طراحی سیستم های نهفته

مبتنی بر FPGA

تکلیف کامپیوتری 1

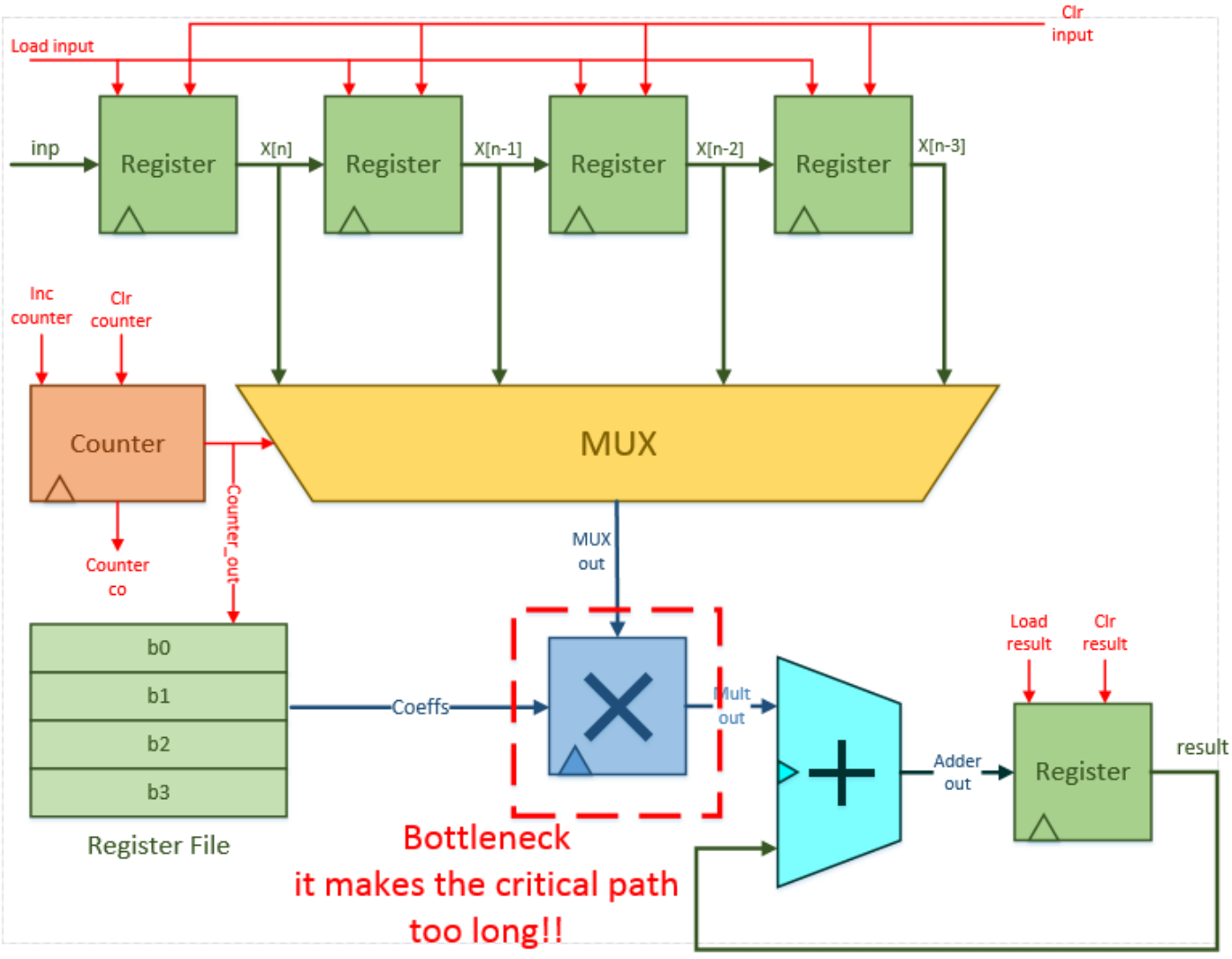
طراحی و پیاده سازی یک فیلتر FIR به همراه درستی سنجی آن

