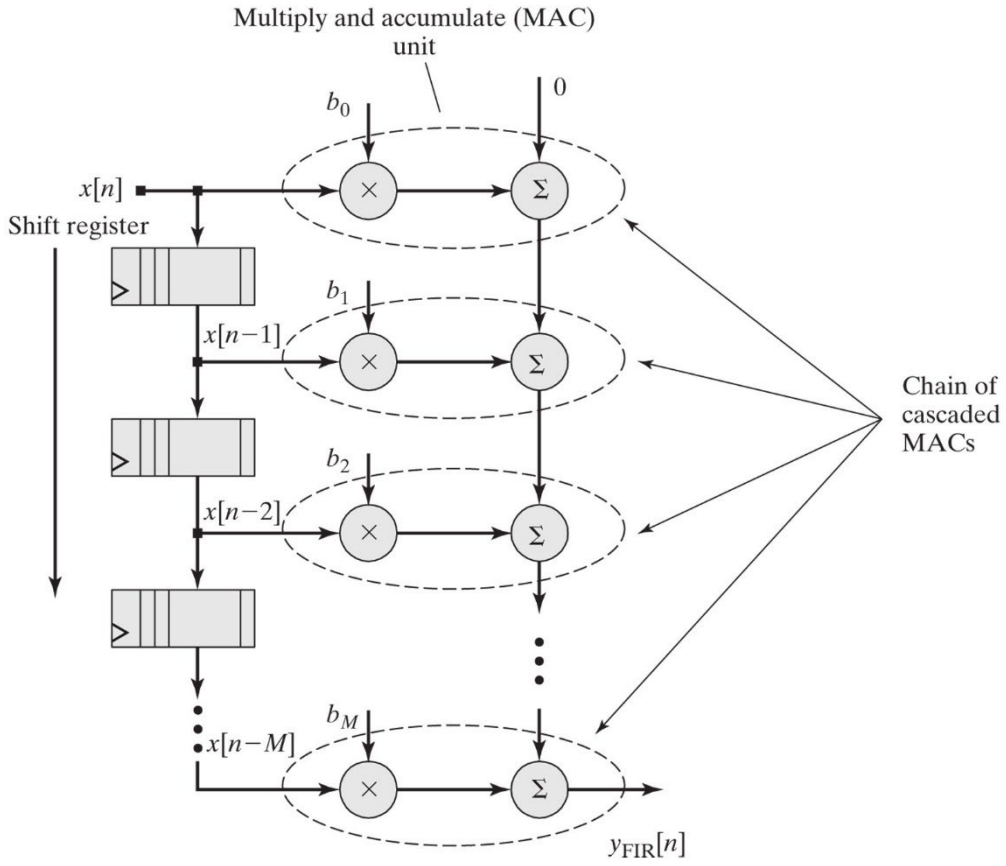
MAC-based architecture for an M^{th}-order FIR filter.

The input samples are pushed into the shift register. Each entry in the shift register is multiplied by a filter coefficient (or weight) and then accumulated to give the output sample.




```
module MAC ( Acc_out, Sample_in, weight_in, Acc_in );     // combinational logic

parameter FIR_order      = 4;
parameter sample_size    = 6;                    // maximum sample value is 63
parameter weight_size    = 5;                    // maximum value may be 31
parameter word_size_out  = 2*sample_size + 2;  // maximum possibe output 63*31*(4+1)

output  [word_size_out -1: 0] Acc_out;
input   [sample_size   -1: 0] Sample_in;
input   [weight_size   -1: 0] weight_in;
input   [word_size_out -1: 0] Acc_in;

assign Acc_out = (Sample_in*weight_in) + Acc_in;    // Multiply and Accumulate MAC

endmodule
```

```
module FIR_noPipeline ( FIR_out_MAC, FIR_out_assign, Sample_in, clock, reset);

parameter FIR_order     = 4;
parameter sample_size   = 6;                    // maximum sample value is 63
parameter weight_size   = 5;                    // maximum value may be 31
parameter word_size_out = 2*sample_size + 2;  // maximum possibe output 63*31*(4+1)

output reg [word_size_out -1: 0] FIR_out_MAC;
output reg [word_size_out -1: 0] FIR_out_assign;

input      [sample_size   -1: 0] Sample_in;
input                            clock, reset;

wire       [word_size_out -1: 0] Acc_0, Acc_1, Acc_2, Acc_3, Acc_4;
wire       [word_size_out -1: 0] comb_out;                              // optional

// Filter coefficients
parameter b0 = 5'd3;
parameter b1 = 5'd7;
parameter b2 = 5'd20;
parameter b3 = 5'd7;
parameter b4 = 5'd3;

reg [sample_size -1: 0] Sample_Array [1: FIR_order];   // 5th coefficient multiplied by Data_in
integer k;

MAC M0 ( Acc_0, Sample_in,         b0, 0     );    // combinational logic
MAC M1 ( Acc_1, Sample_Array [1], b1, Acc_0 );    // combinational logic
MAC M2 ( Acc_2, Sample_Array [2], b2, Acc_1 );    // combinational logic
MAC M3 ( Acc_3, Sample_Array [3], b3, Acc_2 );    // combinational logic
MAC M4 ( Acc_4, Sample_Array [4], b4, Acc_3 );    // combinational logic

// Alternate way to create the combinational logic
assign comb_out = b0 * Sample_in
                + b1 * Sample_Array[1]
                + b2 * Sample_Array[2]
                + b3 * Sample_Array[3]
                + b4 * Sample_Array[4];

always @ (posedge clock)
     if (reset == 1) begin
         for (k = 1; k <= FIR_order; k = k+1)
              Sample_Array[k] <= 0;
         FIR_out_MAC          <= 0;
         FIR_out_assign       <= 0;
         end
   else begin
         Sample_Array [1]  <= Sample_in;
           for (k = 2; k   <= FIR_order; k = k+1)
           Sample_Array[k] <= Sample_Array[k-1];
         FIR_out_assign    <= comb_out;
         FIR_out_MAC       <= Acc_4;
         end

endmodule
```

