



ECSE 325 Take Home Lab Report

Ishraq Hossain 260720764

Glen Xu 260767363

Introduction

The goal of this lab is to become familiar with the concept of pipelining in designing digital systems. This was done by using different implementations of a complex multiplier, a system that computes the square of a complex number, and analyzing their performance metrics.

Implementation

We were first required to implement a simple complex multiplier. The architecture of this was provided to us to aid our understanding of the system which proved to be extremely helpful in implementing the later stages.

The operation is detailed below:

$$Z^2 = (X+iY)*(X+iY) = (X^2-Y^2)+i(2XY)$$

The implementation is attached below. It can be seen that the process is carried out in one clock cycle.

```
4
5 entity complex_square is
6 port (
7     i_clk      : in std_logic;
8     i_rstb     : in std_logic;
9     i_x        : in std_logic_vector(31 downto 0);
10    i_y        : in std_logic_vector(31 downto 0);
11    o_xx, o_yy  : out std_logic_vector(64 downto 0));
12 end complex_square;
13
14 architecture rtl of complex_square is
15     signal r_x, r_y : signed (31 downto 0);
16
17     begin
18     p_mult : process(i_clk, i_rstb)
19     begin
20         if (i_rstb = '0') then
21             o_xx <= (others => '0');
22             o_yy <= (others => '0');
23             r_x <= (others => '0');
24             r_y <= (others => '0');
25         elsif(rising_edge(i_clk)) then
26             r_x <= signed(i_x);
27             r_y <= signed(i_y);
28             o_xx <= std_logic_vector(('0' & (r_x*r_x)) - r_y*r_y);
29             o_yy <= std_logic_vector(r_x*r_y & '0');
30         end if;
31     end process p_mult;
32 end rtl;
```

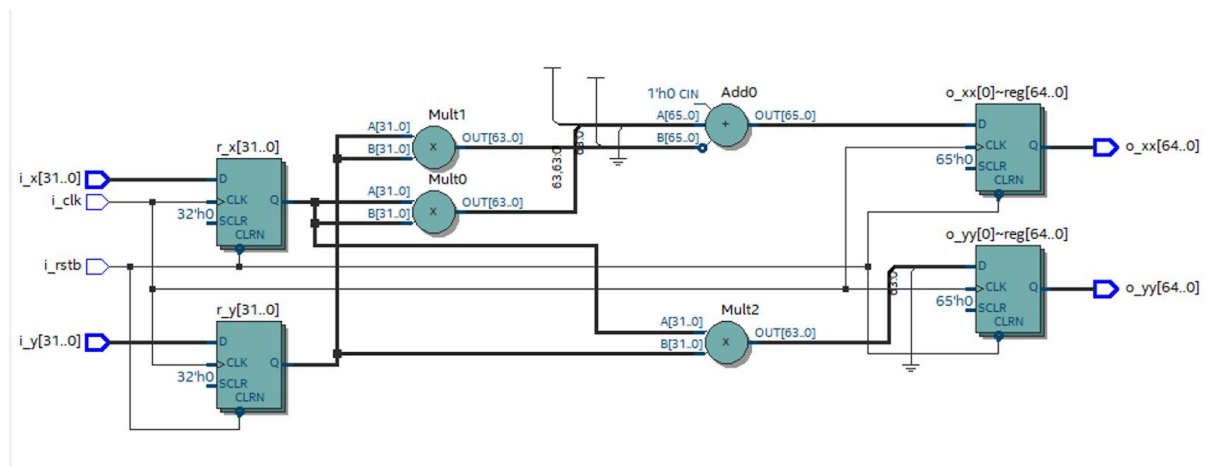
Compilation report

The compilation report for the simple complex multiplier is attached below highlighting the resources used:

Flow Status	Successful - Thu Apr 23 12:49:57 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	complex_square
Top-level Entity Name	complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	103 / 32,070 (< 1 %)
Total registers	65
Total pins	196 / 457 (43 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	9 / 87 (10 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)

We can see that the total number of registers used in our implementation is 65 and the number of logic modules are 103.

The RTL view of our implementation is attached below.



