Project 1: IIR Filter Design, MATLAB and RTL Co-method

Zhuohao Lee ✉[*], Zhenghang Gao ✉[†]

April 23, 2022

Contents

[*]edith_lzh@sjtu.edu.cn | 519021911248 | Project Repo
[†]2454260295@qq.com | 519030910146

1 Introduction

IIR filters are often used to calculate scenarios with limited storage resources, low latency, and wireless phase requirements. After the IIR filter, it can be connected to the all-pass network for phase equalization and optimize the group delay.

Compared with FIR, which is more common in the real world, IIR has an infinite $h(n)$. In general, IIR has a time equation which is shown as $y(n) = \sum_{k=0}^{\infty} h(k)x(n-k)$. Theorically, infinite length convolution can't be implemented directly, so we should induce **feedback loop** in our system. As result, IIR with feedback often shows like $y(n) = \sum_{i=0}^{N} b_i x(n-i) - \sum_{j=1}^{M} a_j y(n-j)$.

In this lab, we're gonna implement IIR by **direct II structure**($H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{i=0}^{N} b_i z^{-i}}{1+\sum_{j=0}^{N} a_j z^{-j}}$) and improve it in Verilog HDL under the design guidance of MATLAB filter designer. Besides, we're gonna simulate it in Vivado HSL. We also tune structures by the method of **retiming, pipeline and folding**, which can improve frequency or reduce area, respectivly.

1.1 Labor Division

Zhenghang finished A2 | MATLAB prototype filter design | A1 original RTL coding, simulation, MATLAB verfication | Word version of report in Introduction, parts of A1 and A2

Zhuohao finished A3 | A1 new RTL coding, debugging, simulation, MATLAB verification | Whole LaTeX version of report, parts of Introduction, A3, Conclusion of content, GitHub version control

1.2 Design parameters

Design parameters from slides are shown as Table.1 Taking Butterworth

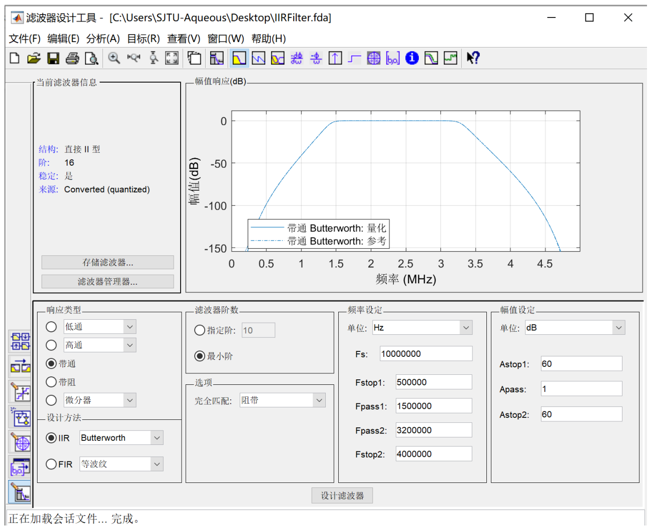
Fs	10Mhz
Fstop1	0.5Mhz
Fpass1	1.5Mhz
Fpass2	3.2Mhz
Fstop2	4Mhz
Apass	1dB
Astop1	60dB
Astop2	60dB

Table 1: Design parameter

analog filter as prototype, the order, fixed point tap coefficient and pole zero of IIR digital filter are determined by using MATLAB filter design tool.

2 MATLAB Pre-design

2.1 Specify filter coefficient using MATLAB FDA tool



滤波器设计工具 - [C:\Users\SJTU-Aqueous\Desktop\IIRFilter.fda]

当前滤波器信息

结构: 直接 II 型
阶: 16
稳定: 是
来源: Converted (quantized)

响应类型: 低通 高通 [✓ 带通] 带阻 微分器

设计方法: [✓ IIR Butterworth] FIR 等波纹

滤波器阶数: 指定阶: 10 [✓ 最小阶]

选项: 完全匹配: 阻带

频率设定
单位: Hz
Fs: 10000000
Fstop1: 500000
Fpass1: 1500000
Fpass2: 3200000
Fstop2: 4000000

幅值设定
单位: dB
Astop1: 60
Apass: 1
Astop2: 60

正在加载会话文件... 完成。

Figure 1: Filter Designer

The filter Designer tool settings screenshot is like in Fig.1

Step1: Choose correct type of filter and fill in the appropriate frequency and parameters.