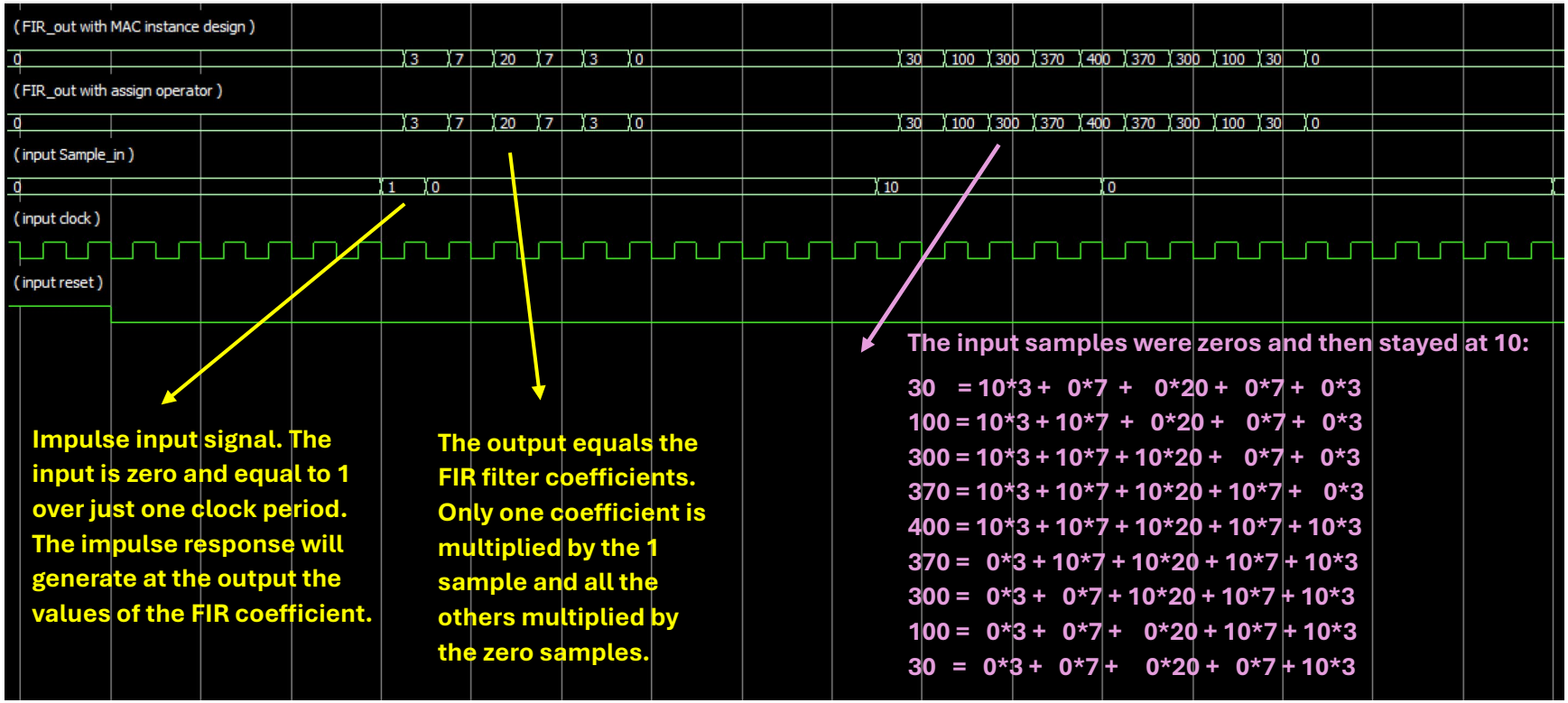
These two blocks of code are just giving the same combinational logic, calculating the value of the output sample.

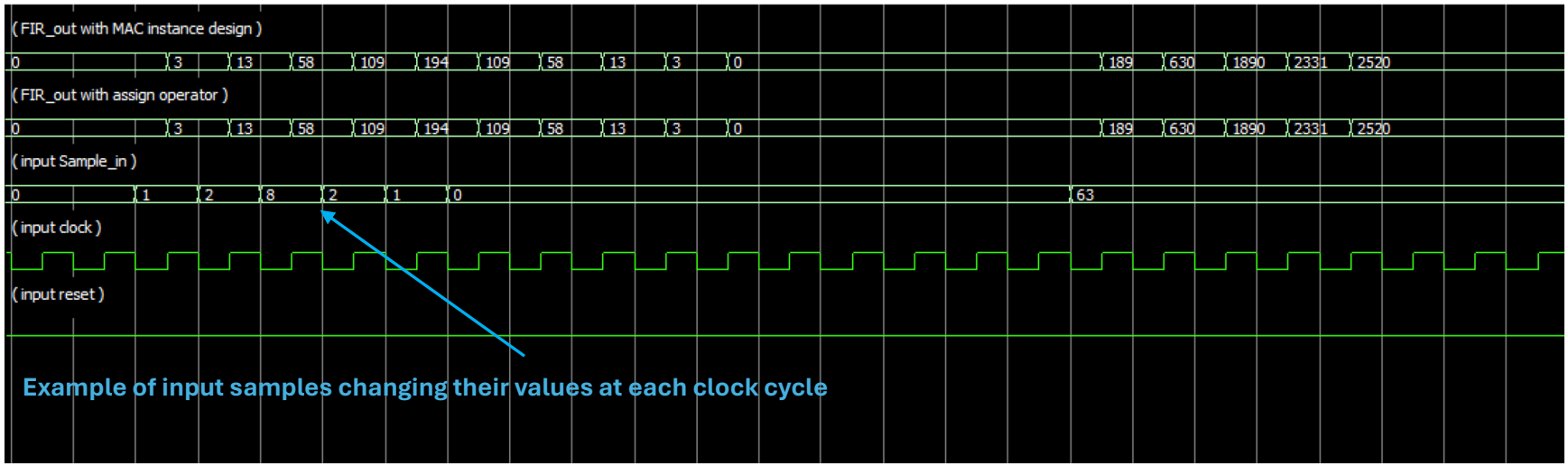
The output is registered to avoid any transitional glitches or intermediate values within one clock period.