نام و نام خانوادگی: سید صدرا قوامی

شماره دانشجویی: 810199474

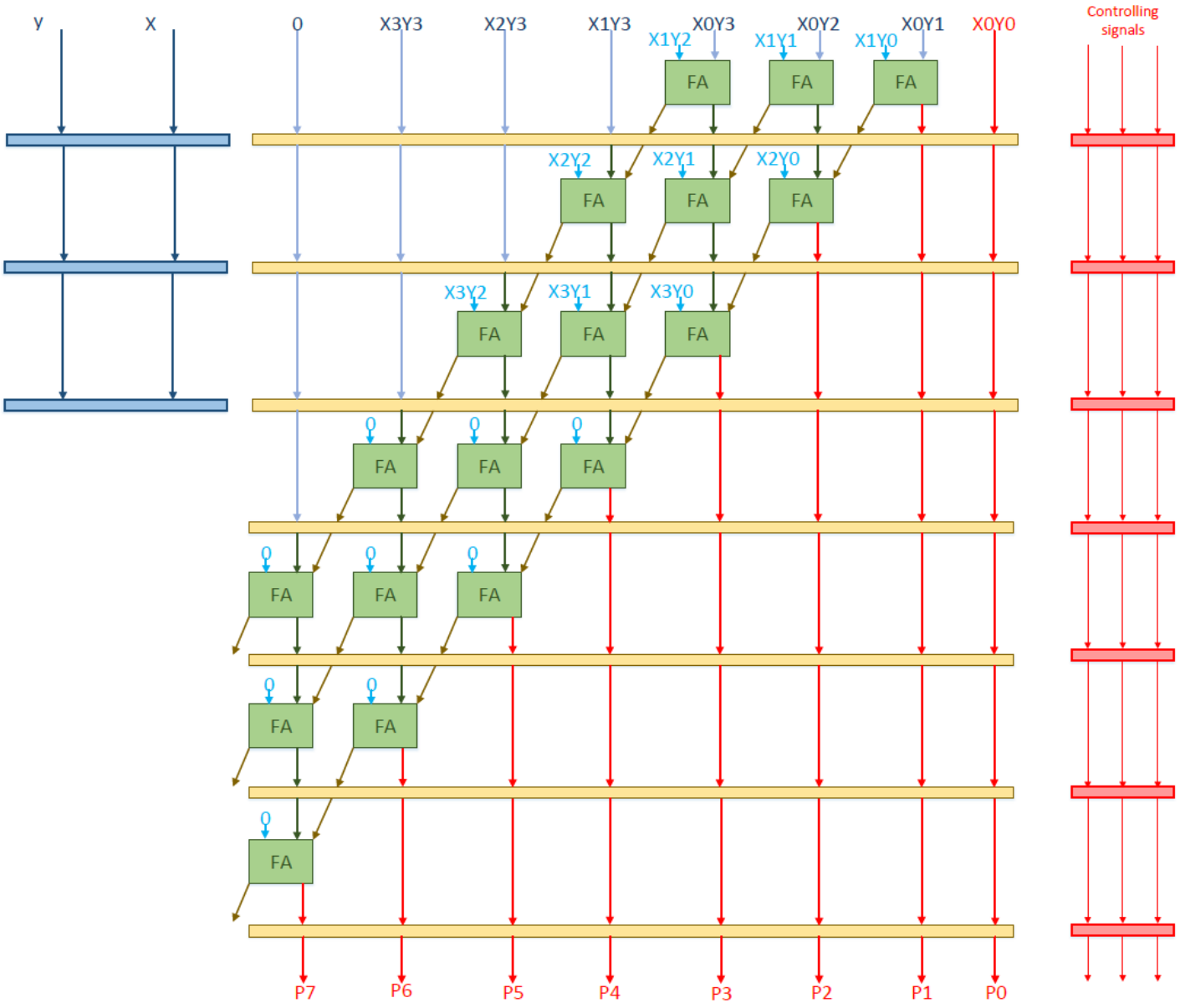
- **طراحی سخت افزاری:**

همانطور که در صورت پروژه خواسته شده، میخواهیم برای پیادهسازی فیلتر FIR یک مسیرداده و کنترلر طراحی کنیم. قیود بیان شده توسط صورت پروژه این است که اولا باید از یک ضرب‌کننده و یک جمع‌کننده استفاده شود و ثانیا باید طراحی مسیرداده و کنترلر تا حد امکان pipeline شده باشد به صورتی که critical path به طرز قابل توجهی کوتاه باشد. طرح اولیه ای که به ذهن بنده رسید در شکل 1 نمایش داده شده است. این پیاده‌سازی یک ایراد قابل توجه دارد و آن هم این است که حتی با وجود قرار دادن رجیسترهای پایپ‌لاین، همچنان critical path بسیار زیاد است زیرا یک ضرب‌کننده کامل که در یک سیکل کلاک مقدار خروجی را آماده میکند بین دو رجیستر قرار میگیرد و همانطور که میدانیم این باعث طولانی‌تر شدن critical path به طرز قابل توجهی میشود.



شکل 1: مسیرداده اولیه در نظر گرفته شده برای فیلتر FIR

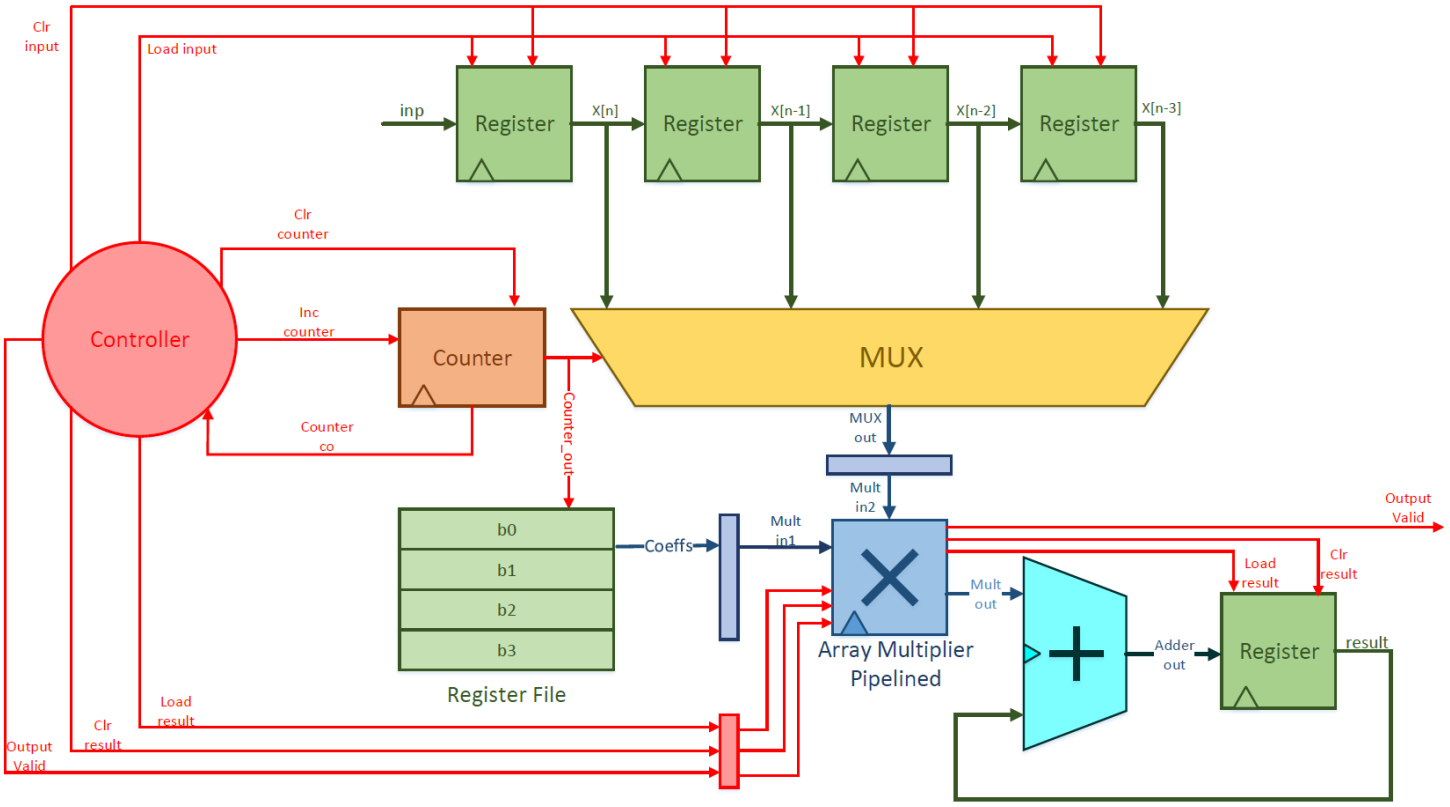
پس در گام اول نیاز است تا ضرب کننده‌ای طراحی کنیم که خود pipeline باشد و critical path کوتاهی داشته باشد. به همین منظور، همان‌گونه که در شکل 2 به تصویر کشیده شده است، یک array multiplier را pipeline کرده ایم. تفاوتی که در نگاه اول به چشم می‌آید این است که به جای اینکه مانند سایر ضرب کننده-های 4 بیتی دارای 4 طبقه full adder باشد، 7 طبقه FA دارد. علت این امر این است که critical path تا حد قابل توجهی کوتاه گردد. به بیان دیگر در طبقه 4 میتوانستیم cout حاصل از هر full adder را به بعدی منتقل کنیم اما این باعث ایجاد یک critical path به طول چهار full adder می‌شد که مطلوب ما نیست و در این پیاده سازی حتی اگر دو عدد 1024 بیتی در هم ضرب شوند، همچنان طول critical path همان یک full adder باقی خواهد ماند و فقط تعداد کلاک لازم برای محاسبه خروجی به واسطه افزایش تعداد طبقات، بیشتر خواهند شد.



شکل 2: پیاده‌سازی ضرب array multiplier 4 بیتی به صورت pipeline

مشکل دیگری که وجود دارد این است که array multiplier قادر به محاسبه ضرب اعداد علامت‌دار نیست و صرفا میتواند اعداد بدون علامت را محاسبه کند. روش حل به کار برده شده به این شکل است که اگر قرار است دو عدد 16 بیتی در هم ضرب شوند که هر دو نیز علامت دار هستند، کافی است این دو عدد را با توجه به روش sign extension به دو عدد 32 بیتی تبدیل کنیم و با استفاده از یک array multiplier حاصل ضرب این دو عدد که 64 بیتی است را محاسبه کنیم و 32 بیت کم ارزش تر آن را به عنوان خروجی ضرب دو عدد 16 بیتی به کار بگیریم.