Step2: Make sure the simulated filter corresponds to the filter we want and then click "design filter".

Step3: Using quantilizer to quantilize coefficients like a and b(Denominator and Numerator).

Step4: Generate coefficients and export them to an ASCII file.

2.2 MATLAB Fixed-point verification

First, we need export the filter coefficient designed by Fdatool to Workspace for our calculation.

The performance of filter is verified by writing MATLAB code. As can be clearly seen from the figure below, the filter I designed can filter out most of the high frequency signals and low frequency signals. It's a pretty good bandpass filter.

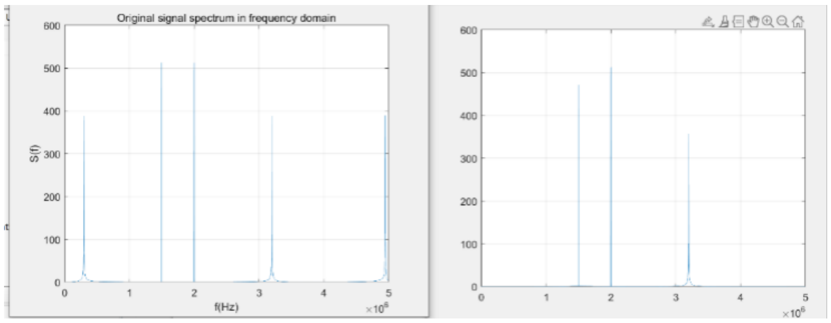


Figure 2: MATLAB Fixed-point verificatio

Since Verilog can only process binary numbers, when exporting coefficients, I did not choose to use the tap coefficients saved in workspace in the previous step, but chose to use MATLAB – Fdatool's own binary export function to export coefficients. In this way, you can not only avoid the calculation error in the process of converting from decimal to binary, but also don't have to worry about whether the bits width represented by the coefficient is accurate.

Also, we should note here that since Verilog can only compute fixed-point numbers, we need to convert floating point numbers to integers. In the single-section filter, since the word length is 22 bits, I uniformly multiply the coefficients by 2^{20}. After the coefficient multiplication operation is completed, the result (binary number) is moved 20 bits to the right arithmetically, so that the high position is filled with sign bits, which does not affect the correctness of the final calculation result.

In second order section filter, I uniformly multiply all the coefficients by $2^{14}$and rightshift the multiplier output 14bits arithmetically to ensure that the result is correct.

3 RTL Validation

3.1 A1

You can see source code in `A1` or `A1_new` file in upload `.zip` or GitHub.

3.1.1 Analysis

A1 is a single-section filter with 16 groups of a and b coefficients. By writing multiplier, adder, register sub-module, we only need to instantiate the sub-module in the main module and connect them together correctly, final IIR filter can be obtained.

In fact, we don't think about routine module writing at the beginning, because it would be very troublesome to write routine module in the final connection. But later, when talking with TA on tencent meeting, I changed my previous idea and chose to write each small module separately. In later debugging, I realized the importance of writing this way, the reasons are as follows:

1. Separate modules can ease the burden of debugging, reducing the workload.
2. Writing RTL code in separate modules can be very easy to modify the circuit structure, add or subtract components, modify the timing topology and determine the critical path.