One way to write the Verilog code is to replace the comb_out with the full expression calculating it, and assign it to FIR_out_assign directly under the always block.

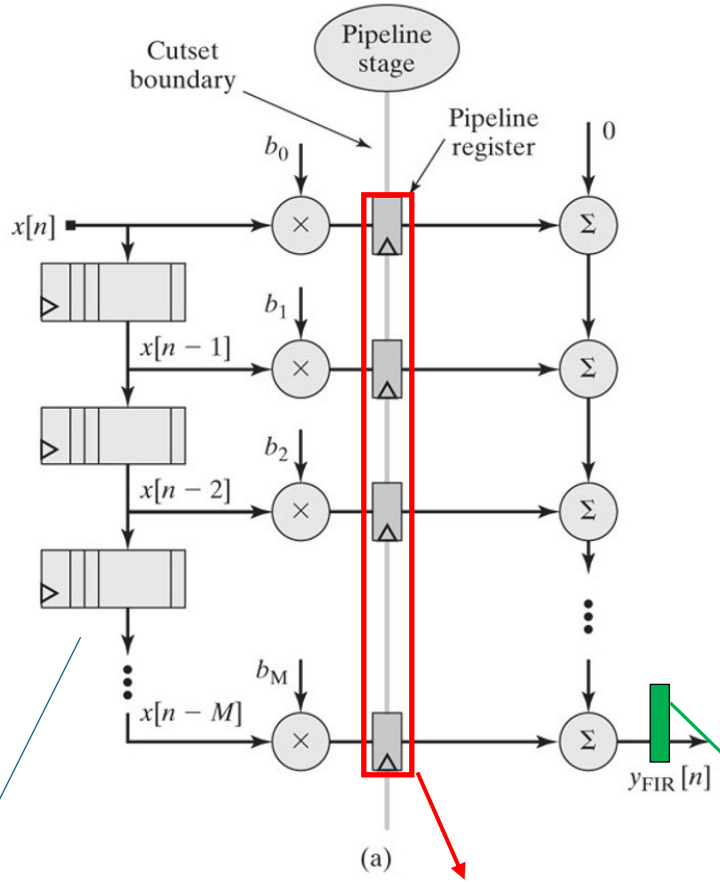
```
module FIR_Pipeline_tb ();

parameter FIR_order     = 4;
parameter sample_size   = 6;                    // maximum sample value is 63
parameter weight_size   = 5;                    // maximum value may be 31
parameter word_size_out = 2*sample_size + 2;    // maximum possibe output 63*31*(4+1)

wire [word_size_out -1: 0] FIR_out_MAC;
wire [word_size_out -1: 0] FIR_out_assign;

reg      [sample_size   -1: 0] Sample_in;
reg                            clock, reset;

FIR_noPipeline UUT  ( FIR_out_MAC, FIR_out_assign, Sample_in, clock, reset);

initial
begin
clock = 0;
forever
#5 clock = ~clock;
end

initial
begin
reset = 1;
#40 reset = 0;
end

initial
begin
Sample_in = 0;    #100
Sample_in = 1;    #10      // impulse response
Sample_in = 0;    #100
Sample_in = 10;   #50      // same input over 5 clock cycles
Sample_in = 0;    #100
Sample_in = 1;    #10
Sample_in = 2;    #10
Sample_in = 8;    #10
Sample_in = 2;    #10
Sample_in = 1;    #10
Sample_in = 0;    #100
Sample_in = 63;   #100
Sample_in = 0;
end

endmodule
```

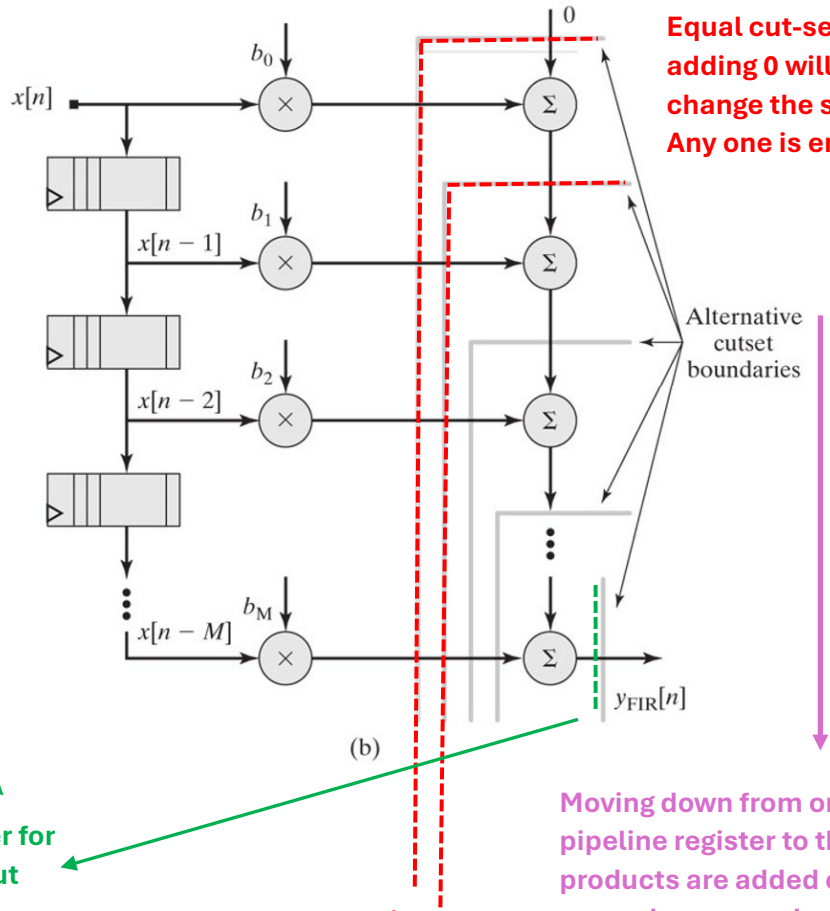
Comparing FIR_out_MAC and FIR_out_assign to show that the two blocks of combinational code are just creating the same logic for the output sample with the same timing.