Timing Analysis

We were then required to create a timing constraint of 5ns which was equivalent to a 200MHz. TimeQuest was used to conduct timing analysis and the timing summary is attached below for the Slow 1100mV Slow 0C model.

	Fmax	Restricted Fmax	Clock Name	Note
1	110.83 MHz	110.83 MHz	i_clk	

	Clock	Slack	End Point TNS
1	i_clk	-4.023	-168.258

As we can see, the design does not pass the timing test as the frequency of the clock is 110.83 MHz. So the time period comes to around 9 ns. We also noticed that in the setup the slack that there were failing paths. The next section highlights our attempt to pipeline the design using a register.

Pipeline Register

We were required to implement the complex multiplier again, but this time with an intermediary pipeline register between the multiplication and subtraction operation. The implementation of this is attached below:

```
14 architecture rtl of complex_square is
15   signal r_x, r_y : signed(31 downto 0);
16   signal p_reg_x : signed(64 downto 0);
17   signal p_reg_y : signed(63 downto 0);
18 begin
19   p_mult : process(i_clk, i_rstb)
20   begin
21     if (i_rstb = '0') then
22       o_xx <= (others => '0');
23       o_yy <= (others => '0');
24       r_x <= (others => '0');
25       r_y <= (others => '0');
26       p_reg_x <= (others => '0');
27       p_reg_y <= (others => '0');
28     elsif(rising_edge(i_clk)) then
29       r_x <= signed(i_x);
30       r_y <= signed(i_y);
31       p_reg_x <= ('0' & (r_x*r_x));
32       p_reg_y <= (r_y * r_y);
33       o_xx <= std_logic_vector(p_reg_x - p_reg_y);
34       o_yy <= std_logic_vector(r_x*r_y & '0');
35     end if;
36   end process p_mult;
37 end rtl;
```

Our design choice was fairly straightforward and based on the previous implementation. We created two intermediate signals called p_reg_x, and p_reg_y. These signals were implemented to act as pipeline registers and hold the value after the multiplication operation. The values of these registers were then used to carry out the subtraction operation and passed onto the output signal.

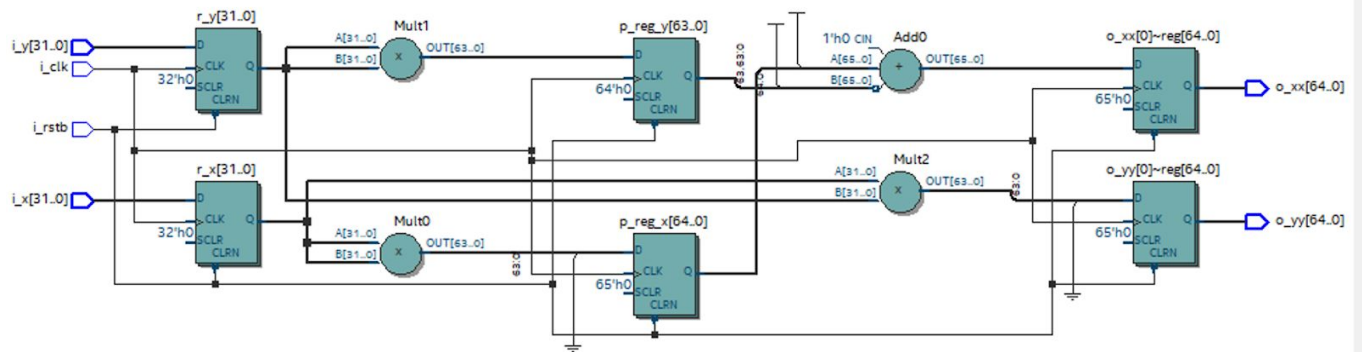
Compilation report

The compilation report for the pipelined complex multiplier is attached below highlighting the resources used.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Thu Apr 23 16:45:14 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	complex_square
Top-level Entity Name	complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	103 / 32,070 (< 1 %)
Total registers	65
Total pins	196 / 457 (43 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	9 / 87 (10 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)

