Experiment 1
Vivado HLS synthesis report for 11-tap FIR Filter

Kaushik Balaji M S *M.Tech in System-on-chip Design*
Indian Institute of Technology, Palakkad
152502010@smail.iitpkd.ac.in

Abstract

The objective of this experiment is to implement and compare multiple C-based realizations of a Finite Impulse Response (FIR) filter using High-Level Synthesis (HLS). Although all implementations realize the same FIR filtering operation, they differ in loop structure, data movement, and optimization directives. The impact of these variations on performance metrics such as latency, initiation interval, and hardware resource utilization is analyzed. This experiment demonstrates how algorithmically equivalent descriptions can result in different hardware architectures after synthesis.

I. Introduction

This experiment compares multiple FIR code variants that implement the same filter coefficients and tap structure but differ in loop organization, degree of parallelism, and memory access patterns. The goal is to analyze how these implementation choices affect synthesis results while preserving identical functional output.

II. Implementation

Five different implementations of FIR Filter was made to compute using the same inputs and the same filter coeffecients. These models differ in how the shifting, multiplication and accumulation operations are done.

A. FIR Model 1

```

#define N 11
#include <ap_int.h>
void FIR1(int *y, int x){
int C[N]={53,0,-91,313,500,313,0,-91,0,53};
static int shift_reg[N];
int acc,i;
acc=0;
SAL: for(i=N-1;i>=0;i--){
        if(i==0){
          acc+=x*C[0];
          shift_reg[0]=x;
        }
        else {
          shift_reg[i]=shift_reg[i-1];
          acc += shift_reg[i]*C[i];
        }
     }
   *y=acc;
}
```

Listing 1. All operations done inside single loop

B. FIR Model 2

```

#define N 11
#include <ap_int.h>
void FIR2(int *y, int x){
int C[N]={53,0,-91,313,500,313,0,-91,0,53};
static int shift_reg[N];
int acc,i;
acc=0;
SAL: for(i=N-1;i>0;i--){
          shift_reg[i]=shift_reg[i-1];
          acc += shift_reg[i]*C[i];
     }
    acc += x*C[0];
   *y=acc;
}
```

Listing 2. Input logic is handled separately

C. FIR Model 3

```

#define N 11
#include <ap_int.h>
void FIR3(int *y, int x){
int C[N]={53,0,-91,313,500,313,0,-91,0,53};
static int shift_reg[N];
int acc,i;
acc=0;
//loop fission
TDL: for(i=N-1;i>0;i--){
          shift_reg[i]=shift_reg[i-1];
      }
      shift_reg[0]=x;
MAC: for(i=N-1;i>=0;i--){
          acc += shift_reg[i]*C[i];
     }
   *y=acc;
}
```

Listing 3. Trap delay separated from other arithmetic operations

D. FIR Model 4

```
#define N 11
#include <ap_int.h>
void FIR4(int *y, int x){
int C[N]={53,0,-91,313,500,313,0,-91,0,53};
static int shift_reg[N];
int acc,i; acc=0;
//loop fission
TDL: for(i=N-1;i>1;i=i-2){
          shift_reg[i]=shift_reg[i-1];
          shift_reg[i-1]=shift_reg[i-2];
      }
      if(i==0)shift_reg[1]=shift_reg[0];
      shift_reg[0]=x;
MAC: for(i=N-1;i>=0;i--){
          acc += shift_reg[i]*C[i];
     }
   *y=acc;
}
```

