Digital filter

Delay modelling

Problem: 4-Point Moving Average (MA4) Filter in RTL

Goal:

Design and verify a synchronous RTL module that smooths a discrete-time input sequence $x[n]$ by replacing each sample with the average of itself and the previous three samples:

$$y[n] = \frac{x[n] + x[n-1] + x[n-2] + x[n-3]}{4}.$$

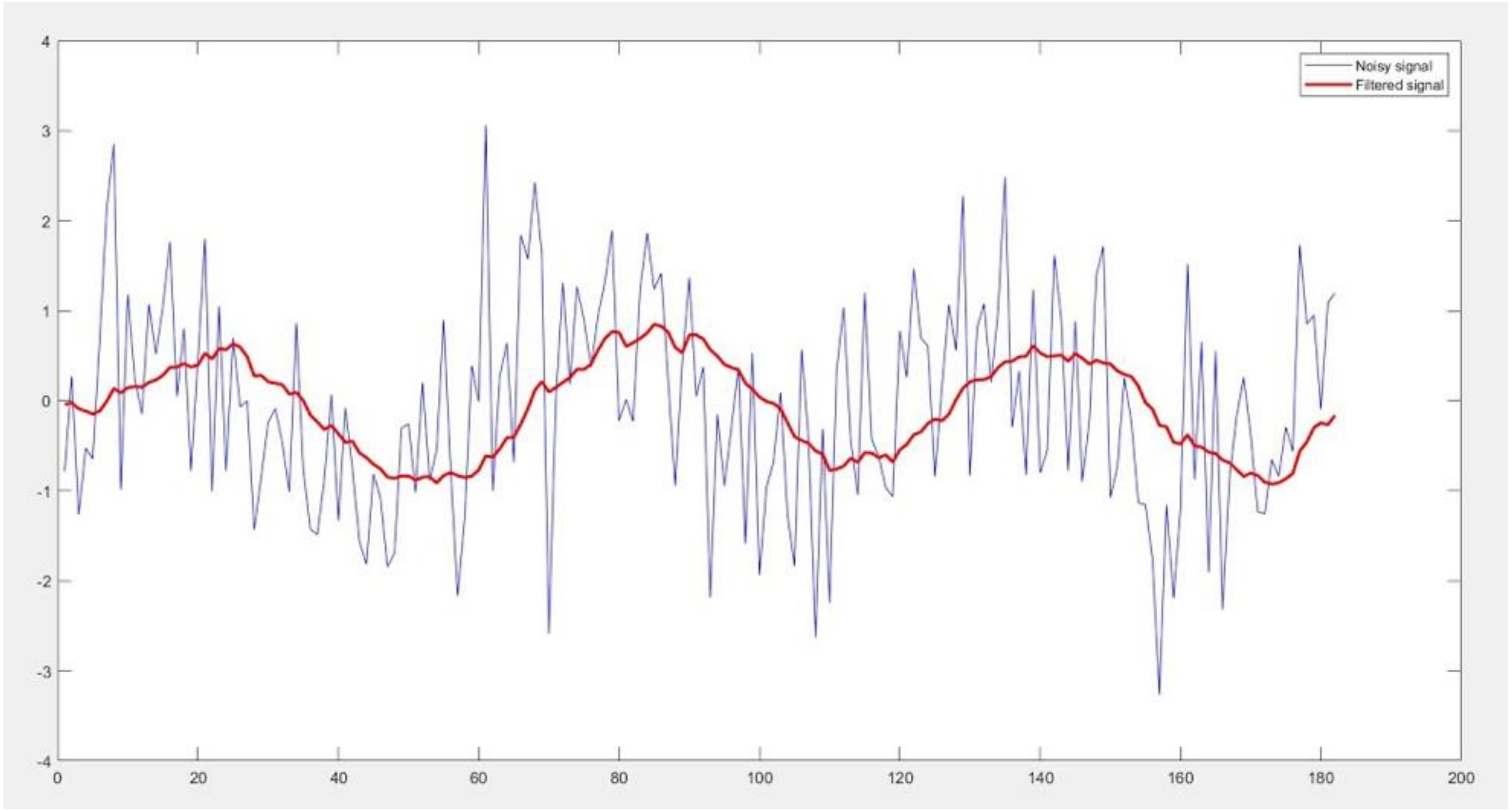
Clock	x_in	x1	x2	x3	y_out = (x_in+x1+x2+x3)/4
1	10	0	0	0	2
2	20	10	0	0	7
3	15	20	10	0	11
4	25	15	20	10	17
5	20	25	15	20	20
6	10	20	25	15	17

✅ Interpretation

•During the **first (N−1) = 3 clock cycles**, the filter **has not yet “filled”** its memory window.

