The University of Texas at Dallas

ASIC Assignment-5

AXM220237, UXK220003

Adithya Prathipna Mullan Koni, Usman Khan
2-26-2024

Module Explanation:

This Verilog module is a filter that processes incoming data based on certain coefficients and shifts. Here's a brief explanation of its functionality:

- Inputs:
  - `clk`: Clock signal
  - `rj`: A 16-bit input representing some control signal
  - `sign`: A single-bit input representing another control signal
  - `shift`: An 8-bit input representing yet another control signal
  - `data`: A 40-bit input representing the data to be processed
- Outputs:
  - `out`: A 40-bit output representing the filtered data
  - `flag`: A single-bit output indicating some flag status
- Internal Registers:
  - Various registers (`x_arr`, `r_arr`, `shift_coeff`, `sign_coeff`, `u`, `temp`, `temp1`, `temp2`) are used for storing intermediate values and coefficients.
- Operation:
  - The module processes incoming data in sequential steps based on the state machine defined within the `always` block triggered by the positive edge of the clock signal (`clk`).
  - Depending on the current state, the module performs tasks such as storing coefficients, storing data, and filtering data using the stored coefficients and shifts.
  - The filtered output is stored in the `out` register, and the `flag` output is updated accordingly.

Overall, this module implements a filter algorithm that operates on incoming data using predefined coefficients and shifts, producing a filtered output. The state machine controls the flow of operations, ensuring that the filtering process is carried out sequentially and efficiently.

States Description:

1. **State 0**:
    - In this state, the module initializes or updates internal registers based on the input data, control signals, and coefficients.
    - It sequentially stores the input data, sign coefficients, shift coefficients, and control signals into appropriate registers (`r_arr`, `shift_coeff`, `sign_coeff`, `x_arr`) until specific conditions are met.
    - Once the necessary data is stored, it transitions to State 1 to begin processing.
2. **State 1**:
    - This state prepares for data processing.
    - It resets certain variables (`i`, `j`, `temp1`, `temp2`, `pos`) and sets the `flag` output to indicate that processing is underway.
    - If there's more data to process, it transitions to State 2; otherwise, it goes back to State 0 to wait for new input.
3. **State 2**:
    - This state processes the incoming data.
    - It iterates through each set of coefficients and performs filtering operations on the data.
    - The filtering process involves accumulating data points according to the coefficients and shifts specified.
    - Once all data points have been processed, it transitions back to State 1 to prepare for the next round of processing.
4. **State 3**:
    - This state is a sub-state of State 2.
    - It iterates through each data point and applies the filtering operation based on the current coefficients and shifts.
    - It accumulates filtered values (`u`) and updates the position (`pos`) within the data stream.
    - Once all data points have been processed for the current set of coefficients, it transitions back to State 2 to continue processing the next set of coefficients.

Overall, these states control the sequential execution of the filter algorithm, ensuring that data is processed systematically and efficiently. Each state performs specific tasks within the filtering process, leading to the generation of filtered output data.

