Final Project Report

PIIR Filter

Team Name: Twinkle Bunnies

ECE 551
Submitted to: Parmesh Ramanathan

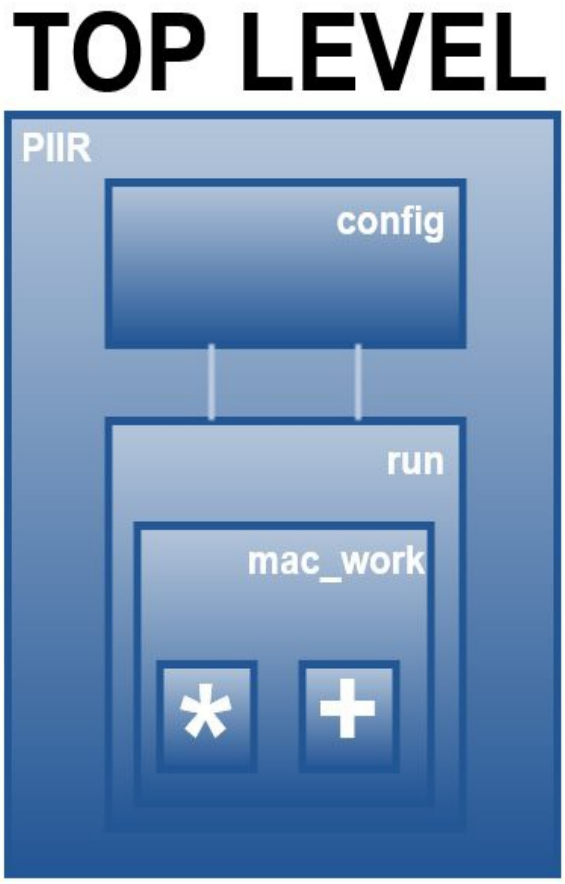
Bob Wagner
Jordan Friendshuh
Aritra Biswas

PIIR Filter Design Specifications

Optimized for	Area (via. Compile Ultra)
Area	2935
Clock Speed (Period of Clk1 & Clk2)	5.5 ns
“k” Value	1
# of Adders	1
# of Multipliers	1
# of Clock Cycles to calculate Y	5

Architectural Specification & Discussion

Our group decided to implement the entire design using one multiplier and one adder. In making this choice, we prioritized the drop in area over the increase in timing. This design produces outputs every five clock cycles. A Block Diagram has been provided to demonstrate the pipelining process:



Design Characteristics:

Optimized for: Area

Area: 2935

Minimum Time Period: 5.5 ns
(Note: Both clocks operate at the same speed)

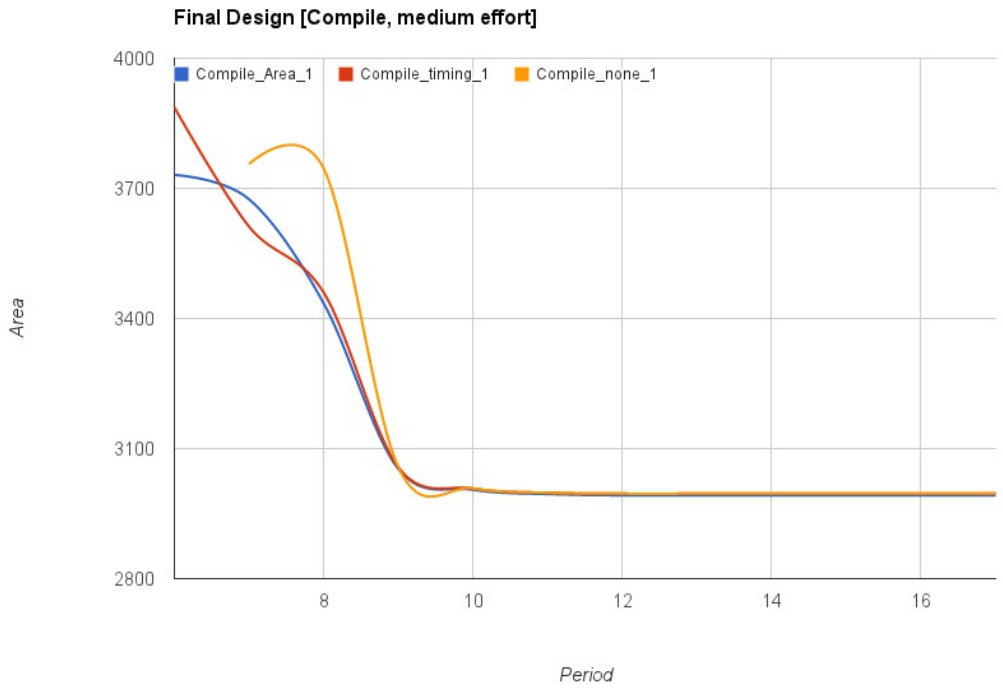
Rising edge of Clk1: 0
Falling edge of Clk1: 2.75

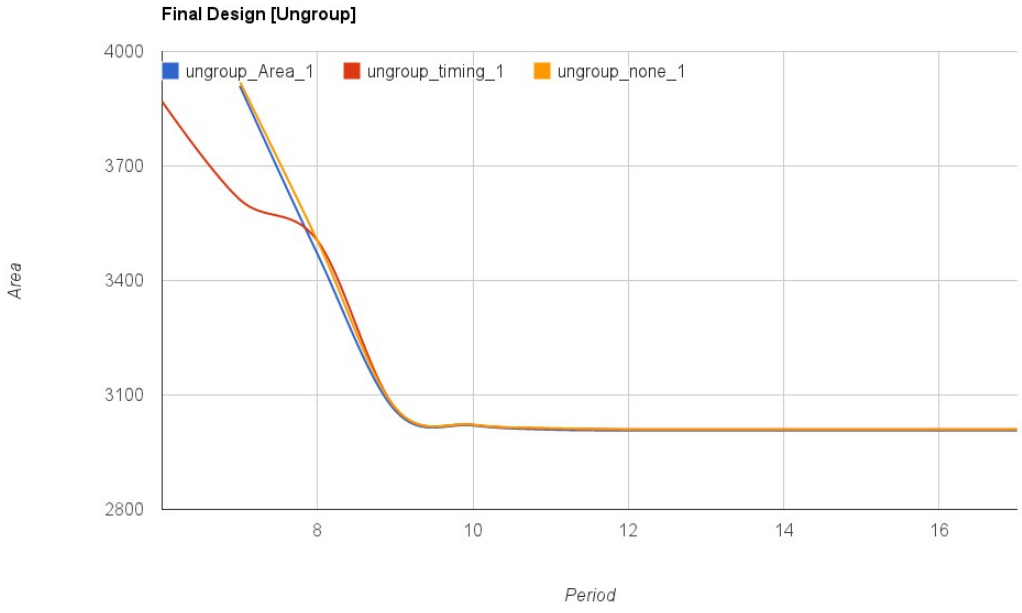
Rising edge of Clk2: 2.75
Falling edge of Clk2: 5.5

