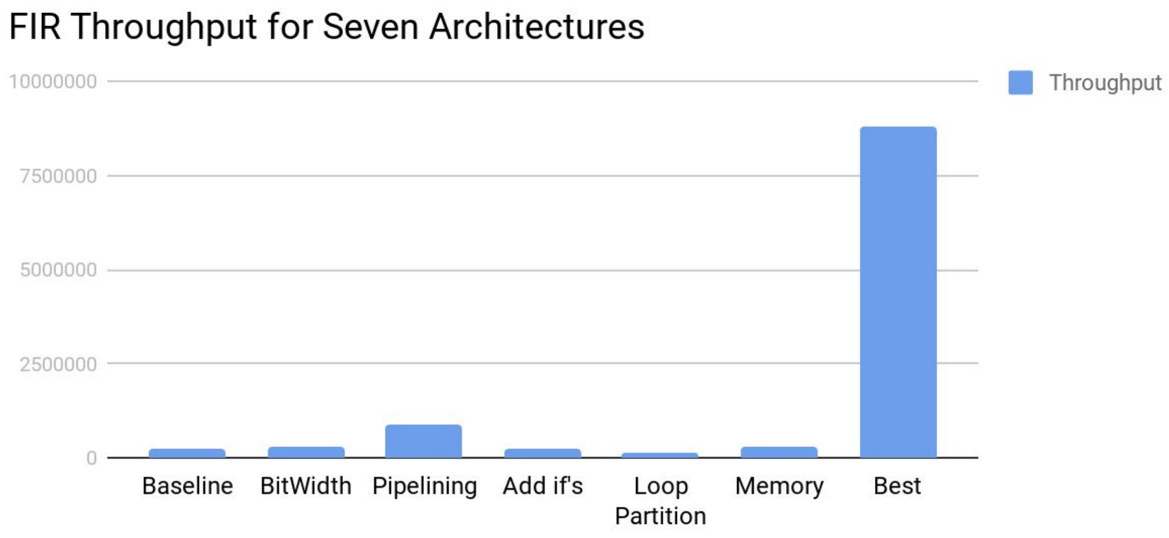
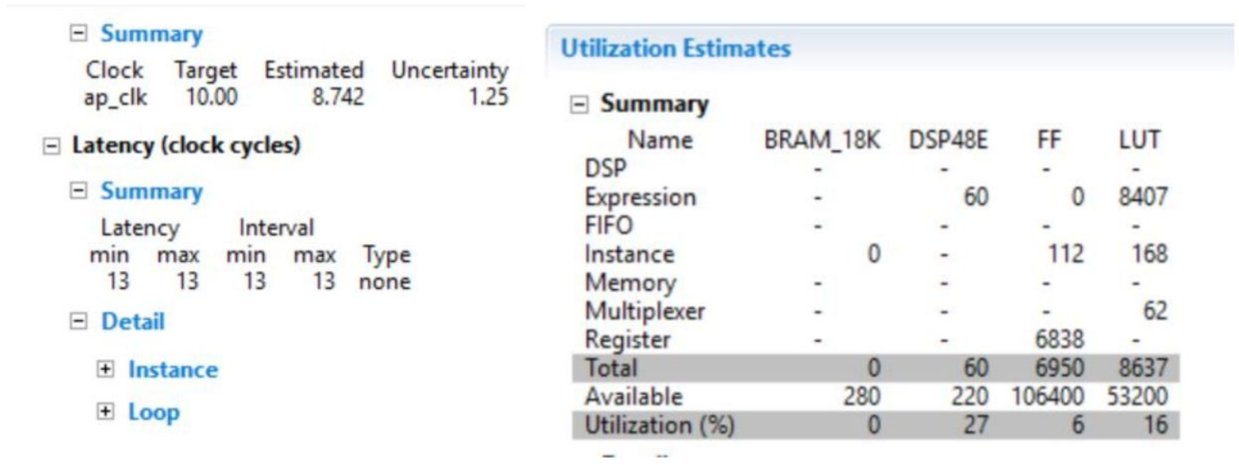


Figure 10: Throughput calculations showing performance with "best" included.



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.742	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
13	13	13	13	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	60	0	8407
FIFO	-	-	-	-
Instance	0	-	112	168
Memory	-	-	-	-
Multiplexer	-	-	-	62
Register	-	-	6838	-
Total	0	60	6950	8637
Available	280	220	106400	53200
Utilization (%)	0	27	6	16

Figure 11: “Best” performance displaying performance (Latency), and Resource consumption in FF’s and BRAM’s.