ECSE 325 - Take Home Final

Group 42

Michael Frajman	260863814
Shi Tong Li	260857759

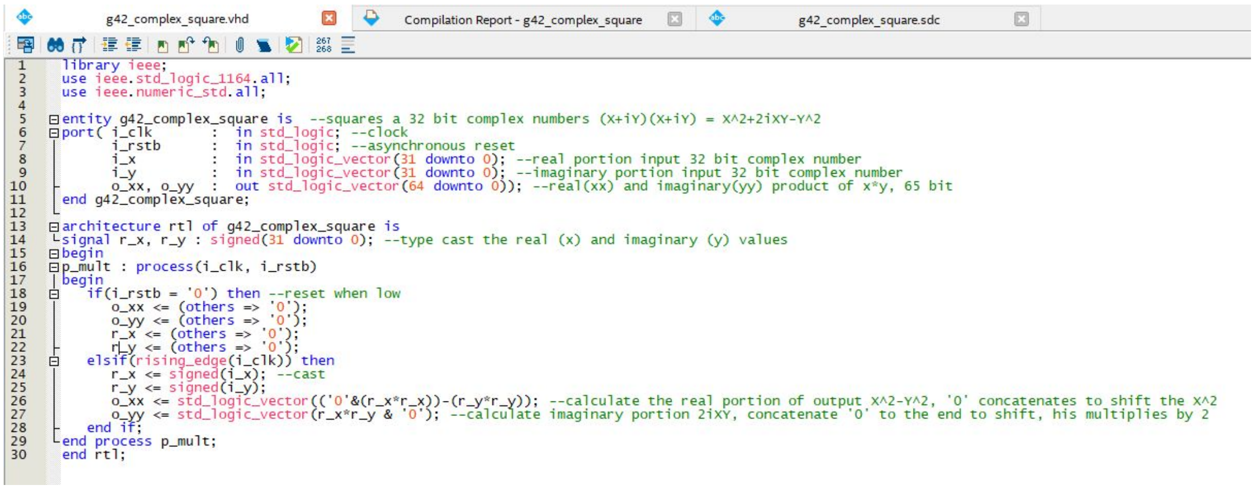
April 24, 2020

Introduction

The goal of the project was to observe the effects of pipelining a hardware design on speed and resource usage by using a timing simulation. The code being analyzed is for calculating the real and imaginary components of a complex number being squared. Two forms of pipelining, one using intermediate registers and another using lpm component multipliers, will be compared against a control program with no pipelining as provided in the assignment document. Additionally the lpm multiplier has the ability to specify the number of layers of additional combinational logic for pipelining, as such designs with different "pipeline" values for the multiplier component will be compared.