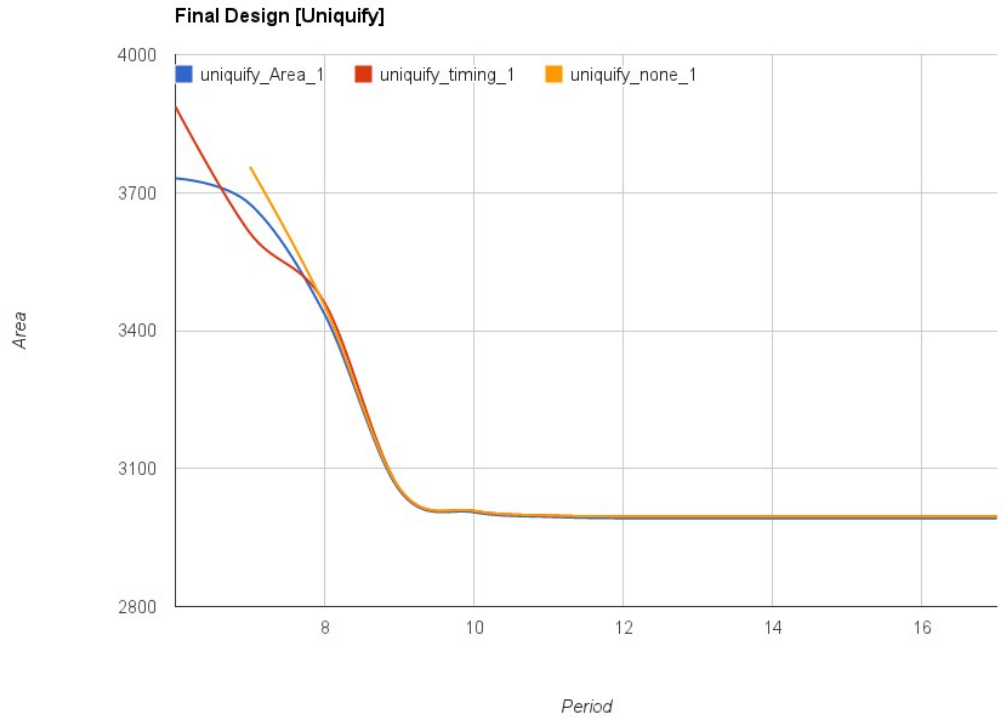
"k" Value: 1
(refer to test values on page 22)

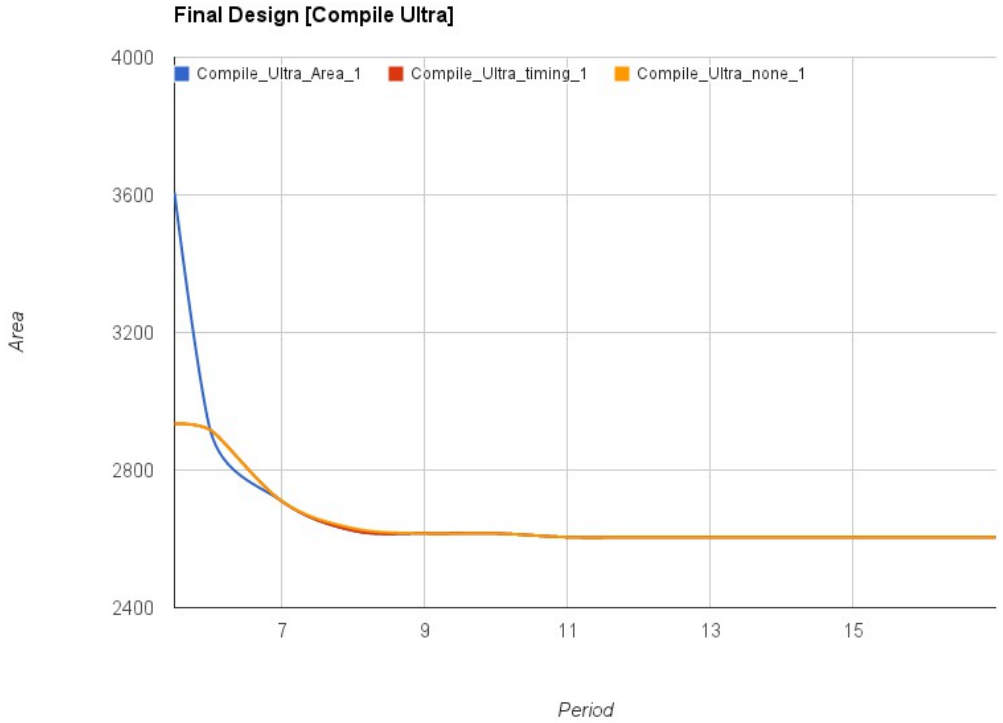
Throughput: 0.2 per cycle

So for optimal functioning, we might not want to chose minimum time period as it increases the area. We have selected a point from the graph based upon our judgment of what is optimal for both timing and area. However, if the user wants to optimize for a different purpose, he may play around with the clocks. We have included graphs to aid this purpose.


Final Design [Compile, medium effort]
Compile_Area_1
Compile_timing_1
Compile_none_1
4000
3700
3400
3100
2800
Area
8
10
12
14
16
Period


Final Design [Ungroup]
ungroup_Area_1
ungroup_timing_1
ungroup_none_1
4000
3700
3400
3100
2800
Area
8
10
12
14
16
Period


Final Design [Uniquify]
uniquify_Area_1
uniquify_timing_1
uniquify_none_1
4000
3700
3400
3100
2800
Area
8
10
12
14
16
Period


Final Design [Compile Ultra]
Compile_Ultra_Area_1
Compile_Ultra_timing_1
Compile_Ultra_none_1
4000
3600
3200
2800
2400
Area
7
9
11
13
15
Period

PIIR Functionality:

1) It must be configured with initial values. The user sets the CFG input to low to configure the PIIR and as long as the CFG input is low we are in configure mode.

2) The user sends in constants corresponding to which initial condition they wish to configure. These values are incorporated at the rising edge of Clk 2.

3) Once all values have configured the user sets CFG to high and run mode is entered.

4) At the next rising clock edge, we output the device address to the address bus and then are ready to read in a value from the user.

5) Once a value is read in, we take 5 clock cycles to compute the value for y. Once we have the value, then at the next available rising edge of clock 1 we send write to high.

6) After that, if CFG is still high we output the device address on the address bus once again and go into read mode at the next two rising edges of Clk 1.

7) In read mode, we update the values of previous y to the last y value we generated and update the x values as well based on the current input. Note: we do not swap if we are transitioning from configure to run mode as we do not have a y value ready to perform swap operations yet. We only update our previous x and y values if we are not coming directly from the configure mode.