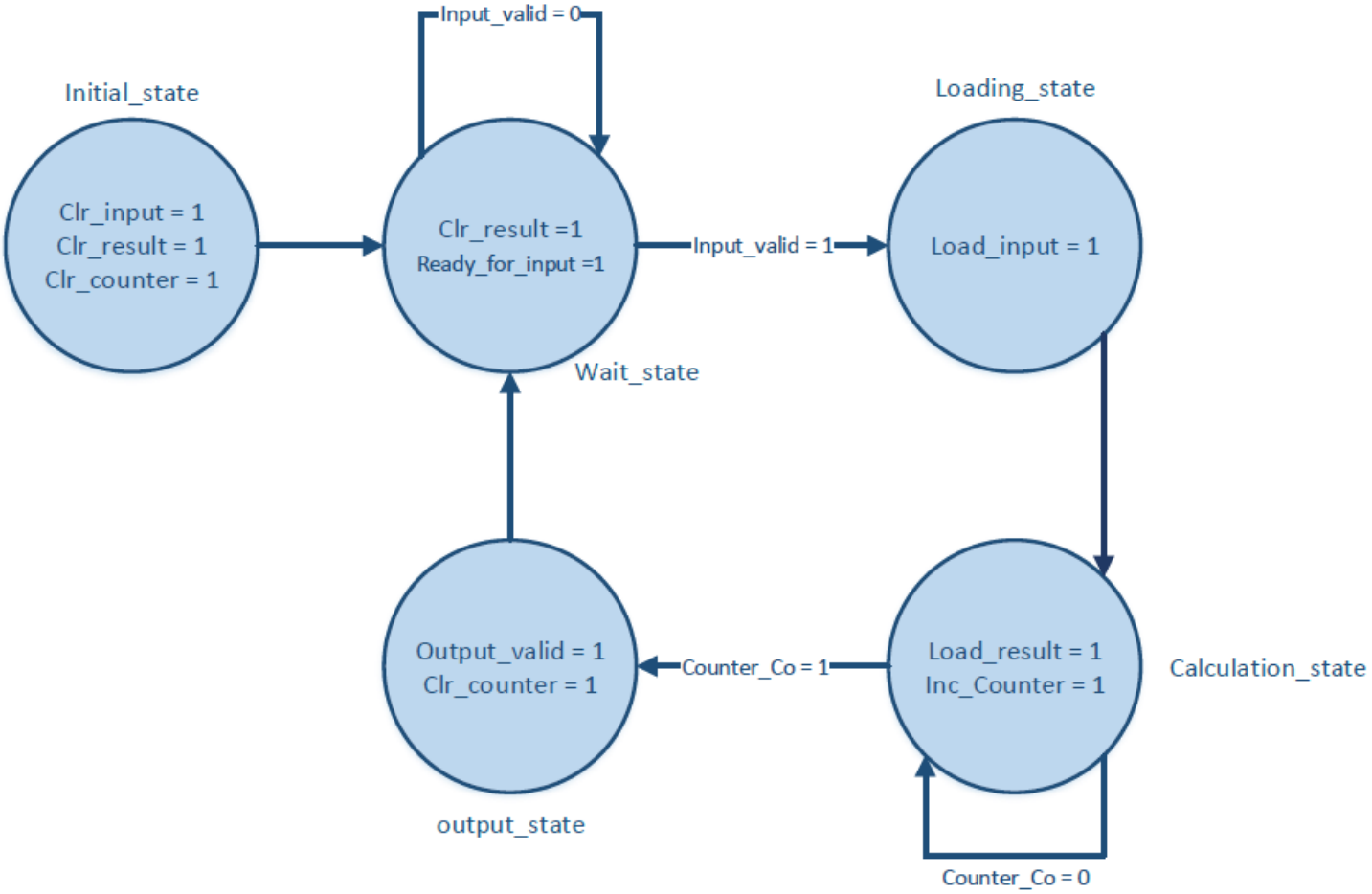
حال با در اختیار داشتن یک ضرب‌کننده که قادر است ضرب اعداد علامت دار را محاسبه کند و همچنین pipeline شده است، میتوانیم یک مسیرداده و کنترلر بهینه‌تر طراحی کنیم. مسیرداده نهایی در شکل 3 به تصویر کشیده شده است. این مسیرداده به صورت کامل pipeline شده است و به این صورت عمل میکند که ابتدا یک ورودی وارد آن میگردد و در رجیستر شماره 0 ذخیره میشود اما به این معنی نیست که داده‌های قبلی از دست می‌روند بلکه در رجیسترهای بعدی لود میشوند تا در روند محاسبه مورد استفاده قرار گیرند. بعد از ورود داده جدید، مولتی‌پلکسر داده تازه وارد را انتخاب میکند و از طرف دیگر ضریب شماره صفر که باید در آن ضرب شود، از یک register file فراخوانی میشوند و هردوی این مقادیر در رجیستر pipe ذخیره میشوند. در کلاک بعدی این دو ورودی وارد خط‌لوله ضرب‌کننده میشوند و باید تعداد قابل توجهی سیکل کلاک را منتظر بمانیم تا نتیجه این ضرب پس از طی کردن تمامی طبقات موجود در ضرب کننده به دست آید. از طرف دیگر، در طی تمامی این مدت، مسیرداده بیکار نمیماند و به واسطه pipeline بودن طراحی، داده های بعدی و ضریب مربوط به آن ها را یکی پس از دیگری میفرستد تا وارد ضرب کننده شوند و حاصلضرب آنها محاسبه گردد. سپس مقادیر محاسبه شده توسط ضرب‌کننده یکی پس از دیگری از آن خارج میشوند و توسط یک جمع‌کننده و یک رجیستر که خروجی آن را ذخیره میکند، حاصل ضرب و مقدار قبلی رجیستر جمع میشود و ذخیره میگردد. این تمام اتفاقی است که باید بیافتد تا مقدار خروجی به درستی محاسبه گردد. وظیفه کنترلر در این مدار برقراری روندی در مسیرداده است که منجر به ایجاد خروجی صحیح گردد.