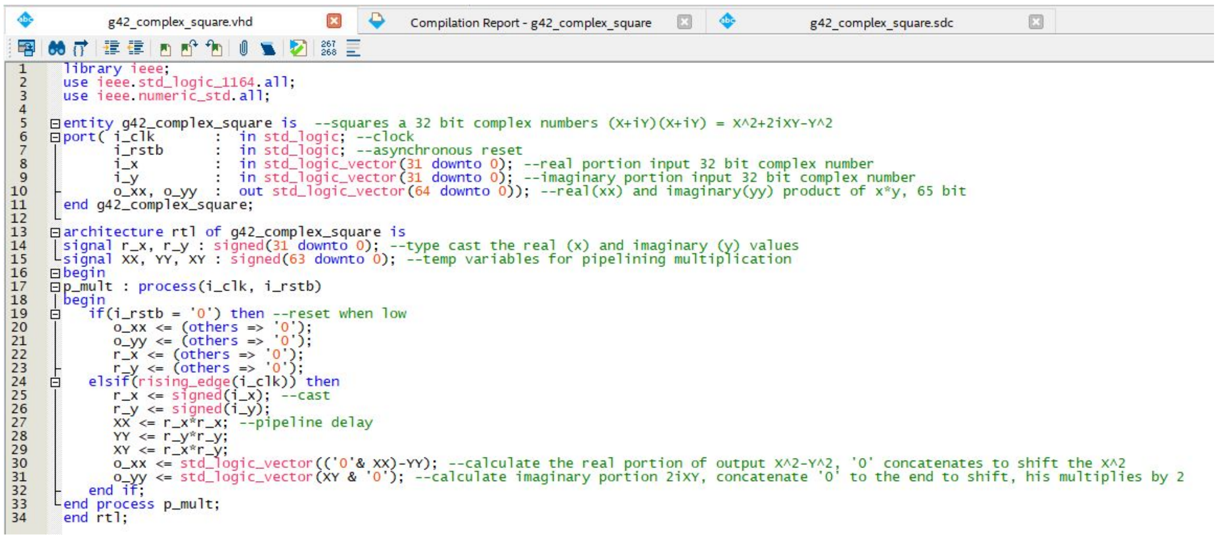
The designs will be compared for their resource utilization through the compilation report as well as their timing frequency against a 5ns clock signal. It is expected that the non-pipelined design will use less registers and FPGA resources but will run at a low frequency and with timing violations. The pipelined designs will require more resources but will run closer to desired clock frequency and possibly without timing violations. These expectations are based off of the in-class discussions on pipelining, where by adding additional layers of combinational logic between registers at the cost of FPGA resources, signals will tend to be more synchronized with the system's clock and timing violations can be avoided.

VHDL code for all versions of the complex square program



```vhdl
 library ieee;
 use ieee.std_logic_1164.all;
 use ieee.numeric_std.all;

entity g42_complex_square is  --squares a 32 bit complex numbers (X+iY)(X+iY) = X^2+2iXY-Y^2
port( i_clk        :  in std_logic; --clock
       i_rstb       :  in std_logic; --asynchronous reset
       i_x          :  in std_logic_vector(31 downto 0); --real portion input 32 bit complex number
       i_y          :  in std_logic_vector(31 downto 0); --imaginary portion input 32 bit complex number
       o_xx, o_yy   :  out std_logic_vector(64 downto 0)); --real(xx) and imaginary(yy) product of x*y, 65 bit
end g42_complex_square;

architecture rtl of g42_complex_square is
signal r_x, r_y : signed(31 downto 0); --type cast the real (x) and imaginary (y) values
begin
p_mult : process(i_clk, i_rstb)
begin
   if(i_rstb = '0') then --reset when low
      o_xx <= (others => '0');
      o_yy <= (others => '0');
      r_x <= (others => '0');
      r_y <= (others => '0');
   elsif(rising_edge(i_clk)) then
      r_x <= signed(i_x); --cast
      r_y <= signed(i_y);
      o_xx <= std_logic_vector(('0'&(r_x*r_x))-(r_y*r_y)); --calculate the real portion of output X^2-Y^2, '0' concatenates to shift the X^2
      o_yy <= std_logic_vector(r_x*r_y & '0'); --calculate imaginary portion 2iXY, concatenate '0' to the end to shift, his multiplies by 2
   end if;
end process p_mult;
end rtl;
```