(FIR_out with MAC instance design)
0 3 7 20 7 3 0 30 100 300 370 400 370 300 100 30 0
(FIR_out with assign operator)
0 3 7 20 7 3 0 30 100 300 370 400 370 300 100 30 0
(input Sample_in)
0 1 0 10 0
(input clock)
(input reset)
Impulse input signal. The input is zero and equal to 1 over just one clock period. The impulse response will generate at the output the values of the FIR coefficient.
The output equals the FIR filter coefficients. Only one coefficient is multiplied by the 1 sample and all the others multiplied by the zero samples.
The input samples were zeros and then stayed at 10:
30 = 10*3 + 0*7 + 0*20 + 0*7 + 0*3
100 = 10*3 + 10*7 + 0*20 + 0*7 + 0*3
300 = 10*3 + 10*7 + 10*20 + 0*7 + 0*3
370 = 10*3 + 10*7 + 10*20 + 10*7 + 0*3
400 = 10*3 + 10*7 + 10*20 + 10*7 + 10*3
370 = 0*3 + 10*7 + 10*20 + 10*7 + 10*3
300 = 0*3 + 0*7 + 10*20 + 10*7 + 10*3
100 = 0*3 + 0*7 + 0*20 + 10*7 + 10*3
30 = 0*3 + 0*7 + 0*20 + 0*7 + 10*3


(FIR_out with MAC instance design)
0 3 13 58 109 194 109 58 13 3 0 189 630 1890 2331 2520
(FIR_out with assign operator)
0 3 13 58 109 194 109 58 13 3 0 189 630 1890 2331 2520
(input Sample_in)
0 1 2 8 2 1 0 63
(input clock)
(input reset)
Example of input samples changing their values at each clock cycle

Alternative pipeline structures for FIR filter, with pipeline registers placed (a) at the outputs of the multipliers and (b) at the inputs of the adders.





Equal cut-sets, as adding 0 will not change the sum. Any one is enough.

The input register is represented by the shift registers that push the input samples through and save recent serial samples to calculate the filter output. The shift register is synchronized by the clock.

The Pipeline Register will hold the product values. It has to hold a total number of bits equal to (FIR_order+1) *product_size. This register can be in the form of an array or a word.

Output register for the filter output

The same PR structure and effect

Moving down from one pipeline register to the next, products are added one by one or by groups depending on the design and the number of pipeline stages.

The Pipeline registers reduce in size.

```
module Pipeline_FIR_a ( FIR_out_pipeline, Sample_in, clock, reset);

parameter FIR_order     = 4;
parameter sample_size   = 6;                                  // maximum sample value is 63
parameter weight_size   = 5;                                  // maximum value may be 31
parameter word_size_out = sample_size + weight_size + 3;  // log2(2^6*2^5*(order+1))

parameter product_size  = sample_size + weight_size;

output reg [word_size_out -1: 0] FIR_out_pipeline;

input      [sample_size   -1: 0] Sample_in;
input                            clock, reset;

// Filter coefficients
parameter b0 = 5'd3;
parameter b1 = 5'd7;
parameter b2 = 5'd20;
parameter b3 = 5'd7;
parameter b4 = 5'd3;

reg [sample_size -1 : 0] Sample_Array [1: FIR_order];    // 5th coefficient multiplied by Data_in
integer k;

reg [product_size -1: 0] PR [0 : FIR_order];             // Array format

 always @ (posedge clock)
      if (reset == 1) begin
          // The input shift register
          for (k = 1; k <= FIR_order; k = k+1)     // to save on code lines
               Sample_Array[k] <= 0;
          // The pipeline register
          for (k = 0; k <= FIR_order; k = k+1)     // to save on code lines
               PR[k]     <= 0;
          // The output register
          FIR_out_pipeline     <= 0;
          end

    else begin
          // The input shift register
          Sample_Array [1]  <= Sample_in;
            for (k = 2; k   <= FIR_order; k = k+1)
            Sample_Array[k] <= Sample_Array[k-1];
          // The Pipeline Register
          PR[0] <= b0 * Sample_in;
          PR[1] <= b1 * Sample_Array[1];
          PR[2] <= b2 * Sample_Array[2];
          PR[3] <= b3 * Sample_Array[3];
          PR[4] <= b4 * Sample_Array[4];
          // The output register
          FIR_out_pipeline  <= PR[0] + PR[1] + PR[2] + PR[3] + PR[4];
          end

endmodule
```

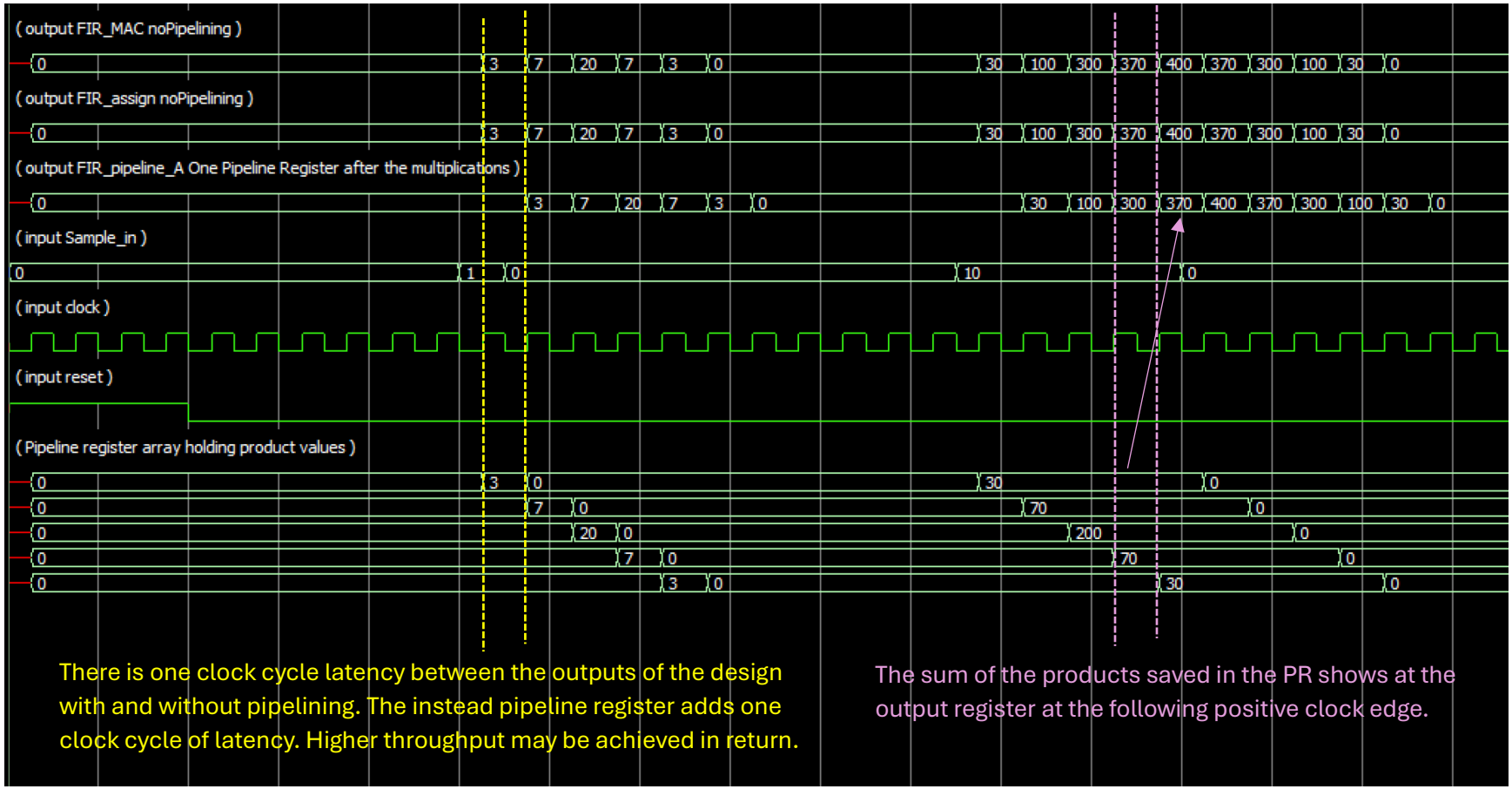
Pipeline registers placed at the outputs of the multipliers.

figure (a)

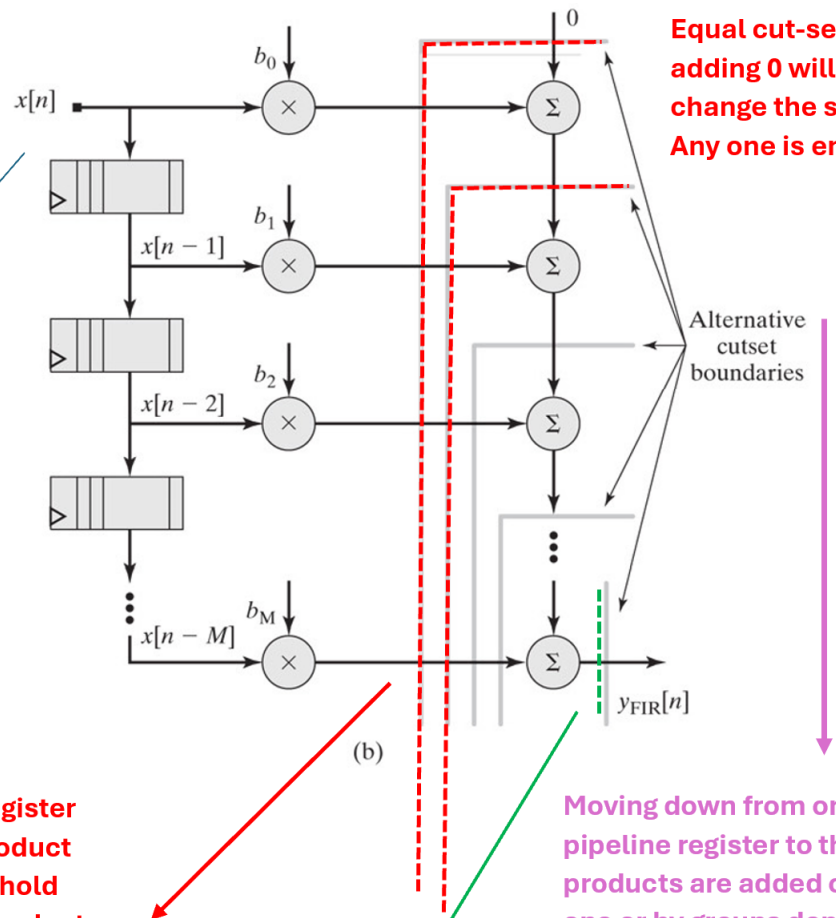
The calculation of the FIR output is split over two steps:

1. **Finding all the products and saving them in the PR array.**
2. **Adding all the products to generate the output.**

Pipelining assures that the multipliers and the adders are always busy with efficient operation and maximized throughput.