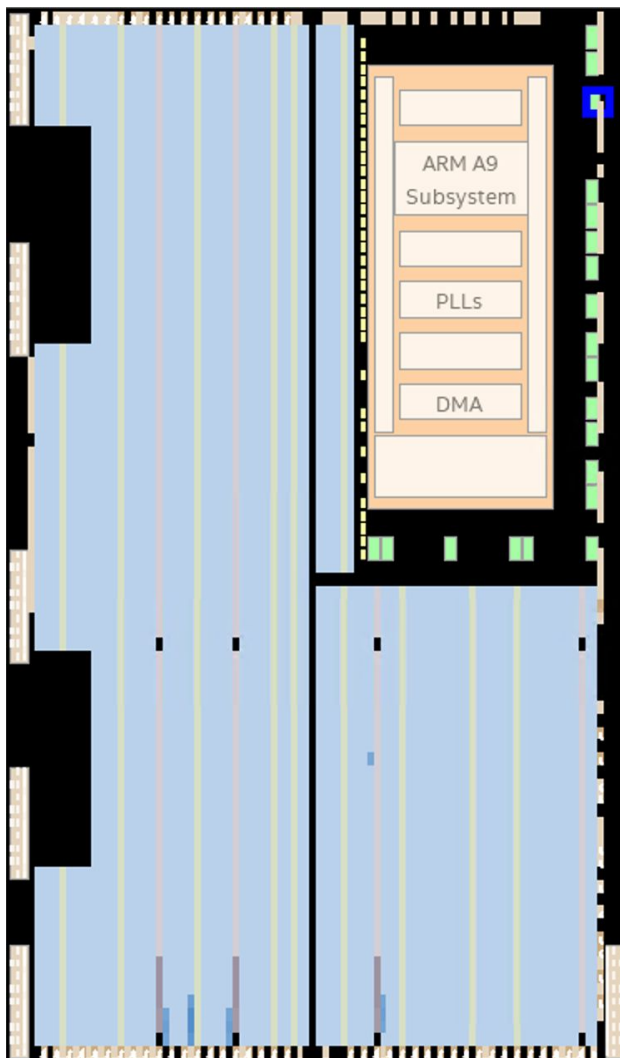
As highlighted above the resources used did not change for when compared to the previous implementation. The number of registers and logic modules were still 65 and 103 respectively. One thing to note is that we observed a significant increase in resources during compilation. The register count went up to 420. This increase was not found in the final summary however. However, the significant improvement in our design was found when we conducted the timing analysis. Which will be detailed in the respective section.

The RTL view of our implementation is attached below.




As expected, the two pipeline registers are present in between our operations.


The resource utilization of our circuit is attached below:



Timing Analysis

We were then required to use the same timing constraint of 5ns and conduct timing analysis. The timing summary is attached below:

Slow 1100mV 0C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	143.99 MHz	143.99 MHz	i_clk	

Slow 1100mV 0C Model Setup Summary			
 <<Filter>>			
	Clock	Slack	End Point TNS
1	i_clk	-1.945	-53.761

We could observe the improvement in the timing analysis because the maximum frequency of the clock was higher. This indicated we were moving closer to our target frequency of 200MHz. By adding a pipeline register we could see our design improved in performance by 2ns. We also noticed that the margin of slack was also lower than in our previous design. We were then required to implement a further pipelined design of the system by implementing the LPM module.

LPM Module


We were then required to further pipeline our design by implementing LPM components to carry out the multiplication operations. The implementation of which is attached below:

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.all;
3  USE IEEE.numeric_std.all;
4  LIBRARY lpm;
5  USE lpm.lpm_components.all;
6
7  entity g14lpm is
8  port (
9      i_clk    : in std_logic;
10     i_rstb   : in std_logic;
11     i_x      : in std_logic_vector(31 downto 0);
12     i_y      : in std_logic_vector(31 downto 0);
13     o_xx, o_yy, o_xy : out std_logic_vector(64 downto 0)
14 );
15 end g14lpm;
16
17 architecture arc of g14lpm is
18     signal r_x : signed(31 downto 0);
19     signal r_y : signed(31 downto 0);
20     signal xx : std_logic_vector(63 downto 0);
21     signal yy : std_logic_vector(63 downto 0);
22     signal xy : std_logic_vector(63 downto 0);
23
24     ----- COMPONENT DECLARATION -----
25     component LPM_MULT
26     generic ( LPM_WIDTHA : natural;
27             LPM_WIDTHB : natural;
28             LPM_WIDTHHP : natural;
29             LPM_REPRESENTATION : string := "SIGNED";
30             LPM_PIPELINE : natural := 0;
31             LPM_TYPE : string := L_MULT;
32             LPM_HINT : string := "UNUSED");
33     port ( DATAA : in std_logic_vector(LPM_WIDTHA-1 downto 0);
34           DATAB : in std_logic_vector(LPM_WIDTHB-1 downto 0);
35           ACLR : in std_logic := '0';
36           CLOCK : in std_logic := '0';
37           CLKEN : in std_logic := '1';
38           RESULT : out std_logic_vector(LPM_WIDTHHP-1 downto 0));
39     end component;
40
41     begin
42         ----- COMPONENT INSTANTIATION -----
43         mult1 : LPM_MULT generic map (
44             LPM_WIDTHA => 32,
45             LPM_WIDTHB => 32,
46             LPM_WIDTHHP => 64,
47             LPM_REPRESENTATION => "SIGNED",
48             LPM_PIPELINE => 4
49         )
50         port map ( DATAA => i_x, DATAB => i_x, CLOCK => i_clk, RESULT => xx );
51         mult2 : LPM_MULT generic map (
52             LPM_WIDTHA => 32,
53             LPM_WIDTHB => 32,
54             LPM_WIDTHHP => 64,
55             LPM_REPRESENTATION => "SIGNED",
56             LPM_PIPELINE => 4
57         )