•From the **4th clock onward**, it produces **true averages** over 3 valid samples.

Clock	x_in	x1	x2	x3	y_out = (x_in+x1+x2+x3)/4
1	10	0	0	0	2
2	20	10	0	0	7
3	15	20	10	0	11
4	25	15	20	10	17
5	20	25	15	20	20
6	10	20	25	15	17


Noisy signal
Filtered signal
4
3
2
1
0
-1
-2
-3
-4
0
20
40
60
80
100
120
140
160
180
200

```
module moving_avg_4_filter (
    input  wire       clk,
    input  wire       rst,
    input  wire [7:0] x_in,
    output reg  [7:0] y_out
);
    // Shift registers for previous samples
    reg [7:0] x1, x2, x3;

    wire [9:0] sum;

    wire [7:0] avg;
```

```verilog
// Intermediate sum (max 4*255=1020 -> needs 10 bits)
assign sum = x_in + x1 + x2 + x3;

// Average (integer division by 4 using right shift)

assign avg = sum >>2;
```

```
// Intermediate sum (max 4*255=1020 -> needs 10 bits)
assign sum = x_in + x1 + x2 + x3;

// Average (integer division by 4 using right shift)
 assign avg = sum[9:2];
```

```verilog
    // Sequential logic: update registers and output
    always @(posedge clk) begin
        if (rst) begin
            x1 <= 0;
            x2 <= 0;
            x3 <= 0;
            y_out <= 0;
        end else begin
            x3 <= x2;
            x2 <= x1;
            x1 <= x_in;
            y_out <= avg;
        end
    end
endmodule
```

Write the code for Parameterized 4-Point Moving Average Filter
Unsigned N-bit input, output each clock

**Concatenation Operator `{ , }`

```verilog
{A, B, C, ...}
```

Concatenation means **joining signals or constants end-to-end** (bitwise).

🔹 Example 1 – Join two 4-bit values

```verilog
wire [3:0] upper = 4'b1010;
wire [3:0] lower = 4'b1100;

wire [7:0] combined = {upper, lower};
```

Result:

```ini
combined = 8'b10101100
```

It literally **glues** the bits of `upper` and `lower` together.

🔹 Example 2 – Combine different sizes

```verilog
wire [1:0] a = 2'b11;
wire [2:0] b = 3'b010;
wire [4:0] c = {a, b};  // total 5 bits
```

Result:

```ini
c = 5'b11010
```

So `{a,b}` → first bits of `a`, then bits of `b`.

**Replication Operator `{N{expression}}`

```verilog
{N{expression}}
```

eplication means **repeat an expression N times and concatenate the results.**

◆ Example 1 – Make all zeros or ones

```verilog
wire [7:0] all_zero = {8{1'b0}};   // 8 zeros: 00000000
wire [7:0] all_ones = {8{1'b1}};   // 8 ones : 11111111
```

🔹 Example 2 – Repeat a 2-bit pattern

```verilog
wire [7:0] pattern = {4{2'b10}};
```

Result:

```ini
pattern = 8'b10101010
```

So `{4{2'b10}}` → `2'b10` repeated 4 times.

◆ Example 3 – Combine replication and concatenation

verilog

```verilog
wire [11:0] mix = {{3{2'b01}}, 2'b11};
```

🔷 Example 3 – Combine replication and concatenation

```verilog
wire [11:0] mix = {{3{2'b01}}, 2'b11};
```

Steps:

- `{3{2'b01}}` → `2'b01 2'b01 2'b01` = `6'b010101`
- Then add `2'b11`
- Final result: `8'b01010111`

Operator	Syntax	Purpose	Example	Result
Concatenation	`{a, b, c}`	Joins signals end-to-end	`{4'b1100, 4'b0011}`	`8'b11000011`
Replication	`{N{expr}}`	Repeats an expression N times	`{4{2'b10}}`	`8'b10101010`

