CSE237C Project 1

Henrik Larsson Hestnes

October 2021

1 Introduction

The goal of this project was to learn the basics of the Vitis HLS tool, by first making a functionally correct HLS design for an 11 tap FIR-filter, and then making and optimizing a 128 tap FIR-filter.

2 FIR11

The implementation of the functionally correct FIR11 can be found in the fir11_baseline folder.

3 FIR128

3.1 Baseline

In the baseline implementation, which is not optimized, we get the following performance wrt. performance and resource usage:
Performance

Estimated Clock Period	Latency	Throughput
6.912 ns	636 cycles	227 KHz

Resource Usage

BRAM_18K	DSP	FF	LUT	URAM
2	1	313	266	0

3.2 Question and optimization 1 - Variable Bitwidth

Performance

Estimated Clock Period	Latency	Throughput
6.508 ns	636 cycles	242 KHz

Resource Usage

BRAM_18K	DSP	FF	LUT	URAM
1	1	41	155	0

By changing the bitwidth of the variables inside the function body one can notice a big difference in the area used for the design. The main reason for this huge difference is the reduced size of the variables shift_reg and c. Since the maximum magnitude of the values in the coefficient array c is 11, every element of this array only needs to use 5 bits instead of the 32 bits they used before. The same goes for the shift_reg, where every element now only consists of 8 bits instead of 32. For this Bitwidth optimization to work we rely on that the values of x will never exceed a magnitude of 127, which might not always be a reasonable assumption, but it underlines how important the bitwidth is for resource usage. Also, the incremental variable i was changed to unsigned 7 bits, and the accumulating variable acc was changed to 16 bits. This reduction of the bitwidth is also based on this exact dataset, and might not work on another dataset. You can also see that the the performance gets slightly better with this optimization.

3.3 Question and optimization 2 - Pipelining

Performance

Loop initiation interval	Estimated Clock Period	Latency	Throughput
1	6.912 ns	134 cycles	1.08 MHz
2	6.912 ns	260 cycles	556 KHz
5, 6, 7,...	6.912 ns	639 cycles	226 KHz

Resource Usage

Loop initiation interval	BRAM_18K	DSP	FF	LUT	URAM
1	2	1	385	332	0
2	2	1	319	355	0
5, 6, 7,...	2	1	317	334	0

As can be seen from the table, the latency goes down and the throughput drastically improves with lower loop initiation interval, while the usage of resources goes up. As also can be seen, having a higher II than 5 makes no difference. That value resembles the amount of clock cycles one single iteration of the loop uses, and thus it makes no sense to have a higher amount of clock cycles per loop initiation than the actual amount of clock cycles one loop uses.

3.4 Question and optimization 3 - Removing Conditional Statements

Performance

Estimated Clock Period	Latency	Throughput
6.912 ns	641 cycles	226 KHz

Resource Usage

BRAM_18K	DSP	FF	LUT	URAM
2	1	379	275	0

Since the baseline implementation already was without if/else, this task added an if/else statement to the loop, to see how that affects the model. Performance wise there is not that much of a difference between the two implementation, although the baseline implementation performs slightly better. Resource wise the difference is bigger, where the implementation with if/else statement has to use 21% more flip-flops(FF), and some more lookup tables(LUT).
When trying to pipeline this implementation with II=1, you get an II Violation alert, and Vitis relaxes the II to 2 so we get 260 cycles of latency. The design also ends up using way more resources when trying to pipeline this implementation. The reason for the II Violation is because the loop has a resource that is needed in two different pipeline stages at the same time, but in different iterations. The result of this is that the performance becomes slower than the baseline performance, and the resource usage higher, which is no way near Pareto optimal. There is no doubt that the baseline implementation without if/else statements is the preferred implementation.

3.5 Question and optimization 4 - Loop Partitioning

Performance

	Estimated Clock Period	Latency	Throughput
Only loop partitioning	6.912 ns	896 cycles	161 KHz
Loop partitioning and pipelining	6.912 ns	267 cycles	542 KHz
Loop partitioning and total unrolling	6.923 ns	120 cycles	1.21 MHz
Loop partitioning, pipelining and unrolling factor of 16	7.396 ns	61 cycles	2.22 MHz

Resource Usage

	BRAM_18K	**DSP**	**FF**	**LUT**	**URAM**
Only loop partitioning	2	1	323	242	0
Loop partitioning and pipelining	3	1	400	359	0
Loop partitioning and total unrolling	2	1	8596	6102	0
Loop partitioning, pipelining and unrolling factor of 16	0	183	27086	18341	0

There is clearly an opportunity for loop partitioning in FIR filters. As can be seen from the tables above, loop partitioning itself just influence the design negatively. Combined with other techniques, it is clearly possible to improve the performance, but at the cost of significantly higher resource usage. The reason for this is that the unrolling requires the physical circuit to do more computations at the same time, and thus needs more components. Some of these configurations clearly uses a bit too much resources compared to performance.

3.6 Question and optimization 5 - Memory Partitioning

Performance

	Estimated Clock Period	**Latency**	**Throughput**
Complete memory partitioning and pipelining	6.912 ns	131 cycles	1.10 MHz
Block memory partitioning and pipelining	6.948 ns	134 cycles	1.07 MHz
Cyclic memory partitioning and pipelining	6.948 ns	260 cycles	554 KHz
Complete memory partitioning, specifying ram_2p and pipelining	6.921 ns	134 cycles	1.08 MHz

Resource Usage

	BRAM_18K	**DSP**	**FF**	**LUT**	**URAM**
Complete memory partitioning and pipelining	0	3	4403	2260	0
Block memory partitioning and pipelining	0	3	967	508	0
Cyclic memory partitioning and pipelining	0	3	899	617	0
Complete memory partitioning, specifying ram and pipelining	6	1	368	370	0

Clearly, the complete memory partitioning and block memory partitioning performs best wrt. throughput. It can also be seen that by specifying that the variables c and shift_reg should be ram/rom, you can replace a huge amount of the flip-flops and lookup tables and digital signal processors with BRAMs. The reason that these performs better is because it makes it possible to do several reads on the same cycle, thus higher throughput.

3.7 Optimization 6

Performance

Estimated Clock Period	**Latency**	**Throughput**
6.923 ns	21 cycles	6.88 MHz

Resource Usage

BRAM_18K	**DSP**	**FF**	**LUT**	**URAM**
0	20	12478	9104	0

By combining a complete array partitioning, loop partitioning, some variable bitwidth and unrolling both loops completely I got this throughput which is the highest I got. On the other hand, the resource usage is also by far the highest I got, which makes it unsuitable if the constraint on resources is strict.

3.8 Question 6 - Best Design

Performance

Estimated Clock Period	**Latency**	**Throughput**
6.912 ns	134 cycles	1.08 MHz

Resource Usage

BRAM_18K	**DSP**	**FF**	**LUT**	**URAM**
6	1	368	378	0

Through a lot of trial an error, the best design I found was this. This design achieves the 1MHz goal set in the assignment, as well as it is really sparse in its resource usage. As you can see from 3.7, I was able to produce a design with much greater throughput, but I felt the cost for resource usage was too high. Therefore I ended up at this design which I think comes close to Pareto optimal. To obtain this result I used memory partitioning and pipelining. If you have enough resources, it would be possible to get a throughput of 100MHz by designing a perfect circuit, but that would most likely be quite an overkill.