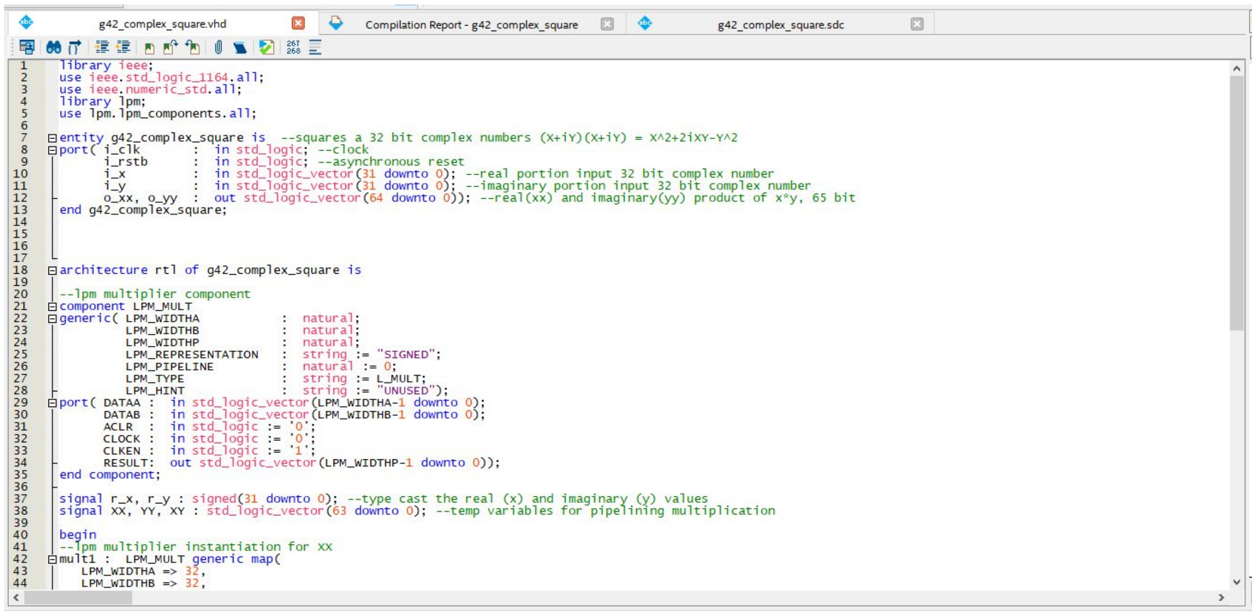
Image 1: Basic code for Complex Square program (as provided in the assignment document) with no attempts at pipelining. Note that partial descriptions of the inputs and outputs are listed in the code comments in the image.



g42_complex_square.vhd | Compilation Report - g42_complex_square | g42_complex_square.sdc

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity g42_complex_square is  --squares a 32 bit complex numbers (X+iY)(X+iY) = X^2+2iXY-Y^2
port( i_clk        :  in std_logic; --clock
      i_rstb       :  in std_logic; --asynchronous reset
      i_x          :  in std_logic_vector(31 downto 0); --real portion input 32 bit complex number
      i_y          :  in std_logic_vector(31 downto 0); --imaginary portion input 32 bit complex number
      o_xx, o_yy   :  out std_logic_vector(64 downto 0)); --real(xx) and imaginary(yy) product of x*y, 65 bit
end g42_complex_square;