شکل 3: مسیرداده نهایی برای پیاده‌سازی فیلتر FIR

برای پیاده‌سازی کنترلر دو روش وجود دارد، اول اینکه کنترلر بعد از اینکه به کمک یک counter هر داده و ضریب متناظر آن را در pipeline قرار داد، منتظر بماند تا تمامی این داده ها pipeline را طی کنند و خروجی به دست بیاید، آنگاه output valid را یک کند و آمادگی خود برای گرفتن داده جدید و محاسبه خروجی را اعلام کند. توجه کنید در طول این بازه، کنترلر باید سیگنال های کنترلی اعم از load result و clear result را به طور مستقیم کنترل کند که این امر نیازمند شمارش های متعدد و state های زیادی در کنترلر است. اما روش دیگری وجود دارد که بسیار بهینه‌تر است. کنترلر میتواند سیگنال هایش را از طریق رجیستر های pipeline به طبقات بعدی منتقل کند. این ویژگی باعث میشود ،همانطور که در شکل 4 نمایش داده شده، کنترلر همراه داده ها و ضرایب سیگنال های کنترلی مربوط به هر کدام را همراه آنها بفرستد و پس از اتمام این فرآیند دیگر منتظر محاسبات آنها نشود بلکه به انتظار داده جدیدی بنشیند. در اینجا بنده پای یک سیگنال دیگر را نیز به طراحی خود باز کرده‌ام. اسم این سیگنال ready for input است. اما این سیگنال چگونه به بهینه‌تر شدن عملکرد سیستم کمک میکند؟ این سیگنال لحظه ای فعال میشود که داده‌ها وارد pipeline شده اند و سیگنال های کنترلی که شامل output valid نیز میشوند، نیز همراه آنها ارسال شده‌اند. در این لحظه شاید هنوز تعداد قابل توجهی سیکل باقی مانده است که خروجی نهایی محاسبه گردد اما میتوان ورودی جدید را دریافت کرد و معطل این محاسبات نماند. سیگنال ready for input اعلام میکند شاید هنوز خروجی های

قبلی به دست دریافت کننده نرسیده است اما میتواند ورودی جدید را وارد کند تا روند محاسبات به طرز قابل توجهی سریعتر انجام گردد. در واقع به کمک این سیگنال میتوان کاری کرد که pipeline تنها برای تعداد بسیار کمی سیکل خالی بماند و دائما پر باشد که این روند باعث افزایش بازدهی به طرز قابل توجهی میگردد.



شکل 4 : تصویر کنترلر طراحی فیلتر FIR

عملکرد کنترلر که در شکل 4 به تصویر کشیده شده است، تا حدودی در بخش های قبل توضیح داده شد. اما اجازه بدهید state به state نیز نگاهی به آن بیاندازیم. در initial state ، تمامی رجیستر های موجود در مدار(غیر از رجیستر های pipeline) مقدار صفر را به خود میگیرند تا مدار آماده شروع محاسبات شود. در state بعد که نام wait state برای آن انتخاب شده است، کنترلر منتظر ورودی جدید است و همانطور که میدانیم، آمدن داده جدید با یک شدن مقدار input valid همراه است به همین دلیل اگر مقدار input valid صفر باشد در همین state میماند و اگر یک شود به state بعدی میرود. در گام بعدی نیاز داریم تا ورودی جدید لود شود و مقادیر قبلی یکی به سمت راست شیفت بخورد. در مسیرداده ما این اتفاق تنها با یک شدن سیگنال load input می‌افتد. پس در loading state تنها همین سیگنال یک میشود و از این حالت عبور میکند. در state بعد که نام calculation برای آن برگزیده شده است، کنترلر میخواهد همه داده‌ها و ضرایب

متناظر آنها را وارد خط‌لوله کند. همانطور که در بخش های قبلی به آن اشاره شد، این کار به کمک یک counter اتفاق می‌افتد که این counter وظیفه تولید سیگنال select مولتی‌پلکسر و همین طور آدرس رجیستر فایل را دارد. کنترلر در این state مقدار counter را با استفاده از inc_counter یک واحد زیاد میکند تا مقادیر جدید وارد خط‌لوله شوند. در کنار این موضوع load result را نیز یک میکند اما توجه کنید که این سیگنال به طور مستقیم به رجیستر مقصد متصل نمیشود بلکه باید تمام طول خط‌لوله را طی کند تا به رجیستر مقصد برسد. نکته مهم دیگر این است که کنترلر تا زمانی که cout مربوط به شمارنده یک نشده است در این state باقی میماند و به محض یک شدن این سیگنال، از این state گذر میکند. در state بعدی که output state نام دارد، سیگنال های output valid و clear counter یک میشوند تا مدار را آماده ورود داده جدید کنند. عبور از این state کنترلر را دوباره به wait state برمیگرداند که مدار انتظار داده جدیدی را میکشد.

- **توصیف به کمک Verilog :**

حال که سخت افزار مورد نیاز برای تولید فیلتر FIR را طراحی کرده ایم، میتوانیم به کمک زبان توصیف سخت-افزاری Verilog آن را پیاده‌سازی کنیم. در این پیاده‌سازی، چند نکته رعایت شده که ذکر آنها خالی از لطف نیست. اول اینکه قطعاتی که در مسیرداده استفاده شده اند به صورت ماژول های جداگانه در فایل مختص به خودشان نوشته شده اند. دومین نکته این است که همه ماژول ها به صورت پارامتری پیاده‌سازی شده اند و میتوان با مقادیر مختلف از آنها instance گرفت. نکته آخر که میتوان به آن اشاره کرد این است که در اسم-گذاری ماژولها و سیم ها تلاش شده تا حد قابل قبولی دارای مفاهیم منطقی باشند و کاربری خود را بیان کنند. در ادامه بخشی از کدها که مهمترند را خواهیم دید.

```
genvar i;
assign x_reg[0] = x;
assign y_reg[0] = y;
assign control_regs[0] = control_signals_in;
assign carry_regs_output[0] = 0;
generate
    assign result_regs_output[0][2*WIDTH-1] = 1'b0;

    for (i=0 ; i<2*WIDTH-1 ; i=i+1) begin: bind_result_registers
       Pipe_register#(.WIDTH(2*WIDTH)) result_registers(.clk(clk), .rst(rst), .par_in(result_regs_input[i]), .par_out(result_regs_ou
       Pipe_register#(.WIDTH(CONTROL_SIGNALS_WIDTH)) control_registers(.clk(clk), .rst(rst), .par_in(control_regs[i]), .par_out(con
    end:bind_result_registers

    for(i=0 ; i<2*WIDTH-2; i=i+1)begin: bind_carry_registers
        Pipe_register#(.WIDTH(WIDTH-1)) carry_registers(.clk(clk), .rst(rst), .par_in(carry_regs_input[i]), .par_out(carry_regs_outp
    end: bind_carry_registers

    for(i=0; i<WIDTH-1; i=i+1)begin: bind_x_y_registers
        Pipe_register#(.WIDTH(WIDTH)) x_register(.clk(clk), .rst(rst), .par_in(x_reg[i]), .par_out(x_reg[i+1]));
        Pipe_register#(.WIDTH(WIDTH)) y_register(.clk(clk), .rst(rst), .par_in(y_reg[i]), .par_out(y_reg[i+1]));
    end: bind_x_y_registers

    for(i=0 ; i<WIDTH ; i=i+1)begin: AND_gates
        assign result_regs_output[0][i] = x_reg[0][0] & y_reg[0][i];
    end: AND_gates

    for(i=WIDTH; i<2*WIDTH-1; i++)begin: result_propagation
        assign result_regs_output[0][i] = x_reg[0][i-WIDTH+1] & y_reg[0][WIDTH-1];
    end: result_propagation
endgenerate
```

