Experiment 3
FIR Filter using Zynq PS, AXI DMA and HLS Accelerator

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

I. Introduction

In this experiment, an FIR filter accelerator is implemented using Vitis HLS, and synthesized into an RTL-based IP core. The generated IP is designed using AXI4-Stream interfaces for high-throughput data transfer. To transfer data efficiently between DDR memory and the accelerator, an AXI DMA engine is used. The ARM processor sends input samples stored in DDR memory to the FIR accelerator through the DMA's MM2S (Memory Mapped to Stream) channel, and receives the filtered output through the S2MM (Stream to Memory Mapped) channel.

The objective of this experiment is to understand PS–PL communication using the AXI protocol, integrate an HLS-generated streaming IP into a Zynq system, and validate real-time data transfer using AXI DMA in a standalone Vitis application.

II. Steps used and Observations made

The experiment was performed using the ZedBoard platform based on the Zynq-7000 SoC. The overall workflow consists of three major phases: (i) FIR accelerator creation using Vitis HLS, (ii) hardware integration using Vivado block design, and (iii) software development using Vitis IDE.

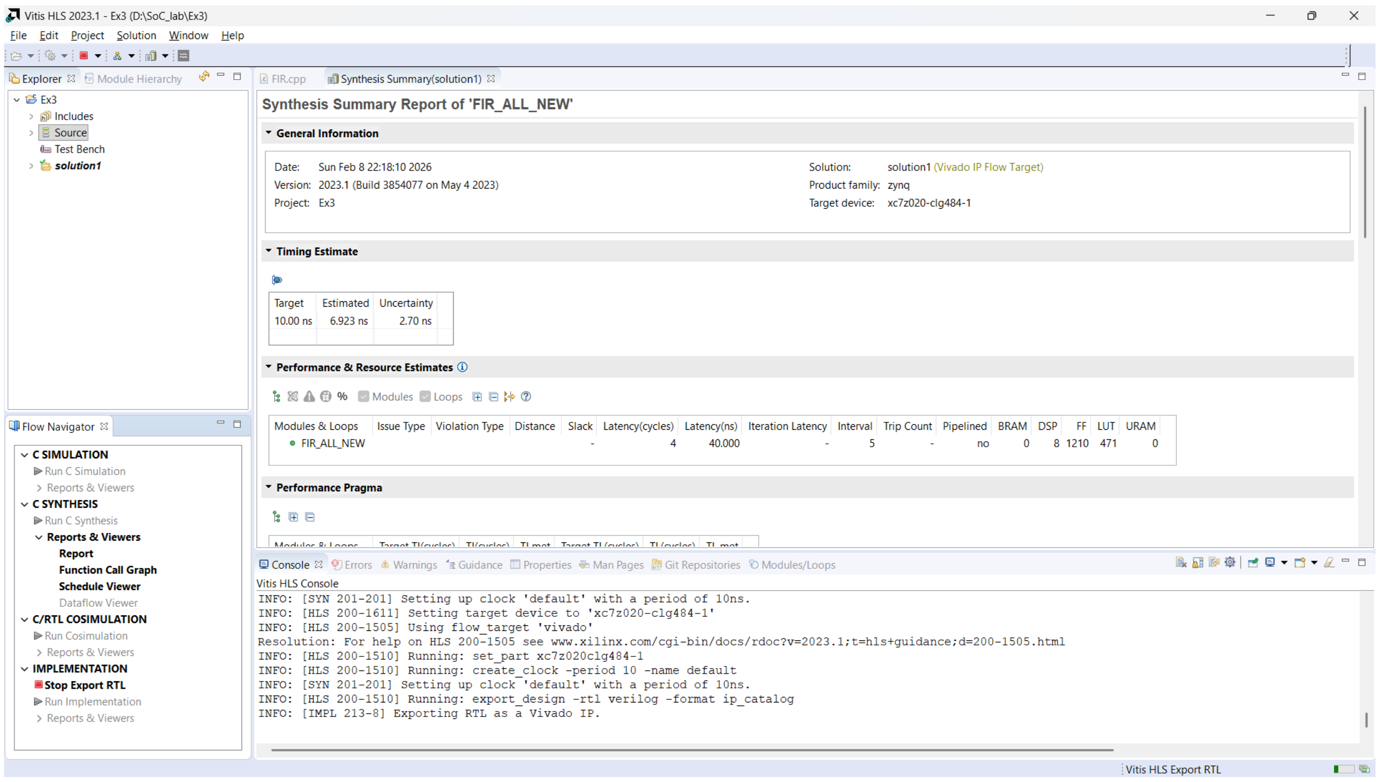
A. FIR Filter Implementation using Vitis HLS

The FIR filter was implemented in C/C++ using Vitis HLS. The design was written using streaming interfaces such that the input and output ports follow the AXI4-Stream protocol. This allows the accelerator to continuously process input samples without requiring register-based control.

The HLS function accepts an AXI stream input packet containing the input sample and generates an output packet containing the filtered result. Internally, a shift register is maintained to store past input samples. Multiply-and-accumulate operations are performed using predefined FIR coefficients. Optimizations such as loop unrolling and array partitioning are applied to increase parallelism and reduce latency.

```
#include <hls_stream.h>
#include <ap_axi_sdata.h>
#define N 10
//for keeping the necessary control signals
typedef ap_axiu<32,0,0,0> axis_t;

void FIR_ALL_NEW(hls::stream<axis_t> &s_axis,
                 hls::stream<axis_t> &m_axis){
    #pragma HLS interface axis port=s_axis
    #pragma HLS interface axis port=m_axis
    #pragma HLS interface ap_ctrl_none port=return

    int C[N] = {53,0,-91,313,500,313,0,-91,0,53};
    static int shift_reg[N];
    #pragma HLS array_partition variable=shift_reg complete

    // ---- Axis handshake was used here ----
    axis_t in_pkt = s_axis.read();    // blocks until VALID&READY
    int x = in_pkt.data;
    int acc = 0;

    // ---- FIR defenition
    FIR_LOOP:
    for(int i=N-1;i>=0;i--){
        #pragma HLS unroll
        if(i==0){
            shift_reg[0] = x;
            acc += x * C[0];
        } else {
            shift_reg[i] = shift_reg[i-1];
            acc += shift_reg[i] * C[i];
        }
    }

    // ---- OUTPUT PACKET ----
    axis_t out_pkt;
    out_pkt.data = acc;

    // Verilog based writing
    out_pkt.last = in_pkt.last;    // TLAST pass-through
    out_pkt.keep = -1;             // full word valid (1111)
    m_axis.write(out_pkt);         // VALID asserted when written
}
```