Design Code:

```
module filter(clk, rj, sign, shift, data, out, flag);

input clk;
input [15:0] rj;
input sign;
input [7:0] shift;
input [39:0] data;
output reg [39:0] out;
output reg flag;

reg [39:0] x_arr [0:1499];
reg [15:0] r_arr [0:15];
reg [7:0] shift_coeff [0:511];
reg sign_coeff [0:511];
reg [39:0] u;
reg [39:0] temp;
reg [39:0] temp1;
reg [39:0] temp2;

integer i;
integer j;
integer n;
integer pos;
integer state = 0;
integer index = 0;
integer rsize = 16;
integer coeff_size = 512;
integer data_size = 1500;


always@(posedge clk) begin
        case(state)
                0: begin // load respective arrays with input values

                        if(index < rsize) begin
                                r_arr[index] = rj; //array of r values
                                sign_coeff[index] = sign; //array of sign values of the coefficent
                                shift_coeff[index] = shift; // array of magnitude of coefficient values
                                x_arr[index] = data; // sign extended input data array
                                index = index + 1;
                        end
                        else if(index < coeff_size) begin
                                sign_coeff[index] = sign;
                                shift_coeff[index] = shift;
                                x_arr[index] = data;
```

```
                index = index + 1;
        end
        else if(index < data_size) begin
                        //out_data_1[index] = 0;
                        x_arr[index] = data;
                        index = index + 1;
        end
        else begin
                index = 0;
                n = 0;
                state = 1;
        end
end
1: begin// start computation for an input value
        flag = 0;
        if(n < data_size) begin
                i = 0;
                j = 0;
                temp1 = 40'b0;
                temp2 = 40'b0;
                pos = 0;
                state = 2;
        end
        else begin
                state = 0;
        end
end
2: begin
        if(i < rsize) begin
                u = 40'b0;
                temp = 40'b0;
                j = pos;
                state = 3;
        end
        else begin
                flag = 1;
                out = temp1;
                n = n + 1;
                state = 1;
        end
end
3: begin //Computation of each u term, i.e. u1, u2, u3...
        if(j < (pos +  r_arr[i])) begin
                temp = 0;
                if(n >= shift_coeff[j])begin
                        if(sign_coeff[j]) begin
                                temp = x_arr[n - shift_coeff[j]];
```

```
                                        temp = (~temp) + 1;
                                    end
                                    else begin
                                        temp = x_arr[n - shift_coeff[j]];
                                    end
                                end
                                u = u + temp;
                                j = j + 1;
                            end
                            else begin//Accumulating u terms
                                pos = pos +  r_arr[i];
                                temp1 = temp1 + u;
                                state =4;
                            end
                        end
                    4: begin//Performing shift in the next cycle
                                temp2[39:0] = {temp1[39], temp1[39:1]};//right shift
                                temp1 = temp2;
                                i = i + 1;
                                state = 2;
                            end
        endcase
end
endmodule
```

TestBench:

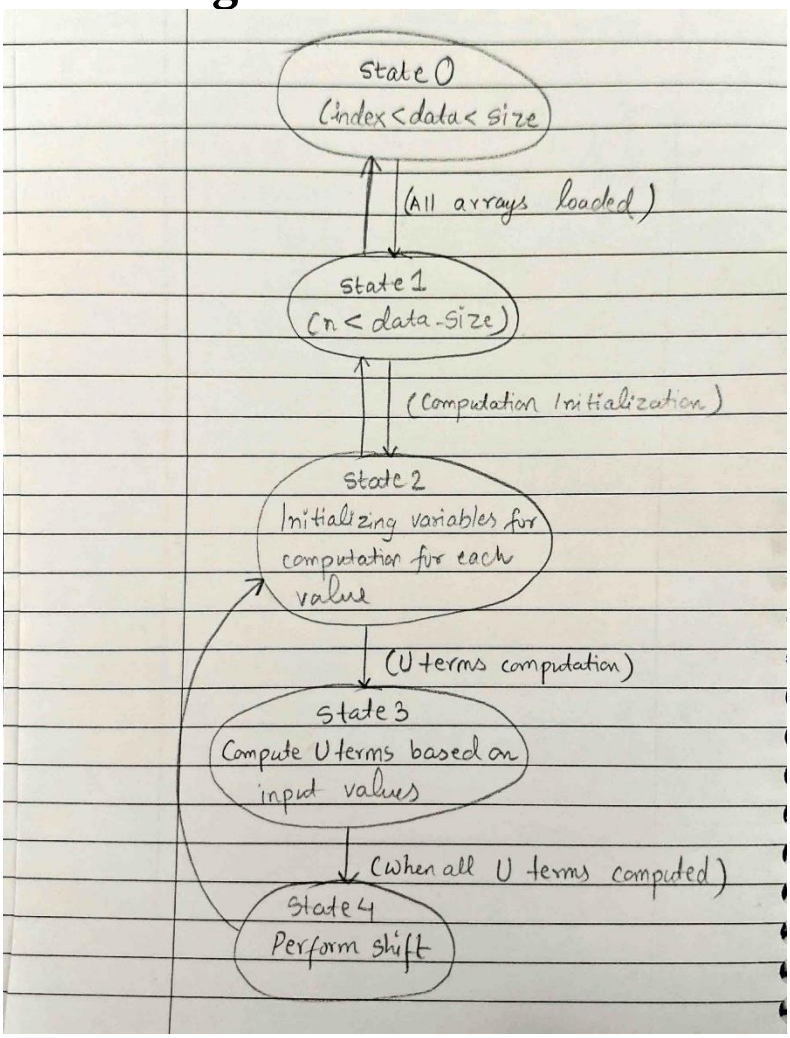
```
`timescale 1ns / 1ps
module filter_tb;

	// Inputs
	reg clk;
	reg [15:0] rj;
	reg sign;
	reg [7:0] shift;
	reg [39:0] data;
	// Outputs
	wire [39:0] out_data;
	wire flag;
	//Memories and variables
	reg [15:0] mem [0:2027];
	reg [15:0] in_data [0:1499];
	reg [39:0] in_new_data [0:1499];
	reg [15:0] temp_in;
	reg [39:0] temp_data;
	reg [15:0] in_rj [0:15];
	reg [15:0] in_coeff [0:511];
	reg [7:0] shift_coeff [0:511];
	reg sign_coeff [0:511];

	integer m;
	integer index = 0;
	integer final_var = 0;
	integer file;
  integer rj_size = 16;
  integer coeff_size = 512;
  integer data_size = 1500;
	integer state_mon;
	reg [39:0] temp_mon;
	reg [39:0] u_mon;

	// Instantiate the Unit Under Test (UUT)
	filter uut (
		.clk(clk),
		.rj(in_rj[index]),
		.sign(sign_coeff[index]),
		.shift(shift_coeff[index]),
		.data(in_new_data[index]),
		.out(out_data),
		.flag(flag)
	);
```

```
  initial
  begin
    clk = 0;
    forever #20 clk = ~clk;
  end

        initial begin
                $system("pwd");
                clk = 0;
                file = $fopen("C:\\Users\\uxk220003\\Desktop\\Verliog
Codes\\Projects\\Filter\\dataout.txt");
                $readmemh("C:\\Users\\uxk220003\\Desktop\\Verliog
Codes\\Projects\\Filter\\data.txt", mem);

                //Seperating inputs, coefficients and rjs from the input file
                for(m = 0; m < rj_size; m = m + 1)
                begin
                        in_rj[m] = mem[m];
                end
                /*$display("Array values:");
                for (integer i = 0; i < 16; i = i + 1)
    begin
    $display("in_new_data[%0d] = %h", i, in_rj[i]);
    end*/

                for(m = 0; m < coeff_size; m = m + 1)
                begin
                        in_coeff[m] = mem[m + rj_size];
                end

                for(m = 0; m < data_size; m = m + 1)
                begin
                        in_data[m] = mem[m + rj_size + coeff_size];
                end

                //Extending the sign bits to make the input of 40 bits

    for (m = 0; m < data_size; m = m + 1)
                begin
                        temp_in = in_data[m];
                        temp_data[39:0] = { {8{temp_in[15] }}, temp_in[15:0] } ; //if temp_in = 6EF4,
then temp_data = 0000 0000 0000 0000 0000 0000 0110 1110 1111 0100
                        in_new_data[m] = temp_data << 16; //left shift the above value by 16 times to
get. in_new_data = 0000 0000 0110 1110 1111 0100 0000 0000 0000 0000
                end

                //Extracting the signs and shift values from the coefficients
```

```
for (m = 0; m < coeff_size; m = m + 1)
        begin
                sign_coeff[m] = in_coeff[m] >> 8;
                shift_coeff[m] = in_coeff[m] & 16'h00FF;
        end

    end

        //Always block for initialization
        always @ (posedge clk) begin : init

                if(index < rj_size) begin
                        rj = in_rj[index];
                        sign = sign_coeff[index];
                        shift = shift_coeff[index];
                        data = in_new_data[index];
                        index = index + 1;
                end

                else if(index < coeff_size) begin
                        sign = sign_coeff[index];
                        shift = shift_coeff[index];
                        data = in_new_data[index];
                        index = index + 1;
                end

                else if(index < data_size) begin
                        data = in_new_data[index];
                        index = index + 1;
                end

                else begin
                        disable init;
                end
        end

        //Always block for generating final output.txt
        always @ (posedge clk)
        begin   : final_txt
                state_mon = uut.state;
                temp_mon = uut.temp1;
                u_mon = uut.u;
                if(flag)
                begin
                $fwrite(file, "%H\n", out_data);
                $display("The value of output is: %h", out_data );
```

```
                final_var = final_var + 1;
                if(final_var == data_size)
                begin
                        $fclose(file);
                        $stop;
                        disable final_txt;
                end
                end
        end

endmodule
```

State Diagram:



Simulation WaveForm: