Bruno E. Gracia Villalobos

Programmable, Multi-Order, Pipelined FIR Filter ASIC

EE 4513

December 3, 2019

Introduction

This report describes the design of an Application Specific Integrated Circuit (ASIC) for filtering digital signals with a Finite-Impulse-Response (FIR) filter. Two Read-Only-Memories (ROMs) are used to store the 'taps' or filter coefficients as well as the input signal as integers. The design is programmable in the sense that the order of the filter as well as the ROM contents for both taps and input signal can be loaded accordingly.

The design is inspired from a Transposed, Direct Form I FIR filter architecture present in Digital Design literature (1). The common architecture is pipelined to meet timing closure, and therefore the ASIC is designed as a Controller and Datapath. The units run on two clock speeds with a 1:3 ratio respectively to compensate for the clock cycles needed to register values and read from the ROMs.

For the ASIC development presented in this report, the input signal as well as the coefficients are of width 16 bits. The output is of width 32 bits. The ROMs are of size 16 bit x 16 row for a total storage capacity of 32 bytes. The design is built in 180, 45, and 90nm technology nodes and compared for power, area, and speed.

Procedure

Datapath Design

Before beginning to design the ASIC at RTL, programmability must be defined. In architectural terms, this means the pipeline must grow and decrease as per defined by the user. In turn, the number of multipliers, adders, and registers also changes; the two ROMs used to store the filter and signal are fixed. In usability terms, although the filter order is static, the ROM can be programmed. Since the design is targeted for an ASIC, the "pseudo" programmability is left to other RTL designers and not the actual user; the design can be implemented on an FPGA with different filter orders.

With programmability defined, the first step to develop the ASIC is to design the Datapath—this is the tunnel where the circuit processes the input signal, and outputs the filtered signal. The design referenced from (1) implements the basic discrete convolution equation used to define a FIR filter, where *h[k]* is the filter, *x[n-k]* is the signal, and *M* is the order of the filter:

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k]$$

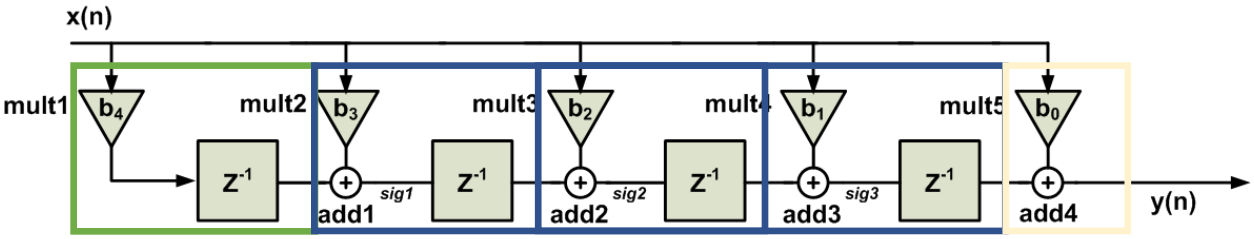

Figure 1: Transposed, Direct Form 1 FIR Filter Architecture (2). (GREEN, BLUE, YELLOW outlines show uniqueness)

For an example, consider the FIR filter architecture in Figure 1. The filter order *M* defines the number of delays *Z^-1,* and *M+1* is the number of taps. The signal is processed in parallel in each block as outlined, and the blocks output the result after a delay by the *Z^-1*. By this time, the same block receives the previous block's output and the cycle repeats. Here is the basic flow of the pipeline to be built, and now the architecture is transcribed into RTL.

The distinct elements present in this architecture are registers (*Z^-1* delays), adders, and multipliers. For this design, a ROM is developed and simulated with variable width and size. The register module is designed with D flip-flops (DFFs) and simulated for operation. From previous assignments, the adder in Carry-Lookahead (CLA) form and multiplier in array form are referenced. A *parameter* in the HDL is used to define the width of the all four elements.

```
317 module ROM #(parameter BITS = 16, ROWS = 16, FILENAME="filter_memory.mem") (
318 input [$clog2(ROWS)-1:0] addr,
319 input rd,
320 input clk,
321
322 output reg [BITS-1:0] out
323 );
324
325 reg [BITS-1:0] rom [0:ROWS-1];
326
327 initial begin
328     $readmemh(FILENAME, rom);
329 end
330
331 always@(posedge clk) begin
332     if(rd) out = rom[addr];
333     else out = {BITS{1'bz}};
334 end
335
336 endmodule
```

The ROM module is of interest in providing programmability. A *parameter* is used to define the file path to a *.mem* file, which Vivado then loads into the specified register ROM with the *$readmemh* system function. The values are stored in hex.

Figure 2: Showcase of programmability in ROM module

Now, with the basic logic elements ready to be used, the programmability factor of the ASIC is assessed once again. As shown in Figure 1, the architecture can be divided into three logical elements that can be developed and then instantiated as needed to build the required filter order. The green outline is the block that is fed data from each ROM. The blue outline communicates with the first and last block, and as far as the inputs are concerned, each block receives the same type of inputs and outputs. The yellow outline is the last element in the chain of the Datapath and outputs the result of the processing to the outside world. While every filter design will have one green block and one yellow block, the blue outline defines the programmability of the filter at compile of the HDL as shown in the table below.

FILTER ORDER	GREEN BLOCK(S)	BLUE BLOCK(S)	YELLOW BLOCK(S)
2 (MINIMUM)	1	1	1
3	1	2	1
4	1	3	1
5	1	4	1
N	1	N-1	1

Table 3: Pattern for filter architecture

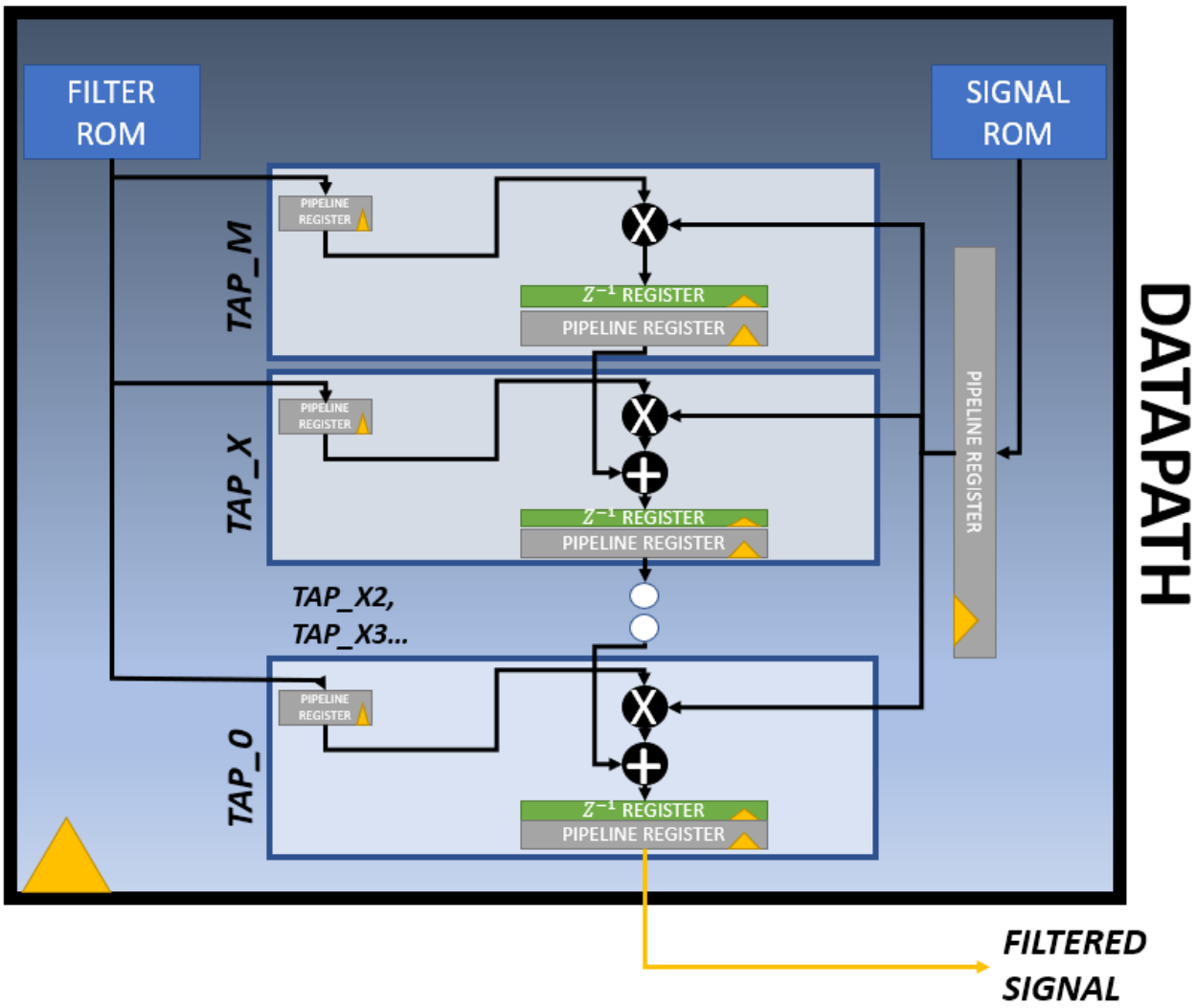

Figure 4: Datapath of ASIC at RTL

The full design of the Datapath at RTL is now ready to be discussed as shown in Figure 3. With the previous discussion, the intuition for developing the multi-order design is to write HDL for three distinct modules for the three distinct taps in the Datapath and instantiate the tap (*TAP_X*) in between the first (*TAP_M*) and last (*TAP_0*) as required. To accomplish this in Verilog, *generate* statements are used to instantiate *N-1* TAP_X modules according to the filter order, and Figure 4 shows a snippet of how the code works. Essentially, the for loop iterates over the number of taps of the filter and instantiates the necessary blocks.

```
586 genvar i;
587 generate
588 for(i=ORDER; i>=0; i=i-1) begin
589
590     //IN BETWEEN
591     if((i!=0) & (i!=ORDER)) begin
592         TAP_X #(BITS) TAP_X_INST(
593             .tap(hn[i]),
594             .x(x),
595             .prev_stage(tap_data[i+1]),
596
597             .clr(clrs[i]),
598             .enable(enables[i]),
599             .clk(clk),
600
601             .out(tap_data[i]),
602
603             //DIAGNOSTICS
604             .tap_reg_out(tap_reg_out[i]),
605             .clk2(clk2[i])
606         );
```

Figure 5: Building the TAP_X's in a *generate* statement

Apart from the TAP modules, the ROMs and pipeline register outside the taps as the remaining logic elements in the Datapath are straightforward. The ROMs are fed control signals from the Controller to select which filter tap and signal value to read, and the pipeline register ensures timing closure of the design. All three elements are wired in without a *generate* statement because they are needed in the same quantity regardless of the order of the filter programmed.

However, since the taps of the filter are not read serially and must be in place before the input signal can be filtered, the controller establishes a routine to clock in the data for each tap's respective pipeline register to hold the tap necessary to process the incoming signal values. This explains why there is a pipeline register for each TAP feeding from the filter ROM; these have a clock enable so that the controller can prevent the TAP from being overwritten by its neighbors. This procedure is explained in more detail in the ASM chart of the *Controller Design* section.

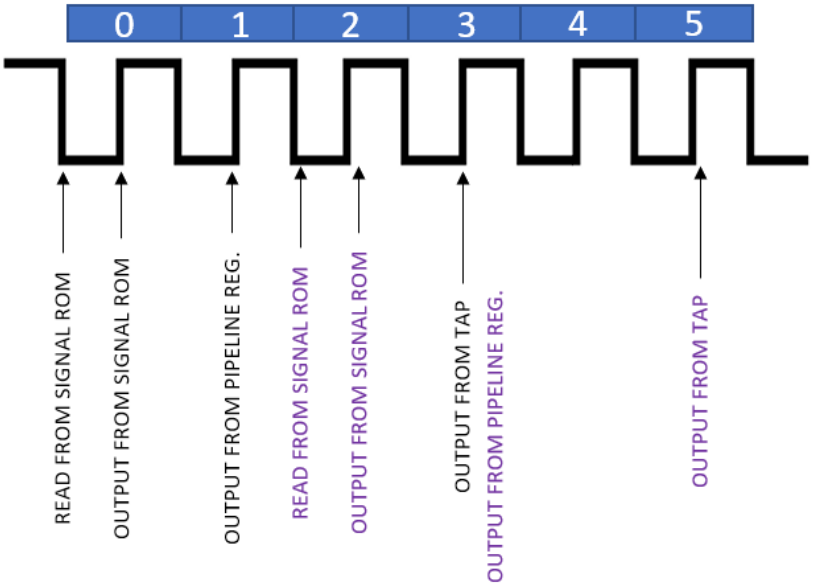

Figure 6: Pipelining of Datapath

Now, let's consider the pipelining of the design in Figure 5. Beginning with the control signal sent to the ROM to output signal data, it takes four clock cycles to get the output from TAP_M. The black text symbolizes the processing of the first output of the ROM, and the purple symbolizes the processing of the second output of the ROM. Since it takes two clock cycles for data to come into each TAP, there is an extra pipeline register after the Z-1 delay in each TAP to account for the added overhead of reading from the ROM.

The signal paths between the multiplier and adder in the TAP_X and TAP_0 modules do not have a pipeline register because they are combinational by design. As a side note, the multiplier is implemented with half-adders and full-adders, which is true for the

CLA as well, and therefore both can be modeled by a large chain of adders that don't need pipelining—given each gate within the adder has the same timing constraint.

Controller Design

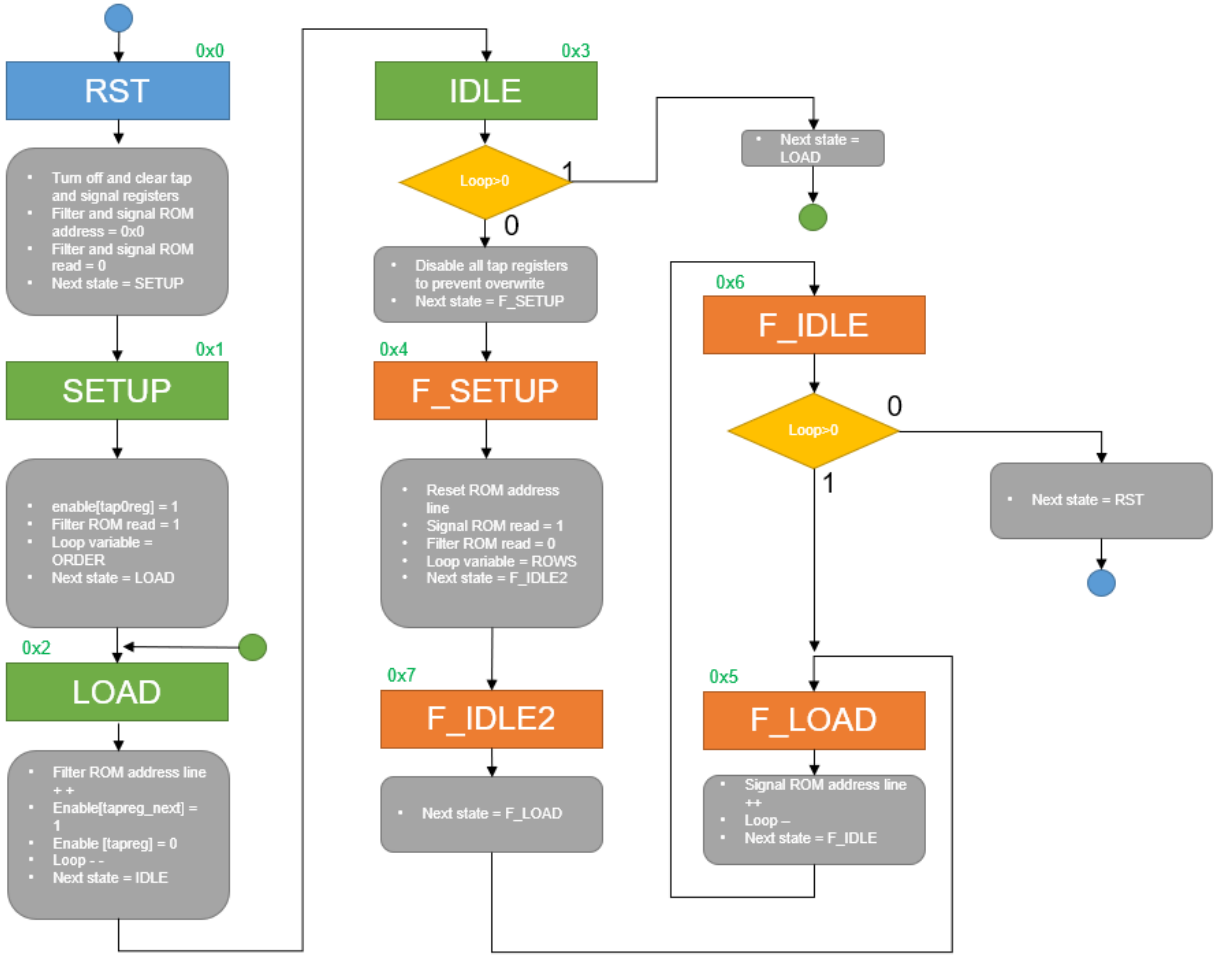

Figure 7: Controller modeled as an ASMD chart

An Algorithmic State Machine-Datapath (ASMD) chart for the Controller of the Datapath is presented in Figure 6 to realize the full-design. Although a Finite State Machine (FSM) would suffice for specifying the architecture of the Controller, the dichotomy of the ASIC's design in Controller and Datapath components leaves the ASMD as the right choice for keeping in mind the Controller's function.

A total of 8 states are used to define the operation of the ASIC. As mentioned briefly in the *Datapath Design* section, the overall operation of the ASIC is as follows: the filter taps are read one by one from the filter ROM and stored in their respective TAP's register. A total of *(M+1)**2 clock cycles are spent on preparing the Datapath for receiving signal input from the signal ROM.

The Controller starts on the reset stage to prepare all tap registers, signal and filter ROMs, as well as the pipeline register outside the TAPs. Once ready, to accomplish the

two main operations of the ASIC, the states above are color-coded to represent the filter setup stage with GREEN states, and the signal feed or filtering stage with ORANGE states. Both stages are composed of a *setup* stage, *load* stage, and *idle* stage, and these are presented next.

In the filter setup stage, the TAP which will be assigned the output of the filter ROM must have its register enabled before the data is received. In the SETUP state, the Controller prepares the filter ROM for read operation and initializes the loop variable for which to iterate on. Once the filter ROM has the output ready, the Controller sends a signal to the register to enable it, and after two clock cycles, it is disabled and lieu of the next register to write to as explained in the pipeline discussion of the *Datapath Design* section. The Controller cycles between LOAD and IDLE states to model this behavior and is ongoing until all TAP registers are populated.

Once the filter setup stage—GREEN—is done, the next stage shown in the ORANGE, color-coded states is next to filter the signal. This stage carries out the same behavior as the filter setup stage: Controller sends address to read from signal ROM, pipeline register stores it, and waits two clock cycles for each TAP to output an answer to repeat again. However, compared to the GREEN stage, the filtering stage must wait an extra clock cycle because after completing the filter setup stage, the ASM is clocked by a faster signal than originally. The reason for this design choice is presented next in consideration of Figure 7.

Full ASIC Design

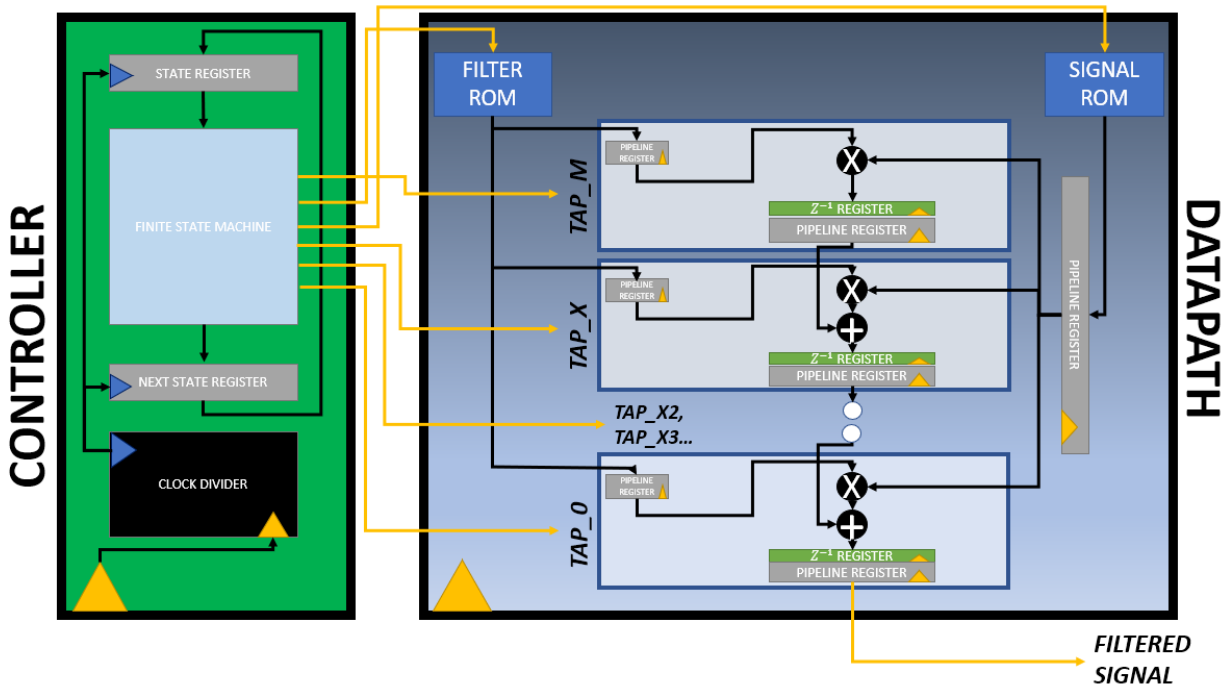

Figure 8: FIR Filter full ASIC design

As shown in Figure 7, the Controller unit contains a Clock Divider that runs the Controller at a lower clock speed than the Datapath to account for the extra clock cycles needed to read from the ROM. Given the filter setup stage writes to registers which are not feeding further Datapath components initially, the Controller can run at a divided clock and therefore send the signals at its own time; the Registers will patiently wait. However, once the filter setup stage is completed, and the filtering stage feeds the signal data from the signal ROM, the Controller must drive the pipeline exactly as designed. Otherwise, the TAPs below the first TAP to receive the signal data will receive erroneous information from the pipeline registers, including *x* (unknown) values, and the integrity of the output at TAP_0—the final TAP—will be compromised.

To rectify the design flaw, the Controller is clocked at the Datapath speed at the F_SETUP state, and to further align the control signals with the pipeline specified in the *Datapath Design* section, the F_IDLE2 state correctly “restarts” the signal ROM read control signal to be sent when the clock is low to prepare to output data at the rising edge. Figure 7 shows the repertoire of control signals sent by the Controller to the Datapath to fulfill the desired operation of the ASIC.

Elaborated Design (Pre-Synthesis)

With the architecture completed, the elaborated design is now presented—the preliminary design interpreted by Vivado’s Verilog engine before synthesis. The elaborated design schematic of the Controller is presented first.

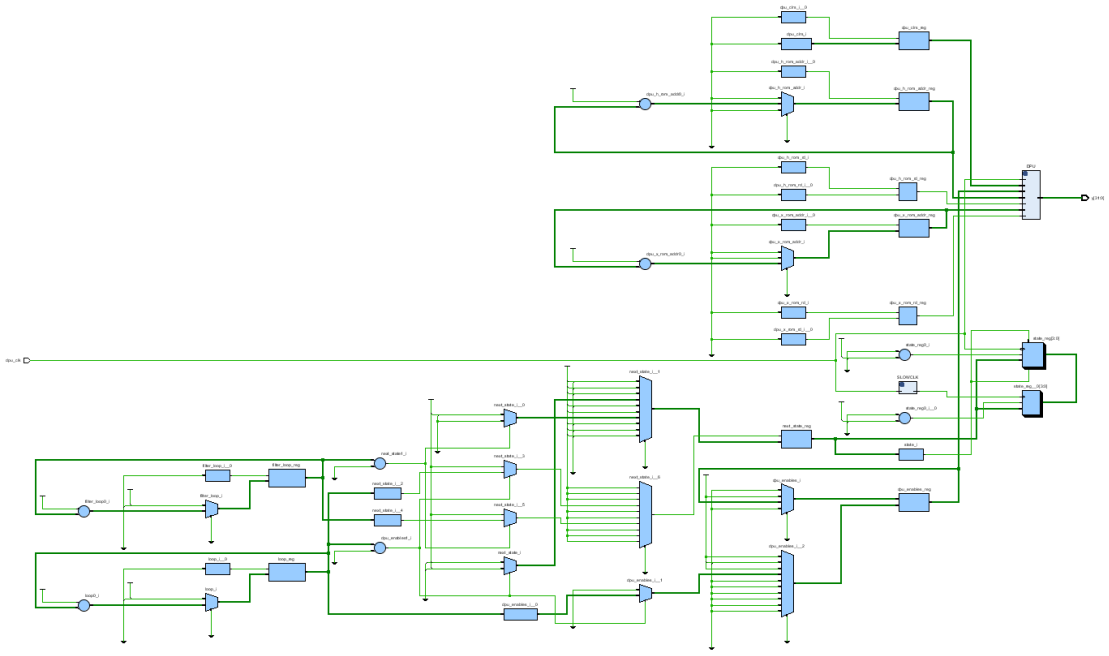

Figure 9: Controller elaborated design

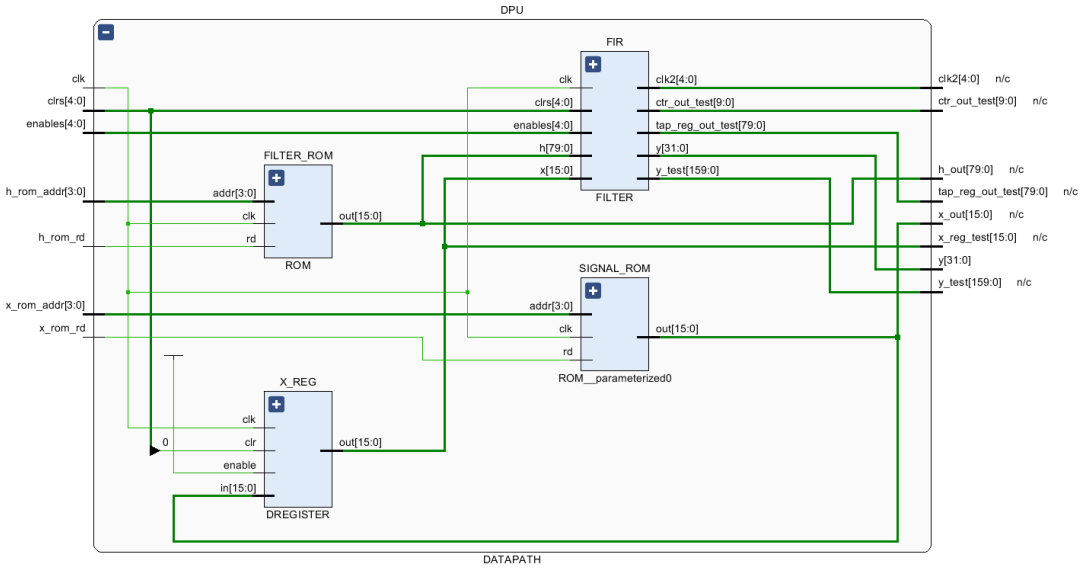

Figure 10: Datapath elaborated design

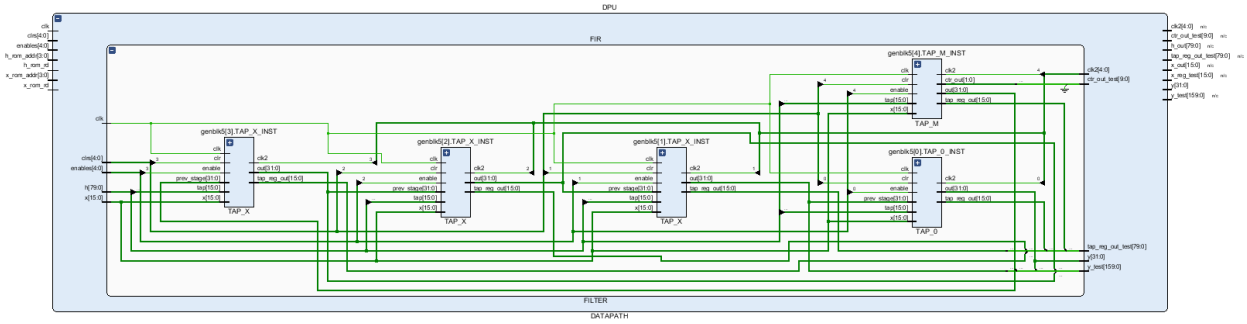

Figure 11: Filter elaborated design for M=4 implementation

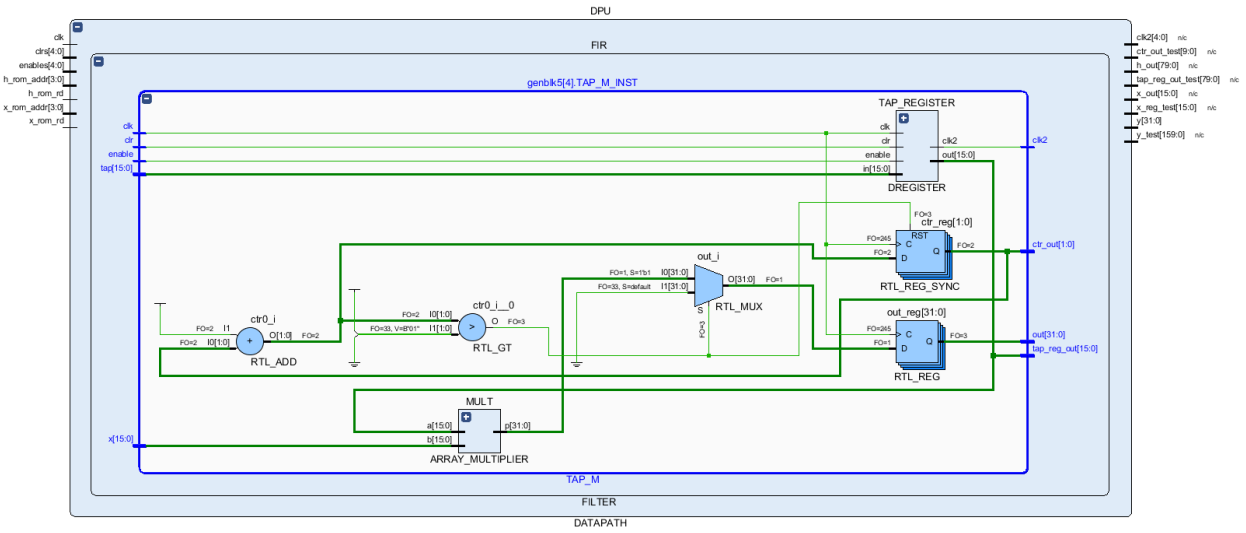

Figure 12: TAP_M elaborated design

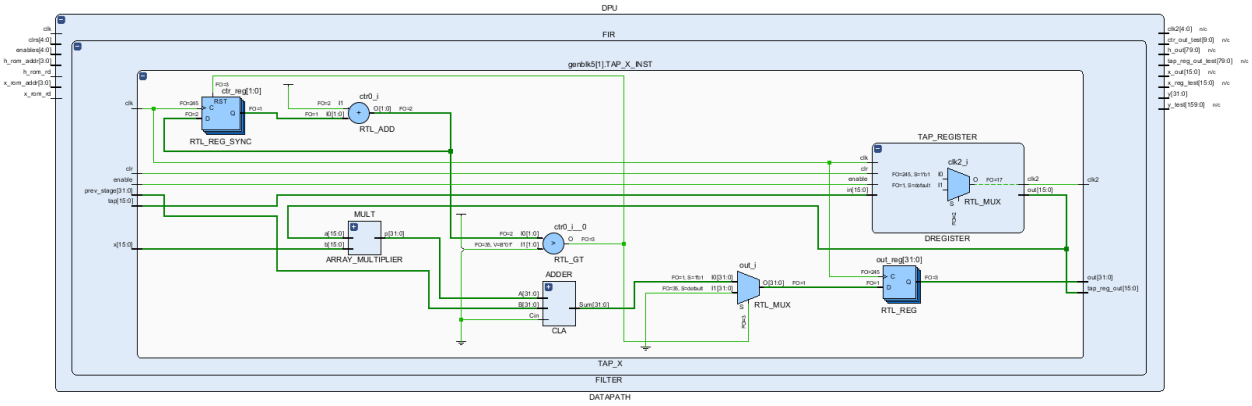

Figure 13: TAP_X elaborated design

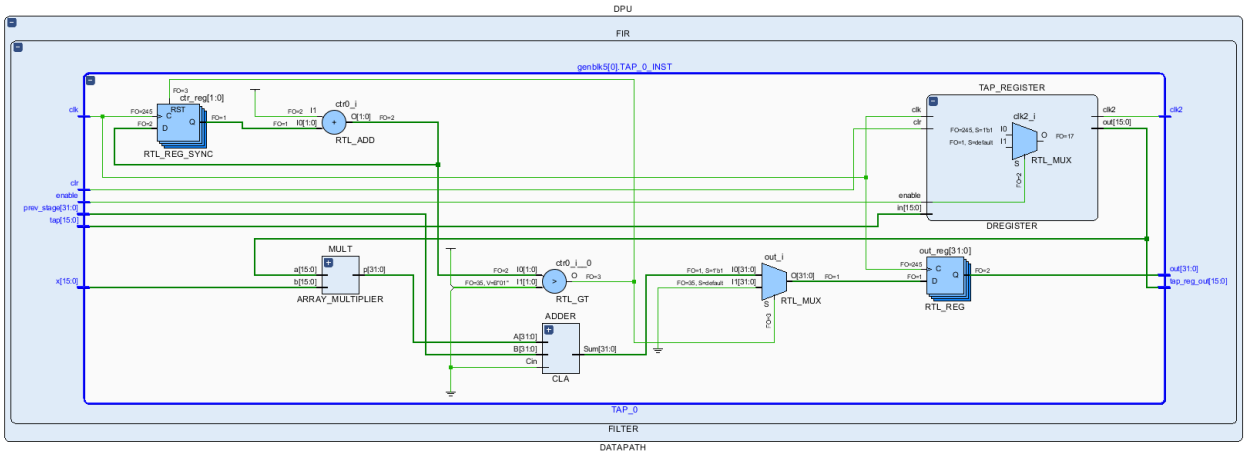

Figure 14: TAP_0 elaborated design

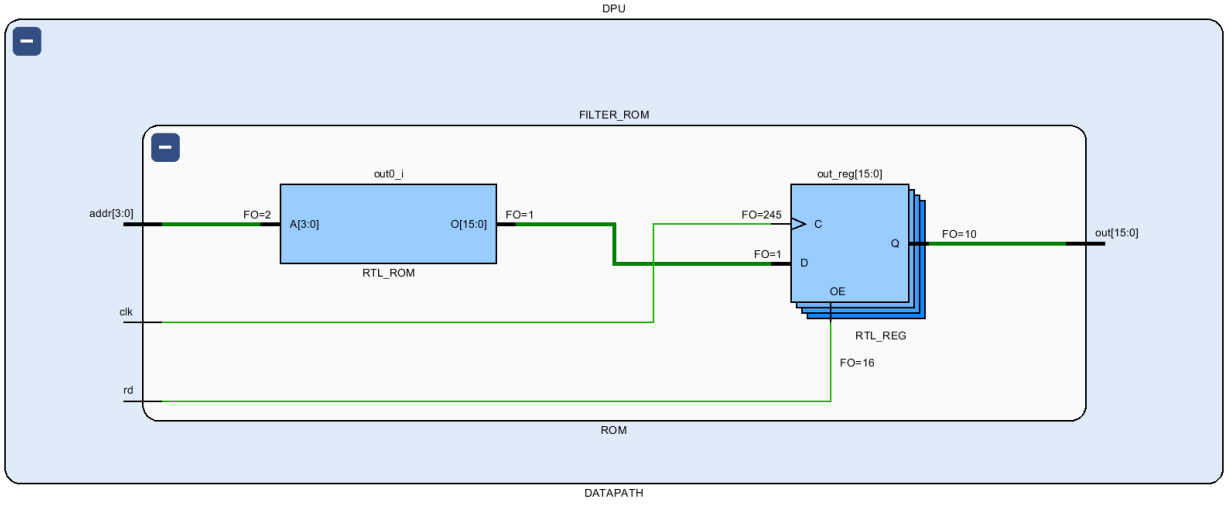

Figure 15: ROM elaborated design

Simulation results are presented in the following section.

Observation and Results

FIR Filter Architecture Simulations

```
h =

    1    1    1

>> x

x =

    1    0    1    2    1

>> y = conv(x,h)

y =

    1    1    2    3    4    3    1

>>
```

The fundamental filter architecture is tested first without external ROM components to verify its functionality before continuing to develop the ASIC. To develop test data, a MATLAB script is used to verify the discrete convolution of arbitrary FIR filters as impulse response h[n] and arbitrary input signals as x[n]. Consider the 2nd order FIR filter example shown in Figure 15. First, the TAP registers must be populated as described in the *Controller Design* and *Datapath Design* sections. Figure 16 showcases the successful load of filter coefficients into the TAP registers by the h[23:0] bus. In this example, an 8-bit word length for both filter and signal points is used, and 24 bits represent all TAP registers.

Figure 17: 2nd order FIR filter

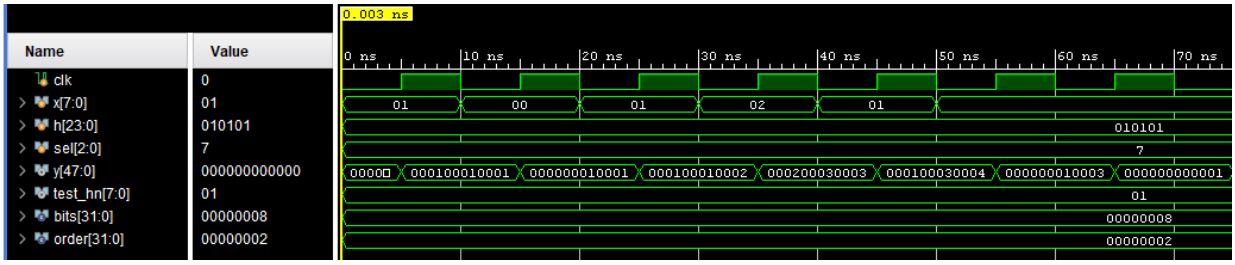

Figure 16: Loading TAP registers (GREEN stage in ASMD)

Now the filtered signal y is tested against the MATLAB script output. Given the input signal is 5 points wide and the filter is 3 points wide, the filtering result is (5+3)-1=7 points wide. This factor as well as the magnitude of the signal points are verified in Figure 16 by the y[15:0] bus's serial output coming from TAP_0, the last tap in the FIR filter pipeline. It is worth mentioning that in this test, TAP_0's output is not pipelined, and therefore each output is valid after 1 clock cycle (repeated values take 2 clock cycles).

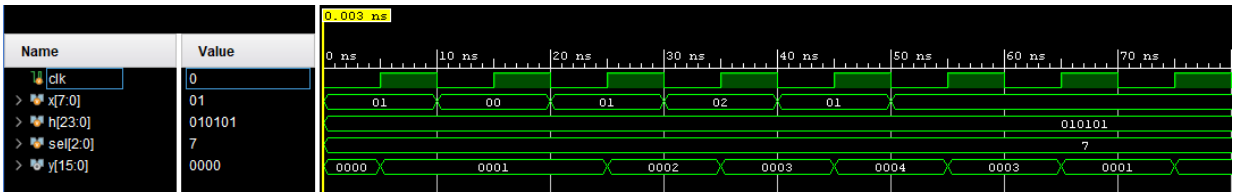

Figure 18: Verified operation of 2nd order FIR filter test

```
h =

    4    2    7    1    9

x =

    1    0    1    2    1

y =

    4    2   11   11   24   17   18   19    9

>>
```

Consider a 4th order FIR filter test next. The input signal is constant for easier verification. In this example, the y output is (5+5)-1 = 9 values in length. With a chosen 8 bits to represent the output, the verified filtered signal is shown in Figure 19 as the y[15:0] bus. The output signal is twice as large as the input signals because of the multiplier operation. Further, the output is valid after each clock cycle as well in this example because the filter has not been pipelined yet.

Figure 19: 4th order FIR Filter test

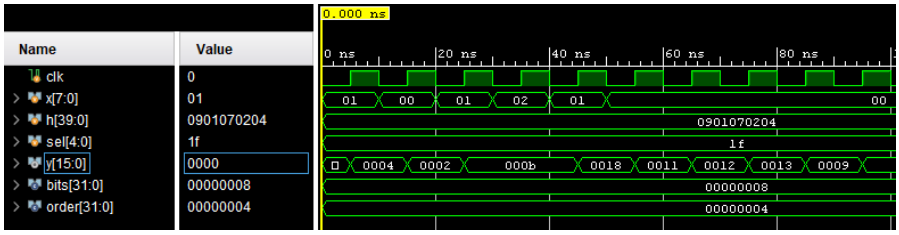

Figure 20: Verified 4th order FIR filter test

```
h =

    4    2    7    1    9    8    7    2    3    4    5

x =

    1    0    1    2    1

y =

Columns 1 through 13

    4    2   11   11   24   25   25   29   35   28   19   12   16

Columns 14 through 15

   14    5
```

Lastly, a 10th order filter is tested as shown in the MATLAB script on Figure 20. However, in this case, a pipeline register is added to the output of TAP_0, and the results are valid after 2 clock cycles each. The results are presented in Figure 21.

Figure 21: 10th order FIR filter test

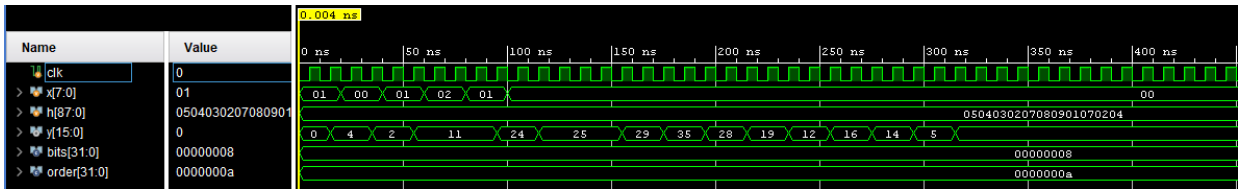

Figure 22: Verified 10th order FIR filter test

Datapath Architecture Simulations

The ASIC is gradually being built, and the next step is to test the full FIR filter pipeline with both filter and signal ROMs. For the first test, the same 3rd order filter is used as the previous section, and Figure 22 displays the serial, verified result in the y[15:0] bus. Both filter setup and filtering stages are shown, and the reader can take note of the h_rom_rd and x_rom_rd signals to specify the end and start of each stage—the h_rom (filter ROM) signal is low after setting up the TAP registers, the x_rom (signal ROM) is high at the beginning of filtering stage. The ORANGE lines in the simulation mark the pipeline stages as described in the *Datapath Design* section: signal data fetch, ROM output, pipeline register saved, output at TAP_0. The lines shown as *x* valued are test registers not in use for the current testbench.

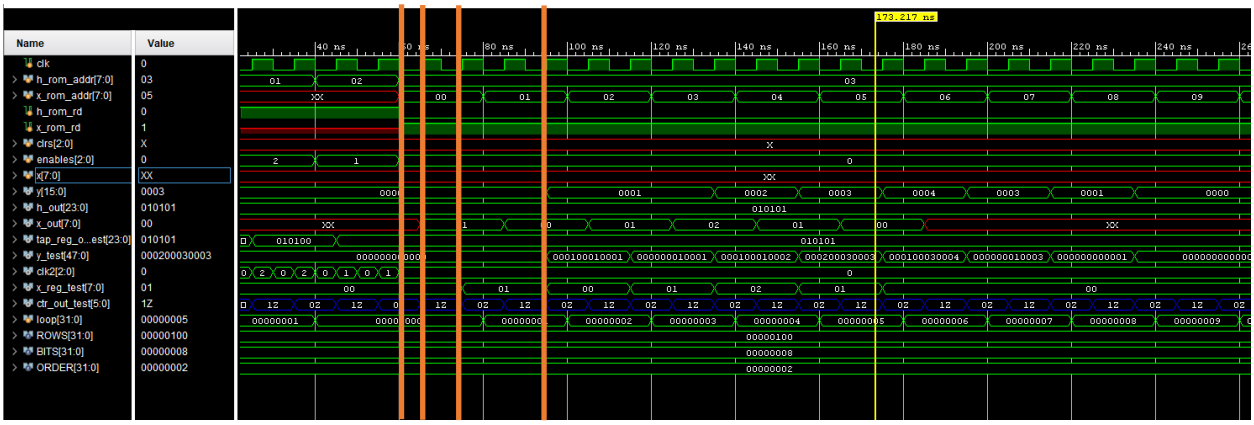

Figure 23: Verified Datapath with a 3rd order filter with pipeline stages marked

Another test is conducted with a 4th order architecture with the exact specifications as used with the *FIR Filter tests* section. The verified results are shown in hex in Figure 23.

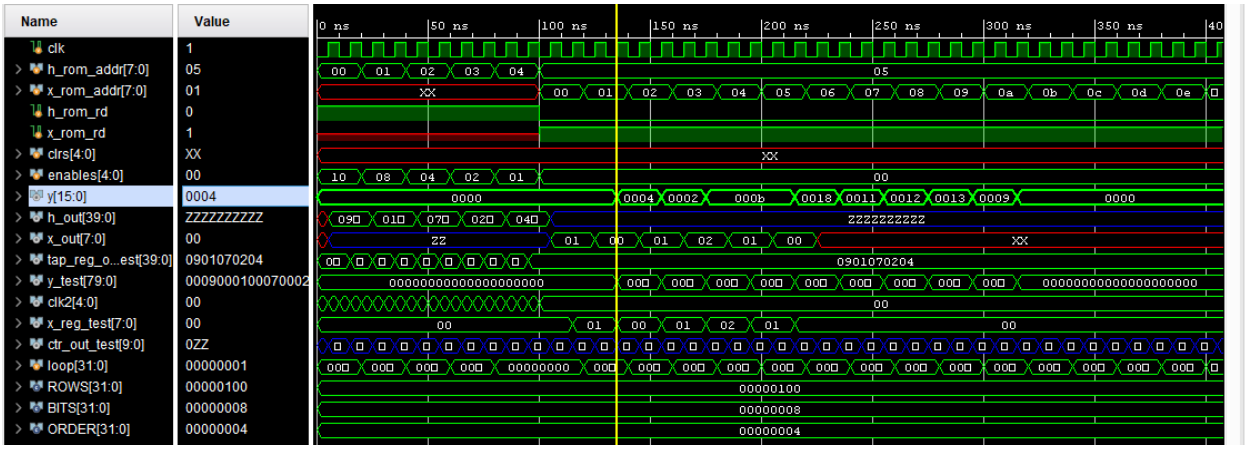

Figure 24: Verified Datapath with 4th order filter

Controller Simulations

The HDL for the controller is written after the full Datapath is verified for desired operation. Similar to the sequential tests conducted on the FIR Filter and Datapath architectures, the Controller is tested with the same methodology: test its main two operations—filter setup and signal filtering—one by one. Testing the filter setup stage is presented first—the GREEN stage as depicted in the ASMD of the Controller in *Controller Design* subsection. The same TAP values as the 4th order test for the *Datapath Architecture* are used. The output of the TAP registers is concatenated in a bus for easier troubleshooting and is shown in the dpu_tap_regs_test[39:0] bus by the 800ns clock mark. This proves a successful routine to write to each TAP register by enabling it whilst disabling the other ones. The states of the FSM are also presented to show how the Controller iterates between LOAD and IDLE stages, although in this example they might have been mis numbered.

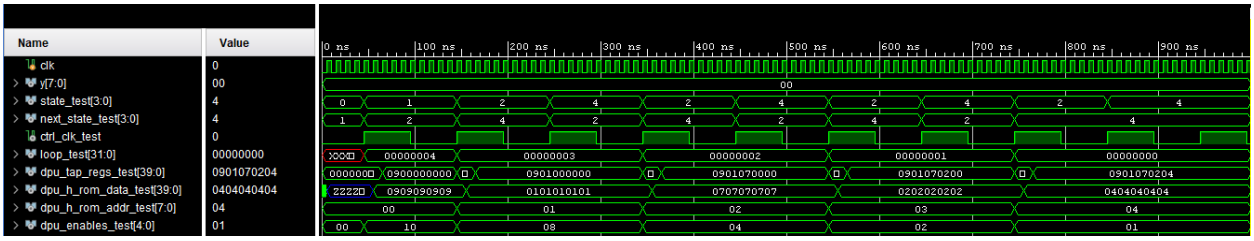

Figure 25: Verified Filter stage setup simulation by Controller

Now, the full FSM is tested, and the Controller cycles through the filtering stage after the filter setup stage (partially hidden from Figure 25). The following test values are used, like the rest of the 4th order FIR filter tests as generated in MATLAB.

$$h = [4\ 2\ 7\ 1\ 9]$$

$$x = [1\ 0\ 1\ 2\ 1]$$

$$y = [4\ 2\ 11\ 11\ 24\ 17\ 18\ 19\ 9]$$

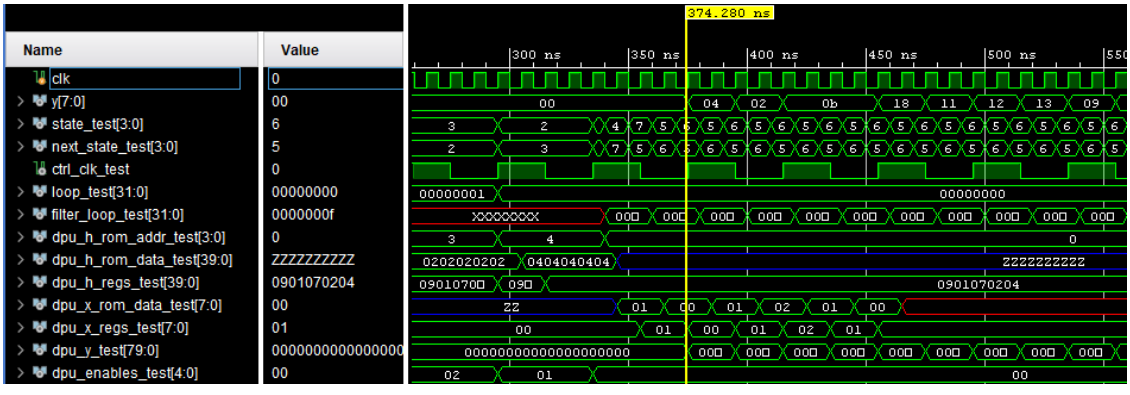

Figure 26: Verified 4th order filter test in full ASIC architecture

The Controller successfully produces the required output in hexadecimal.

To test the full ASIC, more input samples are added to the ROM to test. Consider the following input test parameters and output signal.

$$h = [4\ 2\ 7\ 1\ 9]$$

$$x = [1\ 0\ 1\ 2\ 1\ 7\ 9]$$

$$y = [4\ 2\ 11\ 11\ 24\ 45\ 68\ 86\ 79\ 72\ 81]$$

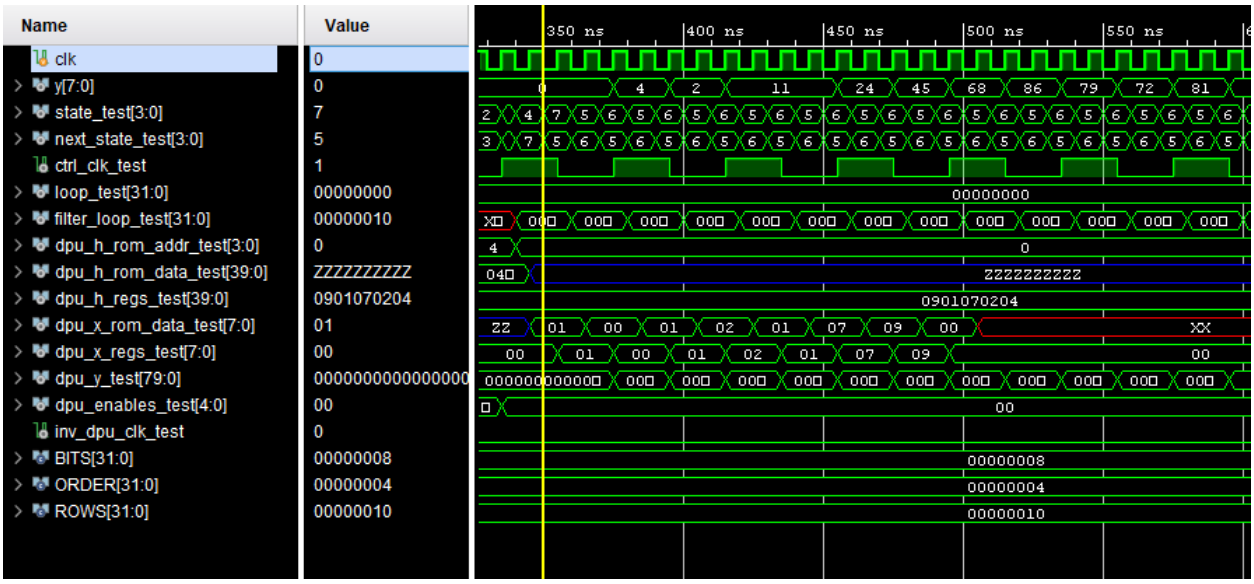

Figure 27: Full ASIC test with a 4th order FIR Filter architecture

The full ASIC design is verified in Figure 27. The Controller successfully operates the Datapath to output the filtered signal as verified by the MATLAB script. The next step is to design the ASIC using Cadence Encounter, and it's implementation in 180nm, 90nm, and 45nm technology nodes are compared.

ASIC Design

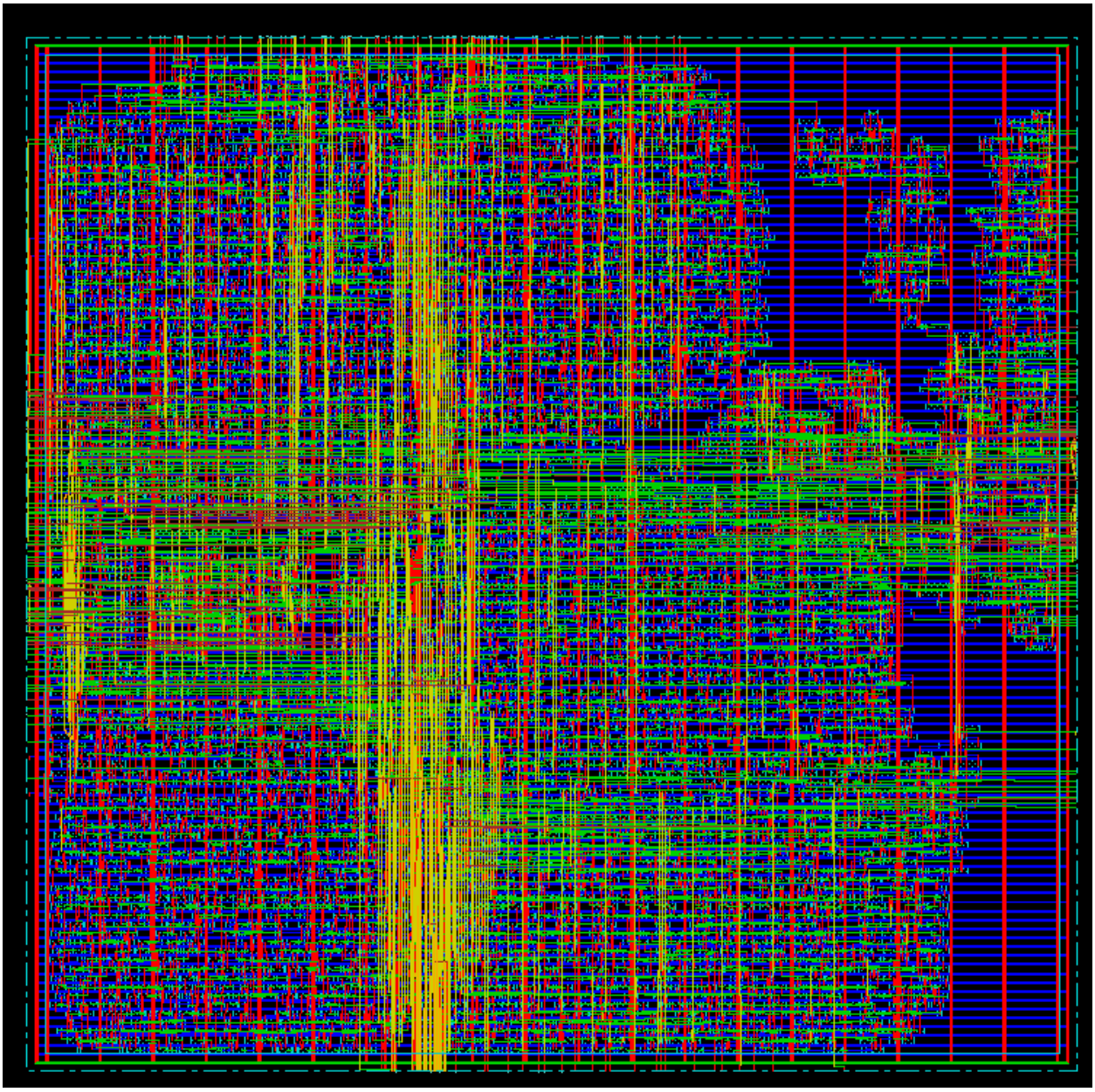

Figure 28: ASIC Design in 180nm using Cadence Encounter

Once the Verilog design is fully simulated for desired operation, the next step is to use Cadence Encounter to realize the ASIC. A 180nm technology node is used to build the design, and Figure 28 showcases the resultant chip layout auto routed by the software. Its comparison in power, area, and delay is presented after 90 and 45nm technology node implementations are presented next.

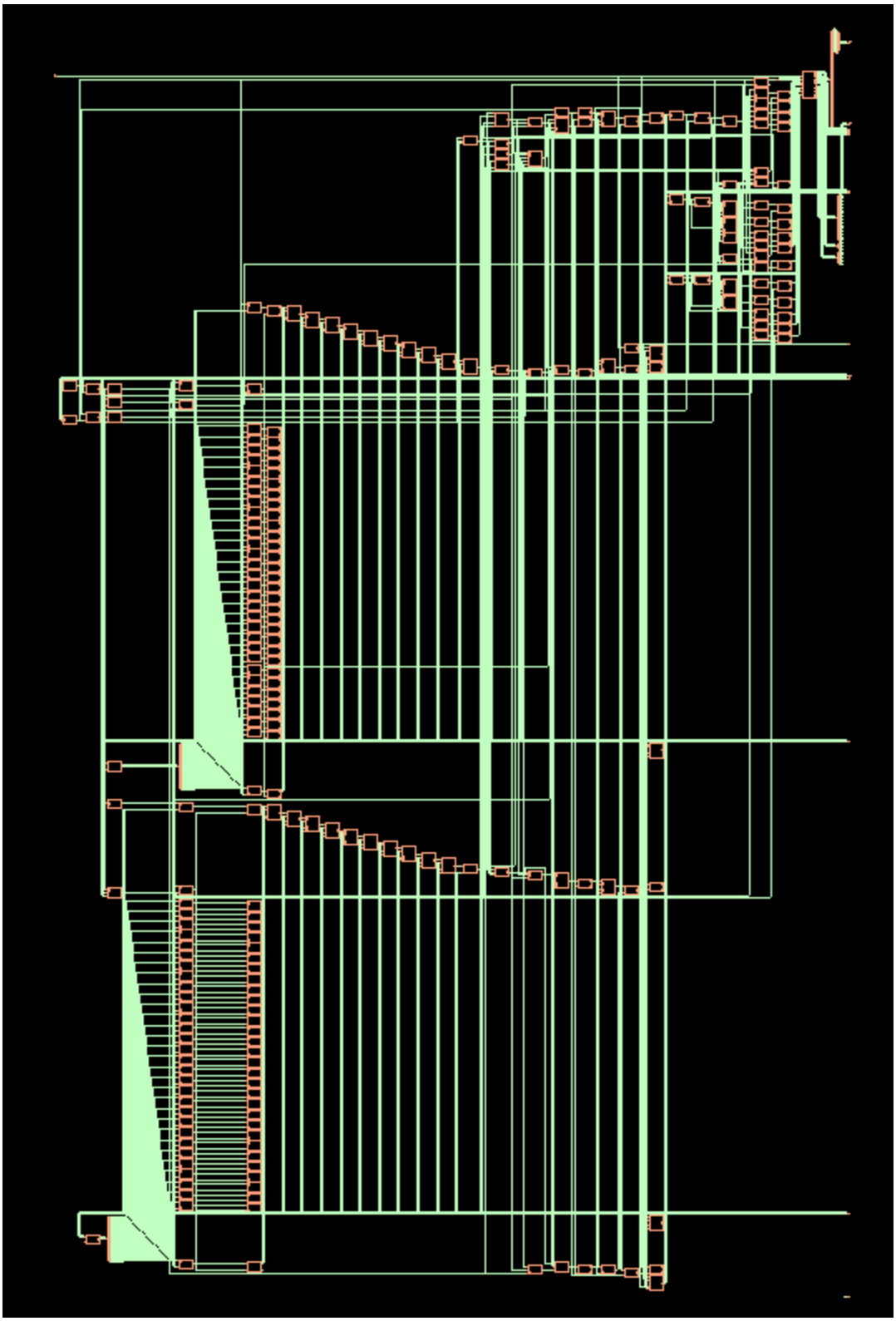

Figure 29: ASIC schematic in 180nm with Cadence Encounter

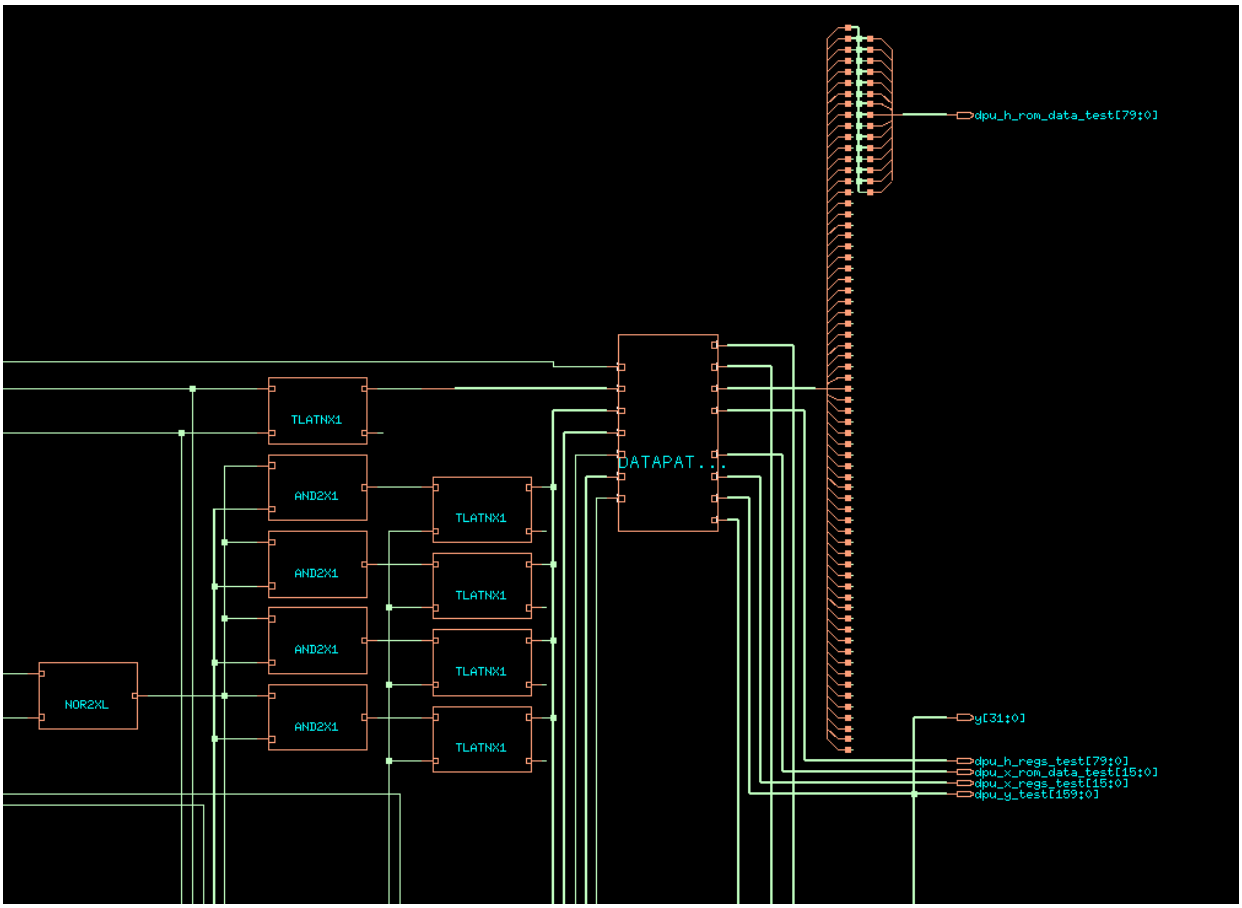

Figure 30: Centered on Datapath placement in ASIC 180nm with Cadence Encounter

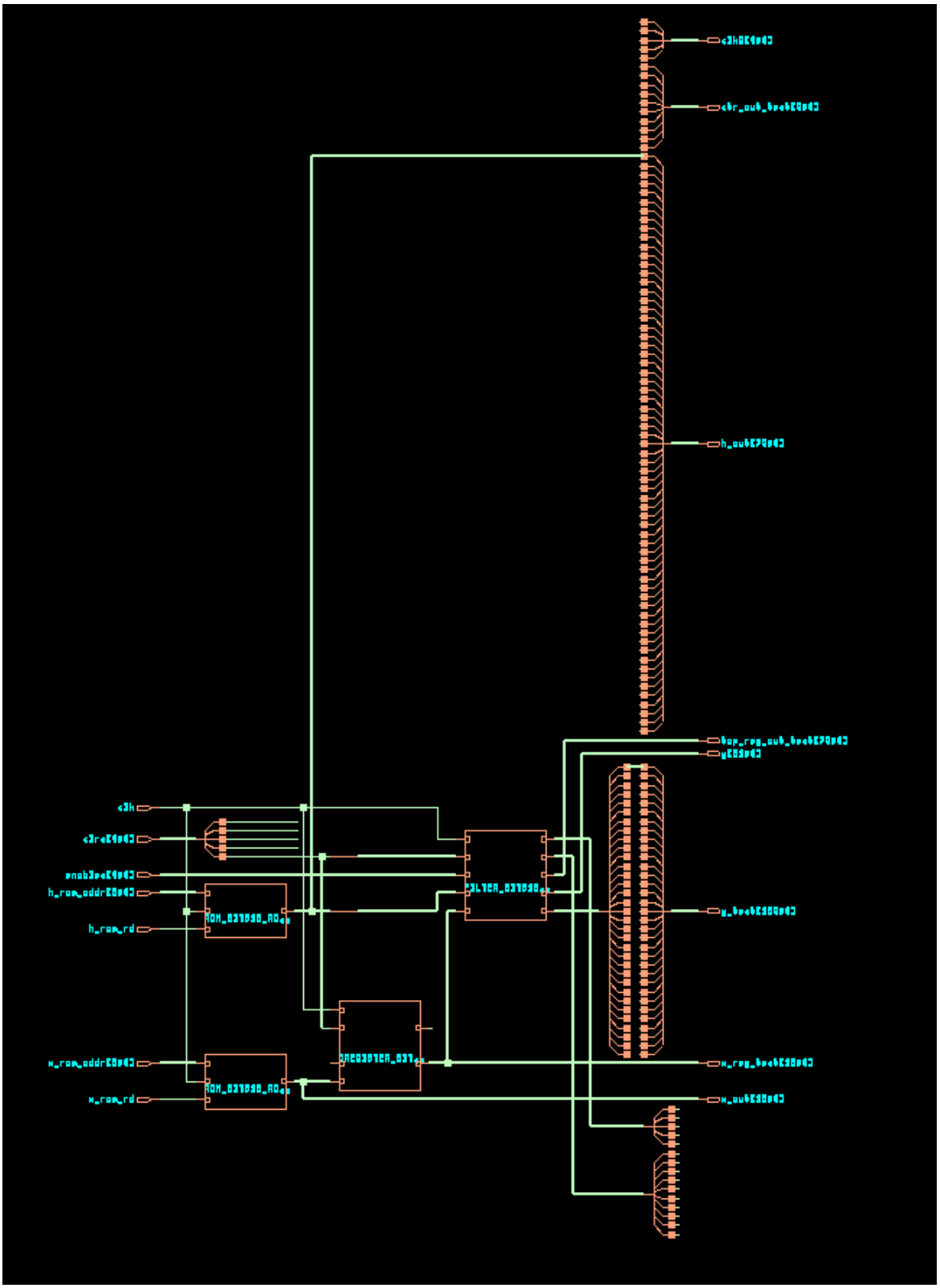

Figure 31: ASIC Datapath in 180nm with Cadence Encounter

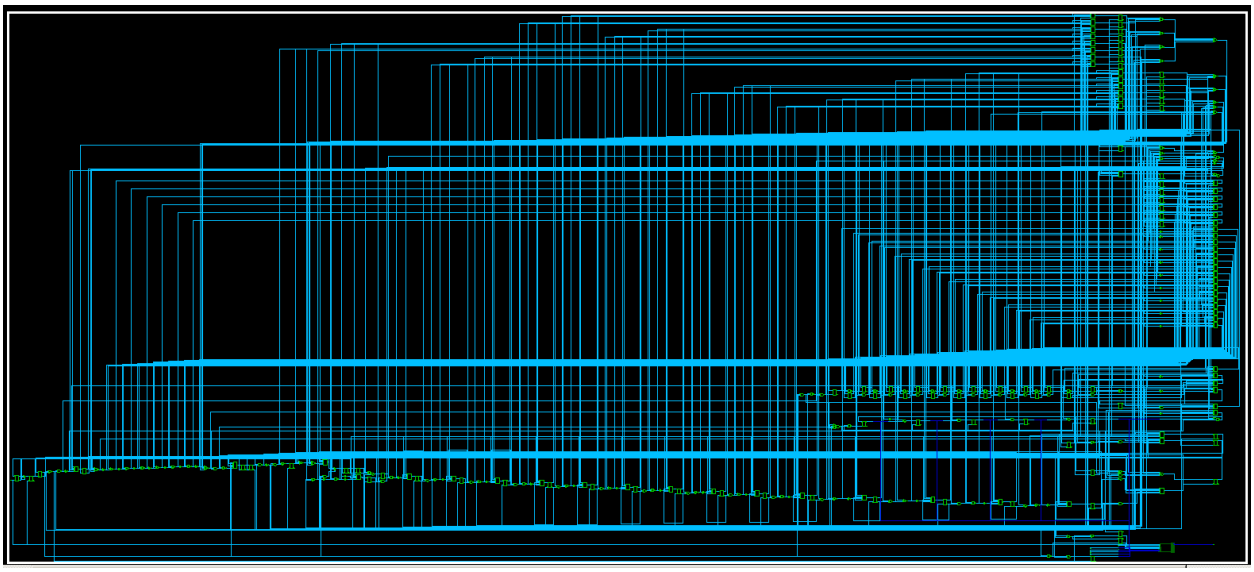

Figure 32: ASIC Schematic in 90nm with Synopsys

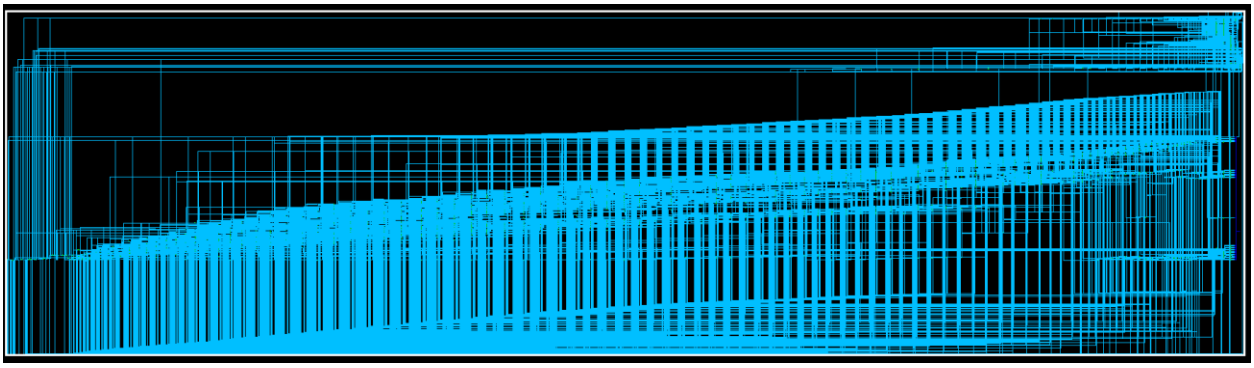

Figure 33: ASIC Schematic in 45nm with Synopsys

Comparing Power, Delay, and Area between Technology Nodes

The characteristics of each Technology Node are compared. The full reports of each node are attached in the appendix for further analysis, and a simplified table is presented below.

	45NM	**90NM**
TIMING	0.46f	0.61 f
TOTAL AREA	10948.894428	64264.217708
TOTAL POWER	3.8015e-02 mW	181.5616 uW

Table 4: Pattern for filter architecture

Conclusion

The design of this ASIC to implement a programmable, multi-order, and pipelined FIR filter is a great capstone to my Intro to VLSI class. It ties into my concentration of Digital Signal Processing (DSP) and Embedded systems, and I learned how to design an integrated circuit at RTL.

I did not understand pipelining too well until endeavoring on this project, and running into errors allowed me to have the intuition for the need of certain pipeline registers at key areas of the Datapath design. Further, I learned better system design skills, especially the preliminary ones by going back to the drawing board when I ran into major issues, such as erroneous output from the TAP_0 register.

As simple as a FIR filter seemed, reading from two separate ROMs added a considerable degree of complexity to this project. I would have liked to further implement full programmability, where the pipeline is static at say, 50 orders, and the user can select the order filter. That is, the output will always be valid although some orders are not in use—this would call for a need of parallel output rather than serial, because the actual filter order is unknown until ROM power up, which is way past the window for programmability of hardware such as in an FPGA.

Above all, the Synopsys and Cadence Encounter tools allowed me to solidify the concepts learned in class, and also brought into my awareness the importance of careful, and detailed design choices to further reduce power, increase speed, and decrease area. I will certainly continue to explore the VLSI technologies in the Digital Signal Processing field after the completion of this project—it is the bridge between both of my concentrations.

References

[1] Ciletti, Michael. “Advanced Digital Design with the Verilog HDL”

[2] https://www.allaboutcircuits.com/technical-articles/pipelined-direct-form-fir-versus-the-transposed-structure/

Appendices

Verilog Code

CNTRL_SIM.V

```
`timescale 1ns / 1ps
//Engineer: Bruno E. Gracia Villalobos
//16oh4.com
//DECEMBER 4, 2019
//MIT LICENSE

module CNTRL_SIM();

localparam BITS = 16;
localparam ORDER = 4;
localparam ROWS = 16;

//CREATE 100 MHz CLOCK
reg clk;
initial clk = 1'b0;
always #5 clk = ~clk;

//CONTROLLER WIRES
wire [BITS*2-1:0] y;

/*
wire [3:0] state_test;
wire [3:0] next_state_test;
wire ctrl_clk_test;
wire [31:0] loop_test;
wire [31:0] filter_loop_test;

//DPU WIRES
wire [$clog2(ROWS)-1:0] dpu_h_rom_addr_test;
wire [BITS*(ORDER+1)-1:0] dpu_h_rom_data_test;
wire [BITS*(ORDER+1)-1:0] dpu_h_regs_test;

wire [BITS-1:0] dpu_x_rom_data_test;
wire [BITS-1:0] dpu_x_regs_test;

wire [BITS*2*(ORDER+1)-1:0] dpu_y_test;

wire [ORDER:0] dpu_enables_test;
wire inv_dpu_clk_test;
*/
CNTRL UUT(
.dpu_clk(clk),

.y(y)
```

```
/*
//DIAGNOSTICS
.state_test             (state_test),
.next_state_test        (next_state_test),
.ctrl_clk_test          (ctrl_clk_test),
.loop_test              (loop_test),
.filter_loop_test       (filter_loop_test),

.dpu_h_rom_addr_test    (dpu_h_rom_addr_test),
.dpu_h_rom_data_test    (dpu_h_rom_data_test),
.dpu_h_regs_test        (dpu_h_regs_test),

.dpu_x_rom_data_test    (dpu_x_rom_data_test),
.dpu_x_regs_test        (dpu_x_regs_test),

.dpu_y_test             (dpu_y_test),

.dpu_enables_test       (dpu_enables_test),
.inv_dpu_clk_test       (inv_dpu_clk_test)
*/
);

defparam UUT.BITS = BITS;
defparam UUT.ORDER = ORDER;
defparam UUT.ROWS = ROWS;

endmodule
```

CNTRL.V

```
`timescale 1ns / 1ps
//Engineer: Bruno E. Gracia Villalobos
//16oh4.com
//DECEMBER 4, 2019
//MIT LICENSE

module CNTRL #(parameter BITS = 16, ORDER = 4, ROWS = 16)(
input dpu_clk,

output [BITS*2-1:0] y,

//CONTROLLER DIAGNOSTICS
output [3:0] state_test,
output [3:0] next_state_test,
output ctrl_clk_test,
output [31:0] loop_test,
output [31:0] filter_loop_test,

//DPU DIAGNOSTICS
output [$clog2(ROWS)-1:0] dpu_h_rom_addr_test,
output [BITS*(ORDER+1)-1:0] dpu_h_rom_data_test,
output [BITS*(ORDER+1)-1:0] dpu_h_regs_test,

output [BITS-1:0] dpu_x_rom_data_test,
```

```
output [BITS-1:0] dpu_x_regs_test,

output [ORDER:0] dpu_enables_test,
output [BITS*2*(ORDER+1)-1:0] dpu_y_test,
output inv_dpu_clk_test
);

localparam ROM_ADDR_SIZE = $clog2(ROWS);
localparam CLK_DIVIDE = 2;

//SETUP CLOCK FOR STATE MACHINE
wire ctrl_clk;
assign ctrl_clk_test = ctrl_clk;

SLOW_CLK #(CLK_DIVIDE) SLOWCLK(
.clk(dpu_clk),
.slow_clk(ctrl_clk)
);

//SETUP DATAPATH
reg [ROM_ADDR_SIZE-1:0] dpu_h_rom_addr;
assign dpu_h_rom_addr_test = dpu_h_rom_addr;

reg dpu_h_rom_rd;

reg [ROM_ADDR_SIZE-1:0] dpu_x_rom_addr;
reg dpu_x_rom_rd;

reg [ORDER:0] dpu_clrs;
reg [ORDER:0] dpu_enables;
assign dpu_enables_test = dpu_enables;

wire [BITS-1:0] dpu_y;

//INVERTER CLOCK WHENEVER X DATA COMES IN
reg inv_dpu_clk;
initial inv_dpu_clk = 1'b0;
assign inv_dpu_clk_test = inv_dpu_clk;

DATAPATH DPU(
.clk(dpu_clk),

.h_rom_addr         (dpu_h_rom_addr),
.h_rom_rd           (dpu_h_rom_rd),

.x_rom_addr         (dpu_x_rom_addr),
.x_rom_rd           (dpu_x_rom_rd),

.clrs               (dpu_clrs),
.enables            (dpu_enables),

.y(y),

//DIAGNOSTICS
.tap_reg_out_test   (dpu_h_regs_test),
```

```
.h_out              (dpu_h_rom_data_test),

.x_reg_test         (dpu_x_regs_test),
.x_out              (dpu_x_rom_data_test),

.y_test(dpu_y_test)
);

defparam DPU.BITS = BITS;
defparam DPU.ORDER = ORDER;
defparam DPU.ROWS = ROWS;

//SETUP STATE MACHINE
reg [3:0] state;
reg [3:0] next_state;
assign state_test = state;
assign next_state_test = next_state;

integer loop;
integer filter_loop;

assign loop_test = loop;
assign filter_loop_test = filter_loop;

localparam RST      = 'h0;

//TAP REGISTERING STATES
localparam SETUP    = 'h1;
localparam LOAD     = 'h2;
localparam IDLE     = 'h3;

//FILTERING STATES
localparam F_SETUP  = 'h4;
localparam F_LOAD   = 'h5;
localparam F_IDLE   = 'h6;
localparam F_IDLE2   = 'h7;

localparam LINGO = 'h8;


//INITIALIZE ALL REGISTERS TO 0
initial begin
dpu_h_rom_addr = {ROM_ADDR_SIZE{1'b0}};
dpu_x_rom_addr = {ROM_ADDR_SIZE{1'b0}};

dpu_h_rom_rd = 1'b0;
dpu_x_rom_rd = 1'b0;

dpu_clrs = {(ORDER+1){1'b0}};
dpu_enables = {(ORDER+1){1'b0}};

state = 'h0;
next_state = 'h0;
end
```

```
always@(posedge ctrl_clk) begin
    if (state < F_SETUP) state = next_state;
end

/*
always@(posedge dpu_clk) begin
    if(state >= F_SETUP ) state = next_state;
end
*/

always@(negedge dpu_clk) begin
    if((next_state == F_SETUP)) state = next_state;
    else if(state >= F_SETUP) state = next_state;
end

always@(state) begin
    case(state)
    RST: begin //STATE 0

        //RESET AND DISABLE ALL REGISTERS
        dpu_clrs = {(ORDER+1){1'b1}};
        dpu_enables = {(ORDER+1){1'b0}};

        //START ALL ROM ADDRESSES TO 0
        dpu_h_rom_addr = {ROM_ADDR_SIZE{1'b0}};
        dpu_x_rom_addr = {ROM_ADDR_SIZE{1'b0}};

        //TURN OFF ROMs
        dpu_x_rom_rd = 1'b0;
        dpu_h_rom_rd = 1'b0;

        next_state = SETUP;
    end
    SETUP: begin // STATE 1

        //DISABLE CLEARS
        dpu_clrs = {(ORDER+1){1'b0}};

        //ENABLE THE TAP REGISTER TO WRITE ROM DATA TO
        //dpu_enables[ORDER] = 1'b1;
        dpu_enables[0] = 1'b1;

        //PREPARE TO READ IN TAPS
        dpu_h_rom_rd = 1'b1;

        //INITIALIZE LOOP VARIABLE
        loop = ORDER;

        next_state = LOAD;
    end
    LOAD: begin //STATE 2

        //INCREMENT ROM ADDRESS TO READ NEXT TAP
```

```
    dpu_h_rom_addr = dpu_h_rom_addr + 1'b1;

    //SHIFT REGISTER ENABLE BIT FOR NEXT TAP TO WRITE FROM ROM
    //dpu_enables = {dpu_enables[0], dpu_enables[ORDER:1]};
    dpu_enables = {dpu_enables[ORDER-1:0], dpu_enables[ORDER]};

    //DECREMENT LOOP
    loop = loop - 1'b1;

    //CONTINUE TO LOOP IN THIS STATE
    next_state = IDLE;

end
IDLE: begin //STATE 3
    if(loop > 0) begin
        next_state = LOAD;
    end
    //WHEN ALL TAPS HAVE BEEN LOADED
    else if(loop == 0) begin

        //DISABLE ALL REGISTERS TO PREVENT OVERWRITE OF TAPS
        dpu_enables = {(ORDER+1){1'b0}};

        next_state = F_SETUP;


    end
end

F_SETUP: begin //STATE 4

    //START ALL ROM ADDRESSES TO 0
    dpu_h_rom_addr = {ROM_ADDR_SIZE{1'b0}};
    dpu_x_rom_addr = {ROM_ADDR_SIZE{1'b0}};

    //START READING FROM SIGNAL ROM
    dpu_x_rom_rd = 1'b1;

    //TAPS ARE REGISTERED SO STOP READING FILTER ROM
    dpu_h_rom_rd = 1'b0;

    //START LOOP VARIABLE
    filter_loop = ROWS;

    //next_state = F_LOAD

    next_state = F_IDLE2;

end

F_LOAD: begin //STATE 5

    //INCREMENT ADDRESS TO READ
    dpu_x_rom_addr = dpu_x_rom_addr + 1'b1;
```

```
        //GO TO IDLE AND COME BACK TO RETRIGGER ALWAYS BLOCK
        next_state = F_IDLE;

        //INCREMENT LOOP COUNTER
        filter_loop = filter_loop - 1'b1;

        next_state = F_IDLE;

    end

    F_IDLE: begin //STATE 6

        if(filter_loop > 0) begin
            next_state = F_LOAD;
        end
        else if(filter_loop == 0) begin
            next_state = RST;
        end

    end

    F_IDLE2: begin //STATE 7
        next_state = F_LOAD;

    end

    LINGO: begin
        next_state = LINGO;
    end

    default: next_state = LINGO;
    endcase
end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//////

module SLOW_CLK #(parameter DIVIDE = 5) (
input clk,
output slow_clk
);

//10ns period = 100MHz = 1E8 clock speed for datapath
//50ns period = 20MHz = 2E7clock speed for controller
reg [31:0] counter;
reg slow_clk_reg;

//ASSUME PARAM IS 32 BITS

initial counter = {32{1'b0}};
initial slow_clk_reg = 1'b0;

assign slow_clk = slow_clk_reg;
```

```
always@(posedge clk) begin
    counter = counter + 1'b1;

    if(counter == DIVIDE) begin
        slow_clk_reg = ~slow_clk_reg;
        counter = {32{1'b0}};
    end
end

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

module ROM #(parameter BITS = 16, ROWS = 16, FILENAME="filter_memory.mem") (
input [$clog2(ROWS)-1:0] addr,
input rd,
input clk,

output reg [BITS-1:0] out
);

reg [BITS-1:0] rom [0:ROWS-1];

initial begin
    $readmemh(FILENAME, rom);
end

always@(posedge clk) begin
    if(rd) out = rom[addr];
    else out = {BITS{1'bz}};
end

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

module DREGISTER #(parameter BITS = 16) (
input [BITS-1:0] in,
input clr,
input clk,
input enable,

output [BITS-1:0] out,

//DIAGNOSTICS
output clk2
);

//Clear all registers if clr flag is set
wire [BITS-1:0] clr_wire;
assign clr_wire = clr ? 1 : 0;
```

```
//"latch" enable
reg en;
initial en = 1'b1;

always@(*) begin
    en <= enable;
end

//Clock enable
assign clk2 = en ? clk : 0;

genvar i;
generate
    for(i = 0; i < BITS; i = i+1) begin
        DFF DFF_INST(
        .d(in[i]),
        .clr(clr),
        .clk(clk2),

        .q(out[i])
        );
    end
endgenerate


endmodule

////////////////////////////////////////////////////////////////////////////////
//////

module DFF(
input d,
input clr,
input clk,

output reg q
);

initial q = 1'b0;

always@(posedge clk, posedge clr) begin
    if(clr) q <= 1'b0;
    else begin
        if((d === 1'bx) | (d === 1'bz)) q <= q;
        else q <= d;
    end

end
endmodule

////////////////////////////////////////////////////////////////////////////////
//////

module DATAPATH #(parameter BITS = 16, ORDER = 4, ROWS = 16) (
input clk,
```

```
input [$clog2(ROWS)-1:0] h_rom_addr,
input h_rom_rd,

input [$clog2(ROWS)-1:0] x_rom_addr,
input x_rom_rd,

input [ORDER:0] clrs,
input [ORDER:0] enables,

output [BITS*2-1:0] y,

//DIAGNOSTICS
output [BITS*(ORDER+1)-1:0] h_out,
output [BITS-1:0] x_out,

output [BITS*(ORDER+1)-1:0] tap_reg_out_test,
output [BITS*2*(ORDER+1)-1:0] y_test,
output [ORDER:0] clk2,
output [BITS-1:0] x_reg_test,
output [2*(ORDER+1)-1:0] ctr_out_test
);

localparam NUM_TAPS = ORDER+1;
localparam H_LENGTH = BITS*(NUM_TAPS);

//INITIALIZE FILTER ROM
wire [BITS-1:0] h_rom_data;

ROM FILTER_ROM(
.addr(h_rom_addr),
.rd(h_rom_rd),
.clk(clk),

.out(h_rom_data)
);

defparam FILTER_ROM.FILENAME = "filter_memory.mem";
defparam FILTER_ROM.BITS = BITS;
defparam FILTER_ROM.ROWS = ROWS;

//INITIALIZE SIGNAL ROM
wire [BITS-1:0] x_rom_data;

ROM SIGNAL_ROM(
.addr(x_rom_addr),
.rd(x_rom_rd),
.clk(clk),

.out(x_rom_data)
);

defparam SIGNAL_ROM.FILENAME = "signal_memory.mem";
defparam SIGNAL_ROM.BITS = BITS;
defparam SIGNAL_ROM.ROWS = ROWS;
```

```
//FOR FILTER
wire [H_LENGTH-1:0] h_rom_data_array;

assign h_rom_data_array = {NUM_TAPS{h_rom_data}};

//DIAGNOSTICS
assign h_out = h_rom_data_array;
assign x_out = x_rom_data;

wire [BITS-1:0] x_reg;
assign x_reg_test = x_reg;

//HOLDS X DATA FROM X_ROM TO PIPELINE
DREGISTER X_REG(
.in(x_rom_data),
.clr(clrs[0]), //WHEN FILTER TAPS ARE CLEARED, CLEAR X REG TOO
.clk(clk),
.enable(1'b1), //ALWAYS ENABLED BECAUSE SHORTED TO X ROM
//TAP REGISTERS HAVE ENABLE WIRING BECAUSE THEY SHARE THE SAME ROM
.out(x_reg)
);

defparam X_REG.BITS = BITS;

FILTER FIR(
.x(x_reg),
.h(h_rom_data_array),

.clk(clk),
.clrs(clrs),
.enables(enables),

.y(y),

//DIAGNOSTICS
.tap_reg_out_test(tap_reg_out_test),
.y_test(y_test),
.clk2(clk2),
.ctr_out_test(ctr_out_test)
);

defparam FIR.BITS = BITS;
defparam FIR.ORDER = ORDER;

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

module FILTER #(parameter BITS = 16, parameter ORDER = 4) (
input [BITS-1:0] x,
input [BITS*(ORDER+1)-1:0] h,
//TAP REGISTERS
input clk,
```

```
input [ORDER:0] clrs,
input [ORDER:0] enables,

output [BITS*2-1:0] y,

//DIAGONSTICS
output [BITS*(ORDER+1)-1:0] tap_reg_out_test,
output [BITS*2*(ORDER+1)-1:0] y_test,
output [ORDER:0] clk2,
output [2*(ORDER+1)-1:0] ctr_out_test
);
//-----------------------------------SETUP---------------------------------
//Wire array for setting up TAPS
wire [BITS-1:0] hn [ORDER:0];

//Feed forward output from each tap module
wire [BITS*2-1:0] tap_data [ORDER:0];

//Breaks down expanded h input
genvar c;
generate
    for(c = ORDER; c>=0; c=c-1) begin
    //TODO: REVERSE ORDER SO USER DOES NOT HAVE TO FLIP IMPULSE RESPONSE
        assign hn[c] = h[BITS*(c+1)-1 : (BITS*(c+1)-1) - (BITS-1)];
    end

endgenerate
//-----------------------------------SETUP---------------------------------

//-------------------------------DEBUGGING-------------------------------

wire [BITS-1:0] tap_reg_out [ORDER:0];

//EXPANDED VECTOR TO SEE THE REGISTER CONTENTS OF EACH TAP
genvar G;
generate
    for(G = ORDER; G >= 0; G = G-1) begin
        assign tap_reg_out_test[BITS*(G+1)-1 : (BITS*(G+1)-1) - (BITS-1)]
        = tap_reg_out[G];
    end
endgenerate


assign y = tap_data[0];

//THIS GENERATE CREATES AN EXPANDED VECTOR TO SEE THE OUTPUT OF EACH TAP
genvar t;
generate
    for(t = ORDER; t >= 0; t = t-1) begin
        assign y_test[BITS*2*(t+1)-1 : (BITS*2*(t+1)-1) - (BITS*2-1)]
        = tap_data[t];
    end
endgenerate

wire [1:0] ctr_out [ORDER:0];
```

```
//THIS GENERATE CREATES AN EXPANDED VECTOR TO SEE THE CTR BITS OF EACH TAP
genvar k;
generate
    for(k = ORDER; k >= 0; k = k-1) begin
        assign ctr_out_test[2*(k+1)-1 : (2*(k+1)-1) - (1)]
        = ctr_out[k];
    end
endgenerate

//--------------------------------DEBUGGING---------------------------------

//--------------------------------LOGIC-------------------------------------
genvar i;
generate
for(i=ORDER; i>=0; i=i-1) begin

    //IN BETWEEN
    if((i!=0) & (i!=ORDER)) begin
        TAP_X #(BITS) TAP_X_INST(
            .tap(hn[i]),
            .x(x),
            .prev_stage(tap_data[i+1]),

            .clr(clrs[i]),
            .enable(enables[i]),
            .clk(clk),

            .out(tap_data[i]),

            //DIAGNOSTICS
            .tap_reg_out(tap_reg_out[i]),
            .clk2(clk2[i])
        );
    end

    //HIGHEST ORDER
    else if(i==ORDER) begin
        TAP_M #(BITS) TAP_M_INST(
            .tap(hn[i]),
            .x(x),

            .clr(clrs[i]),
            .enable(enables[i]),
            .clk(clk),

            .out(tap_data[i]),

            //DIAGNOSTICS
            .tap_reg_out(tap_reg_out[i]),
            .clk2(clk2[i]),
            .ctr_out(ctr_out[i])
        );
    end
```

```
    //LOWEST ORDER (OUTPUT)
    else if(i==0) begin
        TAP_0 #(BITS) TAP_0_INST(
            .tap(hn[i]),
            .x(x),
            .prev_stage(tap_data[i+1]),

            .clr(clrs[i]),
            .enable(enables[i]),
            .clk(clk),

            .out(tap_data[i]),

            //DIAGNOSTICS
            .tap_reg_out(tap_reg_out[i]),
            .clk2(clk2[i])
        );
    end
end
endgenerate

//---------------------------------LOGIC-------------------------------------

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

//Processes the highest order of the filter
//Includes a pipeline register, a multiplier, and a mux
module TAP_M #(parameter BITS = 16) (
input [BITS-1:0] tap,
input [BITS-1:0] x,

//For Tap register
input clr,
input enable,
input clk,

output reg [BITS*2-1:0] out,

//DIAGNOSTICS
output [BITS-1:0] tap_reg_out,
output clk2,
output [1:0] ctr_out
);

wire [BITS-1:0] tap_reg;
assign tap_reg_out = tap_reg;

//Setup Register to hault tap until all registers are done
DREGISTER TAP_REGISTER(
.in(tap),
.clr(clr),
.clk(clk),
```

```
.enable(enable),

.out(tap_reg),
.clk2(clk2)
);

defparam TAP_REGISTER.BITS = BITS;

//Instantiate Array Multiplier
wire [BITS*2-1:0]   mult_out;

ARRAY_MULTIPLIER #(BITS) MULT (
.a(tap_reg),
.b(x),

.p(mult_out)
);

//SETUP CTR FOR PIPELINING ANSWER
reg [1:0] ctr;
initial ctr = 0;

//INITIAL OUTPUT IS 0 TO PREVENT ERRORS
initial out = 0;

//WIRE OUT DIAGNOSTICS
assign ctr_out = ctr;

//Clock pipeline register
always@(posedge clk) begin
    ctr = ctr + 1;

    if((x[0] === 1'bx) | (x[0] === 1'bz)) out <= {(BITS*2){1'b0}};
    if(ctr > 1'b1) begin
        out <= mult_out;
        ctr = 2'b00;
    end

end

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

//Processes stages in between highest order and lowest order of filter
//Includes a pipeline register, a multiplier, an adder, and a mux
module TAP_X #(parameter BITS = 16)(
input [BITS-1:0] tap,
input [BITS-1:0] x,
input [BITS*2-1:0] prev_stage,

//FOR TAP REGISTER
input clr,
input enable,
```

```
input clk,

output reg [BITS*2-1:0] out,

//DIAGNOSTICS
output [BITS-1:0] tap_reg_out,
output clk2
);

wire [BITS-1:0] tap_reg;
assign tap_reg_out = tap_reg;

//Setup Register to hold tap from ROM
DREGISTER TAP_REGISTER(
.in(tap),
.clr(clr),
.clk(clk),
.enable(enable),

.out(tap_reg),
.clk2(clk2)
);

defparam TAP_REGISTER.BITS = BITS;

//Instantiate Array Multiplier
wire [BITS*2-1:0]   mult_out;

ARRAY_MULTIPLIER #(BITS) MULT (
.a(tap_reg),
.b(x),

.p(mult_out)
);

//Instantiate CLA
wire [BITS*2-1:0]   cla_out;

CLA #(BITS*2) ADDER(
.A(mult_out),
.B(prev_stage),
.Cin(1'b0),

.Sum(cla_out),
.Cout()
);

reg [1:0] ctr;
initial ctr = 0;
initial out = 0;

//Clock pipeline register
always@(posedge clk) begin
    ctr = ctr + 1;
```

```
    if((x[0] === 1'bx) | (x[0] === 1'bz)) out <= {(BITS*2){1'b0}};
    if(ctr > 1'b1) begin
        out <= cla_out;
        ctr = 2'b00;
    end
end

endmodule

////////////////////////////////////////////////////////////////////////////////
//////

//Processes the lowest order of the filter
//Includes a a multiplier and a mux
module TAP_0 #(parameter BITS = 16) (
input [BITS-1:0] tap,
input [BITS-1:0] x,
input [BITS*2-1:0] prev_stage,

input clr,
input enable,
input clk,

output reg [BITS*2-1:0] out,

//DIAGNOSTICS
output [BITS-1:0] tap_reg_out,
output clk2
);

//Setup Register to hold tap from ROM
wire [BITS-1:0] tap_reg;
assign tap_reg_out = tap_reg;

DREGISTER TAP_REGISTER(
.in(tap),
.clr(clr),
.clk(clk),
.enable(enable),

.out(tap_reg),
.clk2(clk2)
);

defparam TAP_REGISTER.BITS = BITS;

//Instantiate Array Multiplier
wire [BITS*2-1:0]   mult_out;

ARRAY_MULTIPLIER #(BITS) MULT (
.a(tap_reg),
.b(x),

.p(mult_out)
);
```

```
//Instantiate CLA
wire [BITS*2-1:0]   cla_out;

CLA #(BITS*2) ADDER(
.A(mult_out),
.B(prev_stage),
.Cin(1'b0),

.Sum(cla_out),
.Cout()
);

reg [1:0] ctr;
initial ctr = 0;
initial out = 0;

//Clock pipeline register
always@(posedge clk) begin
    ctr = ctr + 1;

    if((x[0] === 1'bx) | (x[0] === 1'bz)) out <= {(BITS*2){1'b0}};
    if(ctr > 1'b1) begin
        out <= cla_out;
        ctr = 2'b00;
    end
end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//////
//////////////////////////////////////////////////////////////////////////////////
//////
////USE OF PREVIOUSLY WRITTEN MODULES (CLA AND ARRAY MULTIPLIER)

//Wrapper for Carry-Lookahead-Logic and Partial Full Adders
module CLA #(parameter bits = 64) ( //parameter to specify number of bits for adder
input [bits-1:0] A, B, //augend and addend
input Cin,  //first carry in

output [bits-1:0] Sum, //sum output
output Cout //carry out bit

//output [bits-1:0]carries //for testbench
);

wire [bits-1:0]P_in, G_in; //wire bus for propagate and generate bits
wire [bits-1:0]carries; //wire bus for carry bits

assign Cout = carries[bits-1]; //propagate last carry out from msb of carries bus

//create partial full adder instance for the first bit to incorporate Cin
PFA PFA0(
.A(A[0]),
```

```
.B(B[0]),
.Cin(Cin),

.P(P_in[0]),
.G(G_in[0]),
.S(Sum[0])

);

genvar i;
generate //Instantiate N PFA's for bits
    for(i=1; i<bits; i=i+1) begin : PFAS
        PFA PFA_I(
            .A(A[i]),
            .B(B[i]),
            .Cin(carries[i-1]),

            .P(P_in[i]),
            .G(G_in[i]),
            .S(Sum[i])
        );
    end
endgenerate

//Instantiate Carry-Lookahead-Logic block
CLL #(.bits(bits)) CLL_INST(
.Cin(Cin),
.P(P_in),
.G(G_in),

.Cout(carries)
);

endmodule

//This module handles the carry outs for each partial full adder
//Acts as a wrapper to individual carry_gen modules
module CLL #(parameter bits = 8) (
    input Cin,
    input [bits-1: 0] P, G,
    output [bits-1: 0] Cout
);

//Create carry generator block for first bit
CARRY_GEN INST0(
        .G(G[0]),
        .P(P[0]),
        .Cin(Cin),

        .Cout(Cout[0])
);

genvar i;
generate //Create N bits of carry generators
    for(i=1; i < bits; i=i+1) begin : CARRIES
```

```
        CARRY_GEN INST_I(
        .G(G[i]),
        .P(P[i]),
        .Cin(Cout[i-1]),

        .Cout(Cout[i])
        );

    end
endgenerate
endmodule

module ADD_HALF(
    output cout,
    output sum,
    input a,
    input b
);
    xor(sum, a, b);
    and(cout, a, b);
endmodule

//Calculates the propagate, generate, and sum bits
module PFA(
    input A, B, Cin,
    output P, G, S
);

    assign P = A ^ B;   //xor gate for propagate
    assign G = A & B;   // and gate for generate
    assign S = Cin ^ P; // xor gate for sum

endmodule

module FULL_ADDER(
    input a,b,cin,
    output cout,sum

);

wire half1_cout, half1_sum, half2_cout;

ADD_HALF HALF1(
.cout(half1_cout),
.sum(half1_sum),
.a(a),
.b(b)
);

ADD_HALF HALF2(
.cout(half2_cout),
.sum(sum),
.a(cin),
.b(half1_sum)
);
```

```
//for the carry out of both half adders
or(cout, half2_cout, half1_cout);

endmodule

//Calculates the carry using generate, propagate, and carry in bits
module CARRY_GEN(
    input G, P, Cin,
    output Cout
);

assign Cout = G | P&Cin; //wire out carry calculation

endmodule

////////////////////////////////////////////////////////////////////////////////
//////
////////////////////////////////////////////////////////////////////////////////
//////

//PARAMETER BITS SPECIFIES THE INPUTS
module ARRAY_MULTIPLIER #(parameter bits = 32) (
    input [bits-1:0] a, b,
    output [bits*2-1:0] p
);

//rows means the partial sums
//for 4-bit adder:
//carries[1][0] is the carry out from the rightmost halfadder in the 1st partial sum

//DECLARING: [index] carries [rows]
//ACCESSING: carries[rows][index]

wire [bits-1:0]     carries [bits-1:1];
wire [bits*2-1:0]   sums [bits-1:1];

assign p[0] = a[0] & b[0]; //always true

genvar i,j;
generate
    for(i=1; i < bits; i=i+1) begin : GEN_PARTIALS
        for(j=0; j < bits; j=j+1) begin: GEN_ADDERS

            //FIRST LEVEL OF PARTIAL SUMS
            if(i==1) begin

                //FIRST ADDER
                if(j==0) begin
                    ADD_HALF INSTF_HA_FIRST (
                        .a(a[i]&b[j]),
                        .b(a[i-1]&b[i]),

                        .sum(p[i]),
                        .cout(carries[i][j])
```

```
        );
    end

    //LAST ADDER
    else if(j==(bits-1)) begin
        ADD_HALF INSTF_HA_LAST (
            .a(a[i]&b[j]),
            .b(carries[i][j-1]),

            .sum(sums[i][j]),
            .cout(carries[i][j])
        );
    end

    //IN BETWEEN
    else begin
        FULL_ADDER INSTF_FA (
            .a(a[i]&b[j]),
            .b(a[i-1]&b[j+1]),
            .cin(carries[i][j-1]),

            .sum(sums[i][j]),
            .cout(carries[i][j])
        );
    end
end //FIRST LEVEL

//LAST LEVEL OF PARTIAL SUMS
else if(i==(bits-1)) begin

    //FIRST ADDER
    if(j==0) begin
        ADD_HALF INSTL_HA_FIRST (
            .a(sums[i-1][j+1]),
            .b(a[i]&b[j]),

            .sum(p[i+j]),
            .cout(carries[i][j])
        );
    end

    //LAST ADDER
    else if(j==(bits-1)) begin
        FULL_ADDER INSTL_FA_LAST (
            .a(carries[i-1][j]),
            .b(a[i]&b[j]),
            .cin(carries[i][j-1]),

            .sum(p[i+j]),
            .cout(p[i+j+1])
        );
    end

    //IN BETWEEN
    else begin
```

```
                FULL_ADDER INSTL_FA (
                    .a(sums[i-1][j+1]),
                    .b(a[i]&b[j]),
                    .cin(carries[i][j-1]),

                    .sum(p[i+j]),
                    .cout(carries[i][j])
                );
            end
        end //LAST LEVEL

        //ALL LEVELS IN BETWEEN
        else begin

            //FIRST ADDER
            if(j==0) begin
                ADD_HALF INSTB_HA_FIRST (
                    .a(sums[i-1][j+1]),
                    .b(a[i]&b[j]),

                    .sum(p[i]),
                    .cout(carries[i][j])
                );
            end

            //LAST ADDER
            else if(j==(bits-1)) begin
                FULL_ADDER INSTB_FA_LAST (
                    .a(carries[i-1][j]),
                    .b(a[i]&b[j]),
                    .cin(carries[i][j-1]),

                    .sum(sums[i][j]),
                    .cout(carries[i][j])
                );
            end

            //IN BETWEEN
            else begin
                FULL_ADDER INSTB_FA (
                    .a(sums[i-1][j+1]),
                    .b(a[i]&b[j]),
                    .cin(carries[i][j-1]),

                    .sum(sums[i][j]),
                    .cout(carries[i][j])
                );
            end
        end //IN BETWEEN LEVELS
    end //ADDERS FOR LOOP
end //LEVELS FOR LOOP
endgenerate
endmodule
```