Listing 1. FIR Filter C code



Synthesis Summary Report of 'FIR_ALL_NEW'

General Information

Date:	Sun Feb 8 22:18:10 2026	Solution:	solution1 (Vivado IP Flow Target)
Version:	2023.1 (Build 3854077 on May 4 2023)	Product family:	zynq
Project:	Ex3	Target device:	xc7z020-clg484-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	6.923 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

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_NEW				-	4	40.000	-	5	-	no	0	8	1210	471	0

Performance Pragma

Vitis HLS Console

```
INFO: [SYN 201-201] Setting up clock 'default' with a period of 10ns.
INFO: [HLS 200-1611] Setting target device to 'xc7z020-clg484-1'
INFO: [HLS 200-1505] Using flow_target 'vivado'
Resolution: For help on HLS 200-1505 see www.xilinx.com/cgi-bin/docs/rdoc?v=2023.1;t=hls+guidance;d=200-1505.html
INFO: [HLS 200-1510] Running: set_part xc7z020clg484-1
INFO: [HLS 200-1510] Running: create_clock -period 10 -name default
INFO: [SYN 201-201] Setting up clock 'default' with a period of 10ns.
INFO: [HLS 200-1510] Running: export_design -rtl verilog -format ip_catalog
INFO: [IMPL 213-8] Exporting RTL as a Vivado IP.
```

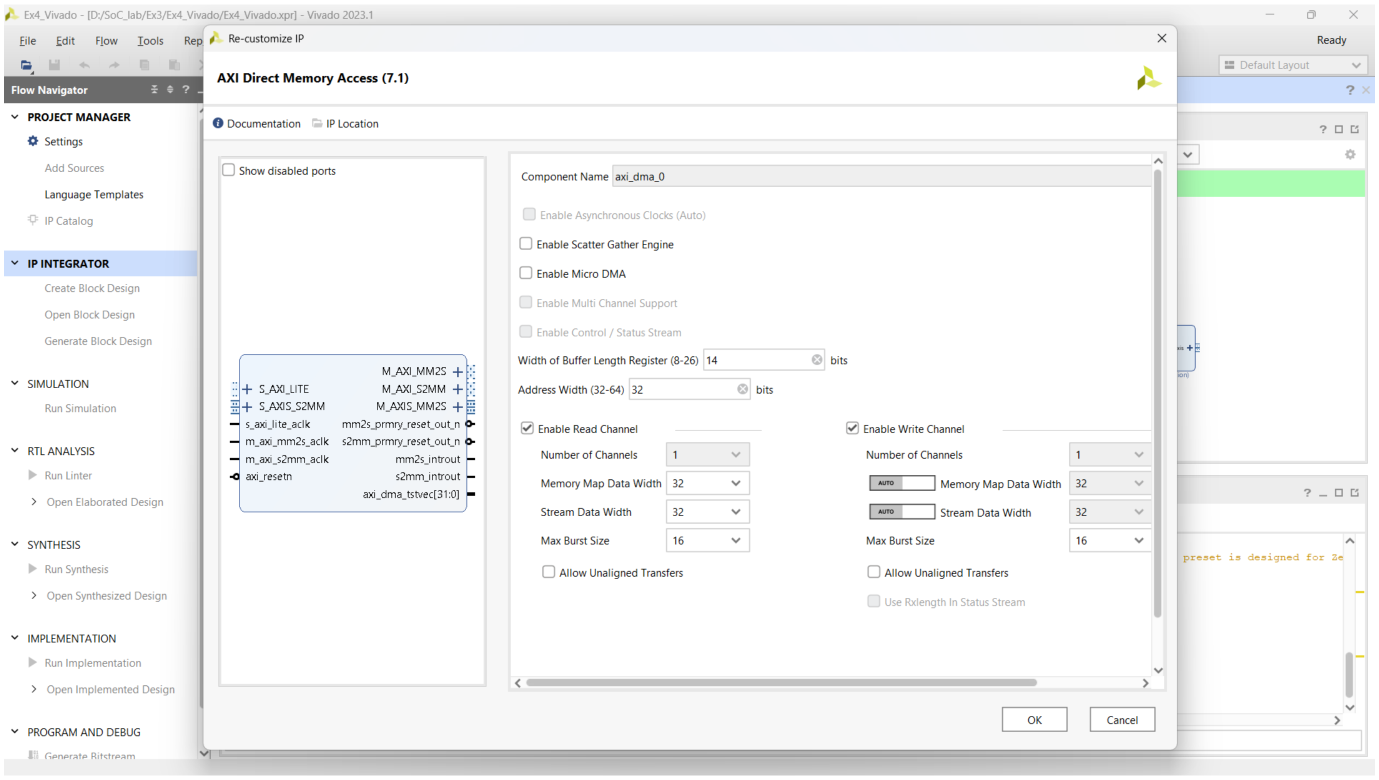
Fig. 1. Vitis HLS Resource Utilization

Once functional verification was completed in HLS simulation, the design was synthesized to generate RTL (Verilog/VHDL). The RTL was then packaged as a Vivado IP core.

B. Vivado Block Design Integration

After generating the FIR IP core, a Vivado block design was created to integrate the Processing System, AXI DMA, and FIR accelerator. The Processing System was configured with DDR memory enabled, and the AXI interface was enabled to allow communication between PS and PL.