3.1.2 Structure

The topology of A1 is as Fig.3 shows

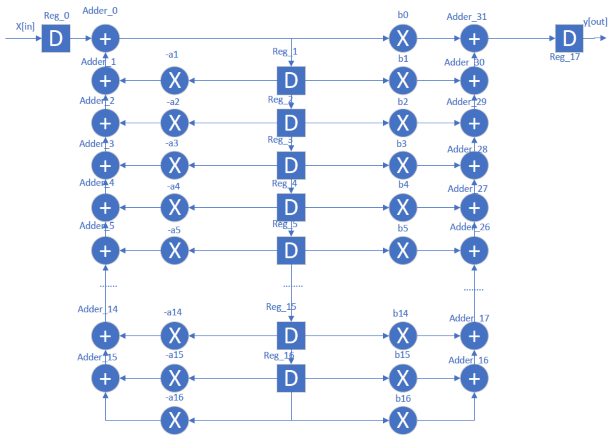


Figure 3: A1 structure

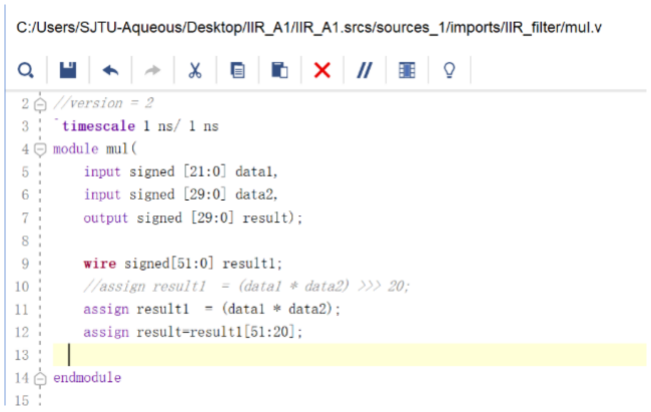
3.1.3 Result

- **Unify bits wide** First we assume input bits wide equals to 8bits. As coefficient a and b is a 20bits number, we finally choose a 30bits wide input and output for Adder Multiplier and Register. For the entire circuit output, a 30bits is carefully chosen since large bits wide results in Large resource consumption but small bits wide results in wrong output data. Bits wide of all components using are defined as follows:
- **Multiplier unit design** Multiplier design is of great significance. Since a multiplier will deal with two different bits wide components. The input coefficient of multiplier is a signed 22bits binary number, and the other input is a signed 30bits binary number. Therefore, the output of the multiplier is a 52bits signed number. However, the next stage is a adder with only 30bits input, so we need to truncate the output number to

Components	Muliplier	Adder	Register	input	output
Input1 Bits wide	22bits	30bits	30bits	8bits	30bits
Input2 Bits wide	30bits	30bits	30bits	8bits	30bits
Output Bits wide	30bits	30bits	30bits	8bits	30bits

Table 2: Design parameter

obtain a 30bits output. As we enlarge the coefficient input by 2^{20}, we should rightshift the output by 20bits. As for the reasons above, we only need [51:20] and the higher bits will be automatically filled with the sign bit by Verilog compiler.



C:/Users/SJTU-Aqueous/Desktop/IIR_A1/IIR_A1.srcs/sources_1/imports/IIR_filter/mul.v

```
//version = 2
`timescale 1 ns/ 1 ns
module mul(
    input signed [21:0] data1,
    input signed [29:0] data2,
    output signed [29:0] result);

    wire signed[51:0] result1;
    //assign result1  = (data1 * data2) >>> 20;
    assign result1  = (data1 * data2);
    assign result=result1[51:20];

endmodule

```

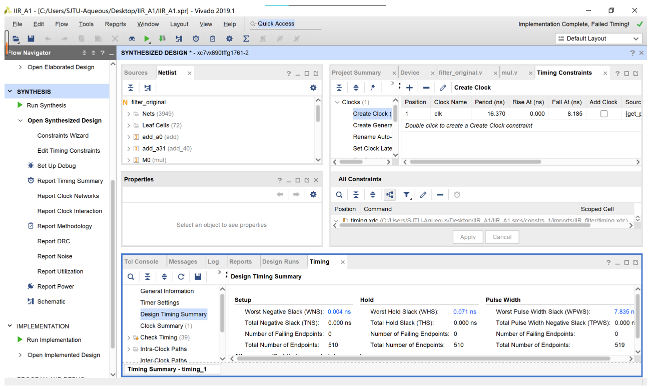
Figure 4: Verilog port definition in A1

- **Adder & Register unit design**This part will be relatively easier since no bit wide mismatch encounted. The adder adds two 30 bits number and the register procuced a clock period delay and restore the number for a while. It should be noted that when two numbers are added, carry out case may occur and we should carefully determine the bit wide to avoid the overflow.

- **Maximum clock frequency** In this design, a clk frequency of **61.1Mhz** can be obtained. (Fig.5 Fig.6)

- **Resource Utilization** We can use vivado timing summary to get critical path. In this design, the critical path is composed of 2 multiplier and 17 adder. The resource usage is listed as Fig.7 (**LUT: 2248, FFT: 558**)

- **Output waveform verification** The MATLAB validation result is like Fig.8. It shows that we filter the frequency of 3Mhz, which is corrsponding to our original thoughts.


IIR_A1 - [C:/Users/SJTU-Aqueous/Desktop/IIR_A1/IIR_A1.xpr] - Vivado 2019.1

Implementation Complete, Failed Timing!