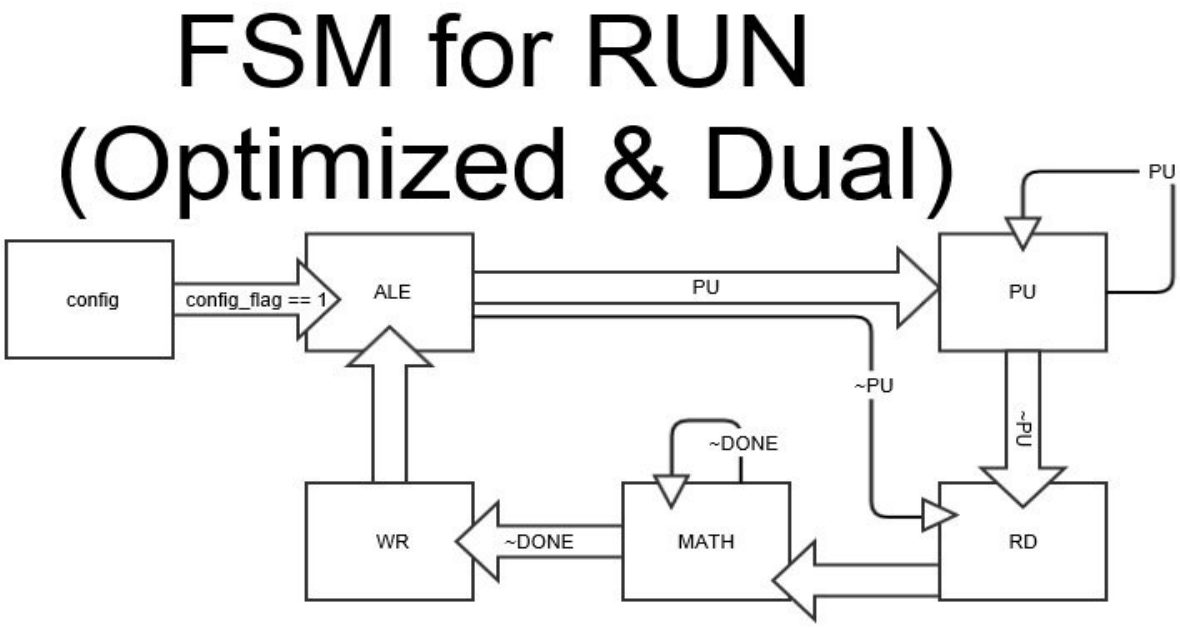
Aside: The code implementation of this behavior is as follows: a flag is set to true if we leave the configure state in our state machine diagram.

8) After the swap, it goes into math mode which gives a Y value in 5 clock cycles and we repeat the process all over again.

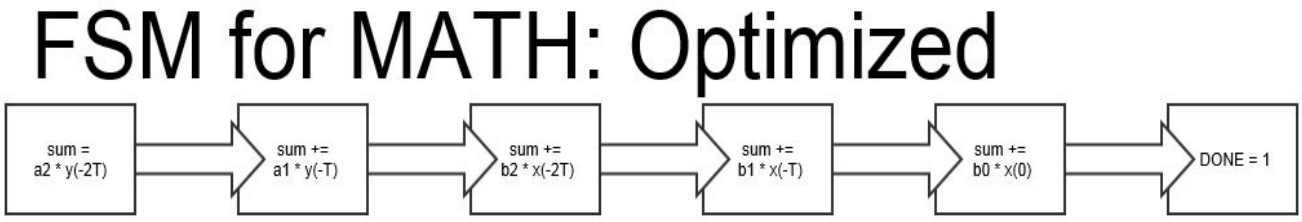
9) The user has the ability to pause the PIIR at the rising edge of Clk 2.

NOTE: The first output of the PIIR is the for the first input we provide the PIIR in Run Mode (see sample output below to get an idea of what this means).

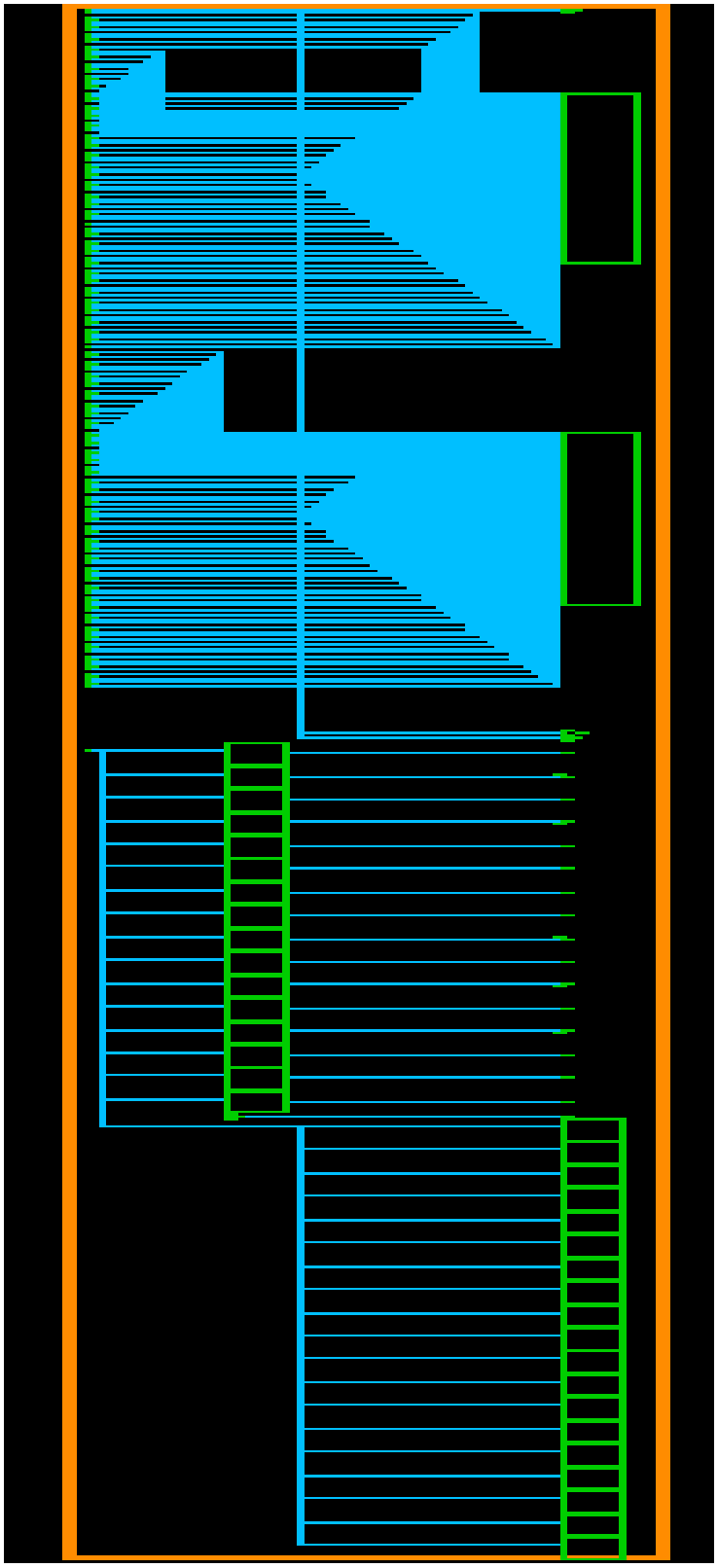
Picture of our Finite State Machine for Run mode:



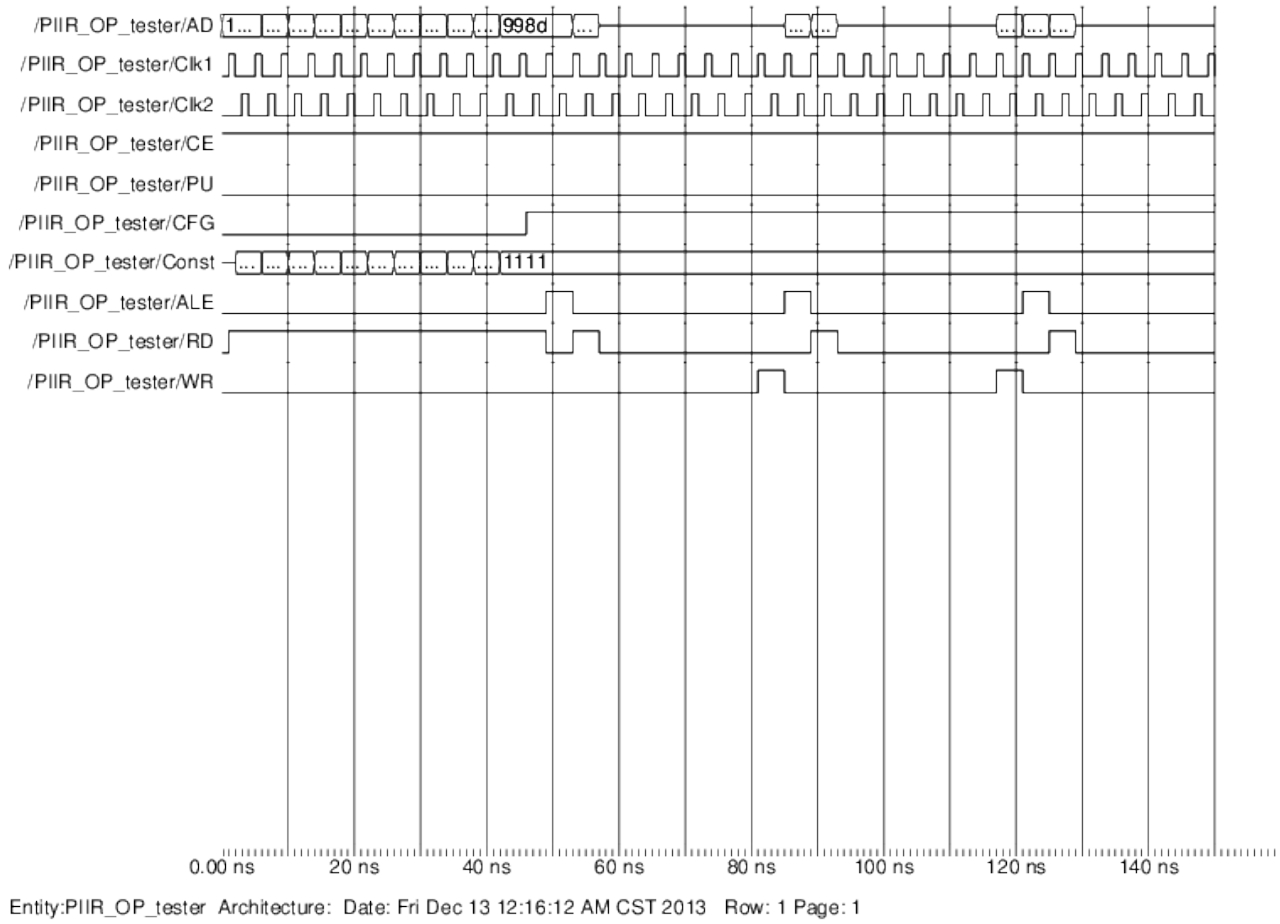
Picture of our Finite State Machine for Math mode: One multiplier & one adder



MAC (Multiply Accumulator) Schematic:



The following waveform is an example of how of read & write cycles operate in our final design. Note how we read in the device address, then read a value, then perform the necessary calculations over 5 clock cycles.



Critical Path Analysis:

Intuitively, the critical path of this design is a path starting from input (entering the multiplier) to an output that is generated at the register—a multiplication and addition operation. The timing report verified that this was the critical path.

Input combinations which would exercise the critical path to its maximum capability for adder and the multiplier are output values like 0xffff. For the adder, when the sum if 0xffff, it means that every single sum logic and carry logic had to implemented to get a result. In other words, data had to go through maximum combinational logic to get a valid output.

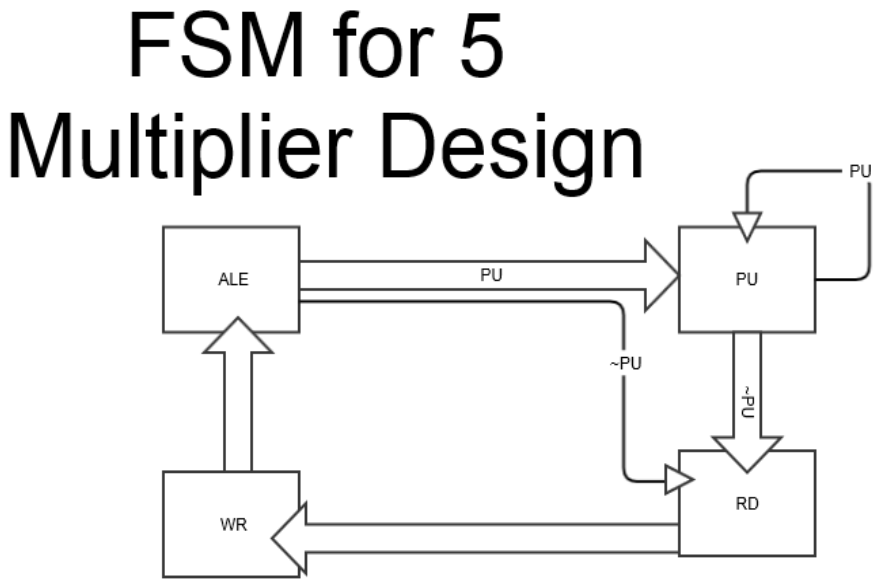
This is the bottleneck of the design and as the timing reports revealed, when the slack for this path is greater than or equal to zero, all other timing constraints are met.

Model Sim changes (writing more efficient code):

First Design:
In the intermediate milestone, we had identified our best design for a fixed-point adder and multiplier. All math operations in this design were performed by these modules.

Originally our group chose to design a PIIR Filter which used five multipliers and four adders. This design would take one cycle to read in the values and output values the next clock cycle. So it had very little latency and very high throughput.

This was the first functionally correct design according our test bench. Additionally, it had a very simple read-write cycle and gave an output immediately. The FSM logic for the design is as follows:



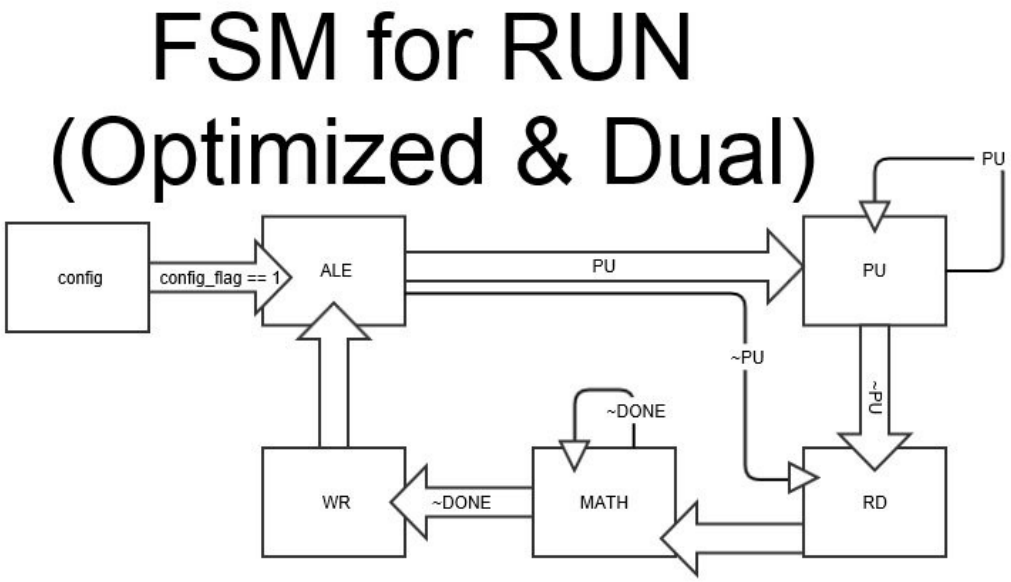
Unfortunately, this design occupied a significant amount of area.