An AXI DMA block was added to the design to facilitate high-speed data transfer between DDR memory and the AXI-stream based FIR accelerator. The MM2S channel of DMA provides the input stream to the FIR accelerator, and the S2MM channel receives the filtered output stream. Connection automation was used to connect clocks, resets, and AXI interconnect paths.



AXI Direct Memory Access (7.1)

Component Name: axi_dma_0

- [ ] Enable Asynchronous Clocks (Auto)
- [ ] Enable Scatter Gather Engine
- [ ] Enable Micro DMA
- [ ] Enable Multi Channel Support
- [ ] Enable Control / Status Stream

Width of Buffer Length Register (8-26): 14 bits

Address Width (32-64): 32 bits

[✓] Enable Read Channel		[✓] Enable Write Channel	
Number of Channels	1	Number of Channels	1
Memory Map Data Width	32	AUTO Memory Map Data Width	32
Stream Data Width	32	AUTO Stream Data Width	32
Max Burst Size	16	Max Burst Size	16
[ ] Allow Unaligned Transfers		[ ] Allow Unaligned Transfers	
		[ ] Use Rxlength In Status Stream	

preset is designed for Ze

Fig. 2. AXI DMA configs

The complete hardware design includes the following major blocks:

1) Zynq Processing System (PS7)
2) AXI DMA (Simple mode)
3) FIR Filter IP (HLS generated)
4) AXI Interconnect and Reset Controller

Figure 3 shows the complete hardware architecture consisting of the Zynq Processing System, AXI DMA, HLS-generated FIR IP, and AXI interconnect modules.

After successful synthesis and implementation, the bitstream was generated and the hardware design was exported as an .xsa file for software development in Vitis.

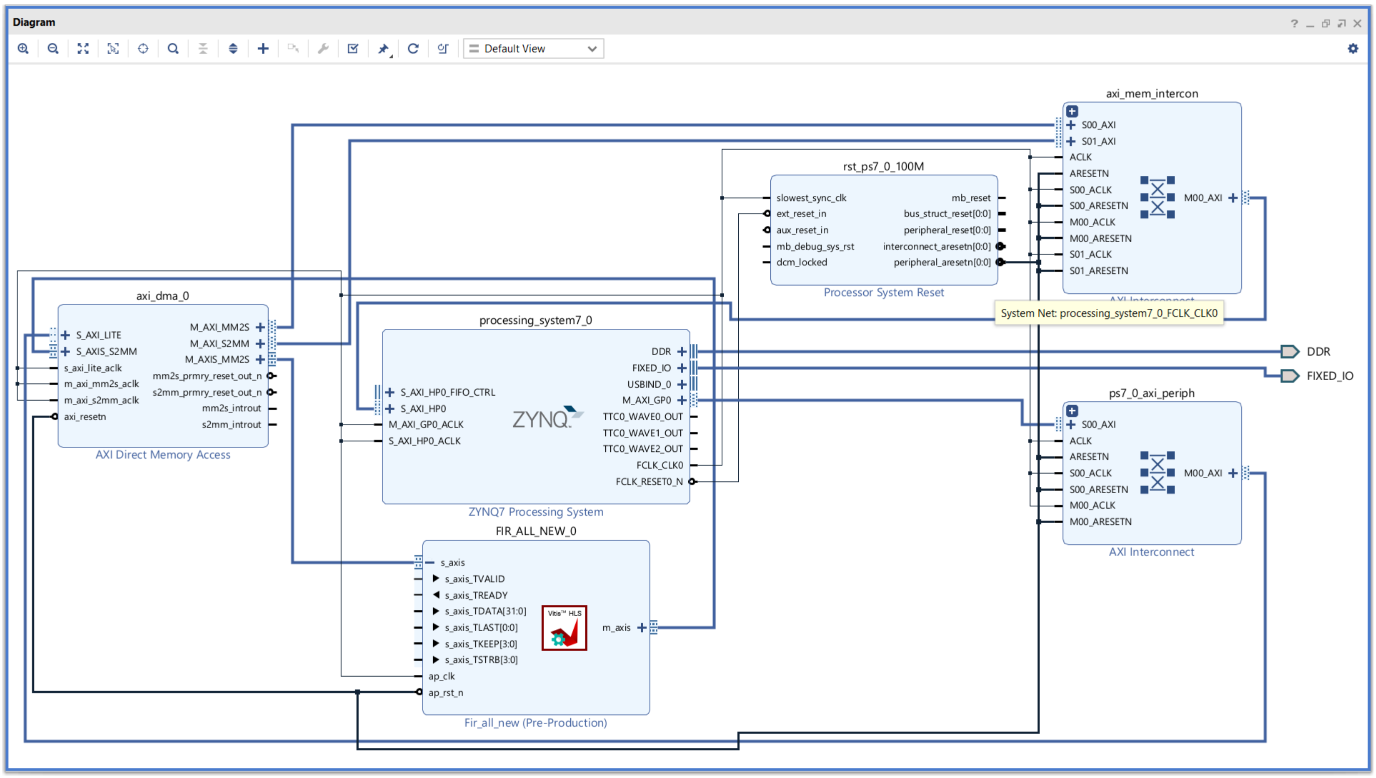
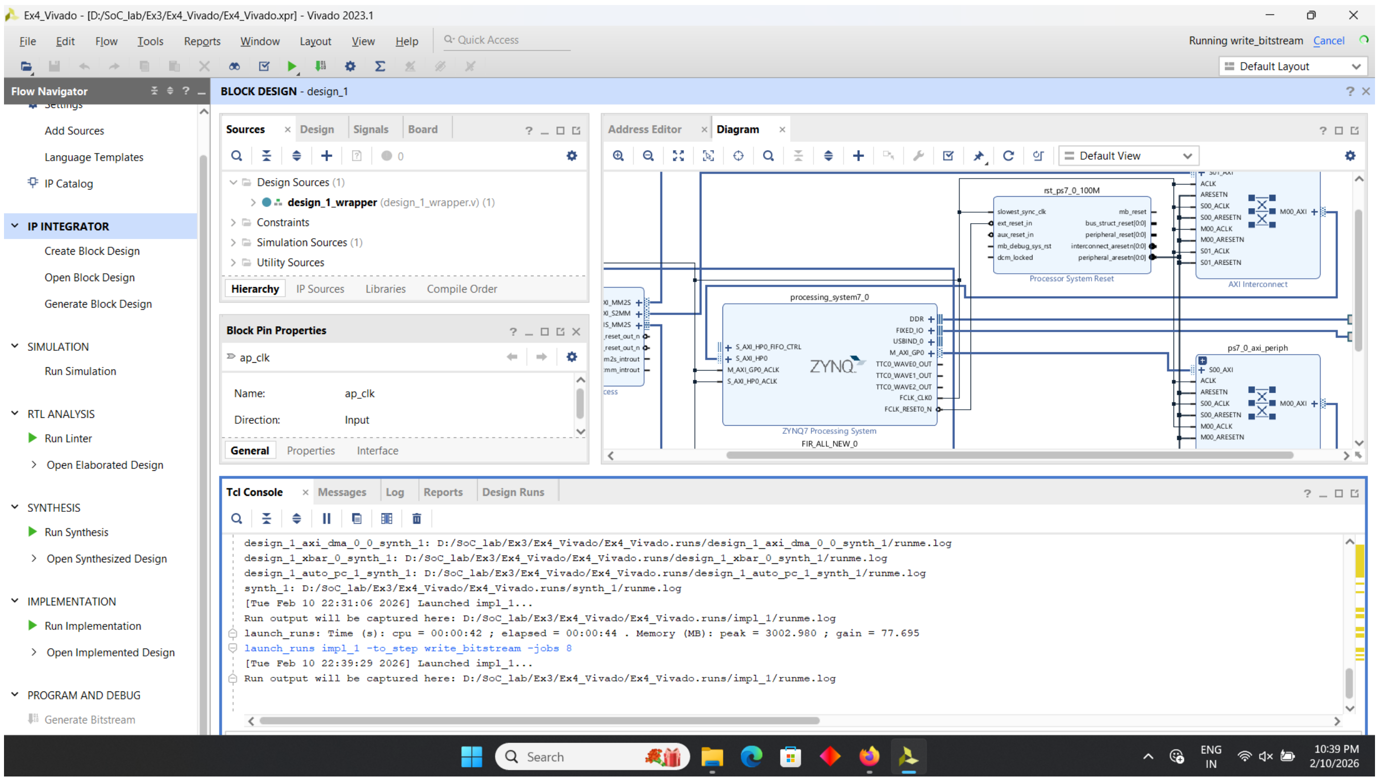