architecture rtl of g42_complex_square is
signal r_x, r_y : signed(31 downto 0); --type cast the real (x) and imaginary (y) values
signal XX, YY, XY : signed(63 downto 0); --temp variables for pipelining multiplication
begin
p_mult : process(i_clk, i_rstb)
begin
   if(i_rstb = '0') then --reset when low
      o_xx <= (others => '0');
      o_yy <= (others => '0');
      r_x <= (others => '0');
      r_y <= (others => '0');
   elsif(rising_edge(i_clk)) then
      r_x <= signed(i_x); --cast
      r_y <= signed(i_y);
      XX <= r_x*r_x; --pipeline delay
      YY <= r_y*r_y;
      XY <= r_x*r_y;
      o_xx <= std_logic_vector(('0'& XX)-YY); --calculate the real portion of output X^2-Y^2, '0' concatenates to shift the X^2
      o_yy <= std_logic_vector(XY & '0'); --calculate imaginary portion 2iXY, concatenate '0' to the end to shift, his multiplies by 2
   end if;
end process p_mult;
end rtl;
```

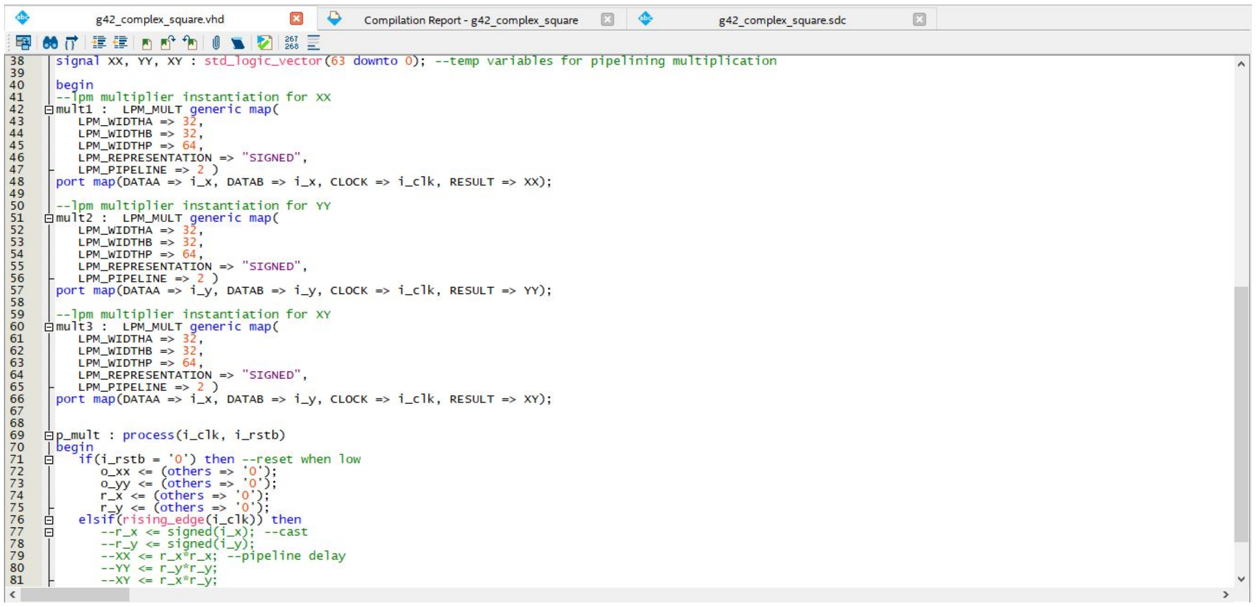
Image 2: Pipelined code for Complex Square program, uses intermediate registers to store the multiplied values with subtractions performed later.



g42_complex_square.vhd | Compilation Report - g42_complex_square | g42_complex_square.sdc

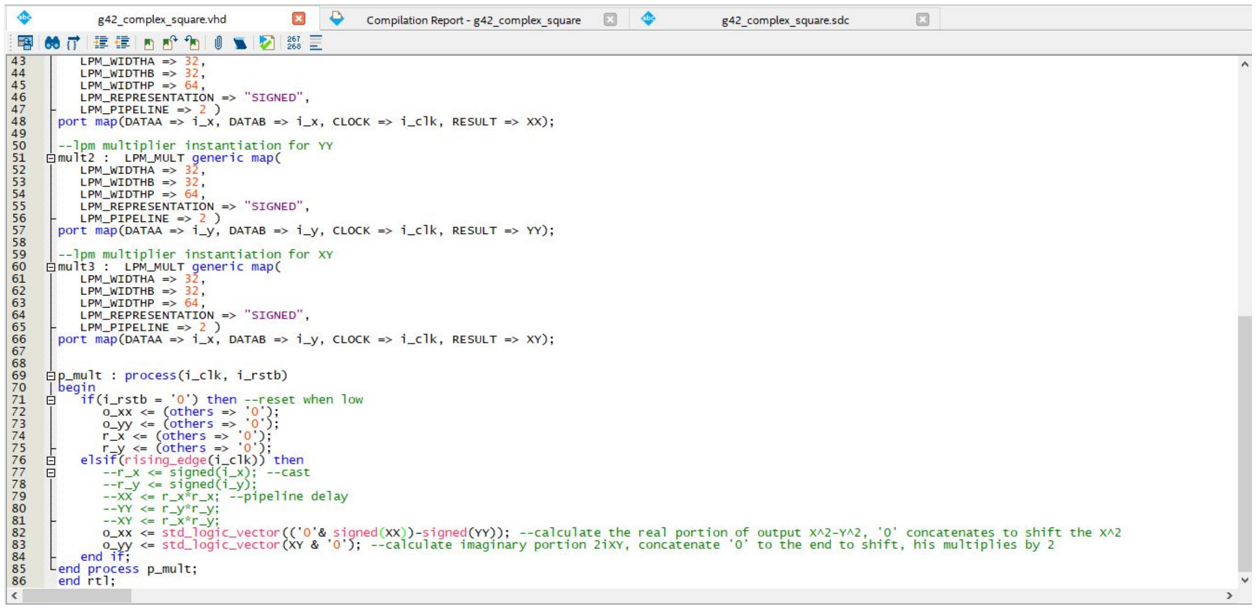
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library lpm;
use lpm.lpm_components.all;

entity g42_complex_square is  --squares a 32 bit complex numbers (X+iY)(X+iY) = X^2+2iXY-Y^2
port( i_clk        :  in std_logic; --clock