Intermediate (Dual) design:
We then refined our design to only use two multipliers and two adders. This dropped the area of our design to half. But now, after reading in values, we needed 3 clock cycles to output a value. So this design takes more time to output a value compared to our previous design but the drop in area was very large. We got it to work functionally, but didn't investigate this design in great detail because we decided to implement the same idea with one multiplier and one adder.

Final Design:
Upon further design, we decided to extend our pipeline design to use just one multiplier and one adder. The trade off was now we get outputs over 5 clock cycles as compared to 3. This further reduced the area for our total design.

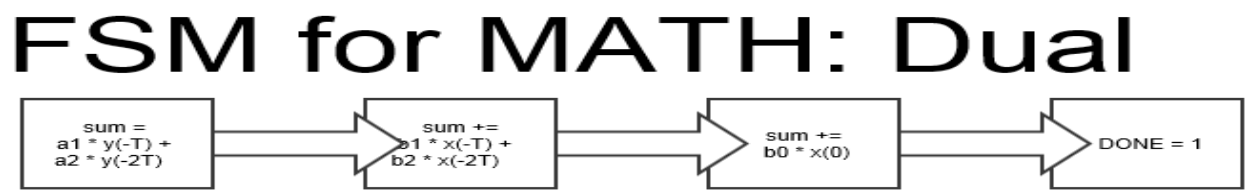
A complete description of this design is listed at the beginning of this report.

The Finite State Machine logic for the Intermediate (Dual) & Final design were as follows:

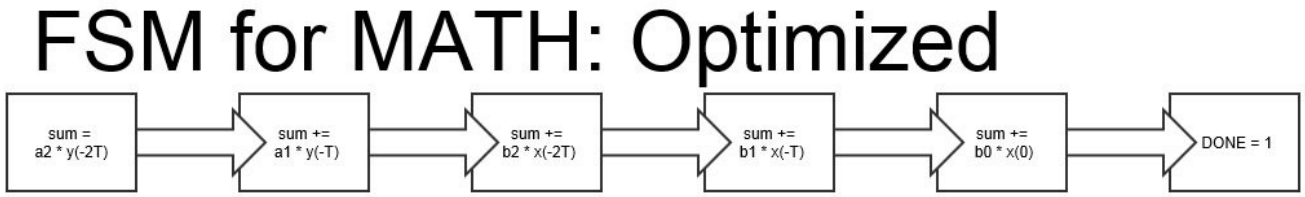


The difference between the two designs was in the Math state, which was it's own Finite State Machine. The 2 multiplier (Dual design) needed 3 clock cycles while our final version needed 5 clock cycles.

Picture of our Finite State Machine for Math mode in Dual design:



Picture of our Finite State Machine for Math mode in Final design:



Design Vision changes (letting the synthesis tool make our design better):

Once we had three different logical designs, we used our synthesis tool to further optimize them, performing multiple kinds of synthesis.

For each design we tried four different compile strategies:

1. Normal (map effort medium)
2. Uniquify
3. Ungroup
4. Ultra

Then each compile strategy at 3 subdivisions:

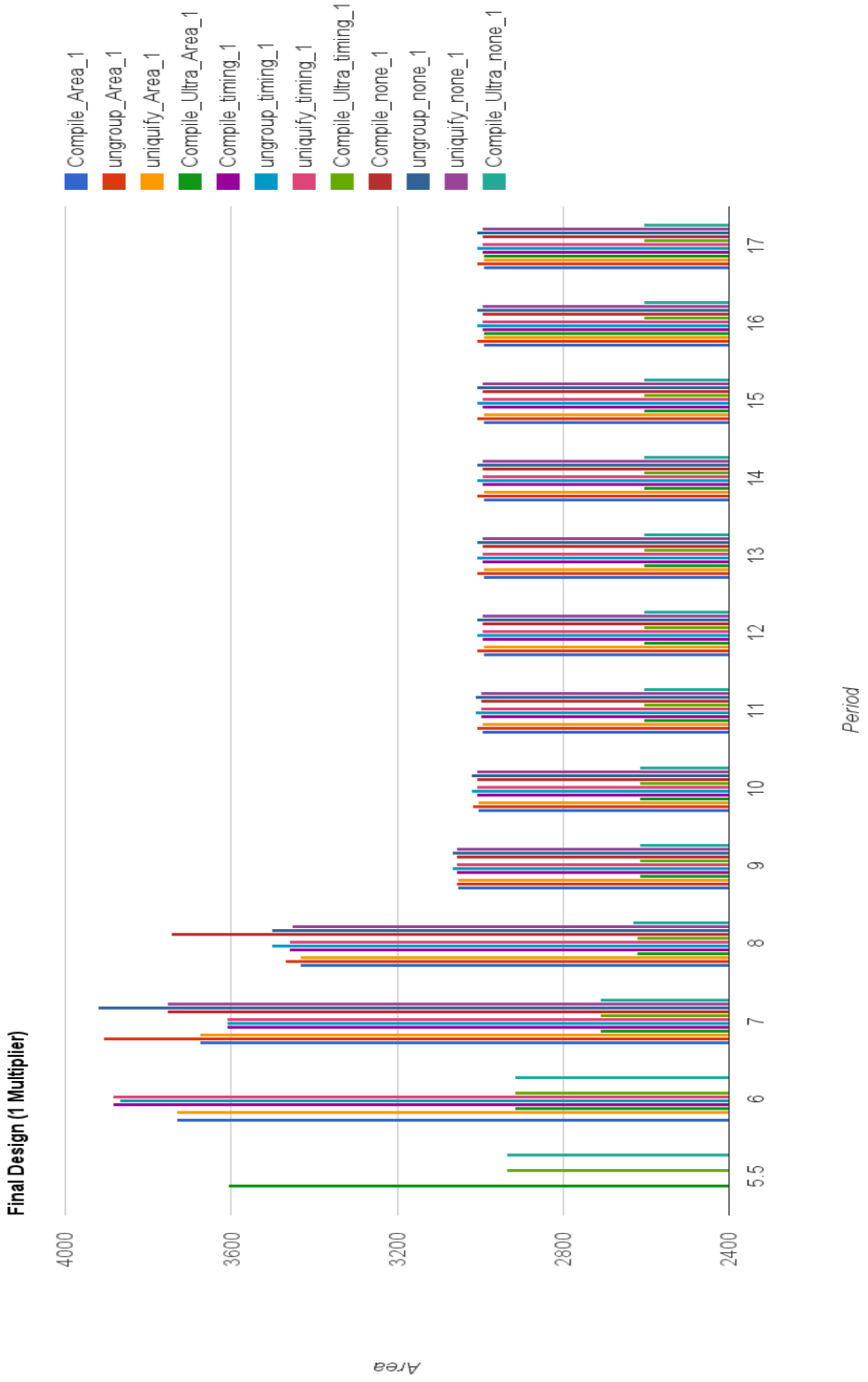
1. Minimize Area
2. Minimize Timing
3. No restrictions

To start out, we synthesized all our designs for a period of 10ns to give a general idea of what hardware would be produced and what area values we would get. We quickly discovered that our 2 multiplier design was inferior to what we wanted, so we didn't dive further into it. It was an intermediate of the other two designs.