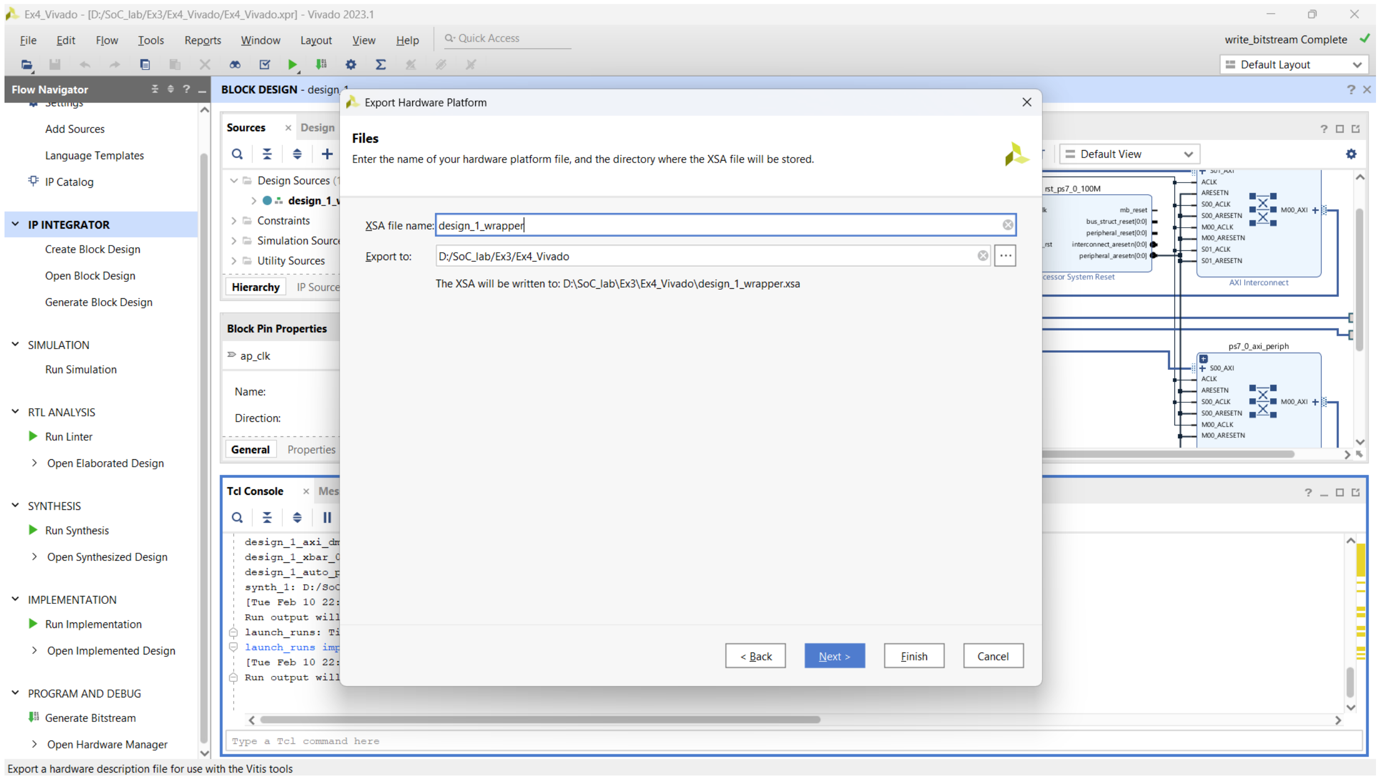
Fig. 3. Full Block Diagram



Name: ap_clk
Direction: Input

```
design_1_axi_dma_0_0_synth_1: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/design_1_axi_dma_0_0_synth_1/runme.log
design_1_xbar_0_synth_1: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/design_1_xbar_0_synth_1/runme.log
design_1_auto_pc_1_synth_1: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/design_1_auto_pc_1_synth_1/runme.log
synth_1: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/synth_1/runme.log
[Tue Feb 10 22:31:06 2026] Launched impl_1...
Run output will be captured here: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/impl_1/runme.log
launch_runs: Time (s): cpu = 00:00:42 ; elapsed = 00:00:44 . Memory (MB): peak = 3002.980 ; gain = 77.695
launch_runs impl_1 -to_step write_bitstream -jobs 8
[Tue Feb 10 22:39:29 2026] Launched impl_1...
Run output will be captured here: D:/SoC_lab/Ex3/Ex4_Vivado/Ex4_Vivado.runs/impl_1/runme.log
```

Fig. 4. Vivado Implementation done for FIR



Export Hardware Platform

Files

Enter the name of your hardware platform file, and the directory where the XSA file will be stored.

XSA file name: design_1_wrapper

Export to: D:/SoC_lab/Ex3/Ex4_Vivado

The XSA will be written to: D:\SoC_lab\Ex3\Ex4_Vivado\design_1_wrapper.xsa

Export a hardware description file for use with the Vitis tools

Fig. 5. Export Hardware platform from Vivado

C. *Vitis Application Development*

A standalone application was created in Vitis IDE using the exported hardware platform. The software runs on the ARM Cortex-A9 processor and is responsible for configuring the DMA engine, sending input samples to the FIR accelerator, and receiving filtered output samples. The sample HelloWorld.c code was replaced with the code as given below in Listing: 2.