Listing 4. Loop iterations reduced by half

E. FIR Model 5

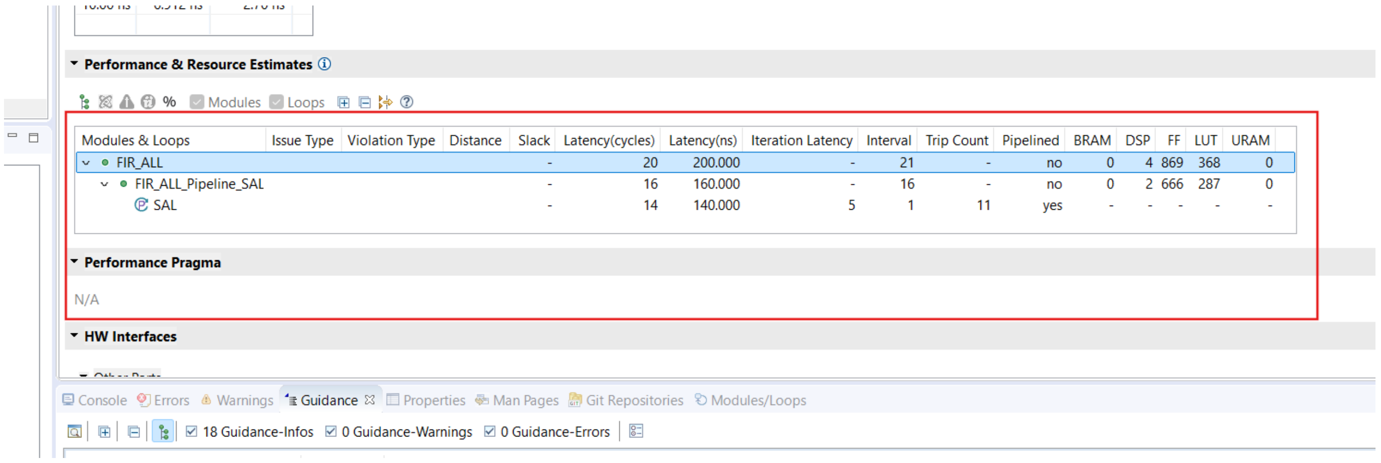
```

#define N 11
#include <ap_int.h>
void FIR5(int *y, int x){
int C[N]={53,0,-91,313,500,313,0,-91,0,53};
static int shift_reg[N];
int acc,i;
acc=0;
#pragma HLS ARRAY_PARTITION variable=shift_reg complete
//loop fission
TDL: for(i=N-1;i>1;i=i-2){//TDL: Tapped Delay Line
          shift_reg[i]=shift_reg[i-1];
          shift_reg[i-1]=shift_reg[i-2];
      }
      if(i==0)shift_reg[1]=shift_reg[0];
      shift_reg[0]=x;
MAC: for(i=N-1;i>=3;i=i-4){//MAC:Multiply  and Accumulate

          acc += shift_reg[i]*C[i]+shift_reg[i-1]*C[i-1]+shift_reg[i-2]*C[i-2]+shift_reg[i-3]*C[i-4];
     }
for(;i>=0;i--) acc +=shift_reg[i]*C[i];
   *y=acc;
}
```

Listing 5. Pragma introduced, so maximum parallelism

III. Report analysis

The synthesis report of every implementation was obtained and their performance and resource estimations are studied.

A. FIR Model 1



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_ALL				-	20	200.000	-	21	-	no	0	4	869	368	0
FIR_ALL_Pipeline_SAL				-	16	160.000	-	16	-	no	0	2	666	287	0
SAL				-	14	140.000	5	1	11	yes	-	-	-	-	-

