CSE237C: Lab 1 FIR Filter Design

Weihong Xu
wexu@ucsd.edu

October 13, 2020

1 Introduction

This report gives the design and optimization processes for FIR filter with different numbers of taps using Vivado HLS design suite. The goal is to utilize different types of HLS optimizations strategies to develop high-performance architectures through making different tradeoffs in performance and resource consumption. The report is organized as follows:

- Section 2 presents a functionally correct baseline design of an 11-tap FIR filter on Vivado HLS.
- I take the advantages of different optimization strategies to optimize the baseline design of FIR128 design in Section 3. These optimization schemes include: bitwidth reduction, pipelining, code hoisting, loop unrolling, and memory partitioning. The performance with associated optimization method is shown.
- The PYNQ demo of the 11-tap FIR is presented in Section 4, which verifies the functionality of developed FIR design.

2 FIR11 Baseline

The folder ***fir11_ baseline*** includes a functionally correct design of the 11-tap filter, FIR11. The core function in *fir.cpp* is given as follows:

```
void fir (
  data_t *y,
  data_t x
  )
{
        coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0,53};


        // Write your code here
        static data_t reg_x[N];
        acc_t ACC = 0;
        TDL:
        for (int i = N-1; i >= 0; i--)
        {
                if(i==0)
                        reg_x[0] = x;
                else
                        reg_x[i] = reg_x[i-1];
        }


        MAC:
        for (int i = N-1; i >= 0; i--)
        {
                ACC += c[i]*reg_x[i];
        }

        *y = ACC;
}
```

The functions are verified using the test bench. Given the clock period constraint of 10 ns, the synthesis results are shown as follows:

Table 1: Synthesis results and throughput of FIR11 baseline.

Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
	Latency	Interval	BRAM	DSP48E	FF	LUT	
8.510	68	68	0	2	235	233	1.728

3 FIR128 Design and Optimization

3.1 FIR128 Baseline

I use the same code of FIR11 baseline to implement the FIR128 baseline design. The code is in folder ***fir128_ baseline***. The synthesis results are shown in Table 2. The input interval and processing latency become longer since the taps increase to 128. The longer interval leads to a lower throughput of 0.152 MHz.

Table 2: Synthesis results and throughput of FIR128 baseline.

Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
	Latency	Interval	BRAM	DSP48E	FF	LUT	
8.510	770	770	1	2	170	235	0.152

3.2 Question 1 - Variable Bitwidths

3.2.1 Solution

The FIR128 baseline design adopts a data type of 32-bit **INT**. However, the input data and tap coefficients are integers and lie in a certain range, which can be quantized using lower bitwidths. Some parts of 32-bit variable bitwidths are redundant, thus providing tradeoff opportunity between resource usage and performance.

The goal is to find the lowest bitwidths that introduce no precision loss for different variables. The optimal bitwidth search process is done in two steps. First, find the maximum absolute value *max_value* of a given variable. Secondly, the optimal bitwidth is given by:

$$\text{Bit width} = \lceil \log_2 max\_value \rceil + 1. \tag{1}$$

The bit widths for coefficients, input data, and output accumulative results are assigned respectively using the following bit width parameters in Table 3.

Table 3: Variable bitwidths for FIR128 design.

Variable	Coefficient	Input	Acc output
Bitwidth	5	8	16

The bitwidth minimization is done through including *ap_int.h* library and defining the data width in the header file *fir.h*. The code is given as follows:

```
#include "ap_int.h"

typedef ap_int<5>          coef_t;
typedef ap_int<8>          data_t;
typedef ap_int<16>         acc_t;
```

Folder ***fir128_optimized1*** includes the code of the FIR128 design with given bitwidths in Table 3. It passes the simulation and matches the ground truth results. The synthesis results and corresponding performance for baseline and optimized design are given in Table 4. The design presented in this section is called **Opt. 1**.

Table 4: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239

3.2.2 Analysis

- *How does the bitwidth affect the performance?*
  The reduction of bitwidth simplifies the complexity of each multiplication and accumulation (MAC) operation, thus reducing the needed clock cycles to perform MAC

calculation from 4 to 3. As a result, the latency and interval decline by 16.6% compared to the baseline while the clock period is shortened to 6.508 ns. The throughput increases by 57% and becomes 0.239 MHz.

- *How does it affect the resource usage?*
  The minimization of bitwidths brings out the saving on resource usage. Compared to the baseline, the required number of DSP68E is reduced by 50%. The required FF and LUT save about 42.9% and 26.4%, respectively. Moreover, BRAM memory size reduces from $4,736$ bits to $1,664$ bits.