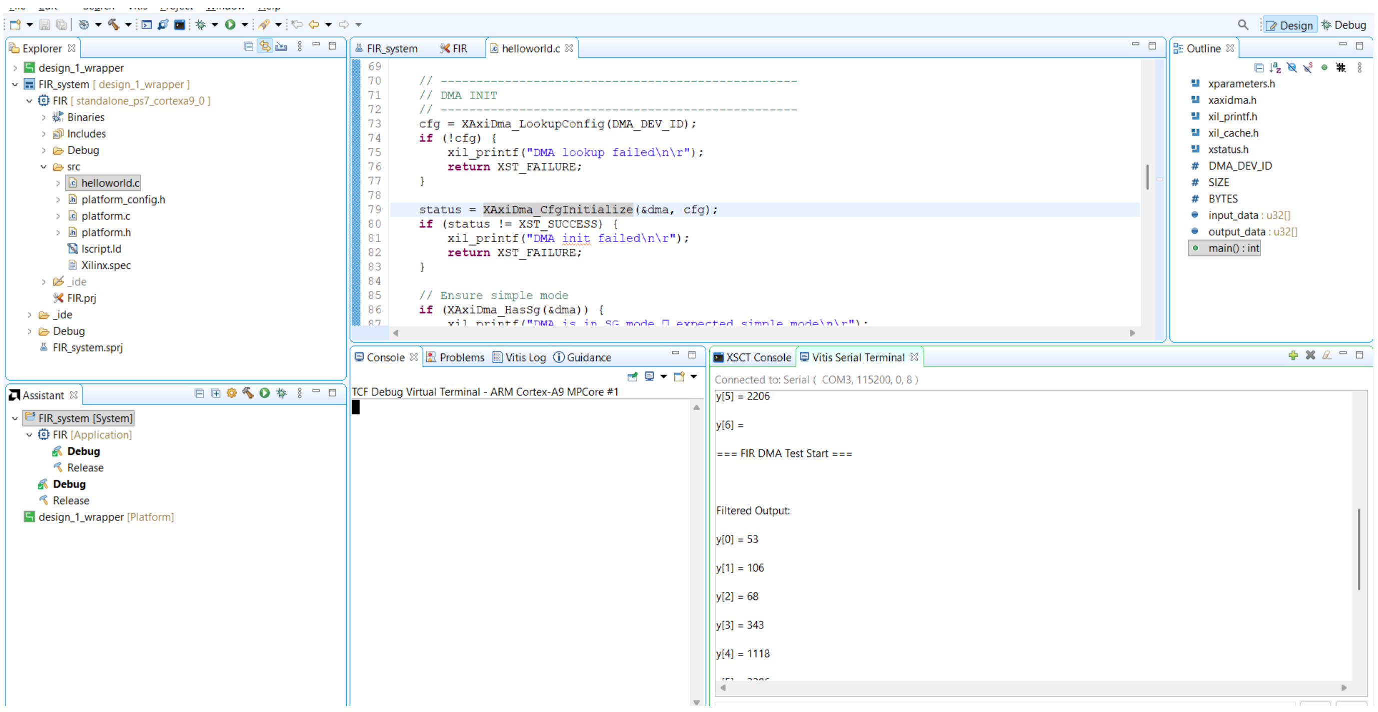
```
#include "xparameters.h"
#include "xaxidma.h"
#include "xil_printf.h"
#include "xil_cache.h"
#include "xstatus.h"

#define DMA_DEV_ID      XPAR_AXI_DMA_0_DEVICE_ID

#define SIZE            12                  // number of samples
#define BYTES           (SIZE * sizeof(u32))

u32 input_data[SIZE]  __attribute__((aligned(64)));
u32 output_data[SIZE] __attribute__((aligned(64)));

int main()
{
    XAxiDma dma;
    XAxiDma_Config *cfg;
    int status;

    xil_printf("\n\r=== FIR DMA Test Start ===\n\r");

    // --------------------------------------------------
    // DMA INIT
    // --------------------------------------------------
    cfg = XAxiDma_LookupConfig(DMA_DEV_ID);
    if (!cfg) {
        xil_printf("DMA lookup failed\n\r");
        return XST_FAILURE;
    }

    status = XAxiDma_CfgInitialize(&dma, cfg);
    if (status != XST_SUCCESS) {
        xil_printf("DMA init failed\n\r");
        return XST_FAILURE;
    }

    // Ensure simple mode
    if (XAxiDma_HasSg(&dma)) {
        xil_printf("DMA is in SG mode ... expected simple mode\n\r");
        return XST_FAILURE;
    }

    // --------------------------------------------------
    // DMA RESET (important after bitstream download)
    // --------------------------------------------------
    XAxiDma_Reset(&dma);
    while (!XAxiDma_ResetIsDone(&dma));

    // Disable interrupts (polling mode)
    XAxiDma_IntrDisable(&dma, XAXIDMA_IRQ_ALL_MASK,
                        XAXIDMA_DMA_TO_DEVICE);
    XAxiDma_IntrDisable(&dma, XAXIDMA_IRQ_ALL_MASK,
                        XAXIDMA_DEVICE_TO_DMA);

    // --------------------------------------------------
    // PREPARE INPUT DATA
    // --------------------------------------------------
    for (int i = 0; i < SIZE; i++) {
        input_data[i] = i + 1;
        output_data[i] = 0;
    }

    // Cache maintenance
    Xil_DCacheFlushRange((UINTPTR)input_data, BYTES);
    Xil_DCacheFlushRange((UINTPTR)output_data, BYTES);

    // --------------------------------------------------
    // START DMA TRANSFERS
    // Order matters: start S2MM (receive) first
    // --------------------------------------------------

    status = XAxiDma_SimpleTransfer(&dma,
                                    (UINTPTR)output_data,
                                    BYTES,
                                    XAXIDMA_DEVICE_TO_DMA);
    if (status != XST_SUCCESS) {
        xil_printf("S2MM start failed\n\r");
        return XST_FAILURE;
    }

    status = XAxiDma_SimpleTransfer(&dma,
                                    (UINTPTR)input_data,
                                    BYTES,
                                    XAXIDMA_DMA_TO_DEVICE);
    if (status != XST_SUCCESS) {
        xil_printf("MM2S start failed\n\r");
        return XST_FAILURE;