58         port map ( DATAA => i_y, DATAB => i_y, CLOCK => i_clk, RESULT => yy );
59         mult3 : LPM_MULT generic map (
60             LPM_WIDTHA => 32,
61             LPM_WIDTHB => 32,
62             LPM_WIDTHHP => 64,
63             LPM_REPRESENTATION => "SIGNED",
64             LPM_PIPELINE => 4
65         )
66         port map ( DATAA => i_x, DATAB => i_y, CLOCK => i_clk, RESULT => xy );
67
68         p_mult : process (i_clk, i_rstb)
69         begin
70             if (i_rstb = '0') then
71                 o_xx <= (others => '0');
72                 o_yy <= (others => '0');
73                 o_xy <= (others => '0');
74                 r_x <= (others => '0');
75                 r_y <= (others => '0');
76             elsif (rising_edge(i_clk)) then
77                 r_x <= signed(i_x);
78                 r_y <= signed(i_y);
79
80                 o_xx <= std_logic_vector ('0' & signed(xx) - signed(yy));
81                 o_yy <= std_logic_vector (r_x*r_y & '0');
82
83             end if;
84         end process p_mult;
85
86     end arc;
87
```


To implement the design we first had to make use of the LPM library which allowed us to use the LPM_MULT component. We were then able to declare the component with the respective sizes of our inputs, outputs and number of pipelines. We then implemented our components to carry out the multiplication operations. We tested our design with 3 different levels of pipelining. We specified the number of pipelines as 2, 3, and 4. The VHDL provided is the one where there were 4 levels of pipelining. Our different implementations are detailed below along with a summary of all our designs at the end.

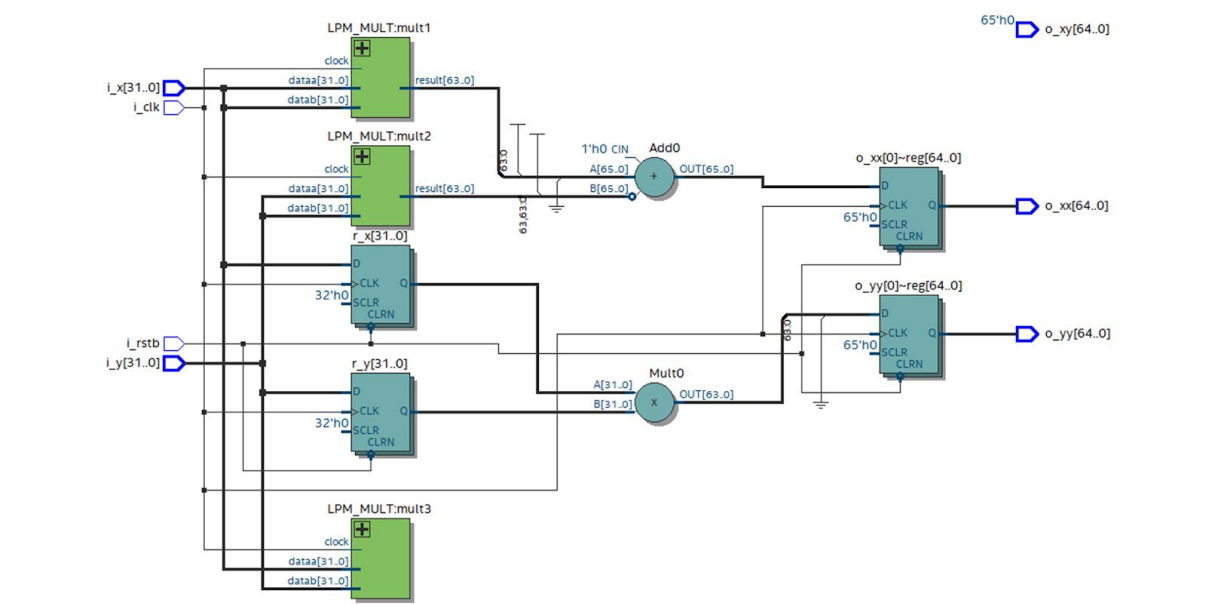
LPM Pipeline = 2

The compilation report after using 2 levels of pipeline is attached below:

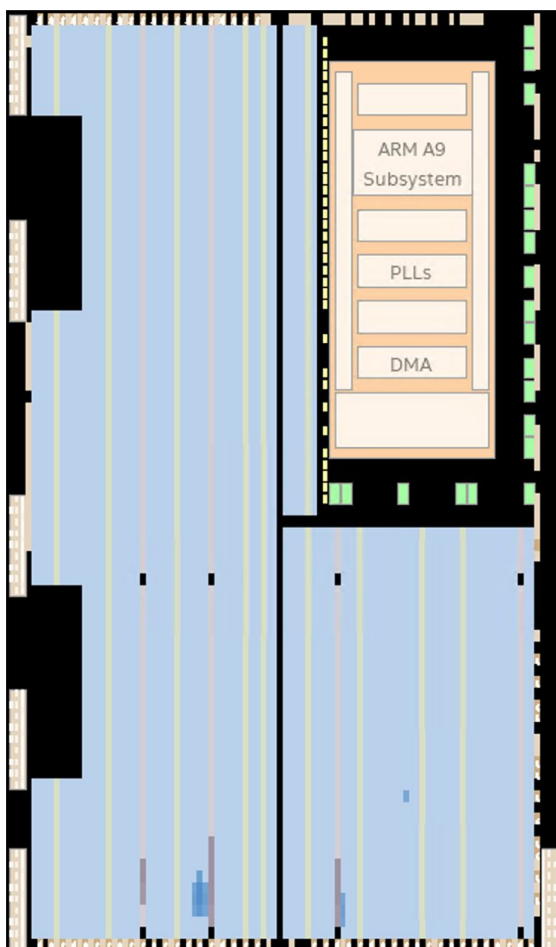
Flow Summary	
 <<Filter>>	
Flow Status	Successful - Fri Apr 24 16:02:23 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G14lpm
Top-level Entity Name	g14lpm