- *What is the minimum data size that you can use without losing accuracy?*
  The minimum data size and bitwidths are given in Table 3.

3.3 Question 2 - Pipelining

3.3.1 Solution

I use the HLS pragma #pragma HLS pipeline II=<INT> to implement pipelining for the MAC loop. The detailed code is modified as follows:

```
TDL:
for (int i = N-1; i >= 0; i--)
{
        if(i==0)
                reg_x[0] = x;
        else
                reg_x[i] = reg_x[i-1];
}

MAC:
for (int i = N-1; i >= 0; i--)
{
        #pragma HLS pipeline II=1
        ACC += c[i]*reg_x[i];
}
```

I tried three different loop initiation interval values from 1 to 3. The code is in folder ***fir128_optimized2***. The synthesis results and performance comparison are listed in Table 5. The design presented in this section is called **Opt. 2**.

Table 5: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239
Opt. 2, II=1	6.508	388	388	1	1	93	190	0.396
Opt. 2, II=2	6.508	515	515	1	1	100	193	0.298
Opt. 2, II=3	6.508	643	643	1	1	90	176	0.239

3.3.2 Analysis

- *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?*
  The increasing of II increases the latency of MAC loop. The TDL loop is mapped to single-port BRAM module which costs fixed 2 cycles to read and write data. Thus, pipelining does not take effect on TDL loop.

- *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?*
  As a result, the overall latency and interval also increase with larger II. The II value does not make sense when II >= 3. For general cases, the II values that bring out performance gain should be less than the latency of optimized loop. Finally, the design with **II** = 1 delivers the best performance without adding to much resource usage.

3.4 Question 3 - Removing Conditional Statements

One conditional statement exists in the TDL loop. This conditional branch is removed by rewriting the code as follows:

```
TDL:
for (int i = N-1; i > 0; i--)
{
        reg_x[i] = reg_x[i-1];
}
reg_x[0] = x;
```

The other optimization strategy is loop fission, where the TDL and MAC loops are merged into single loop as shown below:

```
for (int i = N-1; i > 0; i--)
{
        #pragma HLS pipeline II=1
        reg_x[i] = reg_x[i-1];
        ACC += c[i]*reg_x[i];
}

reg_x[0] = x;
ACC += c[0]*reg_x[0];
```

The code using conditional statement removal and loop fission optimizations is given in folder ***fir128_optimized3***. The pipelining II is set to 1. The synthesis results and performance comparison are listed in Table 6. The design that removes conditional statements is called **Opt. 3, remove**. **Opt. 3, fission** denotes the design that merges two loops into one on the basis of design **Opt. 3, remove**.

Table 6: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239
Opt. 2, II=1	6.508	388	388	1	1	93	190	0.396
Opt. 2, II=2	6.508	515	515	1	1	100	193	0.298
Opt. 2, II=3	6.508	643	643	1	1	90	176	0.239
Opt. 3, remove	6.508	386	386	1	1	66	175	0.398
Opt. 3, fission	8.254	129	129	1	1	48	160	0.946

3.4.1 Analysis

- *Is there a difference in performance and/or resource utilization?*
  From Table 6, the design **Opt. 3, remove** with conditional statement removal saves 29% of FF and 17.9% of LUT compared to design **Opt. 2, II=1**. Besides, two clock cycles are saved. The throughput also slightly increases. The gain comes from the fact that removing the conditional statement saves both cycles and logic used to handle the conditional branch.

  After merging two loops, the clock cycles and resource usage of design **Opt. 3, fission** further decrease at the expense of longer estimated clock period 8.254 ns. The latency and interval drop from 386 to 129, improving the throughput by a speedup factor of 2.38× to 0.946 MHz.

- *Does the presence of the conditional branch have any effect when the design is pipelined? If so, how and why?*
  Yes, the conditional branch has impact on the latency and interval of pipelined design. Compared to the design **Opt.3, remove** without conditional statements, **Opt.2, II=1** requires two more clock cycles. This is because the conditional branch judgment needs additional cycles.

3.5 Question 4 - Loop Partitioning

I choose to use pragma #pragma HLS unroll factor=<INT> to unroll the loop, allowing more loops to be executed in parallel. The following code shows the modified loop partitioning.

```
for (int i = N-1; i > 0; i--)
{
        #pragma HLS pipeline II=1
        #pragma HLS unroll factor=<INT>

        reg_x[i] = reg_x[i-1];
        ACC += c[i]*reg_x[i];
}

reg_x[0] = x;
ACC += c[0]*reg_x[0];
```

The code with loop unrolling is given in folder ***fir128_optimized4***. I tested three different unrolling factors, including 2, 4, and 127. The synthesis results and performance comparison are listed in Table 7.