SYNTHESIZED DESIGN * - xc7vx690tffg1761-2

Netlist: filter_original
- Nets (3949)
- Leaf Cells (72)
- add_a0 (add)
- add_a31 (add_40)
- M0 (mul)

Properties: Select an object to see properties

Timing Constraints — Create Clock

Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Sourc
1	clk	16.370	0.000	8.185	☐	[get_p

Double click to create a Create Clock constraint

All Constraints

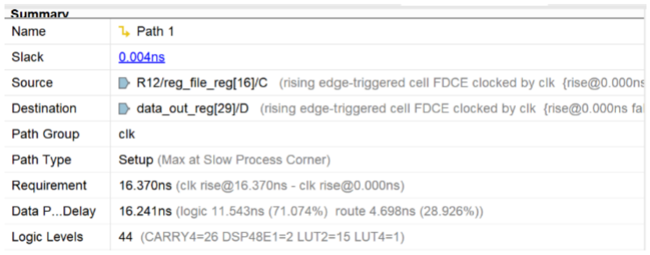
Position	Command	Scoped Cell
timing.xdc (C:/Users/SJTU-Aqueous/Desktop/IIR_A1/IIR_A1.srcs/constrs_1/imports/IIR_filter/timing.xdc)		

Timing — Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.004 ns	Worst Hold Slack (WHS):	0.071 ns	Worst Pulse Width Slack (WPWS):	7.835 n
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 n
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	510	Total Number of Endpoints:	510	Total Number of Endpoints:	519

Timing Summary - timing_1

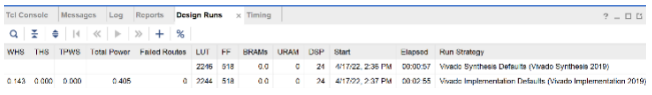
Figure 5: A1 timing report


Summary

Name	Path 1
Slack	0.004ns
Source	R12/reg_file_reg[16]/C (rising edge-triggered cell FDCE clocked by clk {rise@0.000ns
Destination	data_out_reg[29]/D (rising edge-triggered cell FDCE clocked by clk {rise@0.000ns fa
Path Group	clk
Path Type	Setup (Max at Slow Process Corner)
Requirement	16.370ns (clk rise@16.370ns - clk rise@0.000ns)
Data P...Delay	16.241ns (logic 11.543ns (71.074%) route 4.698ns (28.926%))
Logic Levels	44 (CARRY4=26 DSP48E1=2 LUT2=15 LUT4=1)

Figure 6: A1 timing report

- **Hint:** Due to the long structure of A1 naive one, the output is not so stable as we think. In `A1_new`, we proposed a new sysmetrical in/output A1 structure with lightly optimization iunder retiming. To be more specific, we seperates the registers intermediate shown in Fig.3 to be **bit-wise**, which means thay have different bit width. And in order to avoid mistakes in subsections like `adder` or `multiplier`, we induce IP core in Vivado. Then the MATLAB validation result wave is more stable. In fact, we don not design IIR filter like A1 originally presents. MATLAB even uses cascode to implement this. (For TAs: if you wan to reproduce our result, please refer to `A1_new`, which is more stable)


Design Runs

WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Run Strategy
					2246	518	0.0	0	24	4/17/22, 2:36 PM	00:00:57	Vivado Synthesis Defaults (Vivado Synthesis 2019)
0.143	0.000	0.000	0.405	0	2244	518	0.0	0	24	4/17/22, 2:37 PM	00:02:55	Vivado Implementation Defaults (Vivado Implementation 2019)