(output FIR_MAC noPipelining)
0 3 7 20 7 3 0 30 100 300 370 400 370 300 100 30 0
(output FIR_assign noPipelining)
0 3 7 20 7 3 0 30 100 300 370 400 370 300 100 30 0
(output FIR_pipeline_A One Pipeline Register after the multiplications)
0 3 7 20 7 3 0 30 100 300 370 400 370 300 100 30 0
(input Sample_in)
0 1 0 10 0
(input clock)
(input reset)
(Pipeline register array holding product values)
0 3 0 30 0
0 7 0 70 0
0 20 0 200 0
0 7 0 70 0
0 3 0 30 0
There is one clock cycle latency between the outputs of the design with and without pipelining. The instead pipeline register adds one clock cycle of latency. Higher throughput may be achieved in return.
The sum of the products saved in the PR shows at the output register at the following positive clock edge.

(b) Alternative pipeline structures for FIR filter with pipeline registers placed at the inputs of the adders.



Equal cut-sets, as adding 0 will not change the sum. Any one is enough.

The input register is represented by the shift registers that push the input samples through and save recent serial samples to calculate the filter output. The shift register is synchronized by the clock.

The Pipeline Register will hold the product values. It must hold (FIR_order+1) products. This register can be in the form of an array or a word.

Output register for the filter output

Output register will have the sum of all the products with a depth of 1.

Moving down from one pipeline register to the next, products are added one by one or by groups depending on the design and the number of pipeline stages.

The Pipeline registers reduce in size.

Product 0 = b0*Sample_in
Product 1 = b1*Sample [1]
Product 2 = b2*Sample [2]
Product 3 = b3*Sample [3]
Product 4 = b4*Sample [4]

PR0 register array depth = 5

Product 1 + Product 0
Product 2
Product 3
Product 4

PR1 register array depth = 4

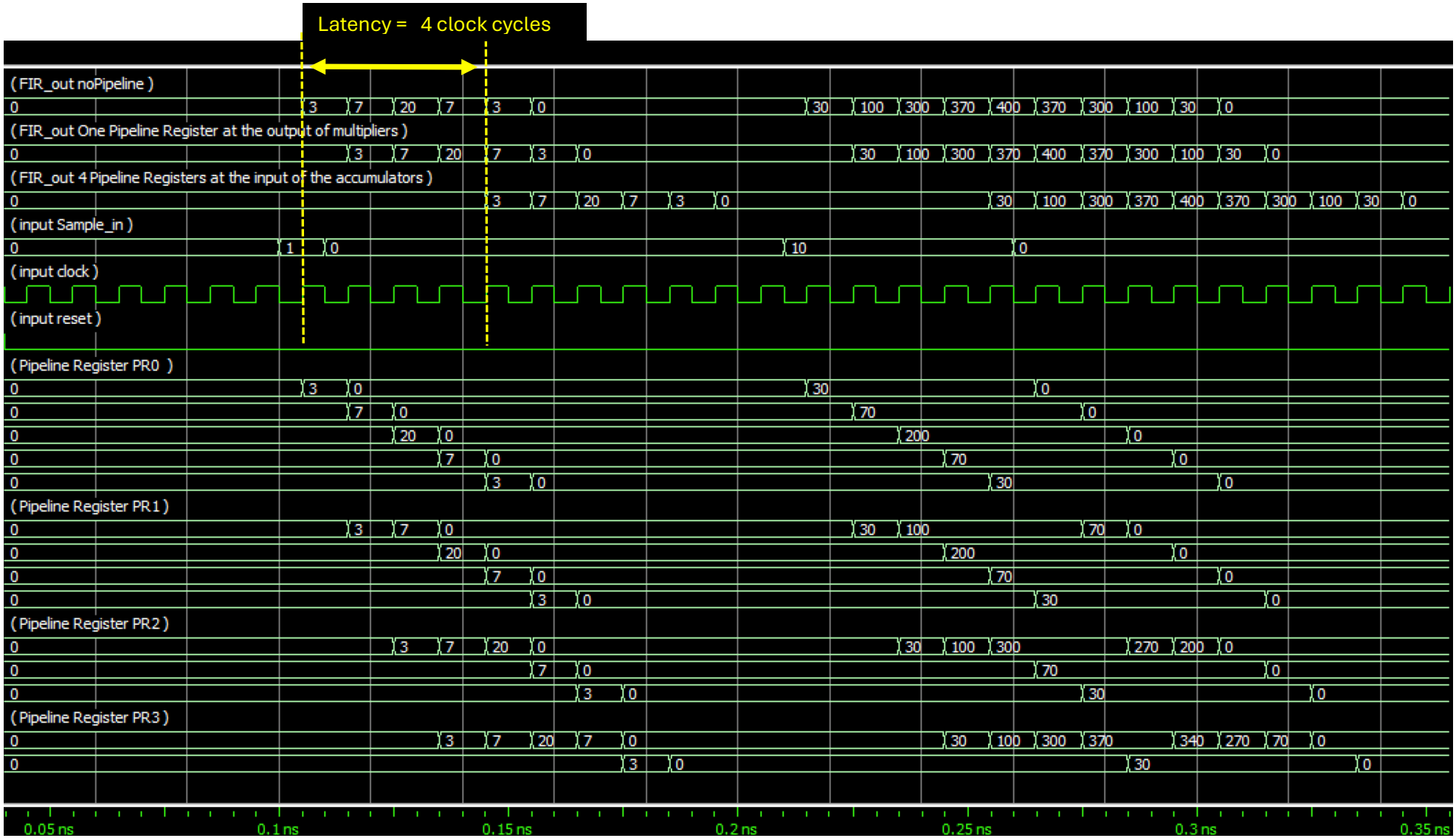
Product 2 + Product 1+ Product 0
Product 3
Product 4

