Question 4

Understanding different concepts of speed

In this question, you will learn about the different concepts of speed. In this question, you will implement a module to calculate the power of 3 inputs. For each section, you will need to calculate the maximum operating frequency, throughput, and latency after implementation and simulation. Consider the delay of each operator as a parameter in your calculations. Analyze and compare the synthesis results in each section.

Part1)

Explain the meaning of throughput, latency, and maximum frequency and their relationship to the operating speed of the circuit.

In digital circuit design, especially in pipelined and synchronous systems, performance is often evaluated based on three primary metrics: maximum operating frequency (F_m), latency, and throughput. These factors directly influence the operating speed and efficiency of the system. This report summarizes their definitions, interdependencies, and impact on the overall performance of a digital circuit.

Maximum Frequency (F_m)

The maximum frequency represents the highest clock rate at which a circuit can function reliably without timing violations. It is inversely proportional to the critical path delay (T_p) — the longest combinational delay between any two sequential elements (flip-flops or registers).

A shorter critical path allows for a higher clock frequency, which is desirable for improving the system's response time and throughput. Techniques such as pipelining and logic optimization are commonly used to reduce the critical path delay and thus increase F_m.

Latency

Latency is defined as the total time taken for a single input to propagate through the circuit and produce a corresponding output. In pipelined systems, it is typically measured as the number of pipeline stages multiplied by the clock period.

$$\text{Latency}=N\times T=N/F$$

where:

- N is the number of pipeline stages,
- T is the clock period,
- F is the clock frequency.

Latency affects how long a single operation takes from input to output. Although pipelining may increase latency due to more stages, it does not necessarily impact the rate at which outputs are generated (throughput).

Throughput

Throughput refers to the rate at which outputs are produced by the system — essentially how many operations are completed per unit time. In a fully pipelined system, throughput is equal to the clock frequency:

Throughput=F_m

This metric is crucial for high-performance applications where large volumes of data must be processed efficiently. Unlike latency, throughput reflects the system's capacity to handle workload in continuous operation.

Relationship to Circuit Operating Speed

The operating speed of a digital circuit refers to how efficiently and quickly it can process data. This is governed by three interrelated metrics: throughput, latency, and maximum frequency (F_m). Each of these contributes to a different aspect of performance, and their balance defines how "fast" a system truly is.

Throughput and Operating Speed

Throughput represents the rate at which data is processed — typically measured in operations per second (e.g., Hz, Mbps, or ops/s). A circuit with high throughput can handle more data over time, making it ideal for applications involving continuous data streams, such as signal processing or image filtering.

Higher throughput directly translates to higher operating speed in sustained workloads.

Latency and Response Time

Latency is the delay between input and corresponding output. Even if throughput is high, a circuit with long latency may not feel fast, especially in applications where quick feedback or real-time response is critical (e.g., control systems or user interfaces).

Lower latency means faster individual operation speed, which improves responsiveness.

Maximum Frequency (F_m) and Clock Speed

The maximum frequency defines the highest reliable clock speed the circuit can sustain. It sets a hard limit on how fast any part of the system can operate. A higher F_m enables: Shorter clock periods, Faster pipeline execution, Higher throughput when fully utilized.

A higher F_m increases the potential speed ceiling of the circuit.

Metric	Affects	Relevance to Operating Speed
Throughput	Rate of output generation	Directly defines how many operations per second
Latency	Delay from input to output	Defines how fast one operation is completed
F_m	Clocking limit of the circuit	Sets upper bound for throughput and timing

A circuit with high throughput, low latency, and high maximum frequency is considered fast and efficient. These three metrics are interconnected, and optimizing one often impacts the others. Effective circuit design finds a balance to achieve the desired operating speed based on application requirements.

Part2)

Design a module to calculate the cube of an 8-bit number and output it. In this section, your structure must be high-throughput. In other words, it must be able to receive an input and produce a corresponding output on all clocks. Use a for loop to implement.

Here is my Verilog code that calculates the cube of an 8-bit input using a combinational for loop. To analyze the timing characteristics of the circuit, I added a register (flip-flop) at both the input and the output. This ensures the input is sampled synchronously and the output is registered, making the design suitable for timing analysis. Since the core logic is purely combinational, the result is available within a single clock cycle after the input is registered.

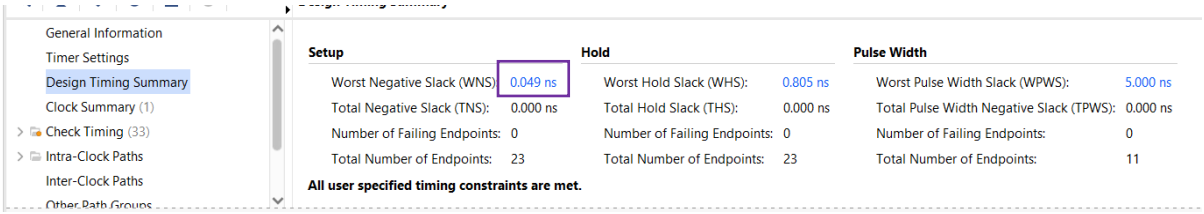
```
module Cube1 (
    input clk,
    input rst,
    input [7:0] in,
    output reg [23:0] out
);

    reg [7:0] in_reg;
    reg [23:0] temp;
    integer i;

    // Input flop
    always @(posedge clk or posedge rst) begin
        if (rst)
            in_reg <= 0;
        else
            in_reg <= in;
    end

    // Combinational logic
    always @(*) begin
        temp = 1;
        for (i = 0; i < 3; i = i + 1) begin
            temp = temp * in_reg;
        end
    end

    // Output flop
    always @(posedge clk or posedge rst) begin
        if (rst)
            out <= 0;
        else
            out <= temp;
    end

endmodule
```

and here is the testbench:

```
`timescale 1ns / 1ps

module Cube1_tb;

    reg clk;
    reg rst;
    reg [7:0] in;
    wire [23:0] out;

    // Instantiate the module under test
    Cube1 uut (
        .clk(clk),
        .rst(rst),
        .in(in),
        .out(out)
    );

    // Clock generation: 11 ns period
    initial begin
        clk = 0;
        forever #5.5 clk = ~clk;
    end

    initial begin
        // Initialize
        rst = 1;
        in = 0;

        // Wait a few cycles with reset high
        #22;
        rst = 0;

        // Apply inputs and observe outputs
        #11 in = 2;   // 2^3 = 8
        #11 in = 3;   // 3^3 = 27
        #11 in = 4;   // 4^3 = 64
        #11 in = 5;   // 5^3 = 125
        #11 in = 10;  // 10^3 = 1000
        #11 in = 255; // Large input

        // Wait before finishing
        #50;

        $finish;
    end

endmodule
```

For timing analysis, I added a constraint file to the project and included the following clock constraint: `create_clock -name clk -period 10.000 [get_ports clk]` `nitially`, I set the clock period to 10 ns (targeting a 100 MHz operating frequency). However, after implementation, the timing summary reported a negative slack of approximately -1 ns, indicating the design was not meeting timing at this frequency.

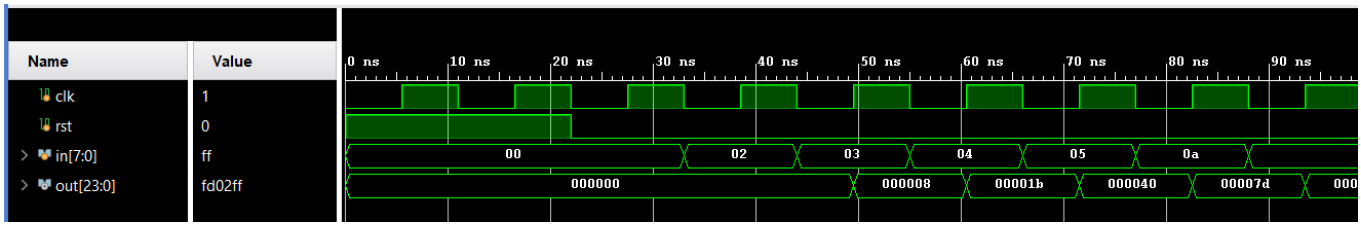
To address this, I adjusted the clock period to 11 ns, which improved the slack and made the circuit timing-viable.



The result implies that the **critical path delay** is roughly **11 ns**, and thus the **maximum operating frequency** is:

OperatingFrequency = 1 / 11 ns = 90.91 MHz

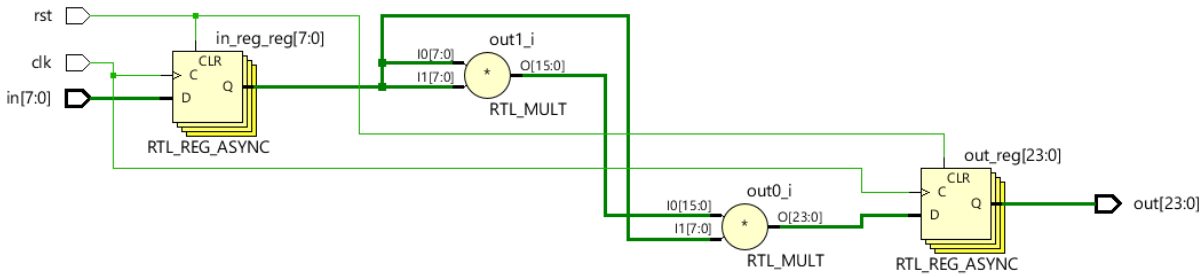
As required, the **latency** of the circuit is **1 clock cycle** (11 ns), which is confirmed by observing the waveform:



Because the output is registered ("flopped out"), we observe a one-clock-cycle delay between the input and the corresponding output.

The **throughput** of the circuit is **24 bits per clock**, or: **24 bits / 11 ns ≈ 2.18 Gbps**

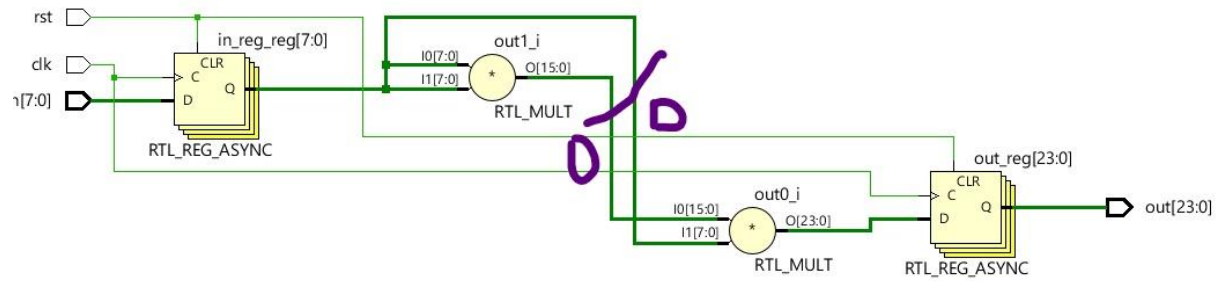
Here is the schematic of the circuit:



Part3)

By opening the loop and pipeline, change your structure to achieve the highest operating frequency.