Tcl Console
Messages
Log
Reports
Design Runs
Timing

Figure 7: A1 resource report

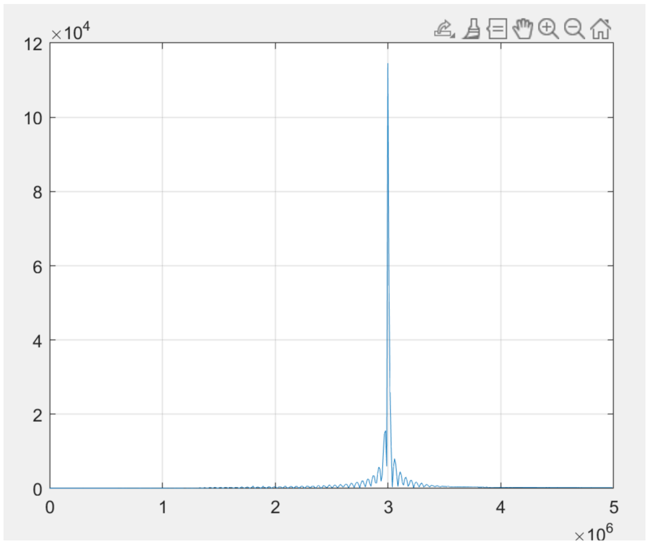


Figure 8: A1 validation

3.2 A2

You can see source code in `A2` in upload `.zip` or GitHub.

3.2.1 Analysis

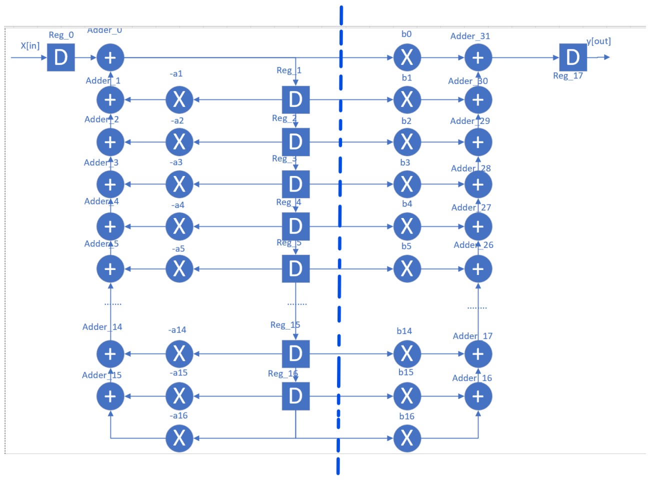


Figure 9: Method to add pipeline

One possible solution to improve clock frequency is to add pipelines on feedforward cut set, as listed on the graph above. However, the clk frequency isn't improved much since there are still 1 multiplier and 16 adders. So we divide IIR-II filter into several segments and cascade them together. In order to improve the clk frequency, I insert several register between each filter. Thanks to cascade structure, it's much more easier to insert register on feedforward cut set. The whole structure is listed Fig.9 and Fig.10.

Structure A2 is a cascade design of 8 IIR-II filter. As resgisters are inserted between each sub-filters using , we assume clk frequency can be boosted to a higher level. Since calculation in each segment is relatively simple, I didn't write reg,adder,multiplier in different modules. Rightshift procedure is carried out once the multiply output is formed. Meanwhile, I write segment filter in an individual module and then the function of top module is to connect them together.

3.2.2 Structure

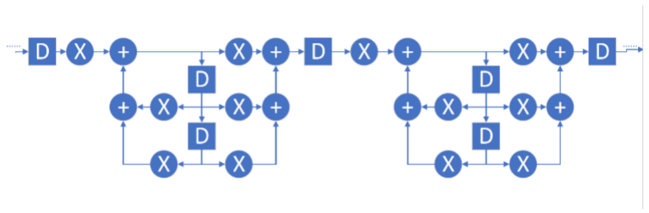


Figure 10: A2 structure

3.2.3 Result

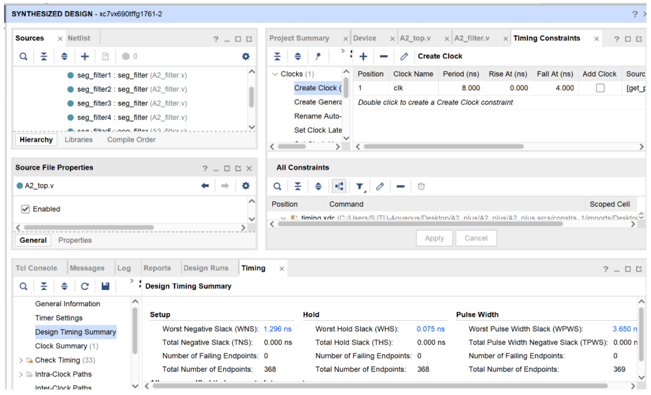
- **Unify bit wide**

Components	Whole structure	subfilter	Register	Parameters(a,b,g)
Input	16bits	16bits	16bits	16bits
Output	16bits	16bits	16bits	16bits

Table 3: Design parameter

- **Filter designed in each stage** Output in previous stage are multiplied by *gn*and then devided by 2^{14}, which is the input of the next stage. The input data then processed by the subfilter and generate an output. After 8 subfilters, the final data can be obtained and stored in a `.txt` file.