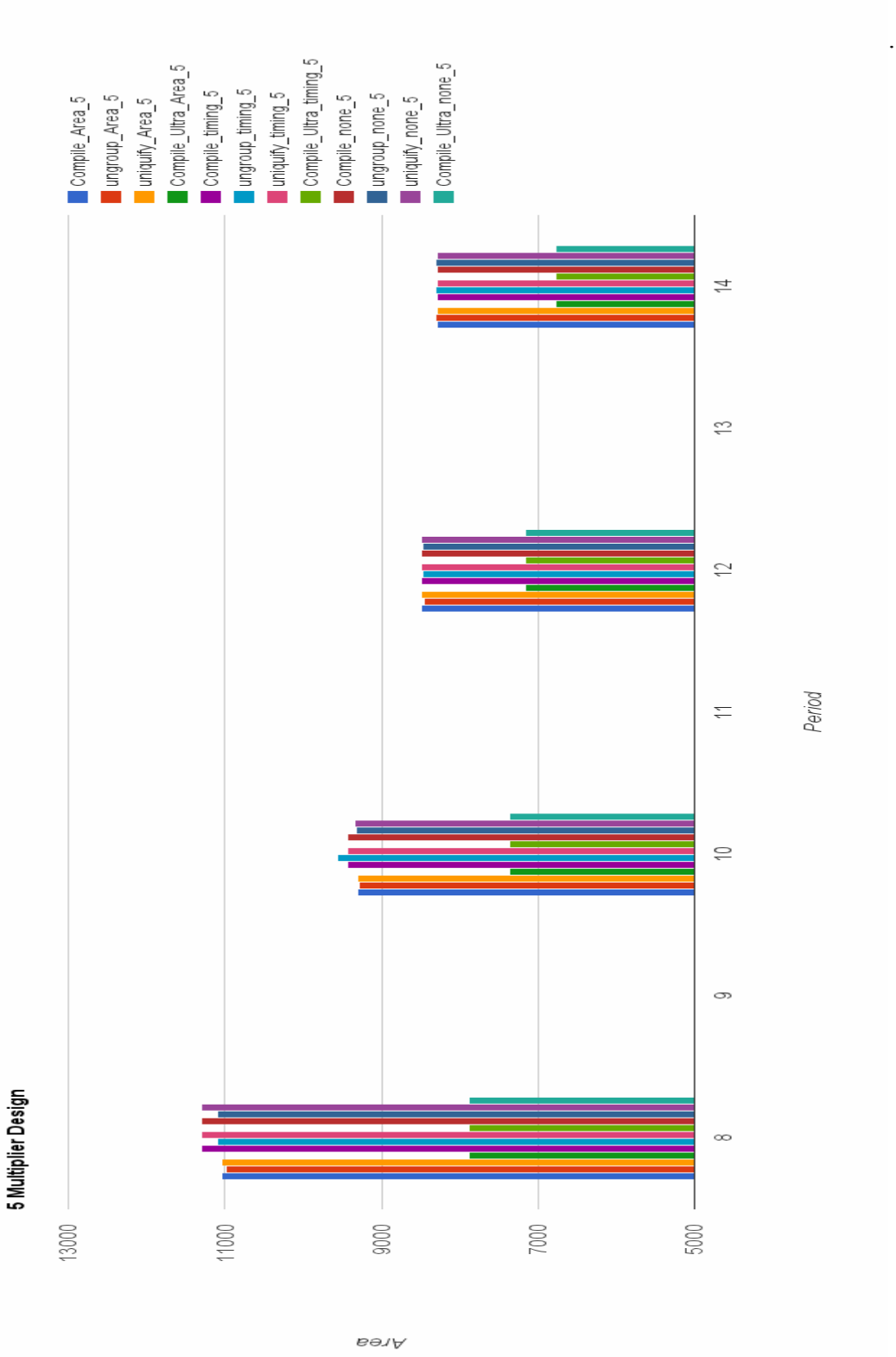
For our other two designs, we found the minimum clock they could synthesize at without giving us a negative slack. We then increased the period by 1 until the area started to plateau. Below is a plot of every single design we synthesized, categorized in terms of period, area, and type of synthesis.

Different synthesis options:

Area vs. Timing for Final design:

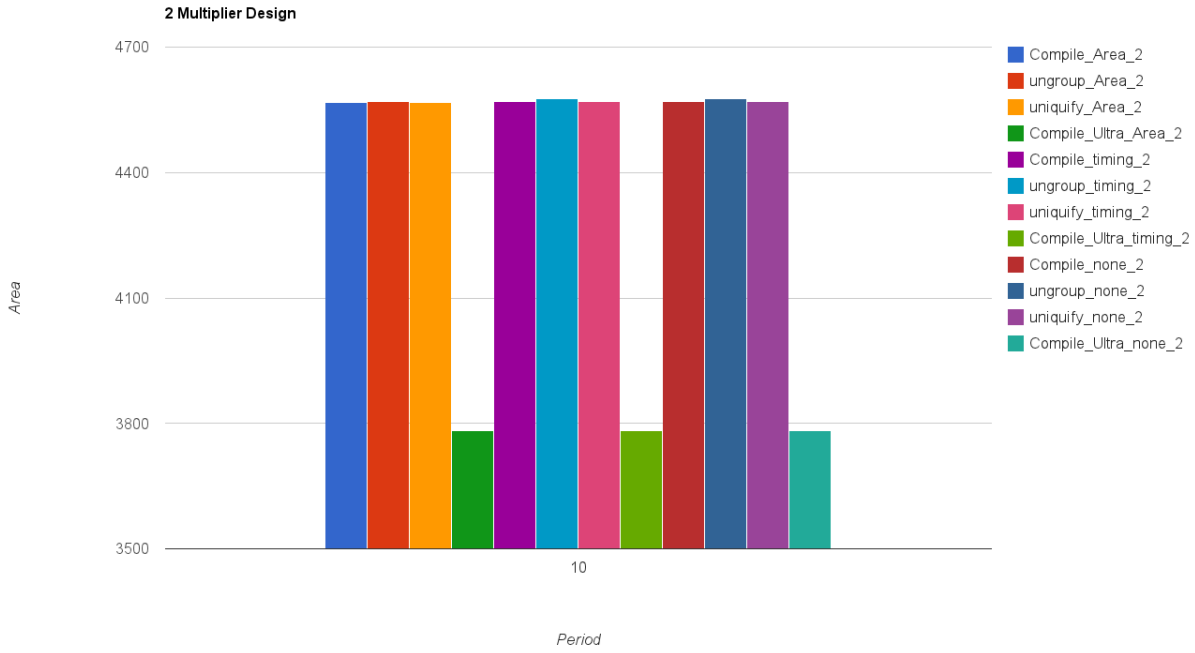


All synthesis options on Original design:

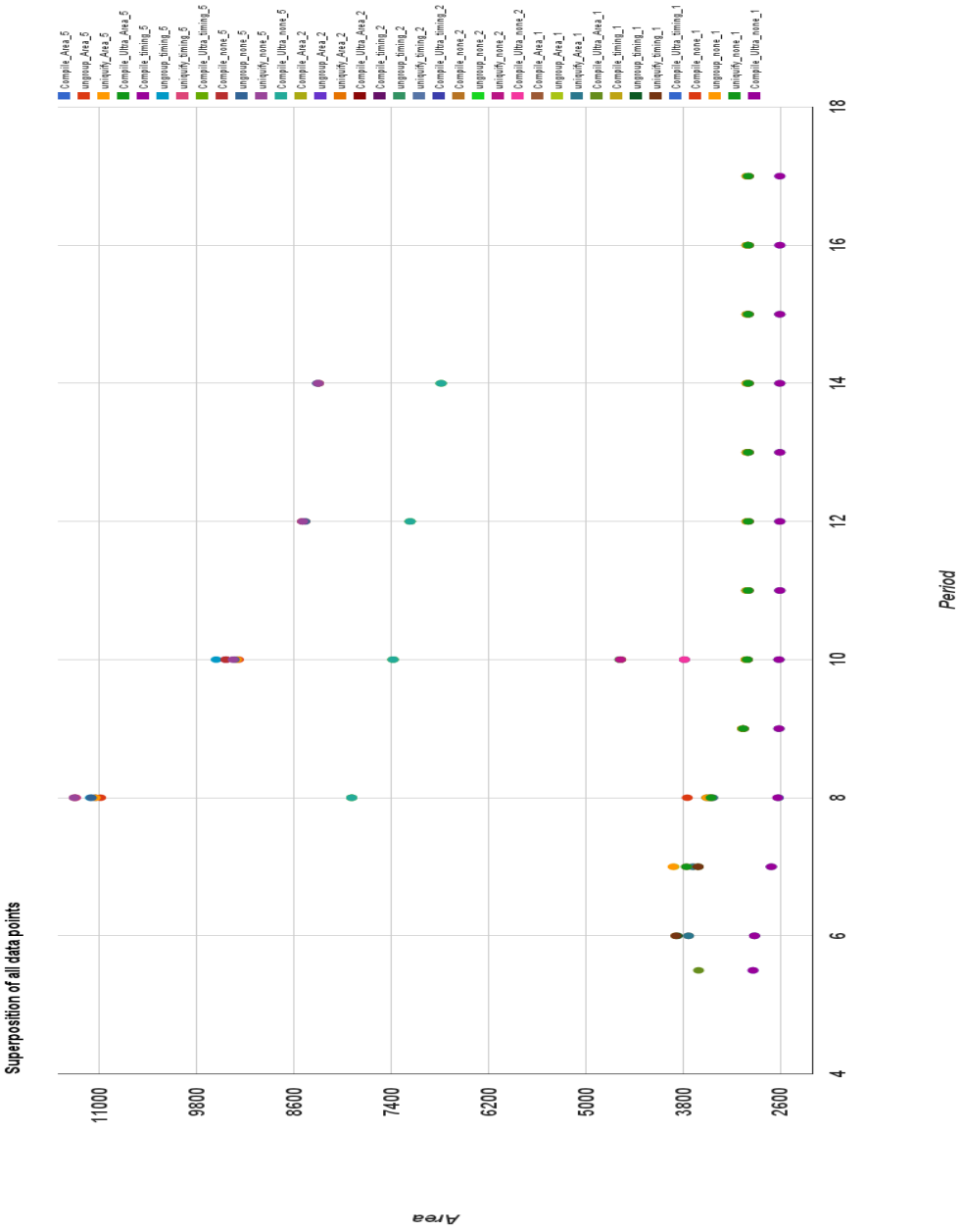


Synthesis for Dual (Intermediate) design just for period of 10ns:

Note: We didn't investigate this design further because we wanted to pick one of the other two as our best (more extreme) design.



Superposition of all synthesized designs:



General comments on Synthesis:

While doing synthesis, we discovered that often times, the best results were found by just letting the synthesis tool run without trying to constrain anything. It was in the constraints that it limited it's hardware options and lost area and time. The other thing we discovered, is that there is a period range that will synthesis with zero slack. In this range, decreasing the period increased the area and vice versa. If you pass the smaller timing edge, it becomes impossible to meet timing requirements. If you pass the larger timing edge, your area stays constant, but you gain more and more slack. Uniquify and Ungroup gave similar results. It was usually better then normal, but almost always lost to Ultra.

We picked the 9 good remaining designs from all of the above plots we did. The highlighted design is our best design.

Note: Latency = time taken between Read and Write.

Alternate Designs

Design #	Area	Throughput (# of new outputs/second)*10^{-9}	"k" Value	Latency *10^{-9} (s)
1	3053	0.11	1	5*9= 45
2	3059	0.11	1	5*9 = 45
3	2605	0.09	1	5*11= 45
4	2935	0.18	1	5*5.5 = 27.5
5	3612	0.14	1	5*7 = 35
6	3731	0.16	1	5*6= 30
7	2631	0.0125	1	5*8= 40
8	3782	0.0125	1	3*8 = 24
9	7884	0.0125	1	1*8= 10
10	7374	0.1	1	1*10=10

Notes

Design #	Type of synthesis
1	1 multiplier, Compile medium effort –minimizing area
2	1 multiplier, Ungroup – minimizing area
3	1 multiplier, Compile Ultra – no parameters targeted
4	1 multiplier, Compile Ultra – no parameters targeted
5	1 multiplier, Ungroup – minimize timing
6	1 multiplier, Uniquify- minimizing area
7	1 multiplier, Compile Ultra none
8	2 multiplier, Compile Ultra – minimize time
9	5 multiplier, Compile Ultra minimize time
10	5 multiplier Compile Ultra minimize time

The first module we tested was the basic math module for the adder and multiplier. We tried extreme values like 0x0000 + 0x000, 0xFFFF – 0x0000, etc.

It took us a while to realize the numbers we were dealing with were not 2's complement numbers and were signed magnitude numbers (to negate a number - just flip sign bit). So that caused us to change the structure of our fixed-point adder and multiplier slightly from what we had before. We added extra logic to handle subtractions.

Logic: While subtracting, we check which number is greater and then decide the value of the sign bit based on that, and do regular subtraction on the magnitude of the number. Bits[14:0]

After we established the math was working, we moved on to the configure mode. By using the wave form for Model Sim, we followed the data paths of our inputs to make sure they were getting assigned in the correct location. We ran into difficulties with getting it to sync exactly how we wanted with the clock, but eventually fixed it. Following the confirmation of the functionality of the configure module, we moved onto the state machine for the Run mode as described above.

The state machine appeared to work correctly right from the beginning, but when we combined it with our configure mode logic, we were having issues with it shifting the values at the correct time.

We update our values in the read state and coming out of configure, it would update previous y to y even before we had a y output. To resolve this issue, we had to put in a the "configure flag", which was set to 1 we came out of configure mode.

We had to change sensitivity lists many, many times to finally get the logic to work just right. We had accidentally based the next state logic on Clk 1, and our states were shifting on two clock cycles. First, the next state would change then the state register would change.

Once we had it working in pre-synthesis mode, we ran synthesis on the design and realized we had three to four latches in our design.

For our configure_flag logic, we had config_flag = config-flag and quickly realized that this assignment was in a combinational always bock, so it would create latches We replaced it with 2 to 1 mux which assigned 1 or 0 to the flag based on its value.

We then realized we didn't have a default case for our state transition logic which also caused latches, and in our Math module, we were assigning a reg to itself in a combinational block.

Through the project we realized that any assignment of the form:

//a is a reg

a = a //gives latches

a<=a // changing at pos clock edge does not give latches.