شکل 5 : کد مربوط به رجیسترهای pipeline در array multiplier

```
52      //Full adders
53      genvar j,k;
54      generate
55          assign result_regs_input[WIDTH-1][2*WIDTH-1] = result_regs_output[WIDTH-1][2*WIDTH-1];
56
57          for(j=0; j<WIDTH-1; j=j+1)begin: First_Part_FA_outer_loop
58
59              for(k=0; k<WIDTH-1; k=k+1)begin: Fisrt_part_FA_inner_loop
60                  Full_adder full_adders(.a(x_reg[j][j+1]&y_reg[j][k]), .b(result_regs_output[j][k+j+1]), .sum(result_regs_input[j][k+j+1
61              end: Fisrt_part_FA_inner_loop
62
63              for(k=0; k<=j; k=k+1)begin: result_propagation_1_1
64                  assign result_regs_input[j][k] = result_regs_output[j][k];
65              end: result_propagation_1_1
66
67              for(k=WIDTH+j; k<2*WIDTH; k=k+1)begin: result_propagation_1_2
68                  assign result_regs_input[j][k] = result_regs_output[j][k];
69              end: result_propagation_1_2
70
71          end: First_Part_FA_outer_loop
72          //
```

شکل 6 : کد مربوط به full adder در array multiplier

```
27      genvar reg_var;
28      assign x[0] = inp;
29      generate
30          for(reg_var=0 ; reg_var < LENGHT ; reg_var = reg_var+1)begin : input_registers
31              Register #(.WIDTH(WIDTH)) inp_reg
32                        (.par_in(x[reg_var]), .par_out(x[reg_var+1]), .clk(clk), .reset(reset), .load(load_input), .clear(clr_input));
33          end: input_registers
34      endgenerate
35
36      MUX #(.WIDTH(WIDTH), .INPUT_NUM(LENGHT)) multiplexer
37          (.inputs(x[1:LENGHT]), .out_put(MUX_out), .select(counter_out));
38
39      Counter #(.COUNT_NUM(LENGHT)) cnt
40              (.clk(clk), .reset(reset), .counter(counter_out), .increament(inc_counter), .clear(clr_counter), .Co(counter_co));
41
42      Register_files #(.WIDTH(WIDTH), .LENGTH(LENGHT)) reg_file
43                     (.address(counter_out), .out_put(coeffs));
44
45      Pipe_register #(.WIDTH(WIDTH)) mux_out_reg (.clk(clk), .rst(reset), .par_in(MUX_out), .par_out(mult_in2));
46      Pipe_register #(.WIDTH(WIDTH)) coeff_reg(.clk(clk), .rst(reset), .par_in(coeffs), .par_out(mult_in1));
47      Pipe_register #(.WIDTH(3)) control_reg(.clk(clk), .rst(reset), .par_in(control_reg_in), .par_out(control_reg_out));
48
49      Signed_multiplier #(.WIDTH(WIDTH), .CONTROL_SIGNALS_WIDTH(3)) multiplier(.clk(clk), .rst(reset), .x(mult_in1), .y(mult_in2),
50                        .result(mult_out), .control_signals_in(control_reg_out), .control_signals_out(control_signals_mult_out));
51
52      assign adder_in = { {(OUTPUT_WIDTH-2*WIDTH){mult_out[2*WIDTH-1]}} , mult_out};
53
54      Adder #(.INPUT_WIDTH(OUTPUT_WIDTH), .OUTPUT_WIDTH(OUTPUT_WIDTH)) adder
55          (.input_1(adder_in), .input_2(result), .out_put(adder_out));
56
57      Register #(.WIDTH(OUTPUT_WIDTH)) result_reg
58              (.par_in(adder_out), .par_out(result), .clk(clk), .reset(reset), .load(control_signals_mult_out[0]), .clear(control_signa
59
60      assign output_valid_out = control_signals_mult_out[2];
```

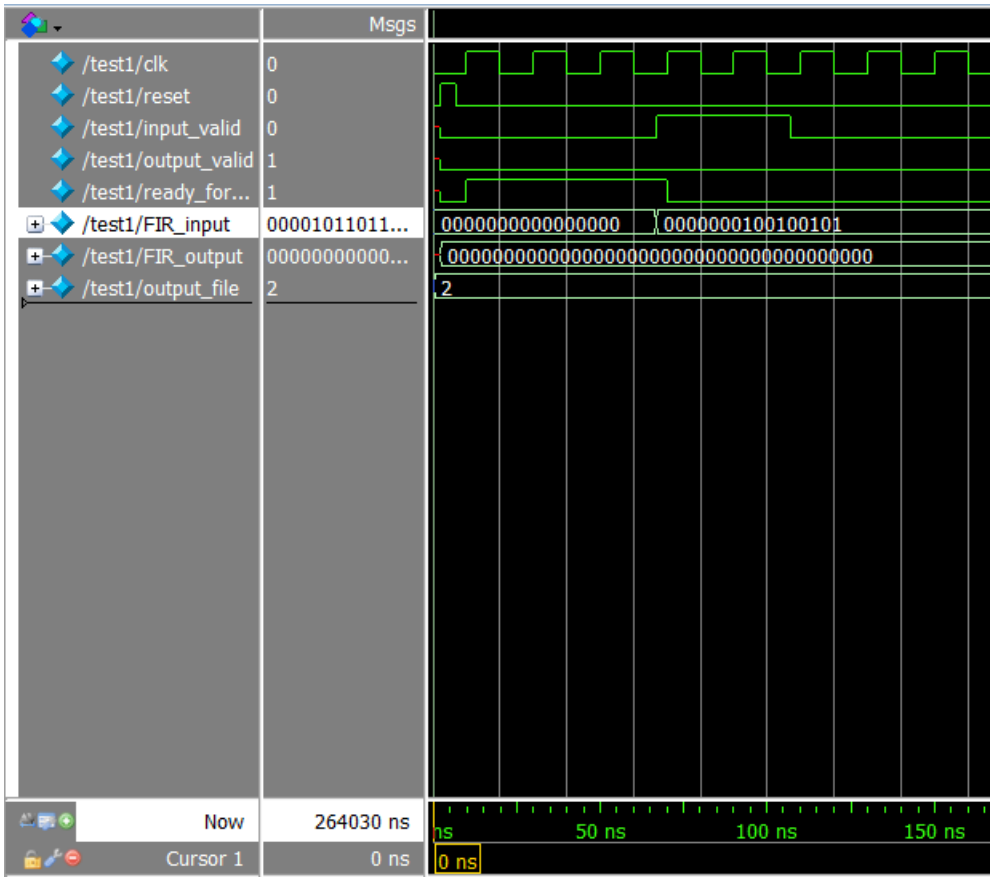
شکل 7 : کد مربوط به datapath

```
14      reg[2:0] present_state, next_state;
15
16      always@(posedge clk, posedge reset)begin
17          if(reset)begin
18              present_state <= INITIAL_STATE;
19          end
20          else begin
21              present_state <= next_state;
22          end
23      end
24
25      always@(present_state, input_valid, counter_co)begin
26          case(present_state)
27              INITIAL_STATE: next_state <= WAIT_STATE;
28              WAIT_STATE: next_state <= input_valid ? LOADING_STATE : WAIT_STATE;
29              LOADING_STATE: next_state <= CALCULATION_STATE;
30              CALCULATION_STATE: next_state <= counter_co ? OUTPUT_STATE : CALCULATION_STATE;
31              OUTPUT_STATE: next_state <= WAIT_STATE;
32              default: next_state <= INITIAL_STATE;
33          endcase
34      end
35
36      always@(present_state)begin
37          {clr_input, load_input, clr_result, load_result, clr_counter, inc_counter, output_valid, ready_for_input} = 8'b0;
38          case(present_state)
39              INITIAL_STATE: {clr_counter, clr_result, clr_input} = 3'b111;
40              WAIT_STATE: {clr_result, ready_for_input} = 2'b11;
41              LOADING_STATE: load_input = 1'b1;
42              CALCULATION_STATE: {load_result, inc_counter} = 2'b11;
43              OUTPUT_STATE: {clr_counter, output_valid} = 2'b11;
44              default: {clr_input, load_input, clr_result, load_result, clr_counter, inc_counter, output_valid, ready_for_input} = 8'b0;
45          endcase
46      end
47  endmodule
```