- **Clock frequency** **125Mhz** clock can be obtained since the critical path is shorten to 3adder and 2multiplier. (Fig.11)



seg_filter1 : seg_filter (A2_filter.v)
seg_filter2 : seg_filter (A2_filter.v)
seg_filter3 : seg_filter (A2_filter.v)
seg_filter4 : seg_filter (A2_filter.v)

A2_top.v

Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Sourc
1	clk	8.000	0.000	4.000	☐	[get_

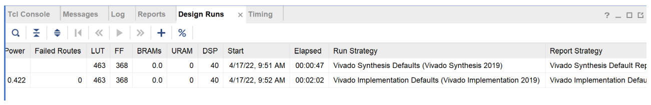
Double click to create a Create Clock constraint

Position	Command	Scoped Cell
timing.xdc (C:/Users/SJTU-Aqueous/Desktop/A2_plus/A2_plus/A2_plus.srcs/constrs_1/imports/Deskto		

```
Setup                                      Hold                                       Pulse Width
Worst Negative Slack (WNS):   1.296 ns     Worst Hold Slack (WHS):       0.075 ns     Worst Pulse Width Slack (WPWS):              3.650 n
Total Negative Slack (TNS):   0.000 ns     Total Hold Slack (THS):       0.000 ns     Total Pulse Width Negative Slack (TPWS):     0.000 n
Number of Failing Endpoints:  0            Number of Failing Endpoints:  0            Number of Failing Endpoints:                 0
Total Number of Endpoints:    368          Total Number of Endpoints:    368          Total Number of Endpoints:                   369
```

Figure 11: A2 time reporting

- **Resourse usage** The cascode structure uses **LUT: 463, FF: 368** (Fig.12)



Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Run Strategy	Report Strategy
		463	368	0.0	0	40	4/17/22, 9:51 AM	00:00:47	Vivado Synthesis Defaults (Vivado Synthesis 2019)	Vivado Synthesis Default Rep
0.422	0	463	368	0.0	0	40	4/17/22, 9:52 AM	00:02:02	Vivado Implementation Defaults (Vivado Implementation 2019)	Vivado Implementation Defau

Figure 12: A2 resource reporting

- **Hint:** We also did more, by adding other levels of pipeline we could make it faster. The structure is like Fig.13: After evaluating, we reached at

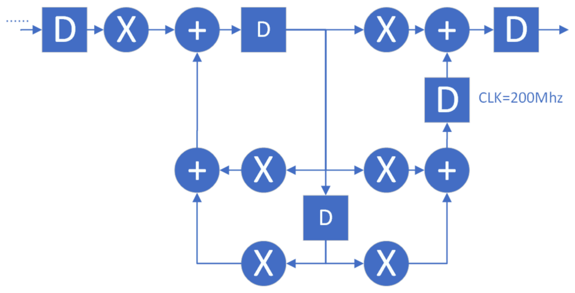


Figure 13: Optimized A2 structure

most **200Mhz**, 75Mhz faster than original A2. The critical path in this structure shown in Fig. 13 is shorter.

3.3 A3

You can see source code in `A3` in upload `.zip` or GitHub.

3.3.1 Analysis

Generally, we're liekly to use parallel processing pipeline structure to speed up when we process data. However, if there're overwhelming multiplier and adder resources, some insufficient resources stuff, here need to consider **folding**. Multiple identical computing units are controlled by time division multiplexing and completed by using one computing unit. Floding will slow the processing speed, but it can save computing resources.

Folded system needs a shift register array to store coefficiency of a and b. To identify regular and reusable hardware components, I fold the regular direct II IIR shown before in Fig.3. It's quite similar to that we discussed in lecture. The main structure is shown in Fig.14 .