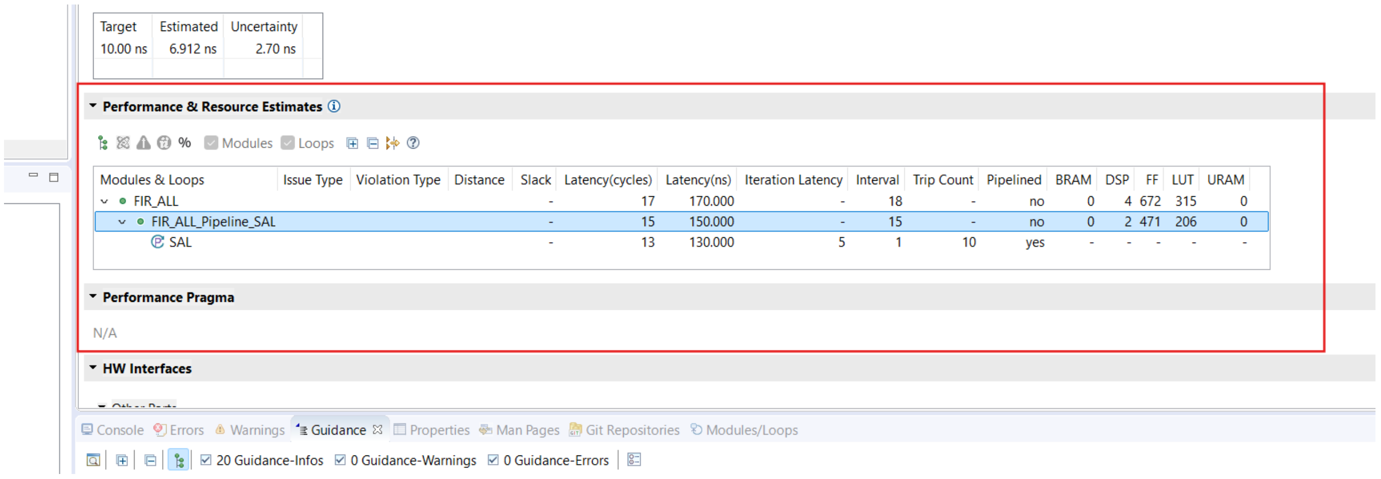
Fig. 1. Base Implementation report

This implementation uses a single loop that performs shift register updates and multiply–accumulate operations together, along with a conditional check to handle the current input sample. **Result obtained:** Latency: 20 cycles , Interval: 21 , DSP: 4 , FF: 869 , LUTs: 368

- The conditional logic inside the loop introduces additional multiplexers and control paths in the datapath.
- The loop dependencies restrict pipelining by the HLS tool.

As a result, this model serves as the reference design with moderate latency and resource usage.

B. *FIR Model 2*



Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_ALL				-	17	170.000	-	18	-	no	0	4	672	315	0
FIR_ALL_Pipeline_SAL				-	15	150.000	-	15	-	no	0	2	471	206	0
SAL				-	13	130.000	5	1	10	yes	-	-	-	-	-

Performance Pragma

N/A

HW Interfaces

Console Errors Warnings Guidance Properties Man Pages Git Repositories Modules/Loops

☑ 20 Guidance-Infos ☑ 0 Guidance-Warnings ☑ 0 Guidance-Errors

Fig. 2. Input Logic Optimized synthesis

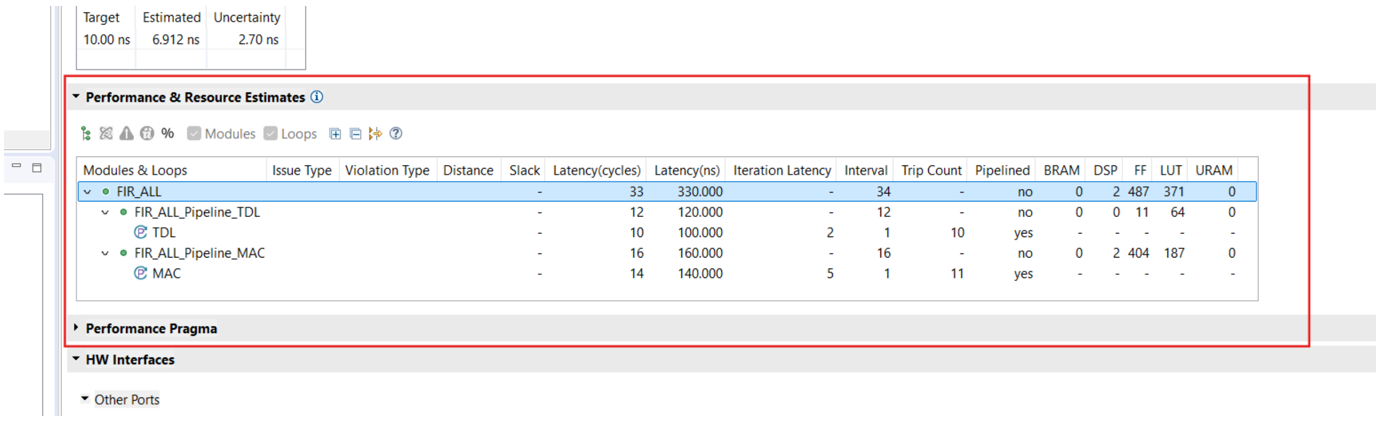
This implementation removes the conditional check from the main loop by handling the boundary condition separately, while still performing shift register updates and multiply–accumulate operations within a single loop.