    }

    // ------------------------------------------------
    // WAIT FOR COMPLETION
    // ------------------------------------------------
    while (XAxiDma_Busy(&dma, XAXIDMA_DMA_TO_DEVICE));
    while (XAxiDma_Busy(&dma, XAXIDMA_DEVICE_TO_DMA));

    // ------------------------------------------------
    // ERROR CHECK
    // ------------------------------------------------
    u32 sr;

    sr = XAxiDma_ReadReg(dma.RegBase, XAXIDMA_TX_OFFSET + XAXIDMA_SR_OFFSET);
    if (sr & XAXIDMA_ERR_ALL_MASK) {
        xil_printf("MM2S DMA error: 0x%08x\n\r", sr);
        return XST_FAILURE;
    }

    sr = XAxiDma_ReadReg(dma.RegBase, XAXIDMA_RX_OFFSET + XAXIDMA_SR_OFFSET);
    if (sr & XAXIDMA_ERR_ALL_MASK) {
        xil_printf("S2MM DMA error: 0x%08x\n\r", sr);
        return XST_FAILURE;
    }

    // ------------------------------------------------
    // INVALIDATE CACHE FOR OUTPUT
    // ------------------------------------------------
    Xil_DCacheInvalidateRange((UINTPTR)output_data, BYTES);

    // ------------------------------------------------
    // PRINT RESULTS
    // ------------------------------------------------
    xil_printf("\n\rFiltered Output:\n\r");
    for (int i = 0; i < SIZE; i++) {
        xil_printf("y[%d] = %d\n\r", i, output_data[i]);
    }

    xil_printf("\n\r=== FIR DMA Test Complete ===\n\r");

    return XST_SUCCESS;
}
```

Listing 2. Vitis code to connect with zedboard

The software initializes the AXI DMA using the Xilinx DMA driver library. Input samples are stored in a buffer located in DDR memory. Cache flush operations are performed to ensure that the updated input data is written back to memory before DMA reads it. Similarly, after DMA writes the output buffer, cache invalidation is performed to ensure that the processor reads the latest data from DDR memory.

The DMA transfer is performed using polling mode. The receive transfer (S2MM) is initiated first, followed by the transmit transfer (MM2S). Once both transfers are completed, the output buffer contains the FIR filtered results. These results are printed on the serial terminal using UART.

Figure 6 shows the UART output when the FIR filter was executed on the ZedBoard using the exported hardware platform.



```
    // --------------------------------------------------
    // DMA INIT
    // --------------------------------------------------
    cfg = XAxiDma_LookupConfig(DMA_DEV_ID);
    if (!cfg) {
        xil_printf("DMA lookup failed\n\r");
        return XST_FAILURE;
    }

    status = XAxiDma_CfgInitialize(&dma, cfg);
    if (status != XST_SUCCESS) {
        xil_printf("DMA init failed\n\r");
        return XST_FAILURE;
    }

    // Ensure simple mode
    if (XAxiDma_HasSg(&dma)) {
        xil_printf("DMA is in SG mode □ expected simple mode\n\r");
```