Based on the previous code schematic I add FFs on this cutset in the figure:

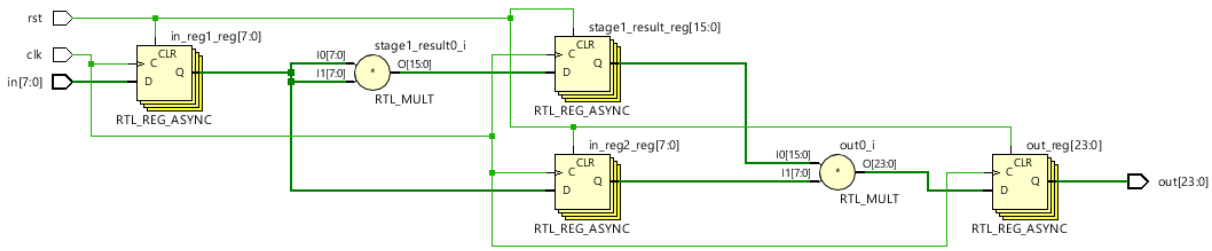


Here is the code that now has 4 FFs and includes almost 2 stages:

```
module Cube1_pipeline (
    input clk,
    input rst,
    input [7:0] in,
    output reg [23:0] out
);

    reg [7:0] in_reg1,in_reg2;
    reg [15:0] stage1_result; // intermediate: in * in

    // Stage 0: Register input
    always @(posedge clk or posedge rst) begin
        if (rst)
            in_reg1 <= 0;
        else
            in_reg1 <= in;
    end

    // Stage 1: Square the input (in * in) and again register input
    always @(posedge clk or posedge rst) begin
        if (rst)
            stage1_result <= 0;
        else
            stage1_result <= in_reg1 * in_reg1;


    end
   always @(posedge clk or posedge rst) begin
        if (rst)
            in_reg2 <= 0;
        else
```

```
            in_reg2 <= in_reg1;
      end
    // Stage 2: Multiply again (square * in) and register output
    always @(posedge clk or posedge rst) begin
        if (rst)
            out <= 0;
        else
            out <= stage1_result * in_reg2;
    end



endmodule
```

here is the schematic:



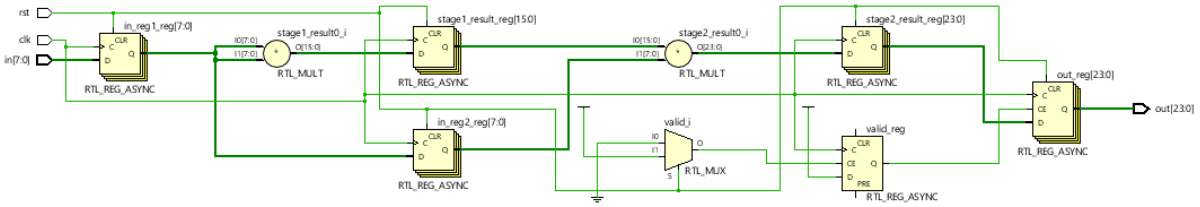
which is exactly what we wanted.

But in waveform we see the out is x for some time we need to consider a valid flag to have valid data in out at any time so here is the new code with extra registers:

```
module Cube1_pipeline (
    input clk,
    input rst,
    input [7:0] in,
    output reg [23:0] out
);

    reg [7:0] in_reg1,in_reg2;
    reg [15:0] stage1_result; // intermediate: in * in
    reg valid;
    reg [23:0] stage2_result;

    // Stage 0: Register input
    always @(posedge clk or posedge rst) begin
        if (rst)
            in_reg1 <= 0;
        else
```

```
            in_reg1 <= in;
    end

    // Stage 1: Square the input (in * in) and again register input
    always @(posedge clk or posedge rst) begin
        if (rst)
            stage1_result <= 0;
        else
            stage1_result <= in_reg1 * in_reg1;


    end
   always @(posedge clk or posedge rst) begin
        if (rst)
            in_reg2 <= 0;
        else
            in_reg2 <= in_reg1;
      end
    // Stage 2: Multiply again (square * in) and register output
    always @(posedge clk or posedge rst) begin
        if (rst )
            stage2_result <= 0;
        else
        begin
            stage2_result <= stage1_result * in_reg2;
            valid <= 1'b1;
        end

   // Stage 3: register output
    end
     always @(posedge clk or posedge rst) begin
        if (rst)
            out <= 0;
        else if(valid==1)

            out <= stage2_result;



    end



endmodule
```

and here is the schematic:



p.s: I didn’t change the testbench and I used the same one for all the parts for better comparison.

now let's get to the timing analysis:

for 11ns we get this result:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.811 ns	Worst Hold Slack (WHS):	0.161 ns	Worst Pulse Width Slack (WPWS):	5.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	69	Total Number of Endpoints:	69	Total Number of Endpoints:	57

All user specified timing constraints are met.

we have almost 4 ns slack time so I try again with 7ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.811 ns	Worst Hold Slack (WHS):	0.161 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	69	Total Number of Endpoints:	69	Total Number of Endpoints:	57

All user specified timing constraints are met.

try again with 6ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.189 ns	Worst Hold Slack (WHS):	0.161 ns	Worst Pulse Width Slack (WPWS):	2.500 ns
Total Negative Slack (TNS):	-0.719 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	5	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	69	Total Number of Endpoints:	69	Total Number of Endpoints:	57

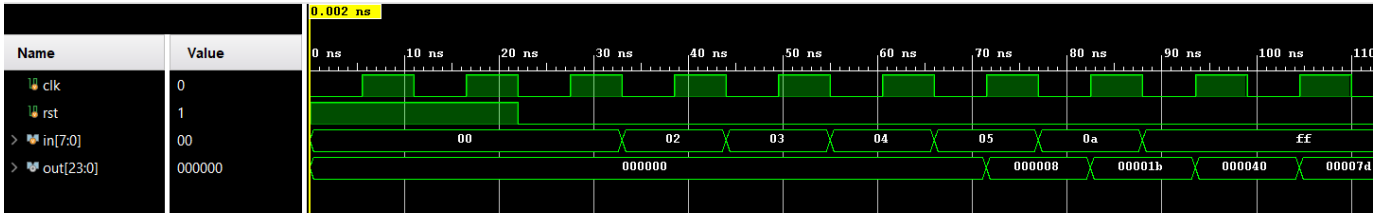
Timing constraints are not met.

trying 6.2 ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.011 ns	Worst Hold Slack (WHS):	0.161 ns	Worst Pulse Width Slack (WPWS):	2.600 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	69	Total Number of Endpoints:	69	Total Number of Endpoints:	57

All user specified timing constraints are met.

so, the critical path time is **6.2ns**. => operating frequency = 1/6.2ns = **161.29 MHz**



as we can see the latency is now 3 clock cycles but the throughput is still 24 bits per cycle or 24.18Mbits/s

(if we didn't have to use valid flag, we would have 2 cycle latency)

Part4)

Change the structure of the second section to achieve the smallest possible area. To do this, you need to use resource sharing.

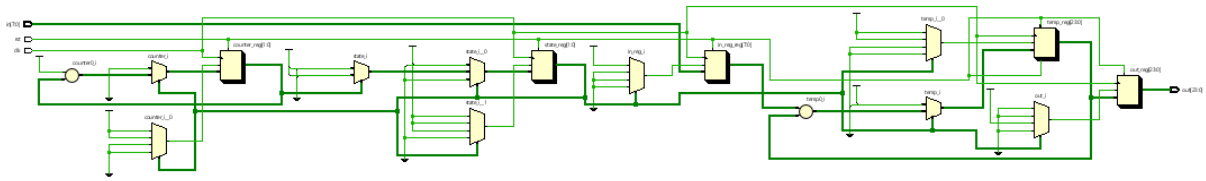
We need to minimize our resources from 2 multipliers to 1 multiplier. In order to do that I used a counter and some muxes to control the input of the.

Here is my code:

```
module Cube1_shared_minimal (
    input wire clk,
    input wire rst,
    input wire [7:0] in,
    output reg [23:0] out
);

    reg [1:0] state;
    reg [1:0] counter;
    reg [23:0] temp;
    reg [7:0] in_reg;
    reg [23:0] temp_out;
    reg valid;

    parameter  IDLE = 2'b00,
               MULT = 2'b01,
               DONE = 2'b10;

    always@(*) begin
    temp_out <= in_reg * temp;
    end

    always @(posedge clk) begin
        if(rst) begin
            state <= IDLE;
            temp <= 24'd1;
            out <= 0;
            in_reg <= 0;
            counter <= 0;
            valid <= 0;
        end
        else begin
        case (state)

             IDLE: begin
                       in_reg <= in;
                       temp <= 24'd1;
                       counter <= 0;
```

```
                        state <= MULT;

             end
            MULT: begin
                    temp <= temp_out;
                    counter <= counter + 1;
                    if ( counter == 2'b10 ) begin

                         state <= DONE;
                         valid <= 1;
                         end
                    else state <= MULT;
             end
            DONE:
                   Begin
                    If(valid) Begin
                    out <= temp;
                    state <= IDLE;
                   end
                   end
        endcase
        end
    end
endmodule
```

Here is the schematic:



timing analysis:

for 10ns we get this result:

we have almost 3.6 ns slack time so I try again with 7ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.347 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	2.500 ns
Total Negative Slack (TNS):	-8.323 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	24	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	103	Total Number of Endpoints:	103	Total Number of Endpoints:	53

Timing constraints are not met.

All user specified timing constraints are met.

try again with 6ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.653 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	103	Total Number of Endpoints:	103	Total Number of Endpoints:	53

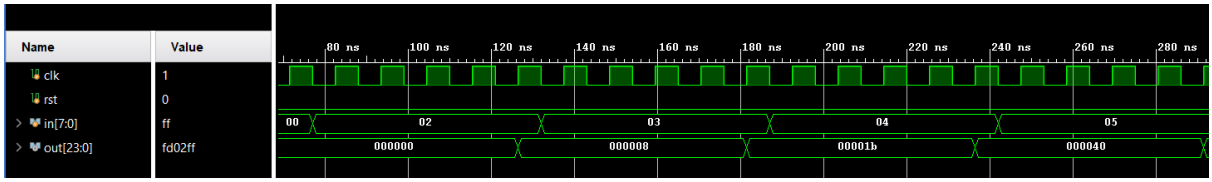
All user specified timing constraints are met.

trying 6.4 ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.053 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	2.700 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	103	Total Number of Endpoints:	103	Total Number of Endpoints:	53

All user specified timing constraints are met.

so, the critical path time is **6.4ns**. => operating frequency = 1/6.4ns = **156.25 MHz**



as we can see the latency is now 4 clock cycles but the throughput is 24 bits per 4 cycles or 6 Mbits/s.

resource analysis:

Name	Slice LUTs (17600)	Slice Registers (35200)	DSPs (80)	Bonded IOB (54)	BUFGCTRL (32)
N Cube1_pipeline	79	55	1	34	1

Name	Slice LUTs (17600)	Slice Registers (35200)	DSPs (80)	Bonded IOB (54)	BUFGCTRL (32)
N Cube1_shared_minimal	29	52	1	34	1

As we can see the resources used is decreased.

part5)

What conclusions do you draw by comparing the time parameters in each section?