Result obtained: Latency: 17 cycles , Interval: 18 , DSP: 4 , FF: 672 , LUTs: 315

- Eliminating the conditional logic simplifies the datapath and reduces the number of multiplexers required.
- The reduced control complexity allows tighter scheduling and improved pipelining by the HLS tool.

As a result, this model achieves lower latency and reduced resource utilization compared to the baseline implementation.

C. *FIR Model 3*



Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_ALL				-	33	330.000	-	34	-	no	0	2	487	371	0
FIR_ALL_Pipeline_TDL				-	12	120.000	-	12	-	no	0	0	11	64	0
TDL				-	10	100.000	2	1	10	yes	-	-	-	-	-
FIR_ALL_Pipeline_MAC				-	16	160.000	-	16	-	no	0	2	404	187	0
MAC				-	14	140.000	5	1	11	yes	-	-	-	-	-

Performance Pragma

HW Interfaces

Other Ports

Fig. 3. Synthesis after Loop Fission

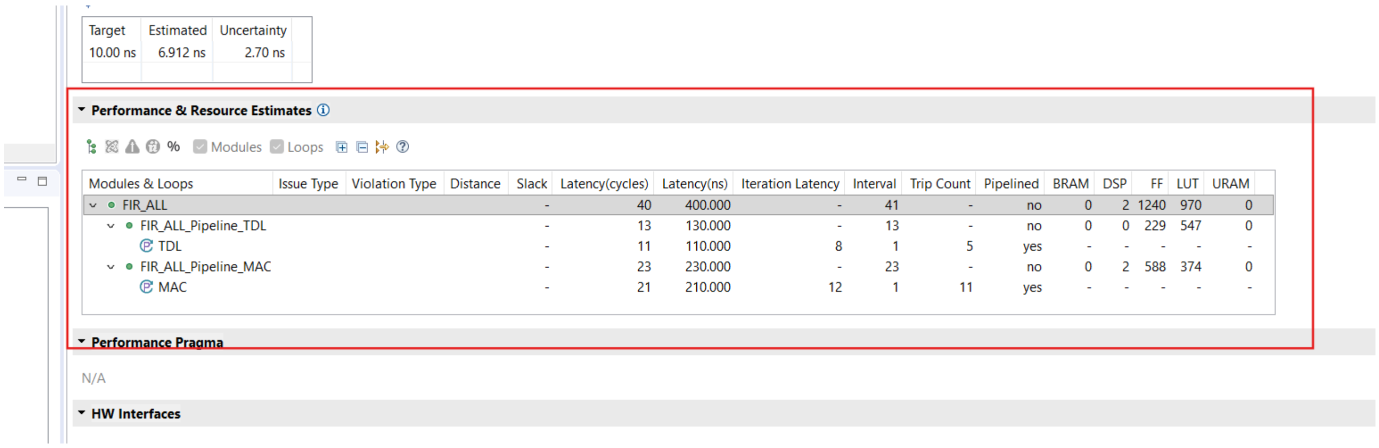
This implementation separates the shift register update and multiply–accumulate operations into two independent loops executed sequentially.

Result obtained: Latency: 33 cycles , Interval: 34 , DSP: 2 , FF: 487 , LUTs: 371

- Sequential execution of the two loops increases the overall latency.
- The separation of operations enables reuse of arithmetic resources across cycles, reducing DSP utilization.

As a result, this model provides improved area efficiency at the cost of increased latency.

D. *FIR Model 4*



Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_ALL				-	40	400.000	-	41	-	no	0	2	1240	970	0
FIR_ALL_Pipeline_TDL				-	13	130.000	-	13	-	no	0	0	229	547	0
TDL				-	11	110.000	8	1	5	yes	-	-	-	-	-
FIR_ALL_Pipeline_MAC				-	23	230.000	-	23	-	no	0	2	588	374	0
MAC				-	21	210.000	12	1	11	yes	-	-	-	-	-