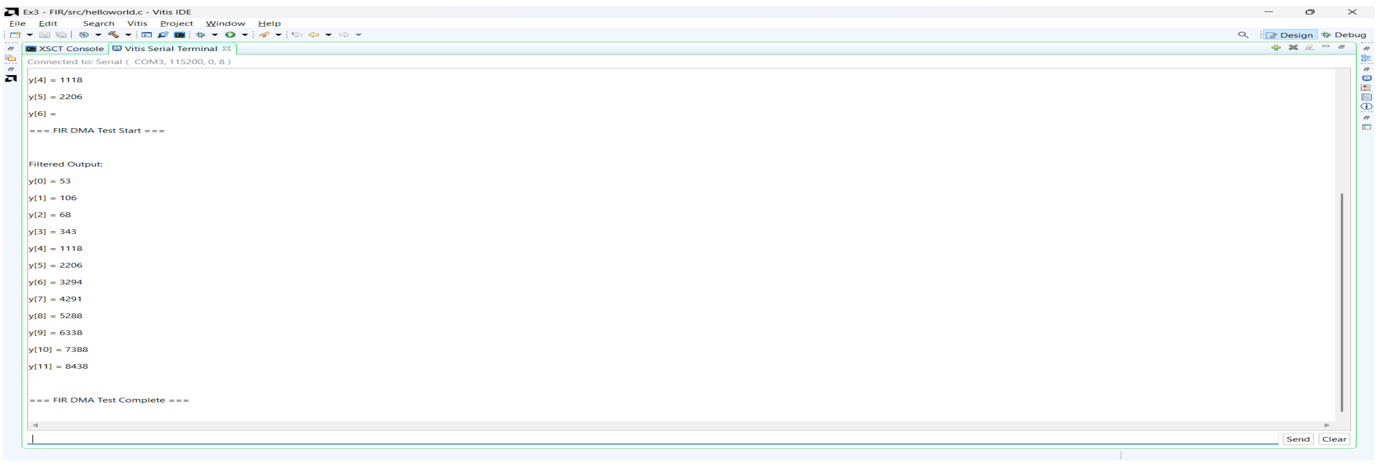
```
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1
```

```
Connected to: Serial ( COM3, 115200, 0, 8 )
y[5] = 2206

y[6] =

=== FIR DMA Test Start ===


Filtered Output:

y[0] = 53

y[1] = 106

y[2] = 68

y[3] = 343

y[4] = 1118
```

Fig. 6. FIR filter running on Vitis

D. Observations

The output results obtained from the UART terminal confirm that the FIR accelerator correctly processes the streamed input samples and produces filtered output values.


Ex3 - FIR/src/helloworld.c - Vitis IDE
File Edit Search Vitis Project Window Help
Design
Debug
XSCT Console
Vitis Serial Terminal
Send
Clear

```
Connected to: Serial ( COM3, 115200, 0, 8 )
y[4] = 1118
y[5] = 2206
y[6] =
=== FIR DMA Test Start ===

Filtered Output:
y[0] = 53
y[1] = 106
y[2] = 68
y[3] = 343
y[4] = 1118
y[5] = 2206
y[6] = 3294
y[7] = 4291
y[8] = 5288
y[9] = 6338
y[10] = 7388
y[11] = 8438

=== FIR DMA Test Complete ===
```

Fig. 7. FIR output on Vitis via UART to the console

The successful execution validates the integration of an AXI4-Stream based hardware accelerator into a Zynq system and demonstrates high-throughput PS–PL communication using AXI DMA.