Method	Syntax Example
Decimal zero	`a = 8'd0;`
Binary	`a = 8'b00000000;`
Hex	`a = 8'h00;`
Replication	`a = {8{1'b0}};`
Concatenation	`a = {1'b0,1'b0,...};`

```verilog
module moving_avg_4_param #(
    parameter integer WIDTH = 8   // input/output width
) (
    input  wire                clk,
    input  wire                rst,     // synchronous active-high reset
    input  wire [WIDTH-1:0]    x_in,    // current input
    output reg  [WIDTH-1:0]    y_out    // averaged output
);
    // Delay elements
    reg [WIDTH-1:0] x1, x2, x3;

    // Sum width = WIDTH + log2(4) = WIDTH + 2
    wire [WIDTH+1:0] sum = x_in + x1 + x2 + x3;

    // Divide by 4 → right-shift by 2 bits (hardware-efficient)
    wire [WIDTH-1:0] avg = sum>>2;

    // Sequential update
    always @(posedge clk) begin
        if (rst) begin
            x1    <= {WIDTH{1'b0}};
            x2    <= {WIDTH{1'b0}};
            x3    <= {WIDTH{1'b0}};
            y_out <= {WIDTH{1'b0}};
        end else begin
            x3    <= x2;
            x2    <= x1;
            x1    <= x_in;
            y_out <= avg;
        end
    end
endmodule
```

Problem Statement:

You are required to design a **4-stage shift register** in Verilog.
The circuit should accept an **8-bit parallel input** `data_in` and output the data after it passes through **four flip-flops connected in series**.

At every positive clock edge:

- The first register should capture the current input (`data_in`).
- Each subsequent register should capture the value of the previous register from the **previous clock cycle**.

The reset should be **synchronous** and active-high.
When reset is active, all registers must be cleared to 0.

Behavioral Description:

- Use four internal 8-bit registers `r0`, `r1`, `r2`, `r3`.
- Use **non-blocking assignments** (`<=`) for sequential updates.
- The output `data_out` should be connected to the output of the last register (`r3`).

Hint:

Each register represents one clock-cycle delay, so:

$$\text{data\_out}(n) = \text{data\_in}(n-3)$$

```
Clock:       ↑    ↑    ↑    ↑    ↑    ↑    ↑
Input x:    x0   x1   x2   x3   x4   x5   x6
Output y:   --   --   --   x0   x1   x2   x3
                 <--- 3-clock delay --->
```

Clock Cycle (n)	Input `data_in = x(n)`	r0	r1	r2	r3	data_out (= r3)
0 *(Reset)*	—	0	0	0	0	0
1	x(0)	x(0)	0	0	0	0
2	x(1)	x(1)	x(0)	0	0	0
3	x(2)	x(2)	x(1)	x(0)	0	0
4	x(3)	x(3)	x(2)	x(1)	x(0)	**x(0)**
5	x(4)	x(4)	x(3)	x(2)	x(1)	**x(1)**
6	x(5)	x(5)	x(4)	x(3)	x(2)	**x(2)**
7	x(6)	x(6)	x(5)	x(4)	x(3)	**x(3)**
8	x(7)	x(7)	x(6)	x(5)	x(4)	**x(4)**

Now **replace all non-blocking assignments (`<=`) with blocking assignments (`=`)** inside the `always` block and simulate again.

verilog Copy code

```verilog
r0 = data_in;
r1 = r0;
r2 = r1;
r3 = r2;
```

Assignment Type	Behavior	Output Relation
Non-blocking (`<=`)	Sequential delay pipeline	`data_out = x(n-3)`
Blocking (`=`)	Cascaded combinational update	`data_out = x(n)`

- `x[n]` = value of `data_in` at clock cycle *n*,
  then:

Register	**Contents after clock *n***	**Expression**
`r0[n]`	current input	`x[n]`
`r1[n]`	previous input	`x[n-1]`
`r2[n]`	input two cycles ago	`x[n-2]`
`r3[n]`	input three cycles ago	`x[n-3]`
data_out[n]	—	`x[n-3]`

Problem Statement

Design a **sequential circuit** in Verilog to generate the **Fibonacci number sequence**.