In the base version of the Cube1 module, all operations happen sequentially in a single cycle, resulting in **low latency** but **low Fmax** due to a long combinational path, and **high throughput**. In the pipelined version, each multiplication is split into stages, allowing for a **higher Fmax** and **higher throughput**, since new inputs can be accepted every cycle, but at the cost of **increased latency** due to multiple clock cycles per result. The resource-shared version reuses a single multiplier across multiple cycles, leading to **low area usage**, **lower Fmax** due to control overhead and sequential dependencies, **high latency**, and **lower throughput** compared to the pipelined design.

Question 5

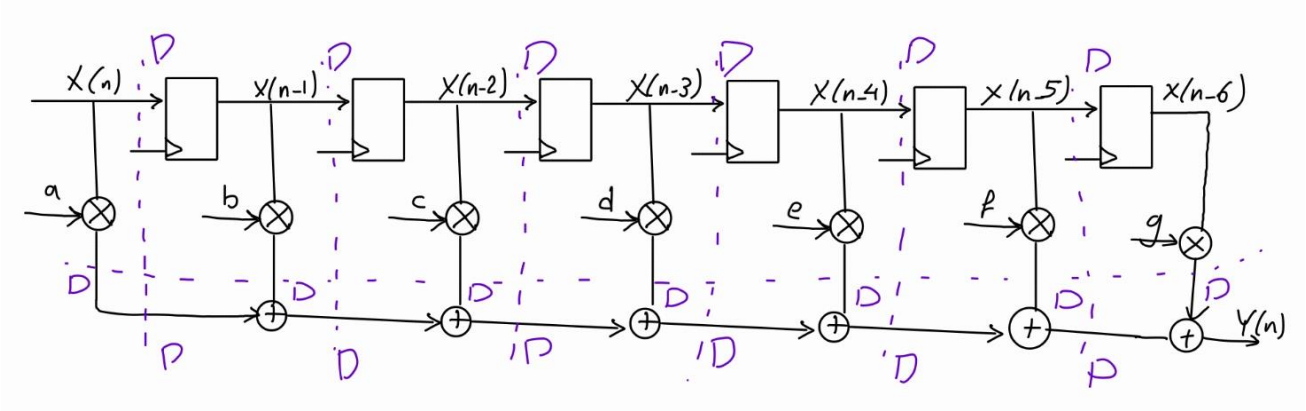
Optimization of FIR filter for different conditions

In this question, implement an FIR filter and optimize the implemented structure for different conditions.

Part1)

Draw a FIR structure with 7 coefficients and pipeline it to safety.

Here is the figure:



In this circuit the critical path is now 1 multiplier where in unpipelined version is 1 multiplier and 6 adders.

Part2)

Implement a 7-coefficient FIR filter with an architectural drawing. Suppose the coefficient values are 8 bits and to determine the coefficients, we have three inputs: the coefficient value, Tlast, indicating the last loaded coefficient, a bit to change the desired coefficient value called writeen. The coefficients desired by the user are loaded into the filter in the form of packet and at the moment the last coefficient is loaded, tlast is set; if this does not happen, the entire packet is discarded. This filter receives an 8-bit input as a stream and the output is also a stream; calculate how many bits the maximum output is with these characteristics and consider the same number of bits for it.

The design allows for runtime coefficient loading via a packetized interface using three control signals:

- coef_val: 8-bit coefficient value
- writeen: enables writing the coefficient
- tlast: marks the last coefficient in the sequence

If tlast is asserted before all seven coefficients are written, the entire set is discarded.

The FIR filter is based on the direct-form structure, which processes input data through a shift register delay line, multiplies each delayed sample by a corresponding coefficient, and finally adds all the products to compute the output.

Key Components:

Coefficient Loader:

A 3-bit index counter writes incoming coefficients into a 7-element register file. If tlast is set and exactly 7 coefficients have been loaded, the valid_coeffs flag is set; otherwise, coefficients are discarded.

Shift Register Delay Line:

A 7-element shift register (shift_reg) stores the latest input samples.

Output Computation:

The output is computed by multiplying each shift register element by its corresponding coefficient and summing all the results. This occurs in one clock cycle when coefficients are valid.

Bit Width Justification:

Input: 8 bits

Coefficients: 8 bits

Output: 18 bits

This ensures safe accommodation of the sum of seven 8×8-bit multiplications (each result up to 16 bits, total max sum ~455,175), without overflow.

Each tap: 8-bit coef × 8-bit input = 16-bit result.

Adding 7 of these: $\log_2(7\times255\times255) \approx 18$ bits.

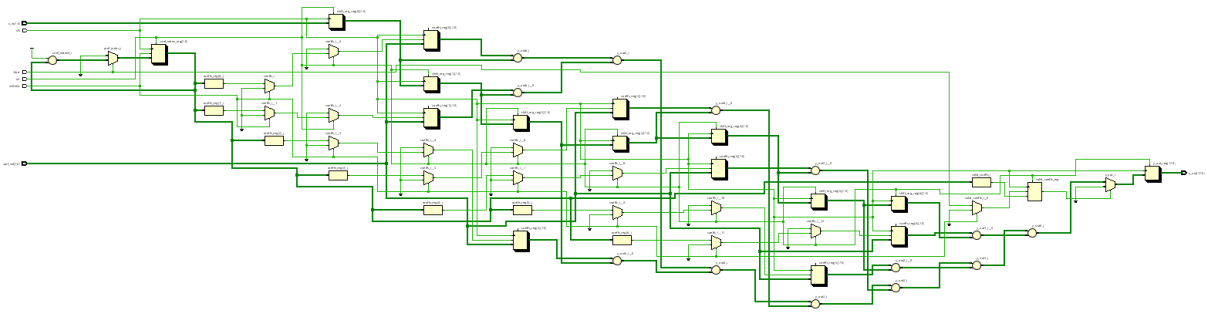
So output: 18 bits max.

Here is the code:

```
module fir_filter_7tap (
    input clk,
    input rst,
    input [7:0] x_in,
    input [7:0] coef_val,
    input writeen,
    input tlast,
    output reg [17:0] y_out
);
    // Internal Coefficient Registers
    reg [7:0] coeffs[0:6];
    reg [2:0] coef_index;
    reg valid_coeffs;

    // Input Shift Register
    reg [7:0] shift_reg[0:6];

    integer i;

    // Load Coefficients
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            coef_index <= 0;
            valid_coeffs <= 0;
        end else if (writeen) begin
            coeffs[coef_index] <= coef_val;
            coef_index <= coef_index + 1;
            if (tlast) begin
                if (coef_index == 6) begin
                    valid_coeffs <= 1;
                end else begin
                    valid_coeffs <= 0; // discard
                end
                coef_index <= 0;
            end
        end
    end

    // Shift input samples
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 7; i = i + 1)
                shift_reg[i] <= 0;
        end else begin
            for (i = 6; i > 0; i = i - 1)
                shift_reg[i] <= shift_reg[i-1];
            shift_reg[0] <= x_in;
        end
    end

    // Output calculation
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            y_out <= 0;
        end else if (valid_coeffs) begin
            y_out <= coeffs[0] * shift_reg[0] +
                     coeffs[1] * shift_reg[1] +
                     coeffs[2] * shift_reg[2] +
                     coeffs[3] * shift_reg[3] +
                     coeffs[4] * shift_reg[4] +
                     coeffs[5] * shift_reg[5] +
                     coeffs[6] * shift_reg[6];
        end else begin
            y_out <= 0;
        end
    end
endmodule
```

here is the schematic:



its not visible enough but we have 7 multipliers, 6 adders + 1 for the counter, 16 flops (7 for the coefficients 6 for the input 2 for reg in reg out and 1 for counter) which is exactly what we wanted.

To demonstrate the correct operation of the implemented structure, write a testbench and set the input to 1 only on one clock edge and zero at the rest of the times; The outputs should be the values of the coefficients. For this section, it is necessary to assume that the additions and multiplications required to calculate the outputs are performed in one clock.

The testbench is designed to validate the functional correctness of the fir_filter_7tap module. It specifically checks:

- The correct reception and storage of coefficient values
- The expected impulse response of the filter once coefficients are loaded and valid
- Proper behavior of control signals (writeen, tlast)
- Correct output size and timing

Test Procedure

Initialization: All inputs set to 0 and rst asserted for 10 ns

Coefficient Loading:

Coefficients are applied sequentially: 10, 20, 30, 40, 50, 60, 70

Each coefficient is sent with writeen = 1

tlast is set high only with the final coefficient(٧٠)

If tlast is not set correctly, the filter ignores the coefficients (safety mechanism)

Impulse Input:

A single non-zero input (x_in = 8'd1) is applied for one clock cycle

All following inputs are zeros

This simulates a unit impulse to test the FIR filter’s impulse response

Output Monitoring:

Outputs are printed to the console using $monitor

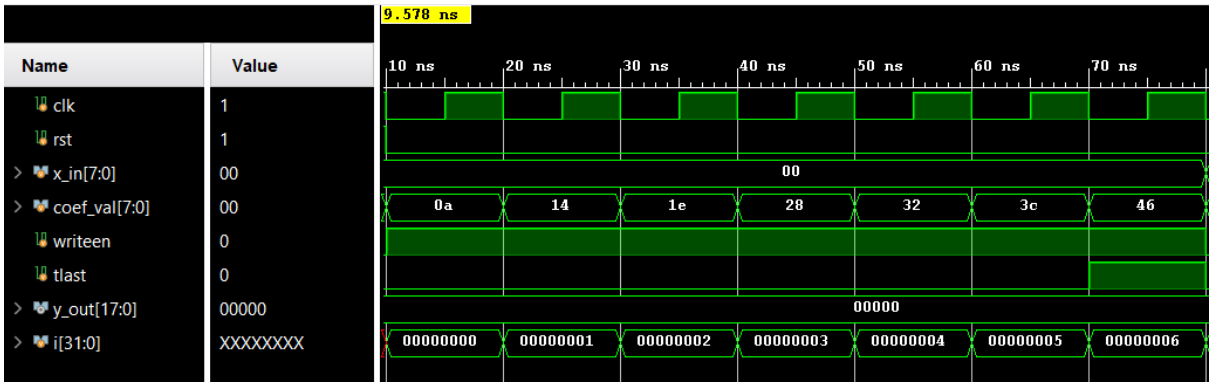
Here is the testbench:

```
module tb_fir_filter_7tap;
    reg clk, rst;
    reg [7:0] x_in, coef_val;
    reg writeen, tlast;
    wire [17:0] y_out;

    fir_filter_7tap uut (
        .clk(clk), .rst(rst),
        .x_in(x_in),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
```

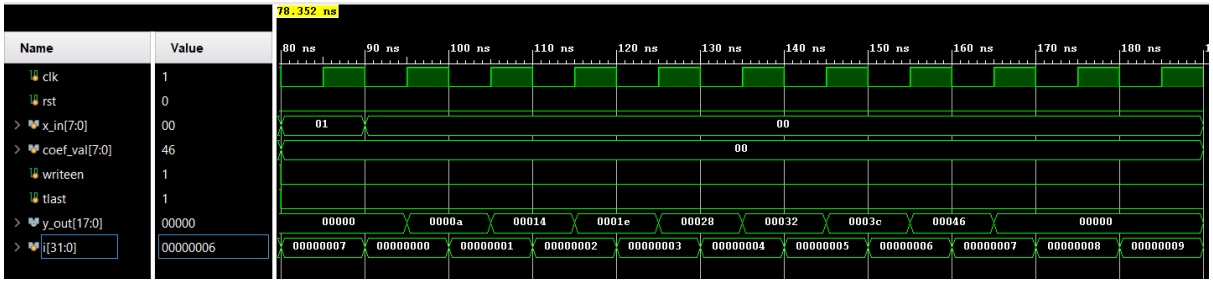
```
        .y_out(y_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;

    integer i;

    initial begin
        $display("Time\tOutput");
        $monitor("%0d\t%d", $time, y_out);

        rst = 1;
        x_in = 0;
        coef_val = 0;
        writeen = 0;
        tlast = 0;
        #10;
        rst = 0;

        // Load coefficients: 10, 20, ..., 70
        for (i = 0; i < 7; i = i + 1) begin
            coef_val = (i+1)*10;
            writeen = 1;
            tlast = (i == 6);
            #10;
        end

        // Disable writing
        writeen = 0;
        tlast = 0;
        coef_val = 0;

        // Apply impulse input (1 followed by 0s)
        x_in = 8'd1; #10;
        x_in = 8'd0;

        // Wait and observe outputs (should match coefficients)
        for (i = 0; i < 10; i = i + 1)
            #10;

        $finish;
    end
endmodule
```

here is the wave form:

loading coefficients:



asserting 1:



TCL console:

the results confirm our observation.

```
Time      Output
0          0
95        10
105       20
115       30
125       40
135       50
145       60
155       70
165        0
$finish called at
```

Part3)

Investigate the symmetry of the coefficients and the modified structure for the time filter. While presenting your structure for a filter with 7 coefficients, write its code and simulate it with the test mentioned in Part B. Draw the architecture of the final structure before implementation.