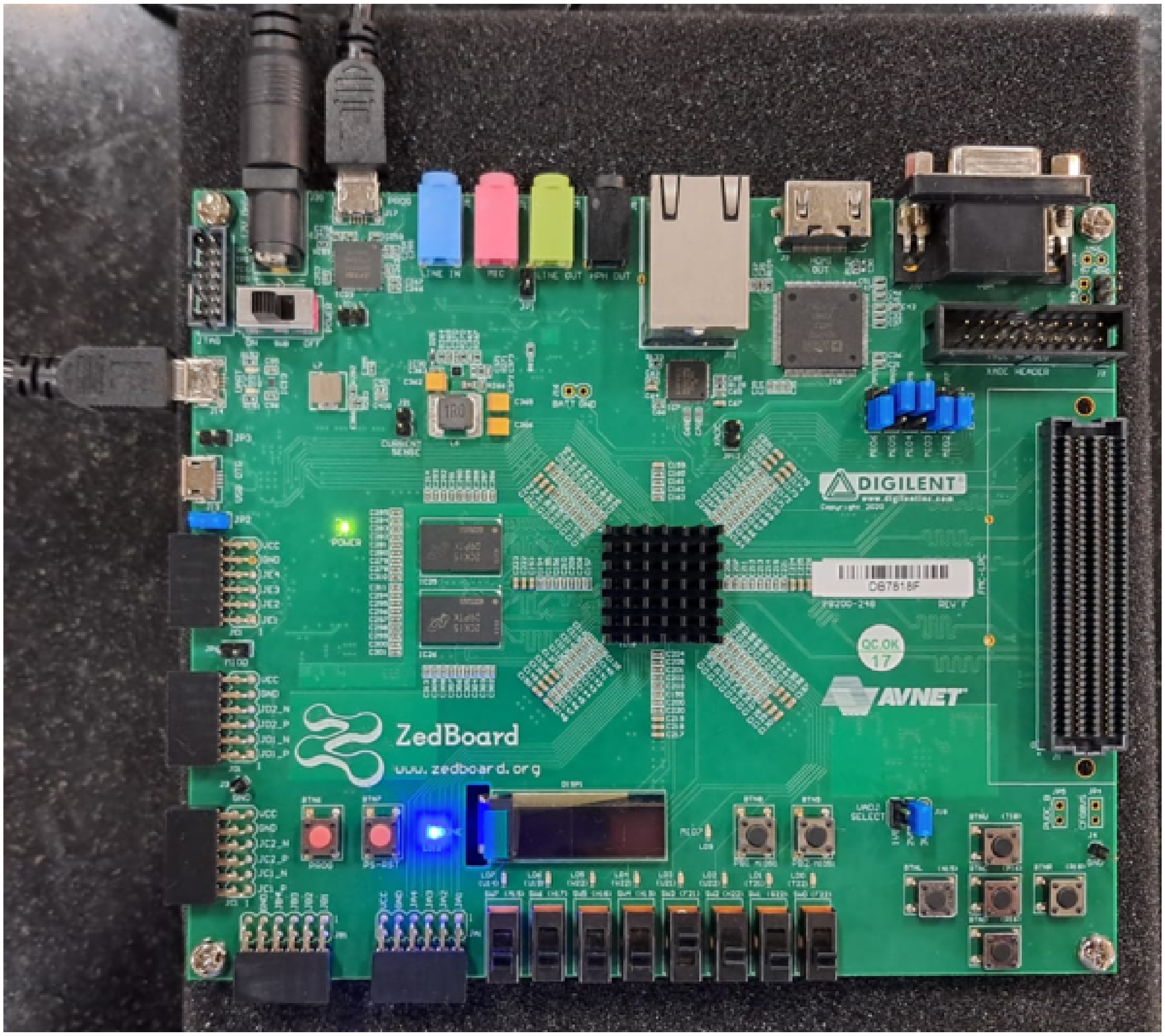


Fig. 8. Zedboard connections

As shown in Figure 8, one connection from the host system to the ZedBoard is through JTAG, which is used to program the FPGA fabric. Another connection is through UART, which enables the processor to transmit output data to the host system.

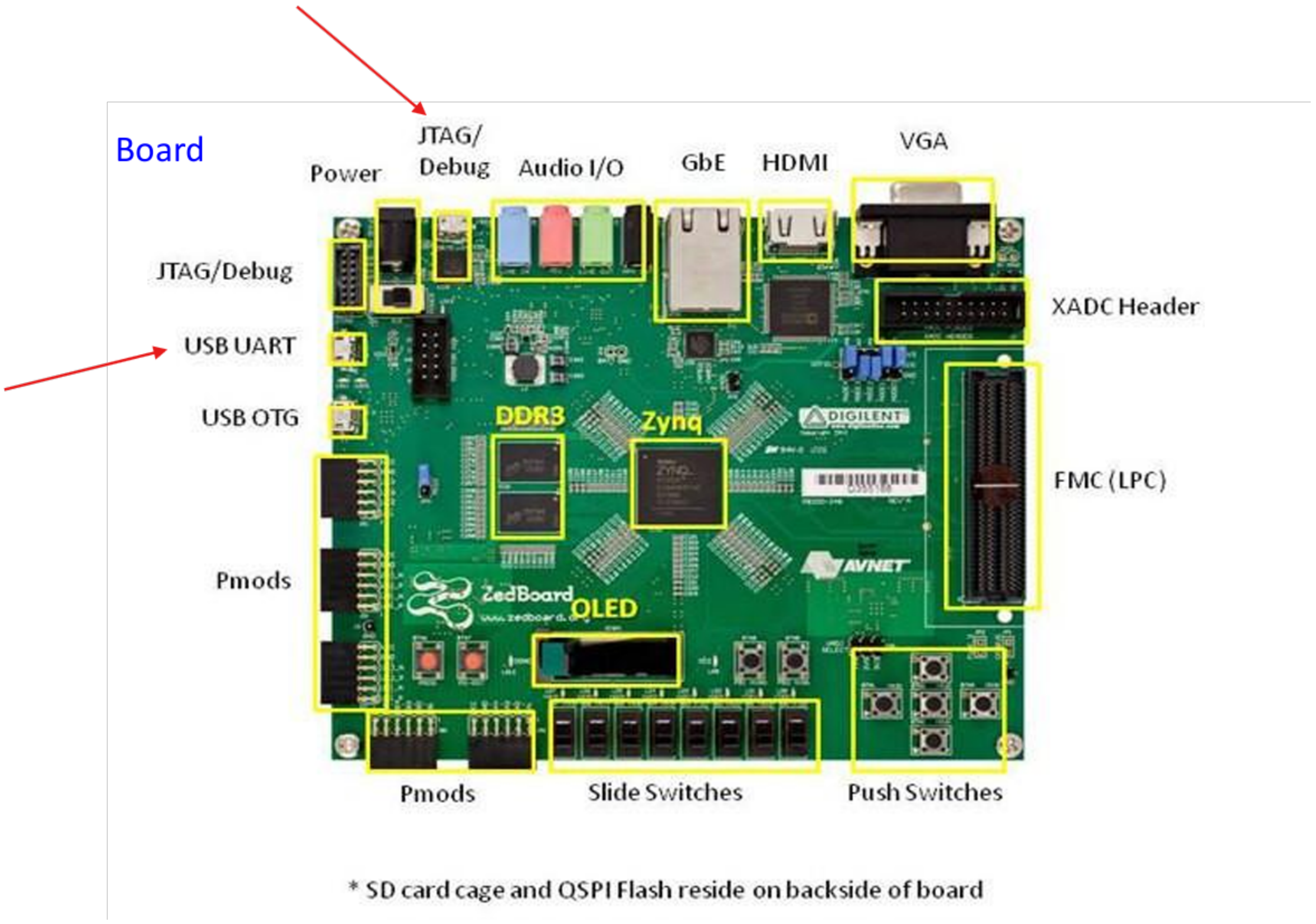


Fig. 9. Zedboard pins

III. Conclusion

A hardware-accelerated FIR filter was successfully implemented on the Zynq-7000 platform using a Vitis HLS generated IP core integrated into the Programmable Logic. AXI4-Stream interfaces were used for input and output communication, and AXI DMA was used to transfer data efficiently between DDR memory and the FIR accelerator. The standalone Vitis application running on the ARM Cortex-A9 processor successfully configured the DMA engine, transmitted input samples, and received filtered output samples.

The results verified correct FIR filter operation, demonstrating the effectiveness of PS–PL hardware acceleration.