      i_rstb       :  in std_logic; --asynchronous reset
      i_x          :  in std_logic_vector(31 downto 0); --real portion input 32 bit complex number
      i_y          :  in std_logic_vector(31 downto 0); --imaginary portion input 32 bit complex number
      o_xx, o_yy   :  out std_logic_vector(64 downto 0)); --real(xx) and imaginary(yy) product of x*y, 65 bit
end g42_complex_square;




architecture rtl of g42_complex_square is

--lpm multiplier component
component LPM_MULT
generic( LPM_WIDTHA              :  natural;
         LPM_WIDTHB              :  natural;
         LPM_WIDTHP              :  natural;
         LPM_REPRESENTATION      :  string := "SIGNED";
         LPM_PIPELINE            :  natural := 0;
         LPM_TYPE                :  string := L_MULT;
         LPM_HINT                :  string := "UNUSED");
port( DATAA :  in std_logic_vector(LPM_WIDTHA-1 downto 0);
      DATAB :  in std_logic_vector(LPM_WIDTHB-1 downto 0);
      ACLR  :  in std_logic := '0';
      CLOCK :  in std_logic := '0';
      CLKEN :  in std_logic := '1';
      RESULT:  out std_logic_vector(LPM_WIDTHP-1 downto 0));
end component;

signal r_x, r_y : signed(31 downto 0); --type cast the real (x) and imaginary (y) values
signal XX, YY, XY : std_logic_vector(63 downto 0); --temp variables for pipelining multiplication

begin
--lpm multiplier instantiation for XX
mult1 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
```