PR2 register array depth = 3

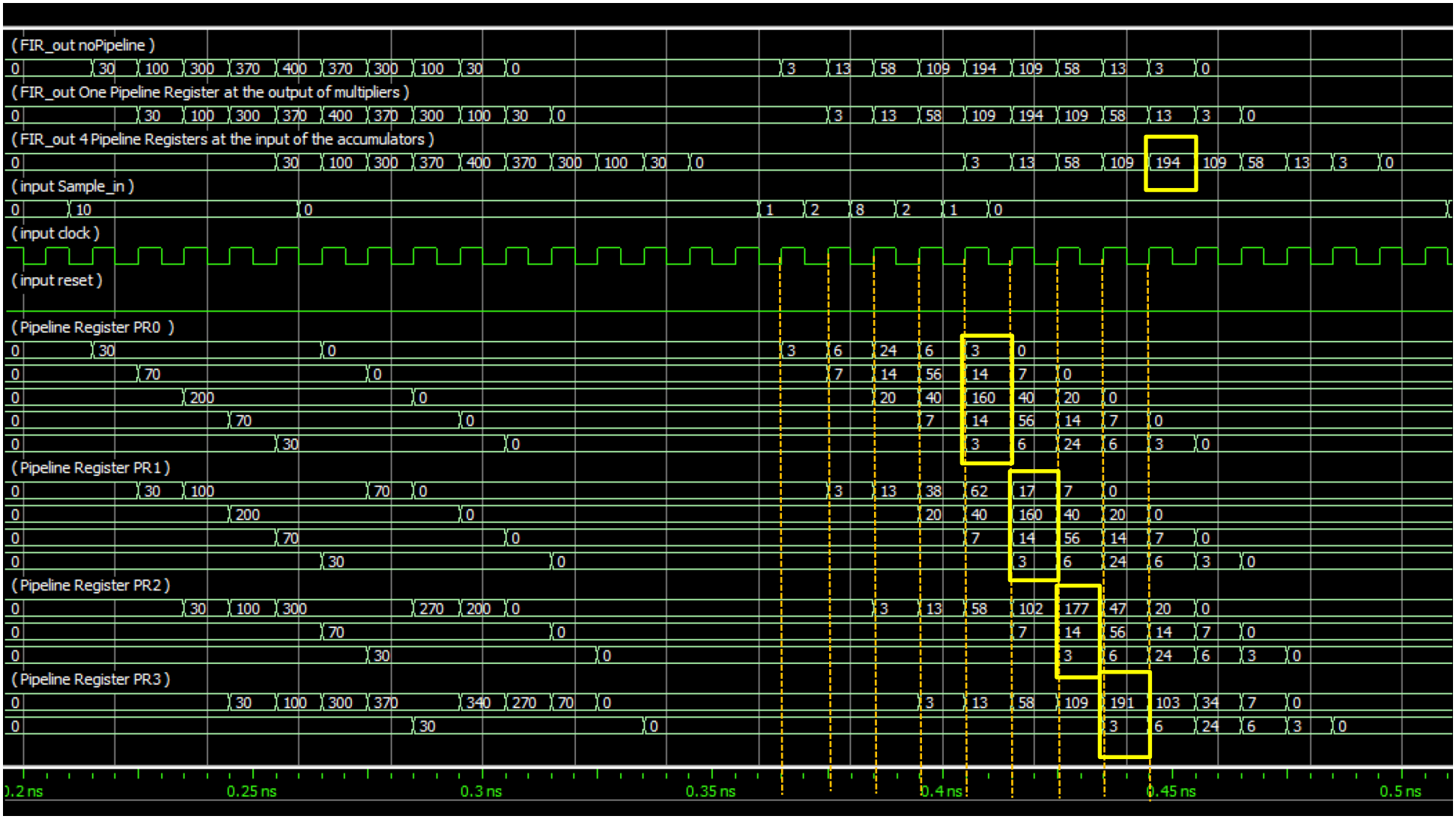
Product 3 + Product 2 + Product 1+ Product 0
Product 4

PR3 register array depth = 2

```
//-------------------------------------------------------------------------------------------
//  (b) Alternative pipeline structures for FIR filter with pipeline registers placed
//      at the inputs of the adders.
//-------------------------------------------------------------------------------------------

   module Pipeline_FIR_b ( FIR_out, Sample_in, clock, reset);

parameter FIR_order     = 4;
parameter sample_size   = 6;                              // maximum sample value is 63
parameter weight_size   = 5;                              // maximum value may be 31
parameter word_size_out = sample_size + weight_size + 3;  // log2(2^6*2^5*(order+1))

output reg [word_size_out -1: 0] FIR_out;

input      [sample_size   -1: 0] Sample_in;
input                            clock, reset;

// Filter coefficients
parameter b0 = 5'd3;
parameter b1 = 5'd7;
parameter b2 = 5'd20;
parameter b3 = 5'd7;
parameter b4 = 5'd3;

reg [sample_size -1 : 0] Sample_Array [1: FIR_order];
integer k;

reg [word_size_out -1: 0] PR0 [0 : FIR_order];
reg [word_size_out -1: 0] PR1 [1 : FIR_order];
reg [word_size_out -1: 0] PR2 [2 : FIR_order];
reg [word_size_out -1: 0] PR3 [3 : FIR_order];


always @ (posedge clock)
     if (reset == 1) begin

         // The input shift register
         for (k = 1; k <= FIR_order; k = k+1)
              Sample_Array[k] <= 0;

         // The pipeline register PR0
         for (k = 0; k <= FIR_order; k = k+1)
              PR0[k]     <= 0;
         // The pipeline register PR1
         for (k = 1; k <= FIR_order; k = k+1)
              PR1[k]     <= 0;
         // The pipeline register PR2
         for (k = 2; k <= FIR_order; k = k+1)
              PR2[k]     <= 0;
         // The pipeline register PR3
         for (k = 3; k <= FIR_order; k = k+1)
              PR3[k]     <= 0;

         // The output register
         FIR_out         <= 0;
         end

     else begin
         // The input shift register
         Sample_Array [1]  <= Sample_in;
           for (k = 2; k   <= FIR_order; k = k+1)
           Sample_Array[k] <= Sample_Array[k-1];

         // The Pipeline Register PR0
         PR0[0] <= b0 * Sample_in;
         PR0[1] <= b1 * Sample_Array[1];
         PR0[2] <= b2 * Sample_Array[2];
         PR0[3] <= b3 * Sample_Array[3];
         PR0[4] <= b4 * Sample_Array[4];

         // Pipeline Register PR1
         PR1[1] <= PR0[0] + PR0[1];
         PR1[2] <= PR0[2];
         PR1[3] <= PR0[3];
         PR1[4] <= PR0[4];

         // Pipleline Register PR2
         PR2[2] <= PR1[1] + PR1[2];
         PR2[3] <= PR1[3];
         PR2[4] <= PR1[4];

         // Pipleline Register PR3
         PR3[3] <= PR2[2] + PR2[3];
         PR3[4] <= PR2[4];

         // The output register
         FIR_out  <= PR3[3] + PR3[4];
         end

endmodule
```