The goal of this part is to explore the symmetry in FIR filter coefficients and implement an optimized architecture for a **7-tap symmetric FIR filter**. The final design should reduce redundant computations by utilizing coefficient symmetry and reusing partial results, followed by simulation with the impulse test from Part 2.

For a linear-phase FIR filter, if the coefficients are symmetric , the circuit can be optimized. Given a 7-tap filter with symmetric coefficients a, b, c, d, c, b, a, the standard filter:

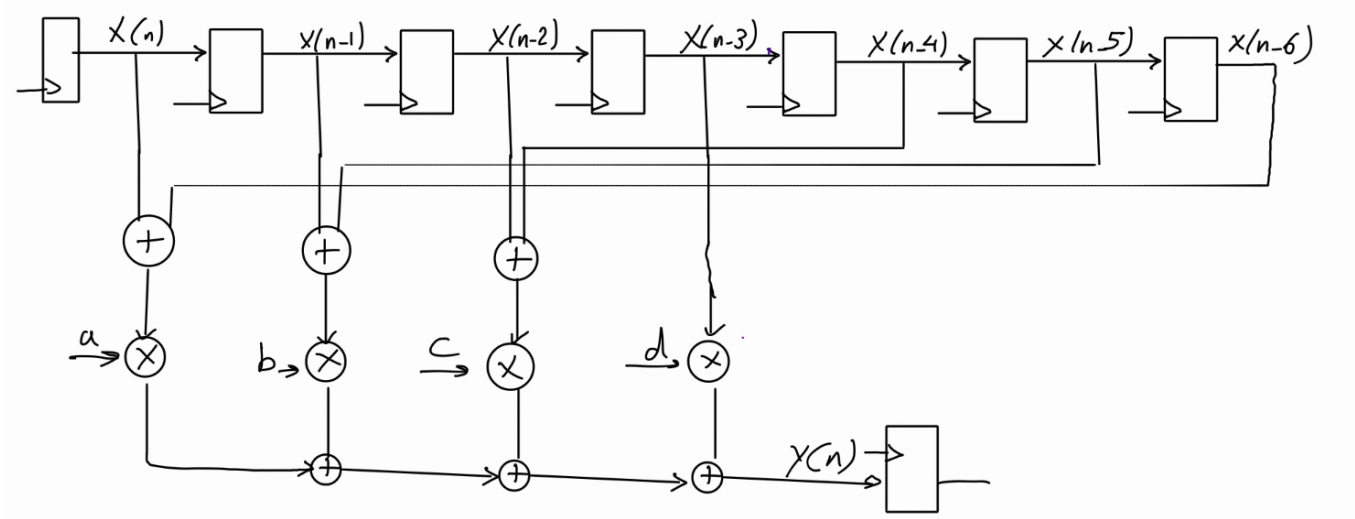
```
y[n] = a·x[n] + b·x[n-1] + c·x[n-2] + d·x[n-3] + c·x[n-4] + b·x[n-5] +
a·x[n-6]
```

can be reorganized as:

```
y[n] = a·(x[n] + x[n-6]) +
       b·(x[n-1] + x[n-5]) +
       c·(x[n-2] + x[n-4]) +
       d·x[n-3]
```

This reduces the number of multiplications from **7 to 4**, significantly optimizing hardware resource usage.

Here is my design:



The architecture is designed to:

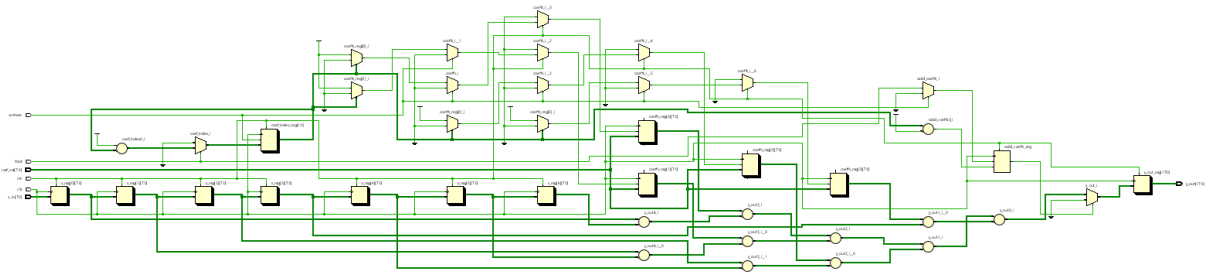
- Store only 4 unique coefficients
- Maintain a 7-element input shift register for storing x[n] to x[n−6].
- Use adders to compute symmetric input pairs.
- Use 4 multipliers for computing partial products.
- Accumulate the final output in a single clock cycle.

Here is my code:

```
module fir_filter_symmetric (
    input clk,
    input rst,
    input [7:0] x_in,
    input [7:0] coef_val,
    input writeen,
    input tlast,
    output reg [17:0] y_out
);
    // Store 4 unique symmetric coefficients
    reg [7:0] coeffs[0:3];
    reg [1:0] coef_index;
    reg valid_coeffs;

    // Shift register for inputs
    reg [7:0] x[0:6];  // x[0] = x[n], x[6] = x[n-6]

    integer i;

    // Coefficient loading
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            coef_index <= 0;
            valid_coeffs <= 0;
        end else if (writeen) begin
            coeffs[coef_index] <= coef_val;
            coef_index <= coef_index + 1;
            if (tlast) begin
                valid_coeffs <= (coef_index == 3); // should be 4 values
                coef_index <= 0;
            end
        end
    end

    // Input shift register
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 7; i = i + 1)
```

```
                x[i] <= 0;
        end else begin
            for (i = 6; i > 0; i = i - 1)
                x[i] <= x[i-1];
            x[0] <= x_in;
        end
    end

    // Output calculation
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            y_out <= 0;
        end else if (valid_coeffs) begin
            y_out <= coeffs[0] * (x[0] + x[6]) +
                     coeffs[1] * (x[1] + x[5]) +
                     coeffs[2] * (x[2] + x[4]) +
                     coeffs[3] * x[3];
        end else begin
            y_out <= 0;
        end
    end
endmodule
```

here is the schematic of the design:



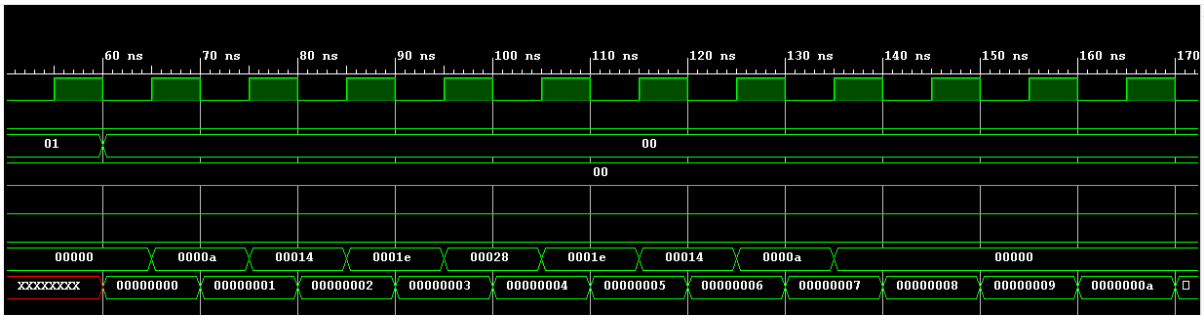
it's not visible enough but we have 4 multipliers, 6 adders + 1 for the counter, 13 flops (4 for the coefficients 6 for the input 2 for reg in reg out and 1 for counter) which is exactly what we wanted.

here is the modifies testbench with symmetric coefficients:

```
module tb_fir_filter_symmetric;
    reg clk, rst;
    reg [7:0] x_in, coef_val;
    reg writeen, tlast;
    wire [17:0] y_out;

    fir_filter_symmetric uut (
```

```
        .clk(clk), .rst(rst),
        .x_in(x_in),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
        .y_out(y_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;

    integer i;

    initial begin
        $display("Time\tOutput");
        $monitor("%0d\t%d", $time, y_out);

        rst = 1;
        x_in = 0;
        coef_val = 0;
        writeen = 0;
        tlast = 0;
        #10;
        rst = 0;

        // Load symmetric coefficients: a=10, b=20, c=30, d=40
        // Final coefficients: [10 20 30 40 30 20 10]
        coef_val = 8'd10; writeen = 1; tlast = 0; #10;
        coef_val = 8'd20; writeen = 1; tlast = 0; #10;
        coef_val = 8'd30; writeen = 1; tlast = 0; #10;
        coef_val = 8'd40; writeen = 1; tlast = 1; #10;

        // Disable writing
        writeen = 0;
        tlast = 0;
        coef_val = 0;

        // Apply impulse input (1 followed by 0s)
        x_in = 8'd1; #10;
        x_in = 8'd0;

        // Wait and observe outputs
        for (i = 0; i < 20; i = i + 1)
            #10;
        $finish;
    end
endmodule
```

loading coefficients is the same but here is the asserting 1 wave form:



TCL console:

the results confirm our observation.

```
Time     Output
0         0
65        10
75        20
85        30
95        40
105       30
115       20
125       10
135        0
$finish called at t:
```

Part4)

State the main advantage of the proposed structure in Part 3 over Part 2 (assuming the coefficients are symmetrical).

The primary advantage of the proposed structure in Part 3 is the optimized utilization of symmetry in the FIR filter coefficients, which leads to significant hardware resource reduction without compromising output accuracy or performance.

Specifically:

Fewer Multipliers:

Part 3 uses only 4 multipliers compared to 7 in Part 2. Since multipliers are among the most resource-intensive components in digital design, this reduction greatly improves area efficiency and power consumption.