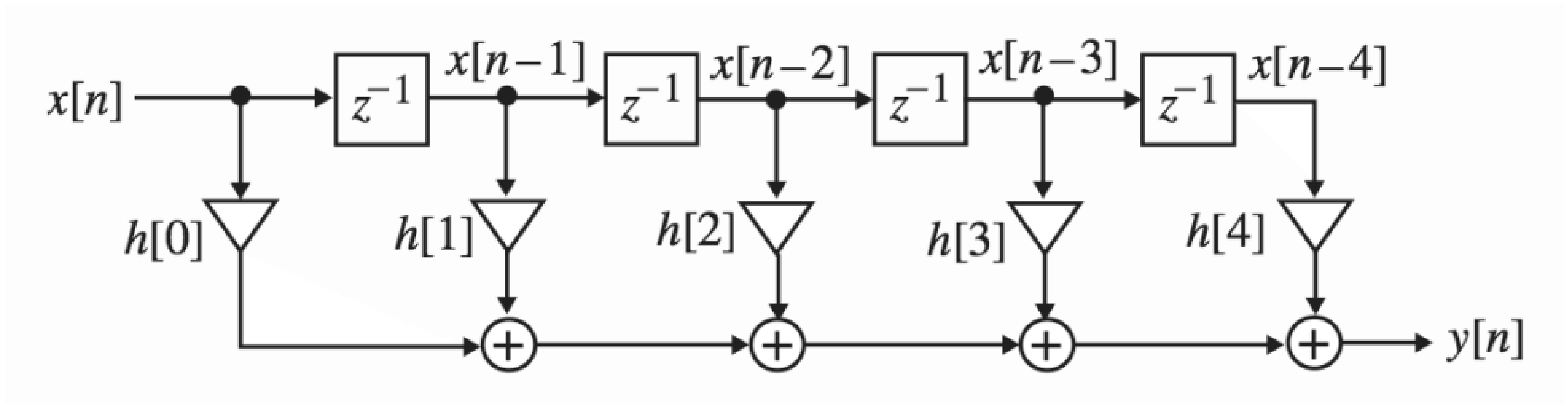
The Fibonacci sequence is defined as:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

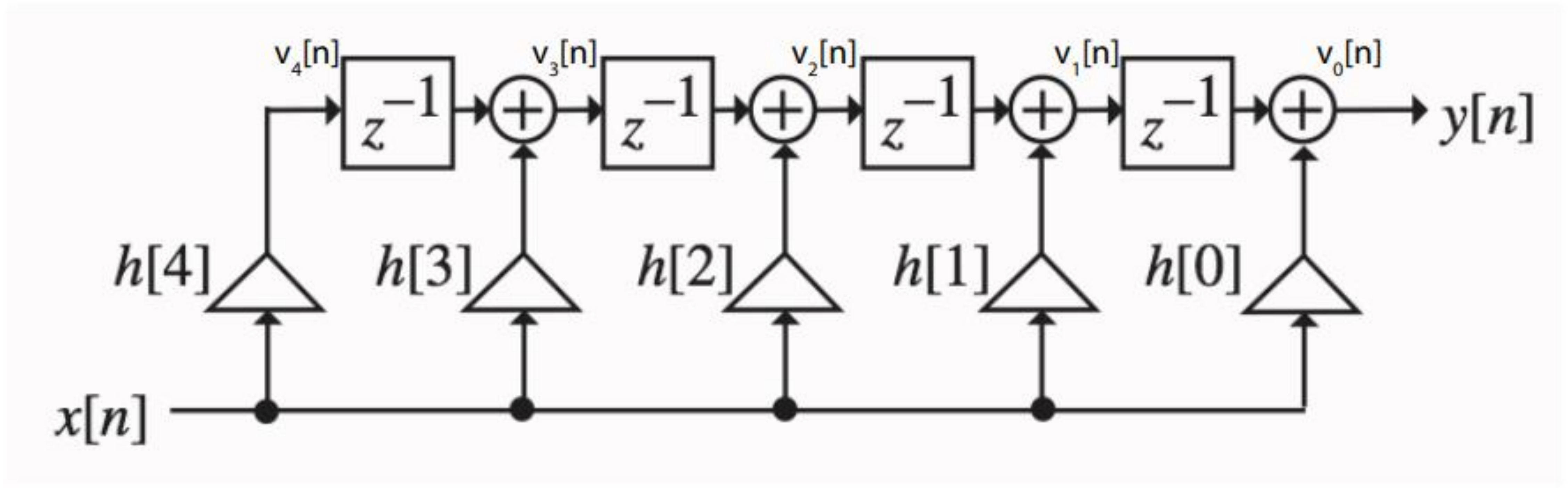
Your Verilog module should use **two registers** to store the previous two numbers and compute the next Fibonacci number at **each positive clock edge**.

Clock	a	b	fib_out (next = a + b)
Reset	0	1	0
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
6	5	8	13

FIR Direct Form



FIR Direct Form Transposed



$v_4[n]$
$v_3[n]$
$v_2[n]$
$v_1[n]$
$v_0[n]$
z^{-1}
z^{-1}
z^{-1}
z^{-1}
$y[n]$
$h[4]$
$h[3]$
$h[2]$
$h[1]$
$h[0]$
$x[n]$