CSE 237C: Project 1 - FIR Filter

Will Zegers

October 4, 2016

1 Introduction

The following report details the design and optimization of a finite infinite response (FIR) filter using the Vivado HLS system design suite. Since the function of the FIR filter is relatively simplistic — implemented using a series of multiply-accumulate (MAC) operations, given a set of coefficients — the main focus of this project was to become acquainted with Vivado and general principles of HLS. Furthermore, this project also introduced the optimizations and parameter tweaking of which the tool is capable, and being able to compare the effect the optimizations have across multiple solutions.

The contents of this report deal mainly with the different categories and capabilities of the optimizations (in the form of inline `#pragma` directives as well as simple code refactoring), and how these optimizations compare across implementations of the FIR filter. In the section 3, I detail the different strategies I used to either increase throughput, decrease total area of the synthesized design, or find some "happy medium" between the two.

2 Baselines

Included in the project folder are two baseline files, one for an 11-tap FIR filter and one for a 128-tap FIR filter. The code in the contained *fir.cpp* files is not optimized and, admittedly, purposely written to make opportunities for optimization through refactoring alone. These files serve as a baseline, and help judge the degree of how much later optimizations are affecting the throughput and area of the filter.

3 Optimizations

3.a Optimizations 1 - 3: Clock period

The first three optimization explored the effect of clock period (which may also impact operation chaining). I experimented with clock frequencies from 3 to 12 ns in increments of 1 ns, and included only results from 3, 6, and 12 ns to highlight some of the key trends of differing period lengths. At shorter periods, throughput increased but also increased the number of flip-flops (FFs) and lookup tables (LUTs) needed in the design, effectively increasing the total area of the synthesized filter. Table 1 gives a summary of the throughput and area (in terms of components used) of these optimizations.

Table 1: Throughput and area of optimizations 1-3 (compared to baseline)

Clock Period (ns)	Estimated (ns)	Throughput (KHz)	FFs	LUTs
3	2.61	249.116	232	178
6	4.59	212.344	159	276
10 (baseline)	8.52	182.821	124	173
12	9.06	171.924	161	175

For periods above 12 ns, there was no noticeable benefit in terms of throughput and area; periods of less than 3 ns ended up with negative slack and were, therefore, obviously not feasible designs.

3.b Optimization 4: Bitwidth

Because many of the typedefs, such as `acc_t` and `data_t` were declared in the "untouchable" *fir.h* file, opportunities for optimizations through bitwidth adjustments were few. However, simply changing the variable `i` (used for iterating through the *for* loop) from `int` to `char` (i.e. from 32 to 8-bits) yielded an approximate 26% reduction in the number of FFs used in the design with no noticeable impact on throughput. While it is hard to tell how much further bitwidth optimizations would impact the number of FFs, the example in this optimization highlights that bitwidth optimization *can* help to reduce the area of the design without impacting throughput.

3.c Optimization 5: Code Hoisting

Optimization 5 removed the conditional `if/else` statement from the loop body, and moved the logic that handles the final iteration of the loop to the end of the `fir` function. While this did also help to reduce the amount of FFs and LUTs from the design (i.e. area reduction), it had a surprisingly small impact on the actual throughput. This ended up in reducing the number of cycles taken from 642 to 637 (a gain of only about 1.5KHz). Nonetheless, code hoisting was beneficial both in terms of area and throughput, so it is included in future optimizations; in fact, removing the extra condtional logic is likely to give better gains with other optimizations such as loop unrolling and loop fissure.

3.d Optimization 6: Loop Pipelining

Loop pipelining with an initiation interval (II) of 1 significantly increased the throughput while also reducing the number of LUTs required. Table 2 gives a summary of the differences between the baseline and optimization 6 throughput and area. Metrics from optimization 5 are also included in the table, since table 6 also used the code hoisting optimization discussed in the previous subsection.

Table 2: Throughput and area of optimizations 5 and 6 (compared to baseline)

Optimization	Throughput (KHz)	FFs	LUTs
Baseline	182.821	124	173
5	184.255	89	163
6	882.487	92	140

For all II parameters above 1, there was no note-worthy gain in performance or area; thus, `II=1` seemed to be the ideal optimization for pipelining the loop.

3.e Optimizations 7 & 8: Loop Unrolling

Optimizations 7 and 8 used code hoisting and loop pipelining described in the previous two subsections, and introduced two different methods of loop unrolling. 8 unrolled the for loop by a factor of 8, and also partitioned the `shift_reg` array as a cyclic type with 8 separate partitions. Optimization 7 did a full loop unroll, converting the `shift_reg` to individually accessible registers. Comparison of the two is shown in table 3.