Reduced Flip-Flops (Registers):

The symmetric design requires storing only 4 unique coefficients, so it uses 13 flip-flops in total versus 16 in Part 2. This is especially valuable for FPGA or ASIC implementations with limited register resources.

Same Throughput:

Despite reducing computation, the structure still calculates one output per clock cycle — just like Part 2 — maintaining the same throughput.

Adders Stay Constant:

Both structures use 6 main adders (for combining input pairs or accumulating results) plus 1 counter-related adder, so no compromise is made on summing efficiency.

Power and Area Efficiency:

By reducing the number of multipliers and registers, the proposed structure in Part 3 inherently consumes less dynamic power and silicon area, making it a better choice for embedded or portable systems.

Summary Table

Metric	Part 2	Part 3	Gain
Multipliers	7	4	−43%
Adders	7	7	—
Flip-Flops	16	13	−19%
Coefficients Stored	7	4	−43%
Clock Cycles/Output	1	1	=

Part5)

Assume that the coefficients take only the values 1, 0, and −1. Optimize the filter structure designed in Part 2.

The goal of this part was to optimize the 7-tap FIR filter structure designed in Part 2 under the assumption that the filter coefficients take values only from the set {1, 0, −1}. This assumption allows us to significantly reduce the hardware complexity, particularly by eliminating general-purpose multipliers.

In traditional FIR filter structures, each tap consists of a multiplier and an adder. However, when coefficients are limited to 1, 0, or -1, multiplication can be completely avoided by using conditional addition or subtraction:

- A coefficient of 1 implies a direct pass-through (i.e., addition).
- A coefficient of -1 implies subtraction.
- A coefficient of 0 implies that the input value is discarded.

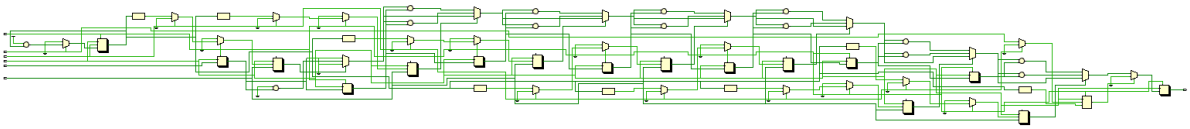
This strategy replaces seven multipliers with a conditional accumulation mechanism. This significantly reduces hardware resource usage, particularly in FPGA designs where multipliers are costly.

Here is the code:

```
module fir_filter_7tap_sparse (
    input clk,
    input rst,
    input [7:0] x_in,
    input signed [7:0] coef_val,
    input writeen,
    input tlast,
    output reg signed [17:0] y_out
);
    // Coefficient Registers
    reg signed [7:0] coeffs[0:6];
    reg [2:0] coef_index;
    reg valid_coeffs;
    reg signed [17:0] acc;

    // Input Shift Register
    reg [7:0] shift_reg[0:6];
    integer i;

    // Load Coefficients
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            coef_index <= 0;
            valid_coeffs <= 0;
        end else if (writeen) begin
```

```
            coeffs[coef_index] <= coef_val;
            coef_index <= coef_index + 1;
            if (tlast) begin
                if (coef_index == 6)
                    valid_coeffs <= 1;
                else
                    valid_coeffs <= 0;
                coef_index <= 0;
            end
        end
    end

    // Shift Input Samples
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 7; i = i + 1)
                shift_reg[i] <= 0;
        end else begin
            for (i = 6; i > 0; i = i - 1)
                shift_reg[i] <= shift_reg[i-1];
            shift_reg[0] <= x_in;
        end
    end

    // Output Calculation (optimized for ±1, 0 coefficients)
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            y_out <= 0;
        end else if (valid_coeffs) begin

            acc = 0;
            for (i = 0; i < 7; i = i + 1) begin
                case (coeffs[i])
                    8'sd1: acc = acc + shift_reg[i];
                    -8'sd1: acc = acc - shift_reg[i];
                    default: acc = acc;
                endcase
            end
            y_out <= acc;
        end else begin
            y_out <= 0;
        end
    end
endmodule
```

here is the schematic:



The optimized filter maintains a shift register of seven past input values and a register bank for the seven coefficients. On each clock cycle, the input value is shifted into the shift register. If valid coefficients are loaded, the output is computed based on the logic described above.

The new design contains:

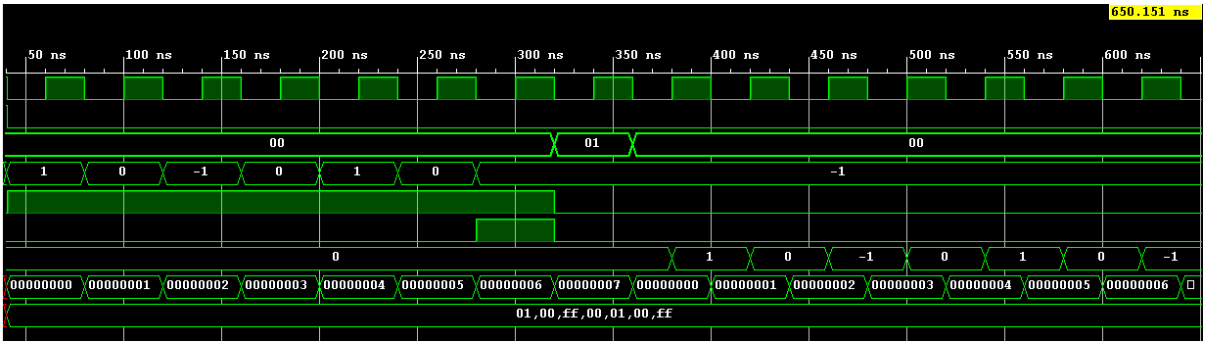
- No multipliers (replaced by conditional add/sub operations or MUXes),
- 6 adders/subtractors in the form of logic inside a case statement,
- 7 registers for storing coefficients,
- 7 registers for the shift register (inputs),
- 2 reg in reg out,
- A total of 16 registers

The design was tested using a custom testbench that:

```
module tb_fir_filter_signed_optimized;
    reg clk, rst;
    reg [7:0] x_in;
    reg signed [7:0] coef_val;
    reg writeen, tlast;
    wire signed [17:0] y_out;

    fir_filter_7tap_sparse uut (
        .clk(clk), .rst(rst),
        .x_in(x_in),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
        .y_out(y_out)
    );

    // Clock generation
    initial clk = 0;
    always #20 clk = ~clk;

    integer i;
    reg signed [7:0] coeff_set[0:6];
```

```
    initial begin
        $display("Time\tOutput");
        $monitor("%0d\t%d", $time, y_out);

        rst = 1;
        x_in = 0;
        coef_val = 0;
        writeen = 0;
        tlast = 0;
        #40;
        rst = 0;

        // Example coefficients: {1, 0, -1, 0, 1, 0, -1}
        coeff_set[0] = 1;
        coeff_set[1] = 0;
        coeff_set[2] = -1;
        coeff_set[3] = 0;
        coeff_set[4] = 1;
        coeff_set[5] = 0;
        coeff_set[6] = -1;

        // Load coefficients
        for (i = 0; i < 7; i = i + 1) begin
            coef_val = coeff_set[i];
            writeen = 1;
            tlast = (i == 6);
            #40;
        end

        writeen = 0;
        tlast = 0;

        // Apply impulse input
        x_in = 8'd1; #40;
        x_in = 8'd0;

        // Wait and observe outputs
        for (i = 0; i < 10; i = i + 1)
            #40;

        $finish;
    end
endmodule
```

Wave form:



TCL Console:

Test outputs confirm that the design behaves correctly, producing expected values based on the coefficient pattern and input data.

```
Time      Output
0           0
380         1
420         0
460        -1
500         0
540         1
580         0
620        -1
660         0
$finish called at
```

Advantages of the Optimized Design

- Eliminates multipliers: Saves significant hardware resources.
- Power-efficient: Lower switching activity compared to multiplier-based design.
- Simpler logic: Easier to debug and verify.
- Ideal for sparse coefficient filters, commonly used in decimation, interpolation, and simple edge-detection kernels in signal processing.

Part6)

Find the maximum possible frequency for the structure implemented in part 3.

Following the same procedure as Question for I start with 10ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-2.570 ns	Worst Hold Slack (WHS):	0.148 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-18.971 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	11	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	101	Total Number of Endpoints:	101	Total Number of Endpoints:	110

Timing constraints are not met.

trying 12.6 ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.042 ns	Worst Hold Slack (WHS):	0.148 ns	Worst Pulse Width Slack (WPWS):	5.800 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	101	Total Number of Endpoints:	101	Total Number of Endpoints:	110

All user specified timing constraints are met.

⇨ maximum operating frequency is 1/12.6ns = **79.37 MHz.**

but I think I should find this frequency for part2 code because in next part we should compare them therefore:

trying 10ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-5.033 ns	Worst Hold Slack (WHS):	0.152 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-46.715 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	15	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	126	Total Number of Endpoints:	126	Total Number of Endpoints:	135

Timing constraints are not met.

trying 15.1ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.067 ns	Worst Hold Slack (WHS):	0.152 ns	Worst Pulse Width Slack (WPWS):	7.050 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	126	Total Number of Endpoints:	126	Total Number of Endpoints:	135

All user specified timing constraints are met.

⇨ maximum operating frequency is 1/15.1ns = **66.23 MHz**

Part7)

Now, by pipelining the structure of part 2, obtain the highest possible frequency of the implemented circuit and justify the difference. At this stage, you also need to ensure the correct operation of your circuit by running a bench test. In this section, you also need to draw the architecture of the desired structure before implementation.