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	103 / 32,070 (< 1 %)
Total registers	65
Total pins	261 / 457 (57 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	9 / 87 (10 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Similar to our compilation of the pipelining with an intermediate register, we noticed an increase in resource utilization during compilation. This showed us a register increase to a total of 484 registers. However, this increase in register use was not reflected in the final summary.

The RTL view of our system is attached below:



The resource utilization of this digital circuit is attached below:



Timing Analysis

We were then required to use the same timing constraint of 5ns and conduct timing analysis. The timing summary is attached below:

Slow 1100mV OC Model Setup Summary			
<<Filter>>			
	Clock	Slack	End Point TNS
1	i_clk	-6.282	-773.283

Slow 1100mV OC Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	137.32 MHz	137.32 MHz	i_clk	

As we can see, using just two pipelines did not improve our design compared to our previous pipelined design.

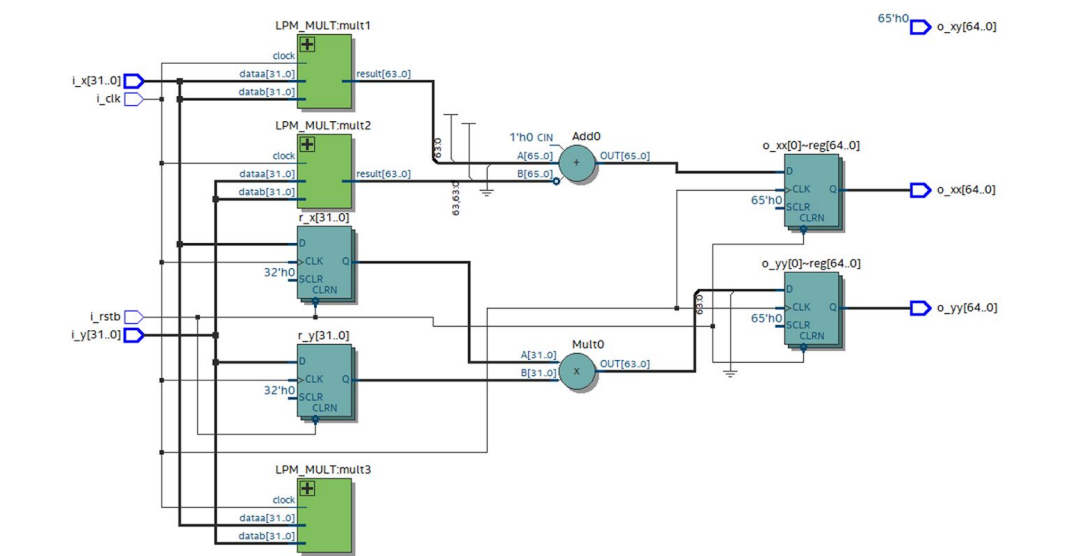
LPM Pipeline = 3

The compilation report after using 3 levels of pipeline is attached below:

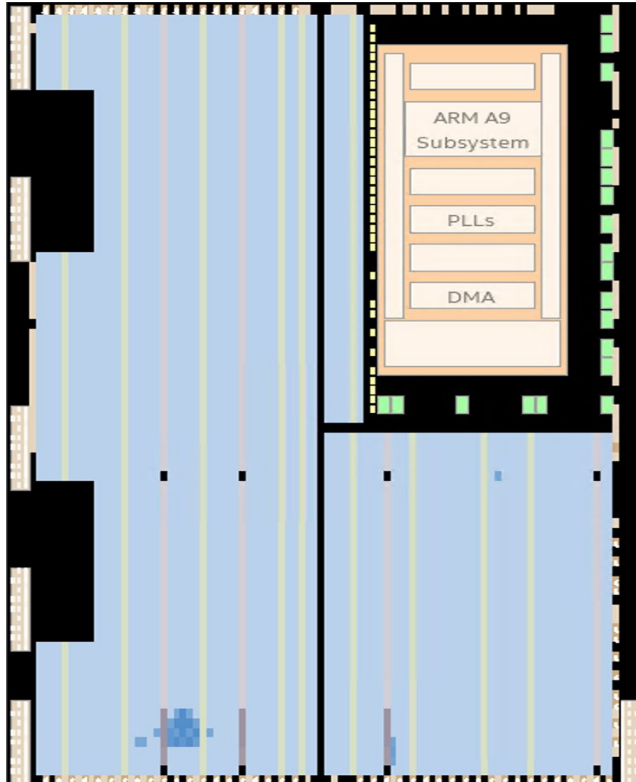
Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri Apr 24 14:34:31 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G14lpm
Top-level Entity Name	g14lpm
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	130 / 32,070 (< 1 %)
Total registers	259
Total pins	261 / 457 (57 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	9 / 87 (10 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

As we can see by increasing the level of the pipeline to 3 we have increased the amount of resources required. This time we can see an increase in the resources used in the final summary. Even though the register count mid compilation was higher. We can see for 3 levels of pipelining we're using 130 logic modules and 259 registers.