Table 7: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239
Opt. 2, II=1	6.508	388	388	1	1	93	190	0.396
Opt. 2, II=2	6.508	515	515	1	1	100	193	0.298
Opt. 2, II=3	6.508	643	643	1	1	90	176	0.239
Opt. 3, remove	6.508	386	386	1	1	66	175	0.398
Opt. 3, fission	8.254	129	129	1	1	48	160	0.946
Opt. 4, factor= 2	6.508	132	132	1	2	123	241	1.164
Opt. 4, factor= 4	9.634	133	133	1	4	230	366	0.780
Opt. 4, factor= 127	9.871	127	127	1	24	1595	5655	0.798

3.5.1 Analysis

- *Compare your hardware designs before and after loop partitioning. What is the difference in performance?*
  After unrolling the loop, the latency and interval do not decline significantly. When unroll factor = 2 or = 4, a slight latency increasement is observed. The performance of unrolling factor = 2 is 1.164 MHz. The bottleneck in this case is not the amount of computing units but the bandwidth of memory. The single-port BRAM can only provide one read or write operation at each clock cycle.

- *How do the number of resources change? Why?*
  The BRAM usage stays the same after unrolling the loop. But the arithmetic resources increase almost linearly with the unrolling factors. This is because the parallel execution of computation replicates multiple copies of MAC units.

3.6 Question 5 - Memory Partitioning

To increase the memory bandwidth, I use pragma #pragma HLS array_partition variable= to perform memory partitioning. The following code shows the complete memory partitioning.

```
        #pragma HLS array_partition variable=c complete
        #pragma HLS array_partition variable=reg_x complete


        acc_t ACC = 0;
```

```
for (int i = N-1; i > 0; i--)
{
        #pragma HLS pipeline II=1
        #pragma HLS unroll
        reg_x[i] = reg_x[i-1];
        ACC += c[i]*reg_x[i];
}

reg_x[0] = x;
ACC += c[0]*reg_x[0];
```

On the basis of design **Opt.4, factor=127**, the code with memory partitioning is given in folder ***fir128_ optimized5***. The synthesis results and performance comparison are listed in Table 8, where **Opt. 5** represents the design with complete memory partitioning.

Table 8: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239
Opt. 2, II=1	6.508	388	388	1	1	93	190	0.396
Opt. 2, II=2	6.508	515	515	1	1	100	193	0.298
Opt. 2, II=3	6.508	643	643	1	1	90	176	0.239
Opt. 3, remove	6.508	386	386	1	1	66	175	0.398
Opt. 3, fission	8.254	129	129	1	1	48	160	0.946
Opt. 4, factor= 2	6.508	132	132	1	2	123	241	1.164
Opt. 4, factor= 4	9.634	133	133	1	4	230	366	0.780
Opt. 4, factor= 127	9.871	127	127	1	24	1595	5655	0.798
Opt. 5	9.871	2	2	0	24	1479	3906	50.653

3.6.1 Analysis

- *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?*
  I tested three memory partitioning types, including cyclic, block, and complete. If the resouce of memory is not explicitly specified as BRAMs, these three types of memory partioning deliver the same performance and resource usage.

  The complete memory partioning delivers the highest throughput 50.653 MHz since it can provide very high memory bandwidth that accommodates the unrolled MAC operations.

3.7 Best Design

To yield the best performance, I integrate the aforementioned optimization strategies. However, the previous designs only achieve a loop pipelining. To further improve the performance, the pragma #pragma HLS pipeline is moved the *fir* function body to achieve the function-level pipelining as following codes.

```
        #pragma HLS array_partition variable=c complete
        #pragma HLS array_partition variable=reg_x complete

        #pragma HLS pipeline II=1

        acc_t ACC = 0;

        for (int i = N-1; i > 0; i--)
        {
                #pragma HLS unroll
                reg_x[i] = reg_x[i-1];
                ACC += c[i]*reg_x[i];
        }

        reg_x[0] = x;
        ACC += c[0]*reg_x[0];
```

The code for the best design is given in folder ***fir128_ best***. The synthesis results and performance comparison are listed in Table 9. As shown in the table, the best design delivers a throughput of 101.3 MHz and is able to output a filtered result every cycle.