Original Design Analysis

The original FIR filter implementation (Part 2) consisted of a straightforward design with:

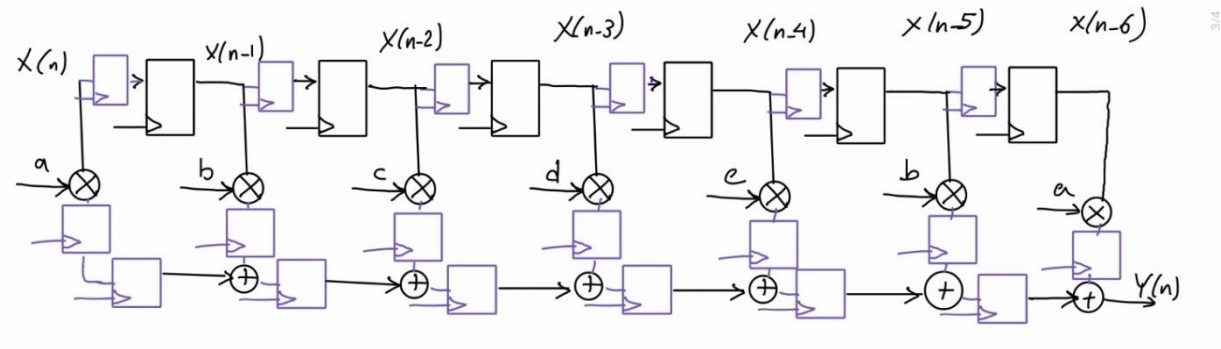
- A coefficient loading mechanism
- An input shift register for sample storage
- A direct output calculation combining all multiplications and additions in a single cycle

This design suffers from a long critical path that limits the maximum operating frequency. The primary bottleneck is in the output calculation, which requires:

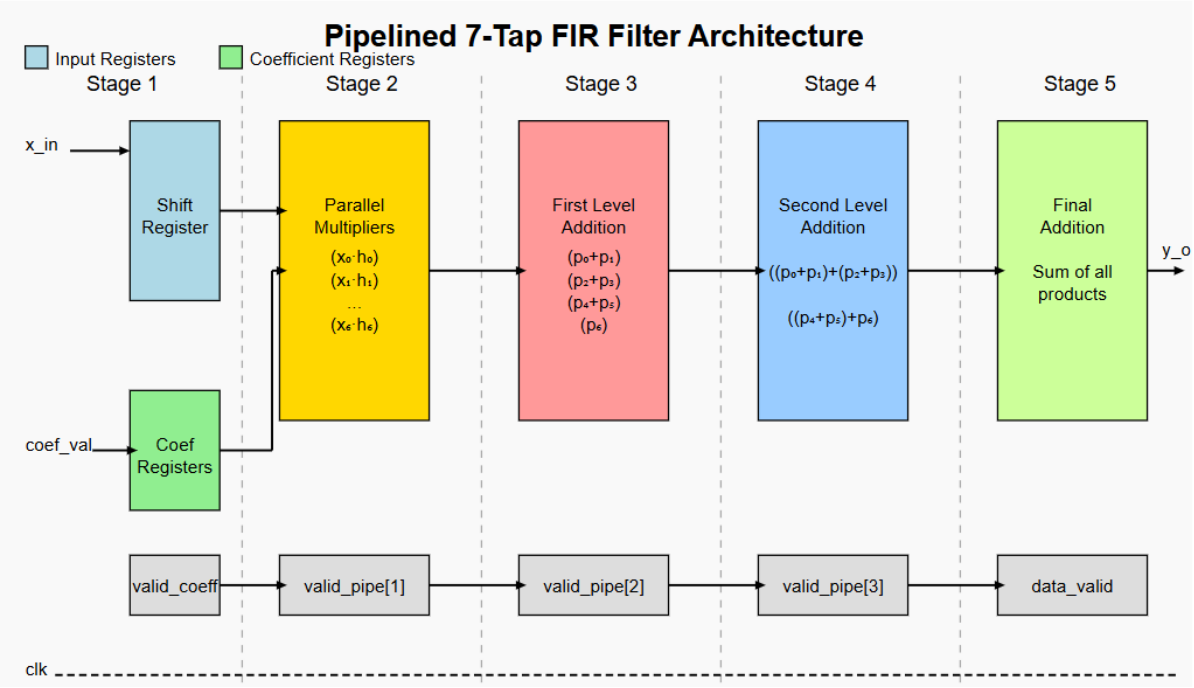
- Seven 8×8-bit multiplications
- Six sequential additions to combine all products
- All operations performed within a single clock cycle

Pipelined Architecture

At first I thought of implementing this fine grain pipelining:



But after doing some research, to improve performance, I implemented a 5-stage pipeline architecture that breaks down the critical path (the difference is that adder stage is tree style instead of sequential style.



Pipeline Stages

Stage 1: Input Registration: Sample shift register/Coefficient storage/Control logic

Stage 2: Multiplication: Seven parallel multipliers (8×8 bit)/Pipeline registers for multiplication results/Valid signal propagation

Stage 3: First Addition Level: Reduces from 7 products to 4 partial sums/Pipeline registers for intermediate results

Stage 4: Second Addition Level: Further reduces to 2 partial sums/Pipeline registers to maintain timing

Stage 5: Final Addition: Combines the remaining partial sums/Output registration with validity indicator

Implementation Details

The pipelined implementation was carefully designed to balance the computation across stages. Key features include:

Tree-based Addition Structure: Instead of sequential addition, a tree structure reduces the number of sequential operations

Valid Signal Propagation: A data validity signal flows through the pipeline to track valid results

Input Buffering: Shift registers ensure proper sample alignment

Pipeline Registers: Strategic register placement to optimize the critical path

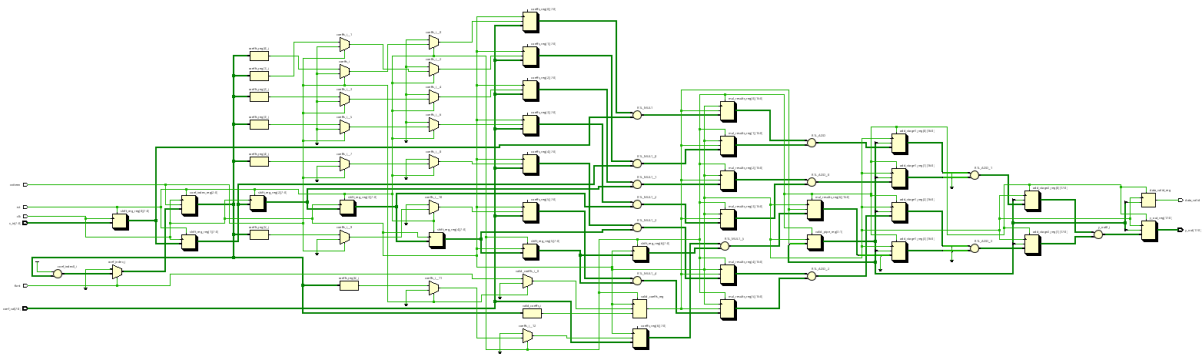
Here is the code:

```
module pipelined_fir_filter_7tap (
    input clk,
    input rst,
    input [7:0] x_in,
    input [7:0] coef_val,
    input writeen,
    input tlast,
    output reg [17:0] y_out,
    output reg data_valid
);
    // Internal Coefficient Registers
    reg [7:0] coeffs[0:6];
    reg [2:0] coef_index;
    reg valid_coeffs;

    // Input Shift Register
    reg [7:0] shift_reg[0:6];

    // Pipeline Registers for Multiplication Results
    reg [15:0] mul_results[0:6];

    // Pipeline Registers for Addition Stages
    // First addition stage
    reg [16:0] add_stage1[0:3]; // 4 partial sums

    // Second addition stage
    reg [17:0] add_stage2[0:1]; // 2 partial sums

    // Pipeline valid signals
    reg valid_pipe[1:4];

    integer i;

    // Load Coefficients
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            coef_index <= 0;
            valid_coeffs <= 0;
        end else if (writeen) begin
            coeffs[coef_index] <= coef_val;
            coef_index <= coef_index + 1;
            if (tlast) begin
                if (coef_index == 6) begin
                    valid_coeffs <= 1;
                end else begin
                    valid_coeffs <= 0; // discard
                end
```

```
                coef_index <= 0;
            end
        end
    end

    // Shift input samples - Stage 1
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 7; i = i + 1)
                shift_reg[i] <= 0;
        end else begin
            for (i = 6; i > 0; i = i - 1)
                shift_reg[i] <= shift_reg[i-1];
            shift_reg[0] <= x_in;
        end
    end

    // Stage 2: Parallel Multiplications
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 7; i = i + 1)
                mul_results[i] <= 0;
            valid_pipe[1] <= 0;
        end else begin
            if (valid_coeffs) begin
                for (i = 0; i < 7; i = i + 1)
                    mul_results[i] <= coeffs[i] * shift_reg[i];
                valid_pipe[1] <= 1;
            end else begin
                valid_pipe[1] <= 0;
            end
        end
    end

    // Stage 3: First level addition
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 4; i = i + 1)
                add_stage1[i] <= 0;
            valid_pipe[2] <= 0;
        end else begin
            if (valid_pipe[1]) begin
                // Add pairs of products and handle the odd one
                add_stage1[0] <= mul_results[0] + mul_results[1];
                add_stage1[1] <= mul_results[2] + mul_results[3];
                add_stage1[2] <= mul_results[4] + mul_results[5];
                add_stage1[3] <= {1'b0, mul_results[6]}; // Zero-extend for
consistency
```

```
                valid_pipe[2] <= 1;
            end else begin
                valid_pipe[2] <= 0;
            end
        end
    end

    // Stage 4: Second level addition
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            add_stage2[0] <= 0;
            add_stage2[1] <= 0;
            valid_pipe[3] <= 0;
        end else begin
            if (valid_pipe[2]) begin
                add_stage2[0] <= {1'b0, add_stage1[0]} + {1'b0,
add_stage1[1]};
                add_stage2[1] <= {1'b0, add_stage1[2]} + {1'b0,
add_stage1[3]};
                valid_pipe[3] <= 1;
            end else begin
                valid_pipe[3] <= 0;
            end
        end
    end

    // Stage 5: Final addition and output
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            y_out <= 0;
            data_valid <= 0;
        end else begin
            if (valid_pipe[3]) begin
                y_out <= add_stage2[0] + add_stage2[1];
                data_valid <= 1;
            end else begin
                data_valid <= 0;
            end
        end
    end

endmodule
```

here is the schematic of the design:



as we can see we have a lot of added FFs but the same multipliers and adders.

Here is the modifies testbench:

```
module pipelined_fir_filter_tb;
    // Parameters
    parameter CLK_PERIOD = 10; // 10ns = 100MHz clock

    // Signals
    reg clk;
    reg rst;
    reg [7:0] x_in;
    reg [7:0] coef_val;
    reg writeen;
    reg tlast;
    wire [17:0] y_out;
    wire data_valid;

    // Instantiate the DUT (Device Under Test)
    pipelined_fir_filter_7tap dut (
        .clk(clk),
        .rst(rst),
        .x_in(x_in),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
        .y_out(y_out),
        .data_valid(data_valid)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
    end
```

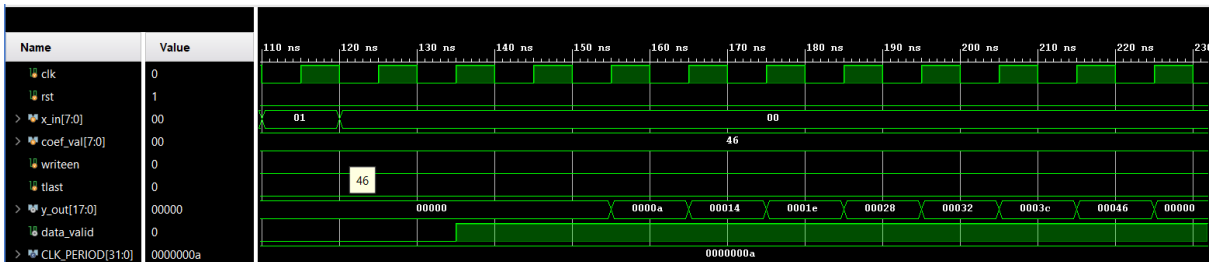
```
// Test sequence
initial begin
    // Initialize
    rst = 1;
    x_in = 0;
    coef_val = 0;
    writeen = 0;
    tlast = 0;

    // Release reset
    #(CLK_PERIOD*2) rst = 0;

    // Load coefficients
    #CLK_PERIOD;
    writeen = 1;

    coef_val = 10; // h[0] = 10
    #CLK_PERIOD;

    coef_val = 20; // h[1] = 20
    #CLK_PERIOD;

    coef_val = 30; // h[2] = 30
    #CLK_PERIOD;

    coef_val = 40; // h[3] = 40
    #CLK_PERIOD;

    coef_val = 50; // h[4] = 50
    #CLK_PERIOD;

    coef_val = 60; // h[5] = 60
    #CLK_PERIOD;

    coef_val = 70; // h[6] = 70
    tlast = 1;
    #CLK_PERIOD;

    // End coefficient loading
    writeen = 0;
    tlast = 0;

    // Apply input sequence
    #CLK_PERIOD;
    x_in = 1;
    #CLK_PERIOD;
    x_in = 0;
```

```
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;

        // Wait for pipeline to complete
        #(CLK_PERIOD*10);

        // Test another sequence with alternating 1s and 0s
        x_in = 1;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;
        #CLK_PERIOD;
        x_in = 0;

        // Wait for pipeline to complete
        #(CLK_PERIOD*10);

        $finish;
    end

    // Monitor results
    initial begin
        $monitor("Time=%0t, x_in=%0d, y_out=%0d, valid=%0b", $time, x_in,
y_out, data_valid);
    end

endmodule
```

let's see the results:

waveform:

coefficient loading is as the same before here is the output after impulse input:



TCL console:

as we can see the drawback is latency. while we could have the result in 1 clock now, we have in 4 clocks!

```
# run 1000ns
Time=0, x_in=0, y_out=0, valid=0
Time=110000, x_in=1, y_out=0, valid=0
Time=120000, x_in=0, y_out=0, valid=0
Time=135000, x_in=0, y_out=0, valid=1
Time=155000, x_in=0, y_out=10, valid=1
Time=165000, x_in=0, y_out=20, valid=1
Time=175000, x_in=0, y_out=30, valid=1
Time=185000, x_in=0, y_out=40, valid=1
Time=195000, x_in=0, y_out=50, valid=1
Time=205000, x_in=0, y_out=60, valid=1
Time=215000, x_in=0, y_out=70, valid=1
```

For finding the maximum operating frequency I followed the same procedure:

Trying 10ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.857 ns	Worst Hold Slack (WHS):	0.152 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	578	Total Number of Endpoints:	578	Total Number of Endpoints:	354

All user specified timing constraints are met.

trying 6ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.143 ns	Worst Hold Slack (WHS):	0.152 ns	Worst Pulse Width Slack (WPWS):	2.500 ns
Total Negative Slack (TNS):	-3.416 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	35	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	578	Total Number of Endpoints:	578	Total Number of Endpoints:	354

Timing constraints are not met.

Trying 6.15 ns:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.007 ns	Worst Hold Slack (WHS):	0.152 ns	Worst Pulse Width Slack (WPWS):	2.575 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	578	Total Number of Endpoints:	578	Total Number of Endpoints:	354

All user specified timing constraints are met.

⇨ maximum operating frequency is 1/6.15ns = **162.60 MHz**

comparing to previous **66.23 MHz** we almost increased the frequency by **2.5 times**.

Conclusion

The pipelined implementation successfully achieves a significant improvement in maximum operating frequency while maintaining correct functionality. The 5-stage pipeline design effectively breaks down the critical path of the original FIR filter, allowing for much higher clock rates.

This implementation demonstrates the power of pipelining techniques in digital signal processing applications. The design trades a small increase in resource usage (pipeline registers) and latency for a substantial improvement in throughput and operating frequency.

Part8)

Assume that the input is such that the inputs are applied to the module at 20-clock intervals.

Optimize the number of multiplications and additions by applying Resource Sharing. At this stage, you also need to ensure the correct operation of your circuit by running a bench test. Synthesize the code from Part 2 and this part and justify the difference in the resources used.

The design uses **resource sharing** for the multiplication and accumulation operations. Instead of using multiple multipliers and adders for each coefficient, the design utilizes a single MAC unit that sequentially handles each multiplication and accumulation. This significantly reduces the hardware requirements.

coefficient Storage: The filter supports up to 7 coefficients, which are stored in a 7-element array.

Shift Register: The input samples are stored in a shift register, which holds 7 previous values of the input data.

Multiplication and Accumulation: A single MAC unit performs the multiply-accumulate operation for each input sample and coefficient, one at a time.

State Machine: The state machine controls the operation of the filter, including loading coefficients, shifting input samples, performing the MAC operation, and outputting the result.

State Machine:

- IDLE: The filter is in the idle state, waiting for input.
- MAC: The filter performs the multiply-accumulate operation.
- DONE: The output is generated after completing the calculation.

MAC Operation:

The mac register holds the result of the accumulation (add + multreg), and the multreg is the product of the coefficient and the input data.

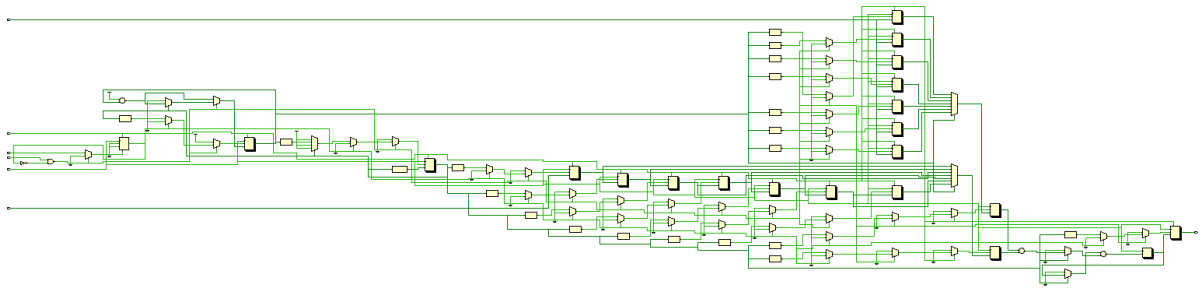
Here is the code:

```
module fir_filter_resourcesharing (
    input wire clk,
    input wire rst,
    input wire [7:0] x_in,
    input wire [7:0] coef_val,
    input wire writeen,
    input wire tlast,
    output reg [18:0] y_out
);

    reg [7:0] coeffs[0:6];
```

```
    reg [7:0] in_reg [0:6];
    reg [2:0] coeff_index;
    reg coeff_done;

    reg [2:0] state;
    reg [18:0] add;
    reg  [18:0] mac;
    reg [15:0] multreg;
    reg  [7:0] multdata;
    reg  [7:0] multcoeff;

   always @(posedge clk)
         mac <= add + multreg;


    integer i;


    parameter IDLE = 3'd0,
              MAC  = 3'd1,
              DONE = 3'd2;

    always @(*)
    begin
    multreg = (state == IDLE) ? (16'b0):(multcoeff * multdata);
    add = (state == IDLE) ? (16'b0):(mac);

    end




    always @(posedge clk) begin
        if (rst) begin
            coeff_index <= 0;
            coeff_done <= 0;
            y_out <= 0;

            state <= IDLE;
            for (i = 0; i < 7; i = i + 1) begin
                coeffs[i] <= 0;
                in_reg[i] <= 0;
            end
        end


        else if (writeen && !coeff_done) begin
            coeffs[coeff_index] <= coef_val;
```

```
            coeff_index <= coeff_index + 1;
            if (tlast)
                coeff_done <= 1;
        end

        else if (coeff_done) begin
            case (state)
                IDLE: begin

                    for (i = 6; i > 0; i = i - 1)
                        in_reg[i] <= in_reg[i-1];
                        in_reg[0] <= x_in;
                      state <= MAC;
                    coeff_index <= 0;


                end

                MAC: begin
                    multdata <= in_reg[coeff_index];
                    multcoeff <= coeffs[coeff_index];
                    coeff_index <= coeff_index + 1;

                    if (coeff_index == 6)
                        state <= DONE;
                end

                DONE: begin
                y_out <= mac;

                    state <= IDLE;
                end

                default: state <= IDLE;
            endcase
        end
    end

endmodule
```

here is the schematic:



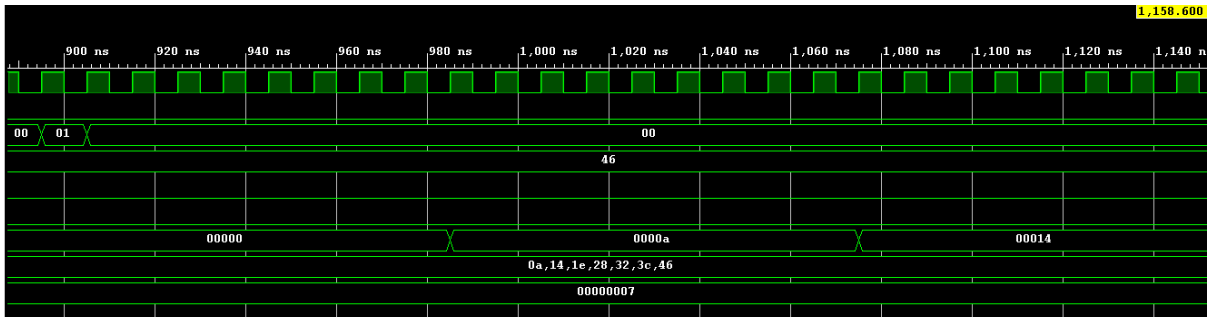
as we can see instead of 7 multipliers and 6 adders we have only 1 mac but with much more MUXes.

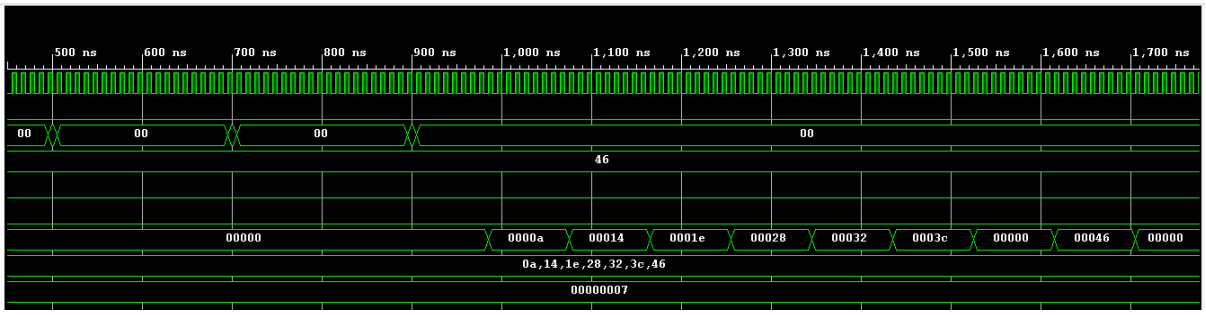
Here is the testbench:

```
`timescale 1ns / 1ps

module fir_filter_resourcesharing_tb;

    reg clk, rst;
    reg [7:0] x_in, coef_val;
    reg writeen, tlast;
    wire [18:0] y_out;


    fir_filter_resourcesharing dut (
        .clk(clk),
        .rst(rst),
        .x_in(x_in),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
        .y_out(y_out)
    );


    always #5 clk = ~clk;

    reg [7:0] coeffs[0:6];
    integer i;

    initial begin


        clk = 0;
        rst = 1;
        x_in = 0;
```

```
        coef_val = 0;
        writeen = 0;
        tlast = 0;

        coeffs[0] = 8'd10;
        coeffs[1] = 8'd20;
        coeffs[2] = 8'd30;
        coeffs[3] = 8'd40;
        coeffs[4] = 8'd50;
        coeffs[5] = 8'd60;
        coeffs[6] = 8'd70;

        #15 rst = 0;


        for (i = 0; i < 7; i = i + 1) begin
            @(posedge clk);
            coef_val <= coeffs[i];
            writeen <= 1;
            tlast <= (i == 6);
        end

        @(posedge clk);
        writeen <= 0;
        tlast <= 0;


        repeat (5) begin

            @(posedge clk);
            x_in <= 8'd1;


            repeat (19) begin
                @(posedge clk);
                x_in <= 8'd0;
            end
        end

        #50;
        $finish;
    end



endmodule
```

here is the result:





the wave form confirms the functionality of the code.