The RTL view of our system is attached below:




The resource utilization of the system is attached below:




By observation of the chip planner, the implementation with 3 levels of pipelining is using significantly more resources when compared to the previous implementations of pipelining.

Timing Analysis

We were then required to use the same timing constraint of 5ns and conduct timing analysis. The timing summary is attached below:

Slow 1100mV OC Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	192.53 MHz	192.53 MHz	i_clk	

Slow 1100mV OC Model Setup Summary			
 <<Filter>>			
	Clock	Slack	End Point TNS
1	i_clk	-4.194	-975.324

We can see that by increasing the level of pipelining to 3 we are now very close to meeting our target of 5ns. This target is expected to be met when we increase the level of pipelining to 4 in the next section

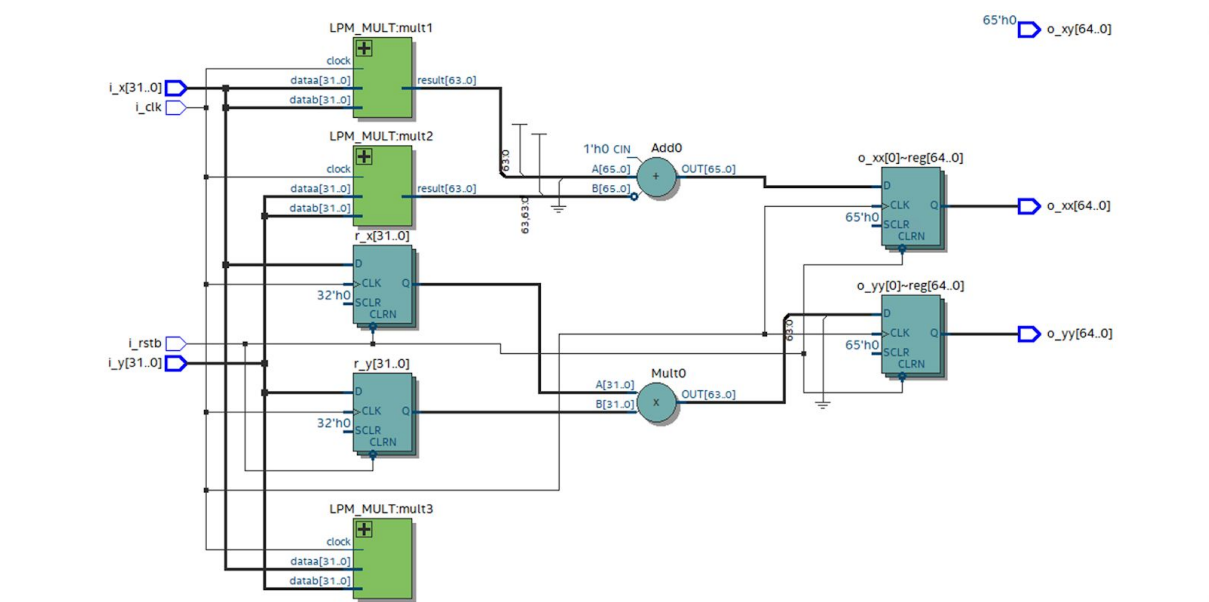
LPM Pipeline = 4

The compilation report after using 4 levels of pipeline is attached below:

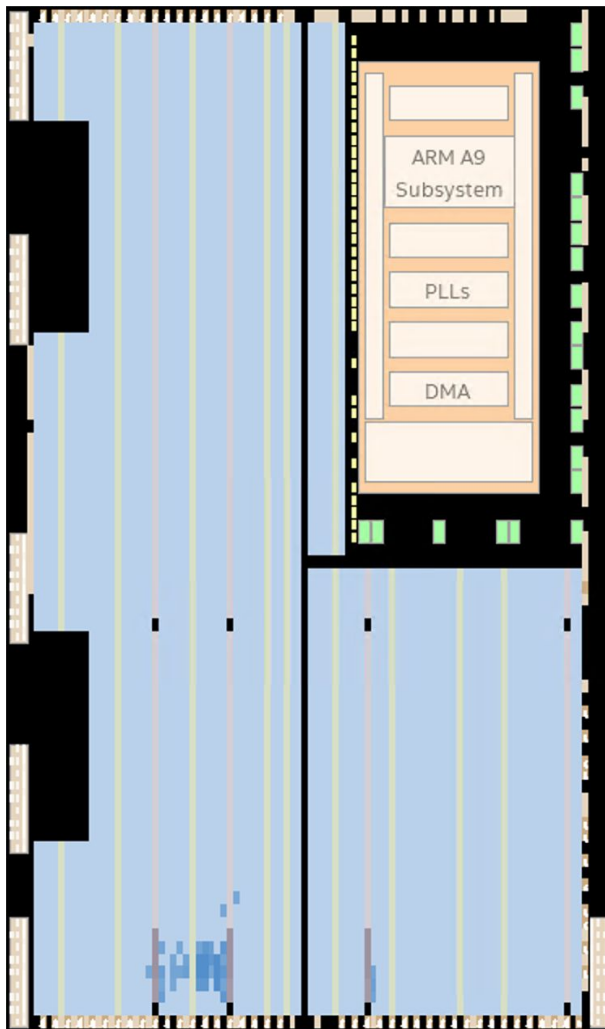
Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri Apr 24 03:10:08 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	G14lpm
Top-level Entity Name	g14lpm
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	139 / 32,070 (< 1 %)
Total registers	387
Total pins	261 / 457 (57 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	9 / 87 (10 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)

As we can see, 4 levels of pipelining requires a lot more resources than the previous implementation. Mid compilation the register count went to 806. However, in the final summary we can see that we require 387 registers. We are also using more logic modules (139).

The RTL view of our system is attached below:



The resource utilization of our system is attached below:



Purely by observation we can see that the 4 levels of pipelining using the LPM_MULT module utilizes the most resources on the chip planner. At this point we are hopeful that by using more resources we will be able to meet our timing constraint.