Table 3: Throughput and area of optimizations 7 and 8 (compared to baseline)

Optimization	Throughput (MHz)	FFs	LUTs
Baseline	0.182	124	173
7	8.383	6300	4422
8	2.7947	1217	697

Clearly, loop unrolling significantly boosts throughput but at the cost of increasing in the number FFs and LUTs needed to support the newly parallizable design. While optimization 7 gives a three-fold gain in throughput, it also requires over five times as many FFs and over six times as many LUTs. Ultimately, the ”better” optimization would be based on design specification: Is the larger area tolerable given the higher or throughput, or does the specification want a compromise in throughput if it means a significantly reduced design area?

3.f Optimizations 9, 10, & 11: Loop Fission and Dependency Removal

Optimizations 9, 10, and 11 examine the effects of splitting a single loop into separate loops with more specific functionality; in particular, for a FIR filter the loop can be split into a *tapped delay line* (TDL) loop — that mainly handles shifting values in the `shift_reg` — and the *multiply-accumuate* — which multiplies each value in the `shift_reg` by its corresponding coefficient in *c*.

To further augment this architecture, which allows for more fined-grained optimizations on each loop, I also explicitly told the HLS tool to remove the *inter*dependency on the `acc` variable. This is justified by the fact that the multiply-accumulate operation can be done in any order and does not need to adhere to a sequence specified in the code; telling the tool to remove dependencies on a variable between loop iterations, thus allowing for greater parallelization. Table 4 compares the results of optimizations 9, 10, and 11 with the baseline (as well as each other). I also included optimizations 7 and 8 to compare to 9 and 11, respectively, respectively, which use the same factors for loop unrolling. Optimization 10 examines a middle ground between the two, using loop unrolling / array partitioning factors of 16.

Table 4: Throughput and area of optimizations 7 and 8 (compared to baseline)

Optimization	Throughput (MHz)	FFs	LUTs
Baseline	0.182	124	173
7	8.383	6300	4422
11	3.0897	7049	4162
10	3.089	1889	1277
8	2.7947	1217	697
9	1.956	993	677

For complete loop unrolling (architectures 7 and 11), loop fissure and dependency removal was actually detrimental to both throughput and area, reducing the throughput by over half and increasing the FF requirements by about 15%.
For the more modest loop unrolling with factor 8, the split-loop architecture throughput dropped but at the factor of about one-third, but came with the gain of a smaller number of FFs and LUTs used. Once gain, the "right" architecture is contigent on specification: Is the reduced area worth the hit to throughput? The same argument can be made when considering architecture 10 as well, which used an unrolling / partitioning factor of 16, yielding a throughput of about 3MHz at the cost of 50% more FFs and 2x the LUTs.

3.g Optimization 12: Resource Specification

Finally, architecture 12 makes a few other tweaks to the optimizations of the baseline and previous architectures. First, it explicitly makes the `shift_reg` a 2-port BRAM divded into eight partitions, allowing for either two simultaneous reads or a simultaneous read write on each partition, permitting greater parallelization. Second, I do something similar for the `c` coefficient array, making it a 2-port ROM (since it is never read, though I did not notice any major difference between using RAM vs ROM), and also partitioning it into *four* partitions.

Why four and not eight partitions for the `c` array? Synthesizing the design with equal partitions for `c` and `shift_reg` caused errors, but by reducing the partitions of `c` to *half* that of `shift_reg` made it possible to synthesize the design. Results comparing architectures 8, 9, and 12 (wherein `shift_reg` is partitioned into eight parts) are shown in table 5.

Table 5: Throughput and area of optimizations 8, 9, and 12

Optimization	Throughput (MHz)	FFs	LUTs
8	2.7947	1217	697
9	1.956	993	677
12	1.956	441	607

Architecture 12 sees no gain in throughput from its optimizations, but a drastic drop in the number of FFs required and roughly 10% drop in the number of LUTs. This, did, however, also increase the number of BRAM18_Ks and DSP48Es used in the design.

4 Final Architecture

For my final architecture, I elected to pick a design with a compromise between throughput and area, using the optimizations discussed in the previous section. The architecture was based on the optimizations discussed in section 3.g, along with all previous architectures on which it was built. However, I took note of the additional gain from both increasing the factor of loop unrolling / array partitioning to 16, as well as reducing the clock period to 6 ns. Both these parameters increased the area of the design, but gave an acceptable throughput. Table 6 shows a final comparison between the baseline and my chosen "best" architecture

Table 6: Comparison between chosen "best" architecture and baseline

Optimization	Throughput (MHz)	BRAM_18Ks	DSP48Es	FFs	LUTs
Baseline	0.182	1	2	124	173
Best	4.766	24	30	822	1230