Table 9: Synthesis results and throughput comparison for various FIR128 designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.510	770	770	1	2	170	235	0.152
Opt. 1	6.508	642	642	1	1	97	173	0.239
Opt. 2, II=1	6.508	388	388	1	1	93	190	0.396
Opt. 2, II=2	6.508	515	515	1	1	100	193	0.298
Opt. 2, II=3	6.508	643	643	1	1	90	176	0.239
Opt. 3, remove	6.508	386	386	1	1	66	175	0.398
Opt. 3, fission	8.254	129	129	1	1	48	160	0.946
Opt. 4, factor= 2	6.508	132	132	1	2	123	241	1.164
Opt. 4, factor= 4	9.634	133	133	1	4	230	366	0.780
Opt. 4, factor= 127	9.871	127	127	1	24	1595	5655	0.798
Opt. 5	9.871	2	2	0	24	1479	3906	50.653
Best	9.871	2	1	0	24	1479	3889	101.3

Figure 1 and Figure 2 illustrate the area results and throughput comparison for different FIR128 designs.

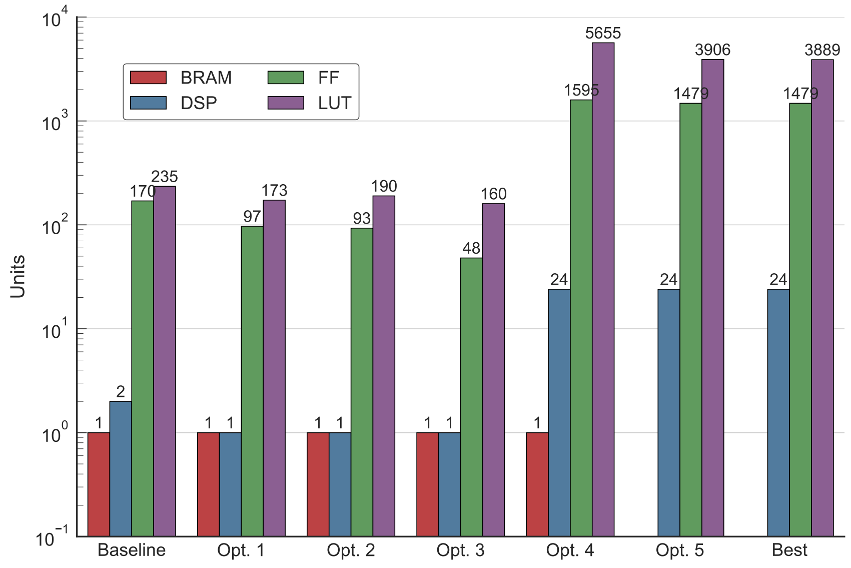

Figure 1: FIR128 area results of different architectures.

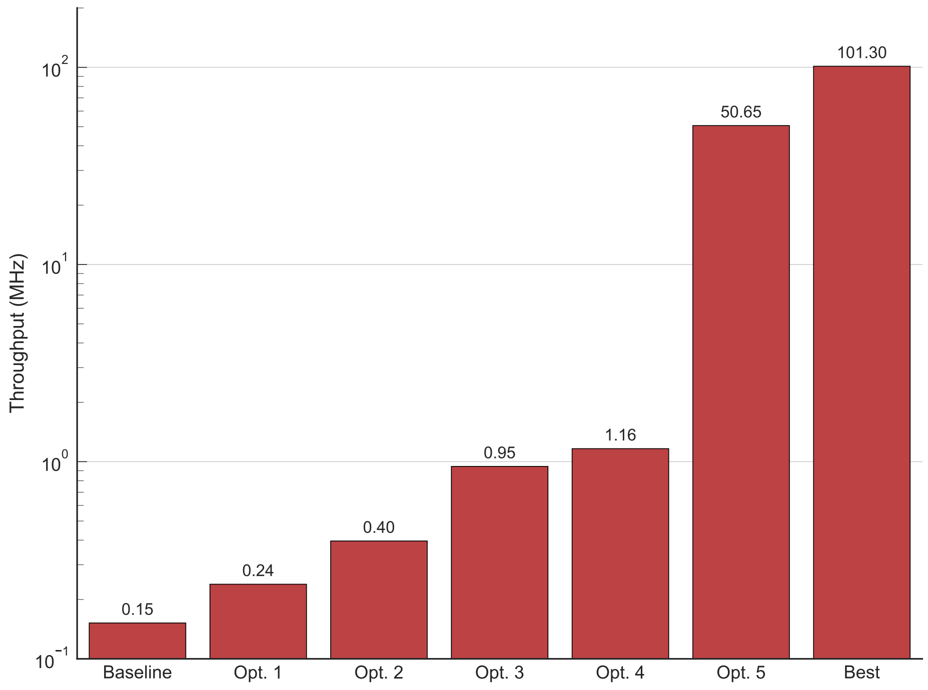

Figure 2: FIR128 throughput of different architectures.