Matching results between all the 3 designs confirm the data coherency for the pipelined design.



As we see from the simulation results, each pipelining register passes the values saved in its array to the next one. The two top values are added together while the rest pass as is to the next register. The throughput (new output) is available at every clock cycle. With the pipelining, we can shorten the longest propagation delay for the combinational logic between the registers, which will consequently allows reducing the clock period and increasing the clock frequency. With an output available at every clock cycle, the throughput of the module increases. The pipelining gives a better efficiency of hardware utilization.

```
module Pipeline_FIR ( FIR_pipeline_A, FIR_pipeline_B, FIR_MAC, Sample_in, clock, reset);

parameter FIR_order     = 4;
parameter sample_size   = 6;                                  // maximum sample value is 63
parameter weight_size   = 5;                                  // maximum value may be 31
parameter word_size_out = sample_size + weight_size + 3;  // maximum possibe output 63*31*(4+1)

//output  [word_size_out -1: 0] FIR_assign;
output  [word_size_out -1: 0] FIR_MAC;
output  [word_size_out -1: 0] FIR_pipeline_A;
output  [word_size_out -1: 0] FIR_pipeline_B;

input      [sample_size   -1: 0] Sample_in;
input                            clock, reset;

wire  [word_size_out -1: 0] FIR_assign;

FIR_noPipeline M1 ( FIR_MAC, FIR_assign, Sample_in, clock, reset);
Pipeline_FIR_a M2 ( FIR_pipeline_A, Sample_in, clock, reset);
Pipeline_FIR_b M3 ( FIR_pipeline_B, Sample_in, clock, reset);

endmodule
```

Decided not to include FIR_assign as an output for the top module as it is identical to the FIR_MAC.

```
module FIR_Pipeline_tb ();

parameter FIR_order     = 4;
parameter sample_size   = 6;                                  // maximum sample value is 63
parameter weight_size   = 5;                                  // maximum value may be 31
parameter word_size_out = sample_size + weight_size + 3;  // maximum possibe output 63*31*(4+1)
parameter product_size  = sample_size + weight_size;

wire [word_size_out -1: 0] FIR_MAC;
wire [word_size_out -1: 0] FIR_pipeline_B;
wire [word_size_out -1: 0] FIR_pipeline_A;

reg      [sample_size   -1: 0] Sample_in;
reg                            clock, reset;

Pipeline_FIR UUT ( FIR_pipeline_A, FIR_pipeline_B, FIR_MAC, Sample_in, clock, reset);
```

```
// Probes to observe the pipleine register PR0
wire [product_size -1 : 0]  PR00;     assign PR00 = UUT.M3.PR0[0];
wire [product_size -1 : 0]  PR01;     assign PR01 = UUT.M3.PR0[1];
wire [product_size -1 : 0]  PR02;     assign PR02 = UUT.M3.PR0[2];
wire [product_size -1 : 0]  PR03;     assign PR03 = UUT.M3.PR0[3];
wire [product_size -1 : 0]  PR04;     assign PR04 = UUT.M3.PR0[4];

// Probes to observe the pipleine register PR1
wire [product_size -1 : 0]  PR11;     assign PR11 = UUT.M3.PR1[1];
wire [product_size -1 : 0]  PR12;     assign PR12 = UUT.M3.PR1[2];
wire [product_size -1 : 0]  PR13;     assign PR13 = UUT.M3.PR1[3];
wire [product_size -1 : 0]  PR14;     assign PR14 = UUT.M3.PR1[4];

// Probes to observe the pipleine register PR2
wire [product_size -1 : 0]  PR22;     assign PR22 = UUT.M3.PR2[2];
wire [product_size -1 : 0]  PR23;     assign PR23 = UUT.M3.PR2[3];
wire [product_size -1 : 0]  PR24;     assign PR24 = UUT.M3.PR2[4];

// Probes to observe the pipleine register PR3
wire [product_size -1 : 0]  PR33;     assign PR33 = UUT.M3.PR3[3];
wire [product_size -1 : 0]  PR34;     assign PR34 = UUT.M3.PR3[4];

initial
begin
clock = 0;
forever
#5 clock = ~clock;
end

initial
begin
reset = 1;
#40 reset = 0;
end

initial
begin
Sample_in = 0;   #100
Sample_in = 1;   #10     // impulse response
Sample_in = 0;   #100
Sample_in = 10;  #50     // same input over 5 clock cycles
Sample_in = 0;   #100
Sample_in = 1;   #10
Sample_in = 2;   #10
Sample_in = 8;   #10
Sample_in = 2;   #10
Sample_in = 1;   #10
Sample_in = 0;   #100
Sample_in = 63;  #100
Sample_in = 0;
end

endmodule
```