After we fixed all these minor bugs:

We used 2 different test benches for checking our design. We had a simple one that was designed to test pause specifically. Our main test bench was simple but effective. It generated 2 different clocks

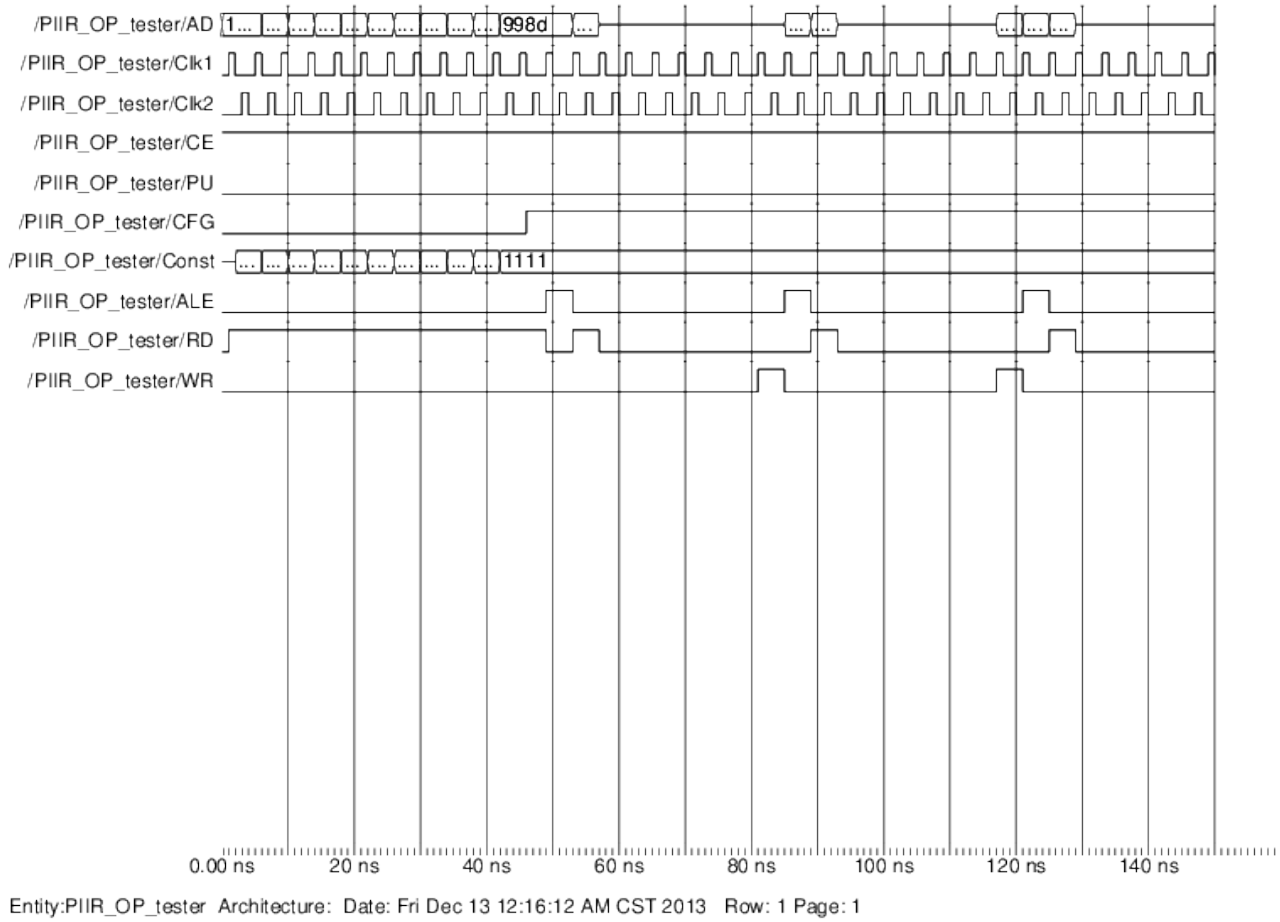
out of phase with the same timing. It tested our configure in the very beginning by setting up some specific values. It was useful to use specific values because tracing the paths becomes much easier. It then went through an array of values submitting a new one every time RD was high. The values we used were hand picked and then copied out of your input list.

Performance results for Instructor-specified Test Vectors (34 values)

```
# A1 = 1000 (0.125000)
# A2 = 9000 (-0.125000)
# Y-1 = 3524 (0.415161)
# Y-2 = 5e81 (0.738312)
# B0 = 9000 (-0.125000)
# B1 = 1000 (0.125000)
# B2 = 9000 (-0.125000)
# X-0 = d609 (-0.672150)
# X-1 = 5663 (0.674896)
# X-2 = 7b0d (0.961334)
# ADDR = 998d (-0.199615)

# k =1
# X_h   Y_h     My values
# 7b0d  x          x
# 5663  x          x
# d609  x           x
# d609  0b57       x
# 8465  8fc7    0b58
# 5212  0356    8fc8
# e301  14cb    0357
# cd0d  8f2f    14cb
# f176  156c    8f30
# cd3d  0087    156c
# 57ed  83d9    0087
# f78c  2422    8389
# e9f9  91af    2420
# 24c6  03d3    91ae
# 84c5  0fc0    03d3
# d2aa  03a6    0fc0
# f7e5  06c3    03a7
# 7277  9359    06c3
# d612  2b51    9539
# db8f  9574    2b4f
# 69f2  85d5    9573
# 96ce  1996    85d6
# 7ae8  a362    1994
# 4ec5  0ffd    a360
# 495c  951b    0ffc
# 28bd  8121    951b
# 582d  9198    8122
# 2665  0330    9128
# 6263  951e    0331
# 870a  0b6a    951e
# 2280  958e    0b6a
# 2120  052c    958d
# 45aa  8c38    052b
# cc9d  1051    8c37
# 3e96  9dac    104f
# b813  1e28    9da9
# 380d  9d51    1e25
# d653  203d    9d4e
                203a
```

Sample Test bench Waveform



Portion of Waveform Showing how PIIR works. This is for the optimal design.

Team Member Contributions

In working on this project, our group attempted to partition the work as fairly as possible. The individual contributions of each team member are as follows:

Bob Wagner:

- ➔ Initial planning of design & modules
  - ➔ Figured how PIIR Filters work
  - ➔ Constructed high-level block diagram for the module connections, see Figure 1
  - ➔ Wrote preliminary module headers (inputs, outputs, etc.)
- ➔ Testing of adder & multiplier designs before submitting for Intermediate Report
- ➔ Wrote intermediate report
- ➔ Wrote the adders and multipliers that ended up being less optimal than the ones included in final design
- ➔ Module writing
- ➔ Synthesis Testing of final design
- ➔ Commented final design code
- ➔ Wrote final report
  - ➔ Includes creating the various figures used in the report

Ari Biswas:

- ➔ Module writing
- ➔ Pipelining of our design
- ➔ Multiply Accumulator design
- ➔ Testing/Debug
- ➔ Improved design to 1 multiplication unit
- ➔ Helped in commenting code
- ➔ Wrote final report

Jordan Friendshuh:

- ➔ Initial planning of design & modules
  - ➔ Figured how PIIR Filters work
  - ➔ Constructed high-level block diagram for the module connections, see Figure 1
  - ➔ Wrote preliminary module headers (inputs, outputs, etc.)
- ➔ Multiply Accumulator design
- ➔ Testing/Debug
- ➔ Improved design to 2 multiplication unit
- ➔ Synthesis

Bob Wagner	*Ari Biswas*	*Jordan Friendshuh*
Bob Wagner	Ari Biswas	Jordan Friendshuh

Statement of Design Functionality

To the best of our knowledge, the design is fully functional.