4 PYNQ Deom of FIR11

The demo on PYNQ platform is shown in this section. Figure 3 and Figure 4 show the original and filtered results, respectively.

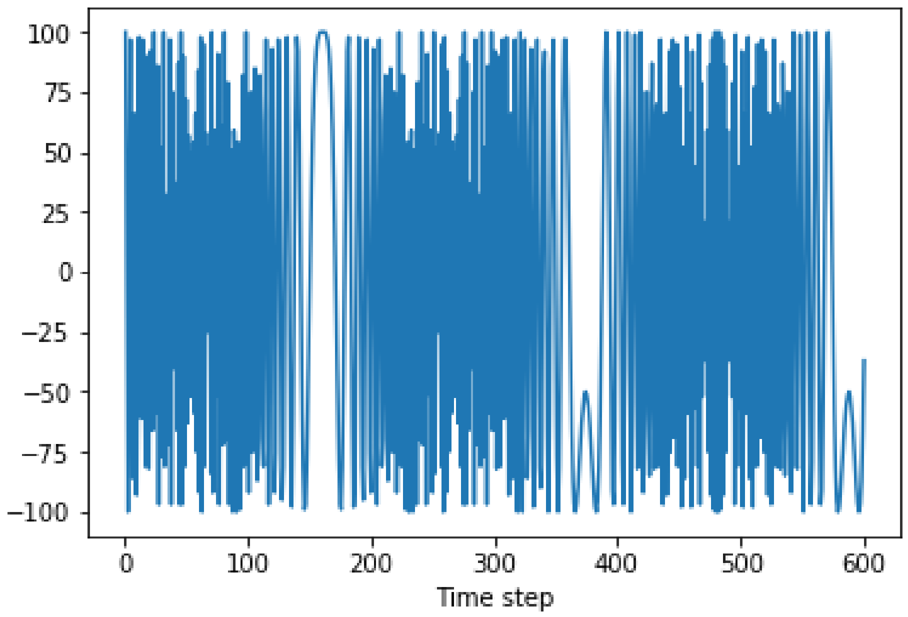

Figure 3: Illustration of original input data.

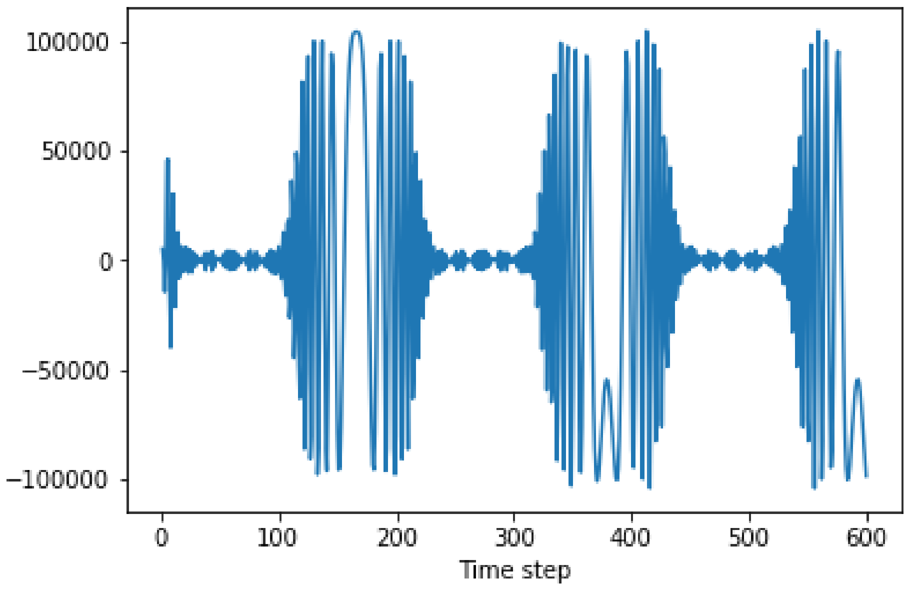

Figure 4: Illustration of results filtered by FIR11.