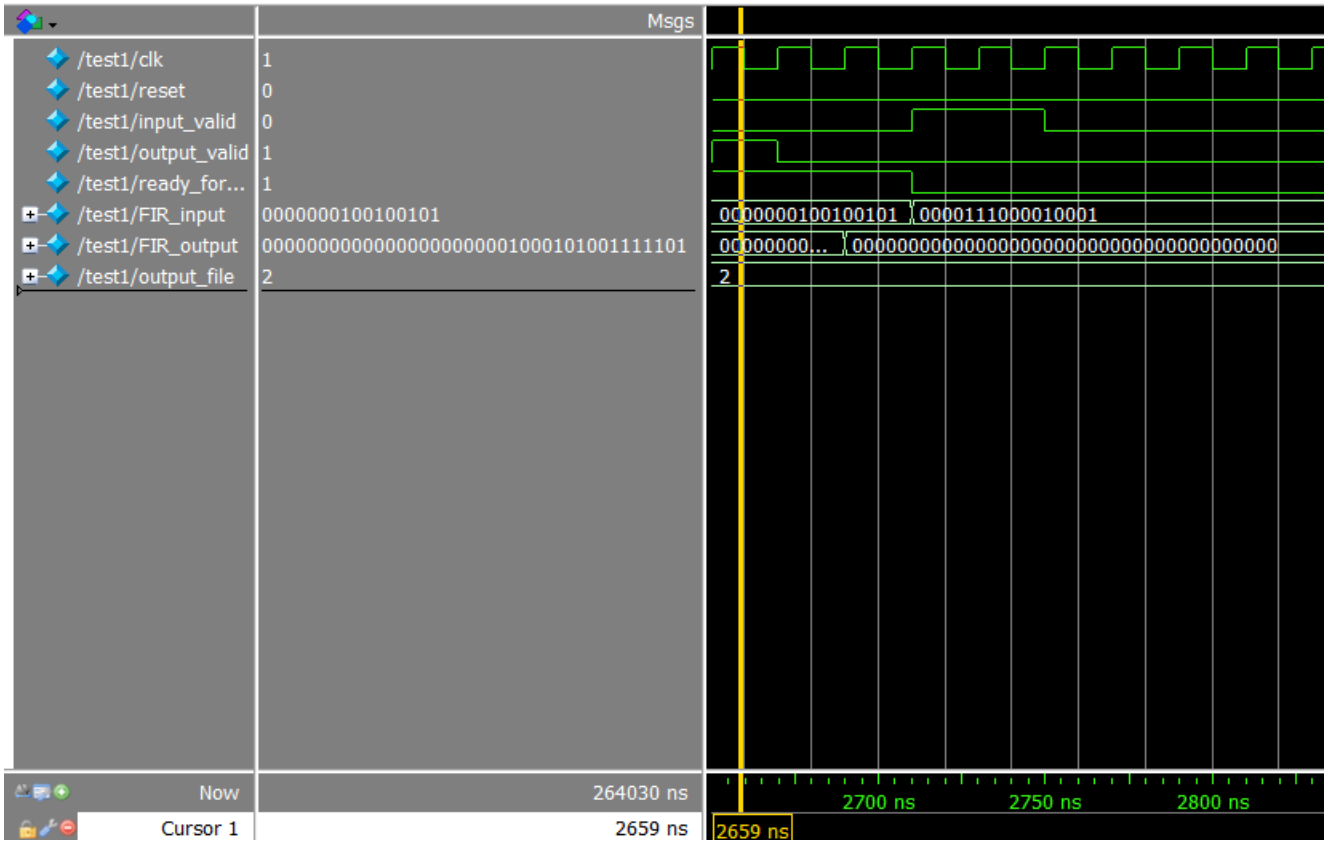
شکل 8 : کد مربوط به controller

- **صحت‌سنجی مدار طراحی شده:**

بعد از پیاده سازی این طراحی با استفاده از وریلاگ، باید به کمک نوشتن testbench از صحت طراحی خود مطمئن شویم. به همین منظور ابتدا از فیلتر طراحی شده با عرض ورودی 16 بیت و طول 64 یک نمونه در testbench خود میگیریم. سپس داده های موجود در فایل "inputs.txt" را یکی پس از دیگری به ماژول مدنظر میدهیم و منتظر میمانیم تا خروجی آن محاسبه شود که با یک شدن output valid از این امر باخبر میشویم و به محض رخ دادن این اتفاق مقدار خروجی به دست آمده را با مقدار موجود در فایل "output.txt" مقایسه میکنیم. در این لحظه اگر خروجی به دست آمده در ماژول با مقدار موجود در فایل یکسان بود، به معنی درستی خروجی و در غیر این صورت ماژول محاسبات را به درستی انجام نداده است. این عملیات را در یک حلقه for قرار میدهیم تا بتوانیم بار ها این کار را تکرار کنیم و به ازای ورودی های متفاوت، از درستی خروجی ماژول اطمینان پیدا کنیم. به علاوه در این تست بنچ خروجی های ماژول در یک فایل با نام "FIR_output.txt" نوشته میشود تا بتوانیم آن را به کد متلب "play_output.m" بدهیم و فایل صوتی بدون نویز را بشنویم. در ادامه تعدادی از شکل موج ها و تصاویر مربوط به این تست بنچ را مشاهده خواهید کرد.



شکل 9 : قرار دادن اولین ورودی با یک کردن input valid



شکل 10 : محاسبه اولین خروجی و فعال شدن سیگنال output valid

```
16      initial begin
17          $readmemb("inputs.txt", inputs);
18          $readmemb("outputs.txt", expected_values);
19      end
20
21      always begin
22          #10 clk <= ~clk;
23      end
24
25      initial begin
26          output_file = $fopen("FIR_output.txt");
27          #2 reset = 1'b1; // active high reset
28             input_valid = 1'b0;
29          #5 reset = 1'b0;
30          for(int i=0 ; i<100 ; i++)begin
31              #60 FIR_input = inputs[i];
32                  input_valid = 1'b1;
33              #40 input_valid = 1'b0;
34              @(posedge output_valid)begin
35                  $fwrite(output_file, "%b\n", FIR_output);
36                  if(FIR_output != expected_values[i])begin
37                      $display("Error in sample %d \n the output is : %b and the expected output is : %b", i+1, FIR_output, expected_value
38                  end
39                  else begin
40                      $display("pass sample %d \n the output is : %b and the expected output is : %b", i+1, FIR_output, expected_values[i
41                  end
42              end
43          end
44          $fclose(output_file);
45          #20 $stop;
46      end
47  endmodule
```

شکل 11: بخش اصلی کد تست بنچ

```
# pass sample          79
#  the output is : 00000000000001001111100101000100010000 and the expected output is : 00000000000001001111100101000100010000
# pass sample          80
#  the output is : 00000000000001001101100010101101101001 and the expected output is : 00000000000001001101100010101101101001
# pass sample          81
#  the output is : 00000000000001001010110110101000100011 and the expected output is : 00000000000001001010110110101000100011
# pass sample          82
#  the output is : 00000000000001000111011010101000110111 and the expected output is : 00000000000001000111011010101000110111
# pass sample          83
#  the output is : 00000000000001000010000011101000101011 and the expected output is : 00000000000001000010000011101000101011
# pass sample          84
#  the output is : 00000000000000111011000011111100101000 and the expected output is : 00000000000000111011000011111100101000
# pass sample          85
#  the output is : 00000000000000110100101110000010110010 and the expected output is : 00000000000000110100101110000010110010
# pass sample          86
#  the output is : 00000000000000110000101111001000010000 and the expected output is : 00000000000000110000101111001000010000
# pass sample          87
#  the output is : 00000000000000101110001000010011011000 and the expected output is : 00000000000000101110001000010011011000
# pass sample          88
#  the output is : 00000000000000101010101001110110011101 and the expected output is : 00000000000000101010101001110110011101
# pass sample          89
#  the output is : 00000000000000100101110111010110011001 and the expected output is : 00000000000000100101110111010110011001
# pass sample          90
#  the output is : 00000000000000100001100011111010100011 and the expected output is : 00000000000000100001100011111010100011
# pass sample          91
#  the output is : 00000000000000011111000100100100111111 and the expected output is : 00000000000000011111000100100100111111
# pass sample          92
#  the output is : 00000000000000011101100001010101100100 and the expected output is : 00000000000000011101100001010101100100
# pass sample          93
#  the output is : 00000000000000011011101011110001001011 and the expected output is : 00000000000000011011101011110001001011
# pass sample          94
#  the output is : 00000000000000011010100011111000000001 and the expected output is : 00000000000000011010100011111000000001
# pass sample          95
#  the output is : 00000000000000011100100001111000101100 and the expected output is : 00000000000000011100100001111000101100
# pass sample          96
#  the output is : 00000000000000100001101110111100110101 and the expected output is : 00000000000000100001101110111100110101
# pass sample          97
#  the output is : 00000000000000100110101011010101100100 and the expected output is : 00000000000000100110101011010101100100
# pass sample          98
#  the output is : 00000000000000100111100100000110001000 and the expected output is : 00000000000000100111100100000110001000
# pass sample          99
#  the output is : 00000000000000100100011100001000011100 and the expected output is : 00000000000000100100011100001000011100
# pass sample         100
#  the output is : 00000000000000100000111001111101000100 and the expected output is : 00000000000000100000111001111101000100
```

- **صحت‌سنجی به کمک assertion :**

در این بخش میخواهیم با استفاده از قابلیت هایی که assertion در اختیار ما قرار میدهد، مدار خود را تست کنیم. شایان ذکر است که ماژولی با عرض ورودی 16 بیت و طول 5 بیت تحت تست قرار گرفته است.

```
Loading : assert property (@(posedge clk) filter.load_input |=> filter.load_result)begin
            $display($stime,,,"\t\ttime  %m\nPASS: transition from loading to calculation state");
        end
        else begin
            $display($stime,,,"\t\ttime:  %m\nFAIL: transition from loading to calculation state");
        end

counter: assert property (@(posedge clk) filter.load_input |-> ##5 filter.counter_co) begin
            $display($stime,,,"\t\ttime %m\nPASS: Counter cout has been asserted after 5 clock cycle");
        end
        else begin
            $display($stime,,,"\t\ttime %m\nFAIL: Counter cout hasn't been asserted yet after 5 clock cycle");
        end

Clear_result: assert property (@(posedge clk) filter.output_valid_interconnect |=> (filter.ready_for_input and filter.clr_result))
        $display($stime,,,"\t\ttime  %m\nPASS: finish calculation -> waiting for new data");
    end
    else begin
        $display($stime,,,"\t\ttime  %m\nFail: finish calculation -> waiting for new data");
    end

output_valid_pipeline: assert property (@(posedge clk) filter.output_valid_interconnect |-> ##64 output_valid) begin
    $display($stime,,, "\t\ttime %m\nPipeline works correctly");
end
else begin
    $display($stime,,, "\t\ttime %m\nPipeline doesn't work correctly");
end
```

شکل 12 : تصویر چهار assertion که برای تست این ماژول نوشته شده است

```
//assertion for transition between loading and calculation states in controller
sequence high_co_in_calculation_state;
    (filter.control.present_state == filter.control.CALCULATION_STATE) and filter.counter_co;
endsequence

sequence low_co_in_calculation_state;
    (filter.control.present_state == filter.control.CALCULATION_STATE) and (filter.counter_co == 1'b0);
endsequence

sequence calculation_state_to_output;
    high_co_in_calculation_state ##1 filter.control.present_state == filter.control.OUTPUT_STATE;
endsequence

sequence calculation_state_to_itself;
    low_co_in_calculation_state ##1 filter.control.present_state == filter.control.CALCULATION_STATE;
endsequence

property loading_and_calculation_state;
    @(posedge clk) (filter.control.present_state == filter.control.LOADING_STATE) |=> calculation_state_to_output or calculation_sta
endproperty

loading_to_calculation_state: assert property(loading_and_calculation_state) begin
    $display($stime,,,"\t\ttime  %m\nPASS: transition between loading and calculating state");
end
else begin
    $display($stime,,,"\t\ttime  %m\nPASS: transition between loading and calculating state");
end
//end assertion
```

شکل 13 : تصویر پنجمین assertion که دارای چندین sequence است و برای تست controller نوشته شده

```
# PASS: transition from loading to calculation state
#         90    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#        150    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#        190    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#        230    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#        250    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#        310    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#        350    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#        390    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#        410    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#        470    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#        510    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#        550    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#        570    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#        630    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#        670    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#        710    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#        730    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#        790    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#        830    time  test_assertion.Clear_result
```

شکل 14 : نتیجه assertion بخش اول

```
# PASS: Counter cout has been asserted after 5 clock cycle
#        990    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#       1030    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#       1050    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#       1110    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#       1150    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#       1190    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#       1210    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#       1270    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#       1310    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#       1350    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#       1370    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#       1430    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#       1450    time test_assertion.output_valid_pipeline
# Pipeline works correctly
#       1470    time  test_assertion.Clear_result
# PASS: finish calculation -> waiting for new data
#       1510    time  test_assertion.Loading
# PASS: transition from loading to calculation state
#       1530    time  test_assertion.loading_to_calculation_state
# PASS: transition between loading and calculating state
#       1590    time test_assertion.counter
# PASS: Counter cout has been asserted after 5 clock cycle
#       1610    time test_assertion.output_valid_pipeline
# Pipeline works correctly
#       1630    time  test_assertion.Clear_result
```

شکل 15 : نتیجه assertion بخش دوم

- **سنتز:**
  - عرض 8 بیت و طول 50:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	233.21 MHz	233.21 MHz	clk	

شکل 16 : بیشترین فرکانس کاری برای عرض 8 بیت و طول 50

Flow Summary	
Flow Status	Successful - Fri Oct 20 22:55:32 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	FIR
Top-level Entity Name	FIR_filter
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,381 / 33,216 (4 %)
Total combinational functions	955 / 33,216 (3 %)
Dedicated logic registers	1,074 / 33,216 (3 %)
Total registers	1074
Total pins	35 / 475 (7 %)
Total virtual pins	0
Total memory bits	148 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 17 : Flow summary برای عرض 8 بیت و طول 50

Analysis & Synthesis Resource Usage Summary		
	Resource	Usage
1	Estimated Total logic elements	1,383
2		
3	Total combinational functions	954
4	◢ Logic element usage by number of LUT inputs	
1	-- 4 input functions	481
2	-- 3 input functions	32
3	-- <=2 input functions	441
5		
6	◢ Logic elements by mode	
1	-- normal mode	908
2	-- arithmetic mode	46
7		
8	◢ Total registers	1074
1	-- Dedicated logic registers	1074
2	-- I/O registers	0
9		
10	I/O pins	35
11	Total memory bits	148
12	Embedded Multiplier 9-bit elements	0
13	Maximum fan-out node	clk
14	Maximum fan-out	1079
15	Total fan-out	6658
16	Average fan-out	3.22

شکل 18 : Resource های استفاده شده برای عرض 8 بیت و طول 50

- عرض 16 بیت و طول 50 :

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	168.1 MHz	168.1 MHz	clk	

شکل 19 : بیشترین فرکانس کاری برای عرض 16 بیت و طول 50

Flow Summary

Flow Status	Successful - Fri Oct 20 23:11:08 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	FIR
Top-level Entity Name	FIR_filter
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,039 / 33,216 (12 %)
Total combinational functions	2,179 / 33,216 (7 %)
Dedicated logic registers	3,502 / 33,216 (11 %)
Total registers	3502
Total pins	59 / 475 (12 %)
Total virtual pins	0
Total memory bits	1,108 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 20 : Flow summary برای عرض 16 بیت و طول 50

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Estimated Total logic elements	4,041
2		
3	Total combinational functions	2178
4	◢ Logic element usage by number of LUT inputs	
1	-- 4 input functions	1272
2	-- 3 input functions	42
3	-- <=2 input functions	864
5		
6	◢ Logic elements by mode	
1	-- normal mode	2108
2	-- arithmetic mode	70
7		
8	◢ Total registers	3502
1	-- Dedicated logic registers	3502
2	-- I/O registers	0
9		
10	I/O pins	59
11	Total memory bits	1108
12	Embedded Multiplier 9-bit elements	0
13	Maximum fan-out node	clk
14	Maximum fan-out	3523
15	Total fan-out	18550
16	Average fan-out	3.22

شکل 21 : Resource های استفاده شده برای عرض 16 بیت و طول 50

- عرض 8 بیت و طول 100:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	185.15 MHz	185.15 MHz	clk	

شکل 22 : بیشترین فرکانس کاری برای عرض 8 بیت و طول 100

Flow Summary

Flow Status	Successful - Fri Oct 20 23:16:49 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	FIR
Top-level Entity Name	FIR_filter
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	2,024 / 33,216 (6 %)
Total combinational functions	1,600 / 33,216 (5 %)
Dedicated logic registers	1,468 / 33,216 (4 %)
Total registers	1468
Total pins	36 / 475 (8 %)
Total virtual pins	0
Total memory bits	948 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 23 : Flow summary برای عرض 8 بیت و طول 100

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Estimated Total logic elements	2,022
2		
3	Total combinational functions	1599
4	◢ Logic element usage by number of LUT inputs	
1	-- 4 input functions	717
2	-- 3 input functions	29
3	-- <=2 input functions	853
5		
6	◢ Logic elements by mode	
1	-- normal mode	1547
2	-- arithmetic mode	52
7		
8	◢ Total registers	1468
1	-- Dedicated logic registers	1468
2	-- I/O registers	0
9		
10	I/O pins	36
11	Total memory bits	948
12	Embedded Multiplier 9-bit elements	0
13	Maximum fan-out node	clk
14	Maximum fan-out	1481
15	Total fan-out	10068
16	Average fan-out	3.23

شکل 24 : Resource های استفاده شده برای عرض 8 بیت و طول 100

- عرض 16 بیت و طول 100:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	171.35 MHz	171.35 MHz	clk	

شکل 25 : بیشترین فرکانس کاری برای عرض 16 بیت و طول 100

Flow Summary

Flow Status	Successful - Fri Oct 20 23:24:49 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	FIR
Top-level Entity Name	FIR_filter
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	5,392 / 33,216 (16 %)
Total combinational functions	3,524 / 33,216 (11 %)
Dedicated logic registers	4,304 / 33,216 (13 %)
Total registers	4304
Total pins	60 / 475 (13 %)
Total virtual pins	0
Total memory bits	1,908 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 26 : Flow summary برای عرض 16 بیت و طول 100

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Estimated Total logic elements	5,386
2		
3	Total combinational functions	3523
4	◢ Logic element usage by number of LUT inputs	
1	-- 4 input functions	1815
2	-- 3 input functions	43
3	-- <=2 input functions	1665
5		
6	◢ Logic elements by mode	
1	-- normal mode	3451
2	-- arithmetic mode	72
7		
8	◢ Total registers	4304
1	-- Dedicated logic registers	4304
2	-- I/O registers	0
9		
10	I/O pins	60
11	Total memory bits	1908
12	Embedded Multiplier 9-bit elements	0
13	Maximum fan-out node	clk
14	Maximum fan-out	4325
15	Total fan-out	25555
16	Average fan-out	3.23

شکل 28 : Resource های استفاده شده برای عرض 8 بیت و طول 100

جدول 1 : مقایسه تعداد resource ها و فرکانس بیشینه کاری برای 4 حالت مختلف سنتز

WIDTH	LENGTH	MAX frequency	Total registers	Total Logic elements
8	**50**	233 MHZ	1074	1383
16	**50**	168 MHZ	3502	4041
8	**100**	185 MHZ	1468	2022
16	**100**	171 MHZ	4304	5386

در این بخش میخواهیم نتایج حاصل از سنتز کردن فیلتر FIR را بررسی کنیم. همانطور که انتظار داشتیم در حالتی عرض ورودی 8 بیت و طول 50 است هم کمترین تعداد Logic element و رجیستر استفاده میشود و هم بیشترین فرکانس کاری ممکن را داریم. در این حالت اگر عرض ورودی را به 16 بیت برسانیم شاهد افزایش قابل توجه تعداد resource های مورد استفاده و همچنین کاهش چشم گیر فرکانس کاری بیشینه مدار هستیم. حال اگر نتایج به دست آمده از سنتز با عرض ورودی 8 بیت و طول 100 را مشاهده کنیم متوجه میشویم که نسبت به حالت ابتدایی logic element و رجیسترهای بیشتری مصرف کرده است و فرکانس کاری مدار بسیار پایین آمده است اما این تغییرات نسبت به تغییرات حاصل از افزایش عرض ورودی بسیار کمتر است. به بیان دیگر، با افزایش عرض بیت ورودی شاهد رشد چشمگیرتری در تعداد resource های مورد استفاده هستیم نسبت به حالتی که طول فیلتر را زیاد میکنیم.