In the direct II IIR recently, the minimum reuse resource is **1 adder** and **1 multiplexer**. The design flow of A3 is briefly summarized to 2 facts:

1. **data flow** mappinmg from algorithm to circuit, including arithmetic units and store elements
2. **control signal** determine when to execute a specific operation. We can implement this logic by a **Finite State Machine (or FSM)**

Store element: We propose a **shift register** to store all 32 coefficients derive from MATLAB, and another 2 to store input **x** and output **y**.

```
wire signed [9:0] coe [32:0]; # store a,b, from a(n), a(n-1),
     ... , a(n-16), b(n-1), ... , b(n-16)
# All coefficients should be stored and no need to be shifted
    , so wire is Ok
# here's coeffient listing

reg signed [9:0] mid_reg_x [16:0]; # store from x[n] to x[n
    -16]
reg signed [9:0] mid_reg_y [15:0]; # store from y[n-1] to y[n
    -16]
# All input value x should be stored and need to be shifted,
    so it's reg type

# mid is another intermediate register array to store
    intermediate valuemid <= mid + coe * reg
reg signed [29:0] mid;
```

Control signal: We propose a **FSM** to control when to read input values, then after input and coeffients computation, store output and go to next. A FSM usually consists of next logic, status register and output logic. The corresponding static diagram of FSM we use here is referred to lecture slides P63

```
reg [5:0] count; //count for pulse, up to number of
    coefficient, which is 33
```

```

always @ (posedge clk or negedge rst_n) begin
    if (!rst_n)
        state_current <= IDLE; # IDLE = 2'b00, referred as Stall status in FSM, other status are
        # READ = 2'b01,
        # CALC = 2'b11,
        # OUT = 2'b10;
        # note the we use Gray Code here, avoiding possible errors
    else
        state_current <= state_next;
end

# 3-substate FSM
# 1) Timing status update. 2) Combinatorial logic jump. 3) Sequential logic output
# when state_current == IDLE and input_valid == 1, control transfers from AVAILABLE to WRITE and make count 0 and out_valid to 0.
# When transfer from AVAILABLE to WRITE, move former x[n-16] out, move x[n] in
# 33 times MUL+AND operations in total, controlled by count. When count == 17, jump out of reg_x and enters reg_y. When count == 33, enters OUT
# When it's at OUT, make output y[n] and write y[n] back to reg_y.

always @ (*) begin
    case (state_current)
        IDLE:
            if (input_valid == 0 || output_valid == 0)
                state_next = IDLE;
            else
                state_next = READ;
        READ:
            state_next = CALC;
        CALC:
            if (count != 33)
                state_next = CALC;
            else
                state_next = OUT;
        OUT:
            state_next = IDLE;
        default:
            state_next = IDLE;
    endcase
end

# reg_x
# reg_y
# mid
```

3.3.2 Structure

The structure is similar to what we mentioned in lecture sildes for FIR. (Fig. 14)

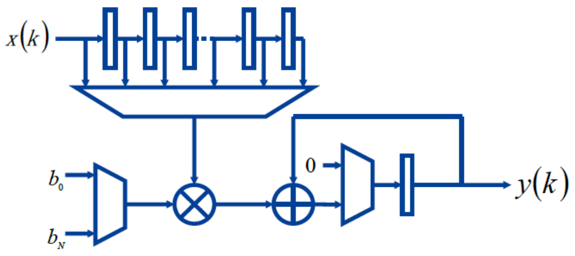


Figure 14: Overview structure of folded system

3.3.3 Result

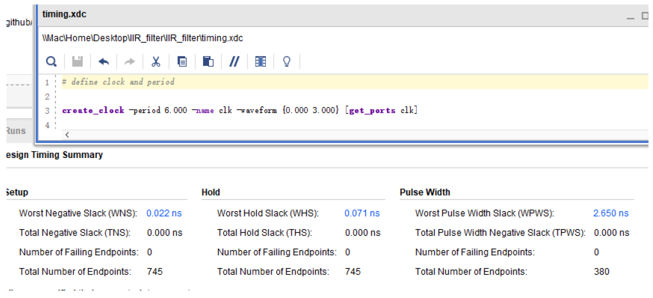
To optimize the structure, I added 2 pipeline level in the intermediate path. Note that this method is under supervision of practice. Critical path can be reduced in this way.

- **Unify bit wide** In this part, we utilize a quanted float number of 10 bits as the input. The output is the same as the input to make it more easy to get regualr inthe inter process. The details are referred in the GitHub repo. Here are the parameters of the data width in Table.4. As for the testbech, I generated a 10-bit `input.txt`. Vivado will read the data in it and compute the outcome and store in `output.txt`. Then I load this `.txt` to `IIR.mat` to check the output wave.