Performance Pragma

N/A

HW Interfaces

Fig. 4. Synthesis after Manual loop unrolling

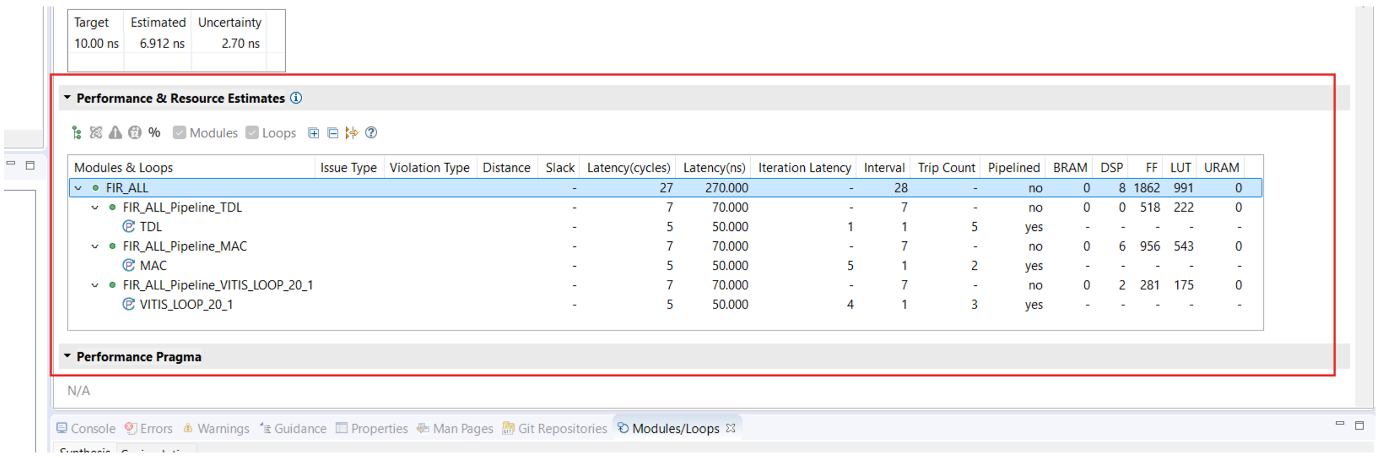
This implementation manually unrolls the shift register update loop by a factor of two without applying array partitioning.

Result obtained: Latency: 40 cycles , Interval: 41 , DSP: 2 , FF: 1240 , LUTs: 970

- The unrolled shift logic increases control complexity and register usage.
- Multiple read and write operations to the shift register in a single iteration increase scheduling overhead.

As a result, this model exhibits increased latency and significantly higher resource usage, making it less efficient than previous implementations.

E. *FIR Model 5*



Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR_ALL				-	27	270.000	-	28	-	no	0	8	1862	991	0
FIR_ALL_Pipeline_TDL				-	7	70.000	-	7	-	no	0	0	518	222	0
TDL				-	5	50.000	1	1	5	yes	-	-	-	-	-
FIR_ALL_Pipeline_MAC				-	7	70.000	-	7	-	no	0	6	956	543	0
MAC				-	5	50.000	5	1	2	yes	-	-	-	-	-
FIR_ALL_Pipeline_VITIS_LOOP_20_1				-	7	70.000	-	7	-	no	0	2	281	175	0
VITIS_LOOP_20_1				-	5	50.000	4	1	3	yes	-	-	-	-	-

Performance Pragma

N/A

Fig. 5. Synthesis after Pipelining

This implementation applies complete array partitioning to the shift register and introduces parallelism in the multiply–accumulate operations.

Result obtained: Latency: 27 cycles , Interval: 28 , DSP: 8 , FF: 1862 , LUTs: 991

- Array partitioning using *PRAGMA* enables simultaneous access to all filter taps, increasing parallel execution.
- Increased parallelism results in higher DSP and flip-flop utilization.

As a result, this model achieves improved performance compared to sequential designs, with a substantial increase in hardware resource usage.

FIR Model	Latency (cycles)	Latency (ns)	Interval	DSP	FF	LUTs
FIR1	20	200	21	4	869	368
FIR2	17	170	18	4	672	315
FIR3	33	330	34	2	487	371
FIR4	40	400	41	2	1240	970
FIR5	27	270	28	8	1862	991

TABLE I
COMPARISON OF FIR CODE VARIANTS: LATENCY AND RESOURCE UTILIZATION

CONCLUSION

This experiment shows that, although all FIR implementations compute the same filter function, differences in coding style significantly impact latency, throughput, and hardware resource utilization after HLS synthesis. Hence, careful selection of implementation style is essential to achieve the desired performance–area trade-off.