Timing Analysis

We were then required to use the same timing constraint of 5ns and conduct timing analysis. The timing summary is attached below:

Slow 1100mV 0C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	265.11 MHz	265.11 MHz	i_clk	

Slow 1100mV 0C Model Setup Summary			
<<Filter>>			
	Clock	Slack	End Point TNS
1	i_clk	-2.772	-1034.219

As expected our design now meets our timing constraint and goes further. By increasing our level of pipelining to 4 we have successfully met our timing constraint of 5ns. We achieved a maximum time period of 3.78 ns. We can also see a massive improvement in the setup slack. Therefore this experiment was a success. The results of all our implementations are detailed in the next section.

Summary

	<u>No Pipelining</u>	<u>Register Pipeline</u>	<u>LMP_MUL T P = 2</u>	<u>LMP_MUL T P = 3</u>	<u>LMP_MUL T P = 4</u>
<u># ALMs</u>	103	103	103	130	139
<u># Registers</u>	65	65 (420)*	65 (484)*	259 (678)*	387(806)*
<u># DSP Blocks</u>	9	9	9	9	9
<u>Fmax</u>	110.83MHz	143.99	137.32	192.53	265.11
<u>Slack</u>	-4.023	-1.945	-6.282	-4.194	-2.772

* register count during compilation

Conclusion

Overall, this design was perfect to get us acquainted with the idea of pipelining in digital systems. We pipelined the design of a complex multiplier based on the code provided to us by first adding a register. We then used the LPM library to pipeline using the LPM_MULT component which allowed us to meet our timing goal of 200MHz. This was previously unattainable using other implementations. We observed the resource usage and conducted timing analysis of each of our implementations. This allowed us to understand the importance of keeping track of the trade off between resource usage and speed in design. Our next goal is to use our design and implement it into an ARM FPGA circuitry. Ideally we would've liked to do this in conjunction with our designing. However, due to a time constraint we were unable to complete that.

This concludes the assignment