The utilization report:

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (54)	BUFGCTRL (32)
N fir_filter_resourcesharing	140	172	39	1

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (54)	BUFGCTRL (32)
N fir_filter_7tap	630	134	38	1

We have much less LUTs almost the same IOB but we have more registers which is neglectable against LUTs.

Part 9)

Suppose that to increase the processing rate, two inputs are applied to your module simultaneously on each clock edge. Change the structure so that two inputs are applied to the circuit on each clock edge and the corresponding two outputs appear on the output after a few clocks. At this stage, you also need to ensure the correct operation of your circuit by running a test bench.

The solution was to implement parallel processing, which nearly doubles the resource usage but also achieves twice the throughput without changing the operating frequency. To accomplish this, I essentially duplicated the data paths and computations, allowing two inputs to be processed simultaneously.

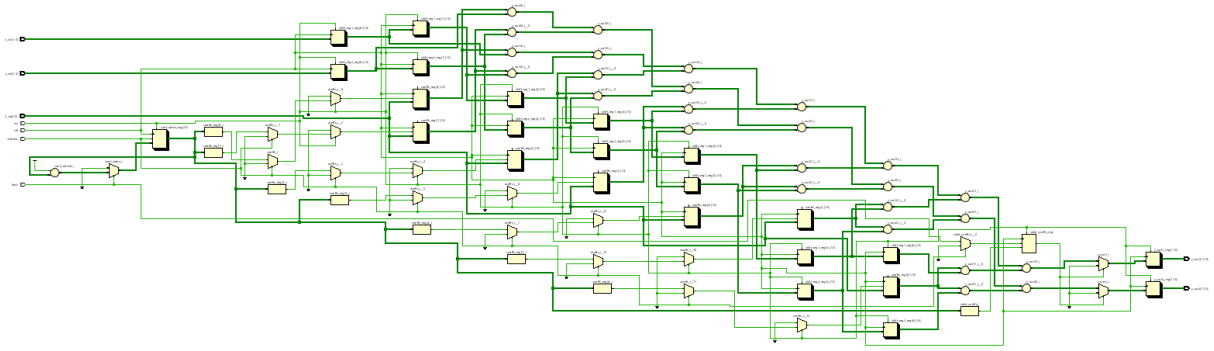
Here is the code:

```
module fir_filter_7tap_parallel (
    input clk,
    input rst,
    input [7:0] x_in1,
    input [7:0] x_in2,
    input [7:0] coef_val,
    input writeen,
    input tlast,
    output reg [17:0] y_out1,
    output reg [17:0] y_out2
);
    // Internal Coefficient Registers
    reg [7:0] coeffs[0:6];
    reg [2:0] coef_index;
    reg valid_coeffs;

    // Input Shift Registers for two parallel streams
    reg [7:0] shift_reg1[0:6];
    reg [7:0] shift_reg2[0:6];

    integer i;

    // Load Coefficients
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            coef_index <= 0;
            valid_coeffs <= 0;
        end else if (writeen) begin
            coeffs[coef_index] <= coef_val;
            coef_index <= coef_index + 1;
            if (tlast) begin
                if (coef_index == 6) begin
                    valid_coeffs <= 1;
                end else begin
```

```
                valid_coeffs <= 0; // discard
            end
            coef_index <= 0;
        end
    end
end

// Shift input samples for stream 1 and stream 2
always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < 7; i = i + 1) begin
            shift_reg1[i] <= 0;
            shift_reg2[i] <= 0;
        end
    end else begin
        for (i = 6; i > 0; i = i - 1) begin
            shift_reg1[i] <= shift_reg1[i-1];
            shift_reg2[i] <= shift_reg2[i-1];
        end
        shift_reg1[0] <= x_in1;
        shift_reg2[0] <= x_in2;
    end
end

// Output calculation for both streams
always @(posedge clk or posedge rst) begin
    if (rst) begin
        y_out1 <= 0;
        y_out2 <= 0;
    end else if (valid_coeffs) begin
        y_out1 <= coeffs[0] * shift_reg1[0] +
                  coeffs[1] * shift_reg1[1] +
                  coeffs[2] * shift_reg1[2] +
                  coeffs[3] * shift_reg1[3] +
                  coeffs[4] * shift_reg1[4] +
                  coeffs[5] * shift_reg1[5] +
                  coeffs[6] * shift_reg1[6];

        y_out2 <= coeffs[0] * shift_reg2[0] +
                  coeffs[1] * shift_reg2[1] +
                  coeffs[2] * shift_reg2[2] +
                  coeffs[3] * shift_reg2[3] +
                  coeffs[4] * shift_reg2[4] +
                  coeffs[5] * shift_reg2[5] +
                  coeffs[6] * shift_reg2[6];
    end else begin
        y_out1 <= 0;
        y_out2 <= 0;
```

```
        end
    end
endmodule
```

here is the scematic:



as we can see the resources used is 2 times more!

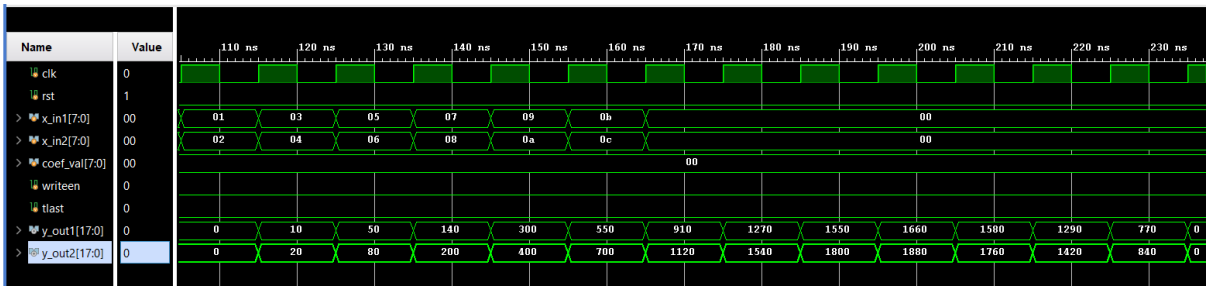
Here is the testbench:

```
`timescale 1ns/1ps

module tb_fir_filter_7tap_parallel();

    reg clk;
    reg rst;
    reg [7:0] x_in1;
    reg [7:0] x_in2;
    reg [7:0] coef_val;
    reg writeen;
    reg tlast;
    wire [17:0] y_out1;
    wire [17:0] y_out2;

    // Instantiate the DUT
    fir_filter_7tap_parallel dut (
        .clk(clk),
        .rst(rst),
        .x_in1(x_in1),
        .x_in2(x_in2),
        .coef_val(coef_val),
        .writeen(writeen),
        .tlast(tlast),
        .y_out1(y_out1),
        .y_out2(y_out2)
    );
```

```
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10ns clock period
    end

    initial begin
        // Initial values
        rst = 1;
        x_in1 = 0;
        x_in2 = 0;
        coef_val = 0;
        writeen = 0;
        tlast = 0;

        // Reset
        #20;
        rst = 0;

        // Load coefficients: 10, 20, 30, 40, 50, 60, 70
        @(posedge clk); writeen = 1; coef_val = 8'd10;
        @(posedge clk); coef_val = 8'd20;
        @(posedge clk); coef_val = 8'd30;
        @(posedge clk); coef_val = 8'd40;
        @(posedge clk); coef_val = 8'd50;
        @(posedge clk); coef_val = 8'd60;
        @(posedge clk); coef_val = 8'd70; tlast = 1;

        @(posedge clk);
        writeen = 0;
        tlast = 0;
        coef_val = 0;

        // Apply inputs
        @(posedge clk);
        x_in1 = 8'd1;
        x_in2 = 8'd2;

        @(posedge clk);
        x_in1 = 8'd3;
        x_in2 = 8'd4;

        @(posedge clk);
        x_in1 = 8'd5;
        x_in2 = 8'd6;

        @(posedge clk);
        x_in1 = 8'd7;
```

```
        x_in2 = 8'd8;

        @(posedge clk);
        x_in1 = 8'd9;
        x_in2 = 8'd10;

        @(posedge clk);
        x_in1 = 8'd11;
        x_in2 = 8'd12;

        // Stop inputs
        @(posedge clk);
        x_in1 = 0;
        x_in2 = 0;

        // Let the filter settle
        repeat(10) @(posedge clk);

        $stop;
    end

    // Monitor outputs
    always @(posedge clk) begin
        $display("At time %0t : y_out1 = %d, y_out2 = %d", $time, y_out1,
y_out2);
    end

endmodule
```

here is the wave form:



here is the TCL console:

```
t time 5000 : y_out1 =      0, y_out2 =      0
At time 15000 : y_out1 =      0, y_out2 =      0
At time 25000 : y_out1 =      0, y_out2 =      0
```

```
At time 35000 : y_out1 =      0, y_out2 =      0
At time 45000 : y_out1 =      0, y_out2 =      0
At time 55000 : y_out1 =      0, y_out2 =      0
At time 65000 : y_out1 =      0, y_out2 =      0
At time 75000 : y_out1 =      0, y_out2 =      0
At time 85000 : y_out1 =      0, y_out2 =      0
At time 95000 : y_out1 =      0, y_out2 =      0
At time 105000 : y_out1 =      0, y_out2 =      0
At time 115000 : y_out1 =      0, y_out2 =      0
At time 125000 : y_out1 =     10, y_out2 =     20
At time 135000 : y_out1 =     50, y_out2 =     80
At time 145000 : y_out1 =    140, y_out2 =    200
At time 155000 : y_out1 =    300, y_out2 =    400
At time 165000 : y_out1 =    550, y_out2 =    700
At time 175000 : y_out1 =    910, y_out2 =   1120
At time 185000 : y_out1 =   1270, y_out2 =   1540
At time 195000 : y_out1 =   1550, y_out2 =   1800
At time 205000 : y_out1 =   1660, y_out2 =   1880
At time 215000 : y_out1 =   1580, y_out2 =   1760
At time 225000 : y_out1 =   1290, y_out2 =   1420
At time 235000 : y_out1 =    770, y_out2 =    840
At time 245000 : y_out1 =      0, y_out2 =      0
At time 255000 : y_out1 =      0, y_out2 =      0
```

They confirm its correction.

Here is the utilization report:

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (54)	BUFGCTRL (32)
N fir_filter_7tap_parallel	1243	208	64	1

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (54)	BUFGCTRL (32)
N fir_filter_7tap	630	134	38	1

As we can see the resources used are doubled.