Components	Whole structure	Register	valid	Parameters
Input	10bits	30bits	1bit	16bits
Output	10bits	30bits	1bit	16bits

Table 4: Design parameter in A3

- **Clock frequency:** **166Mhz**(period of time: 6ns) clock can be obtained since the critical path is shorten to 1adder and 1multiplier. It reached **2.76** times as the baseline. The result from vivado is shown as Fig. 15


timing.xdc

\\Mac\Home\Desktop\IIR_filter\IIR_filter\timing.xdc

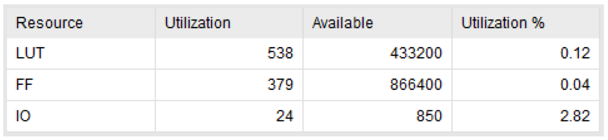
```
# define clock and period

create_clock -period 6.000 -name clk -waveform {0.000 3.000} [get_ports clk]
```

esign Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.022 ns	Worst Hold Slack (WHS):	0.071 ns	Worst Pulse Width Slack (WPWS):	2.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	745	Total Number of Endpoints:	745	Total Number of Endpoints:	380

Figure 15: Clock frequency A3

- **Resourse usage:** The resource usage is shown as Fig.16. The LUT is **538** utilization and FF is **379** one. The usage reduces over **50%** compared to the non-optimized one.



Resource	Utilization	Available	Utilization %
LUT	538	433200	0.12
FF	379	866400	0.04
IO	24	850	2.82

Figure 16: Resourse usage

- **Wave Validation:** I tested it in MATLAB. As I input are **0.5MHz**, **3Mhz**, **4.2MHz**. The expected output is only filtered **3Mhz**. And the result shown in Fig.17 is correct!

- **More thinking:** Actually, I found that folding structure is pretty much similar to a CPU scheduling. We can imagine each input data as an instruction and the folded IIR just do: instrcution fetch → execution → memory access/write back. This is similar to a classic CPU when executes instructions because a real CPU pipes with cycles in limited resources. That's why we need 3 shift registers in A3. Shift register just acts like a memory system. And FSM, which is also vital in A3, acts like a signal controler or **scheduler** in CPU architecture. So, another idea about folded A3, consider it as a CPU, and design memory(register file), scheduler(FSM), input/output pad(IO data) and simulates like a CPU. I can do this later.

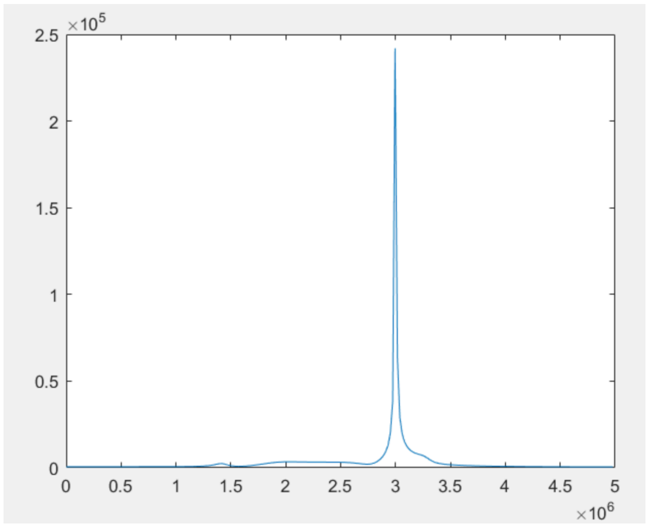


Figure 17: Wave Validation in A3

4 Conclusion

4.1 Problems

- **A1:** The original circuit is not so stable due to the long path and properties in IIR system.
- **A2:** The second-order IIR filter is easy to be realized. It performs good stability and clock frequency. And this is why MATLAB choose this form of filter as default type. No critical problem encounted in this case.
- **A3:** The fold circuits are hard to depict manually due to so many elements in the original one. And the debugging experience is not so good. To resolve the problem, be sure that 1) comments regularly 2)folding set can be rationally distributed 3)never give up

4.2 Achievements

In this lab, we manage strong Verilog and Vivado HSL skills. Blessed with strong debugging skills and construct a better conmmunication with TAs and teammates. Additionally, we have a better understanding of VLSI design techniques like retiming, pipeline, folding. Although folding is not so common because of its compelxity and insecurity, but it really help us to get familiar with these classic techniques then.

Thanks for TAs for discussing and resolvutions!