Image 3: Pipelined code for Complex Square program, uses the built-in multiplier in order to achieve pipelining. Part 1.



```vhdl
signal XX, YY, XY : std_logic_vector(63 downto 0); --temp variables for pipelining multiplication

begin
--lpm multiplier instantiation for XX
mult1 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_x, DATAB => i_x, CLOCK => i_clk, RESULT => XX);

--lpm multiplier instantiation for YY
mult2 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_y, DATAB => i_y, CLOCK => i_clk, RESULT => YY);

--lpm multiplier instantiation for XY
mult3 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_x, DATAB => i_y, CLOCK => i_clk, RESULT => XY);


p_mult : process(i_clk, i_rstb)
begin
   if(i_rstb = '0') then --reset when low
      o_xx <= (others => '0');
      o_yy <= (others => '0');
      r_x <= (others => '0');
      r_y <= (others => '0');
   elsif(rising_edge(i_clk)) then
      --r_x <= signed(i_x); --cast
      --r_y <= signed(i_y);
      --XX <= r_x*r_x; --pipeline delay
      --YY <= r_y*r_y;
      --XY <= r_x*r_y;
```

Image 4:Pipelined code for Complex Square program, uses the built-in multiplier in order to achieve pipelining. Part 2.

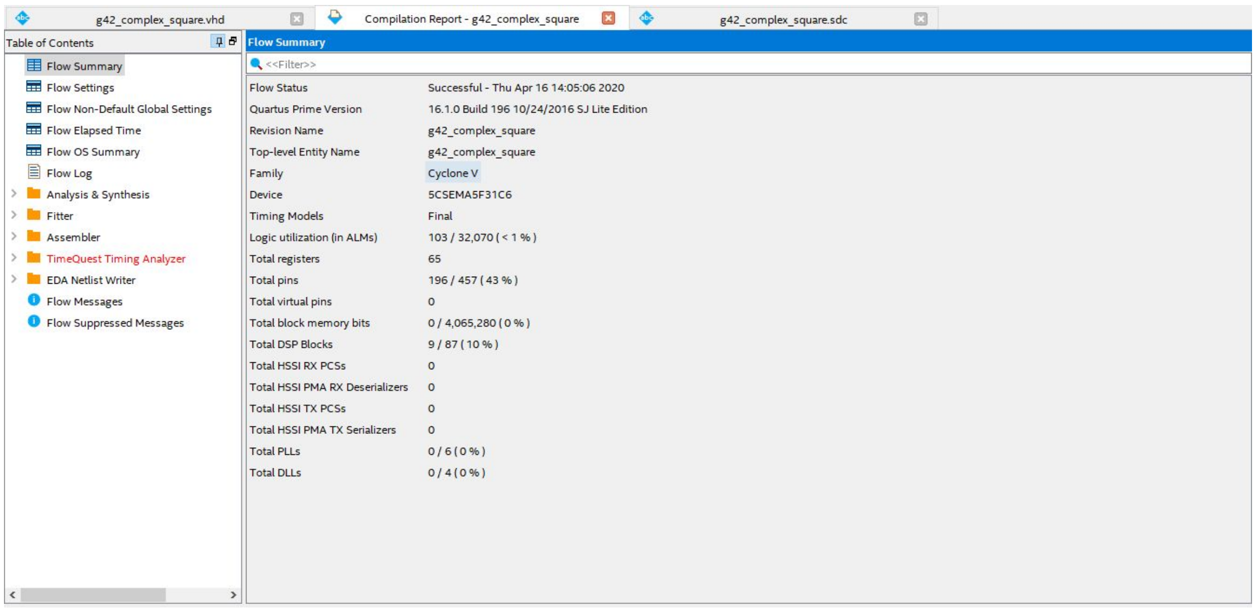


```vhdl
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_x, DATAB => i_x, CLOCK => i_clk, RESULT => XX);

--lpm multiplier instantiation for YY
mult2 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_y, DATAB => i_y, CLOCK => i_clk, RESULT => YY);

--lpm multiplier instantiation for XY
mult3 :  LPM_MULT generic map(
   LPM_WIDTHA => 32,
   LPM_WIDTHB => 32,
   LPM_WIDTHP => 64,
   LPM_REPRESENTATION => "SIGNED",
   LPM_PIPELINE => 2 )
port map(DATAA => i_x, DATAB => i_y, CLOCK => i_clk, RESULT => XY);


p_mult : process(i_clk, i_rstb)
begin
   if(i_rstb = '0') then --reset when low
      o_xx <= (others => '0');
      o_yy <= (others => '0');
      r_x <= (others => '0');
      r_y <= (others => '0');
   elsif(rising_edge(i_clk)) then
      --r_x <= signed(i_x); --cast
      --r_y <= signed(i_y);
      --XX <= r_x*r_x; --pipeline delay
      --YY <= r_y*r_y;
      --XY <= r_x*r_y;
      o_xx <= std_logic_vector(('0'& signed(XX))-signed(YY)); --calculate the real portion of output X^2-Y^2, '0' concatenates to shift the X^2
      o_yy <= std_logic_vector(XY & '0'); --calculate imaginary portion 2iXY, concatenate '0' to the end to shift, his multiplies by 2
   end if;
end process p_mult;
end rtl;
```

Image 5:Pipelined code for Complex Square program, uses the built-in multiplier in order to achieve pipelining. Part 3.

Note: the lpm_mult code for the complex square program is the same, except the value of LPM_PIPELINE is changed to the corresponding number (2,3, or 4).

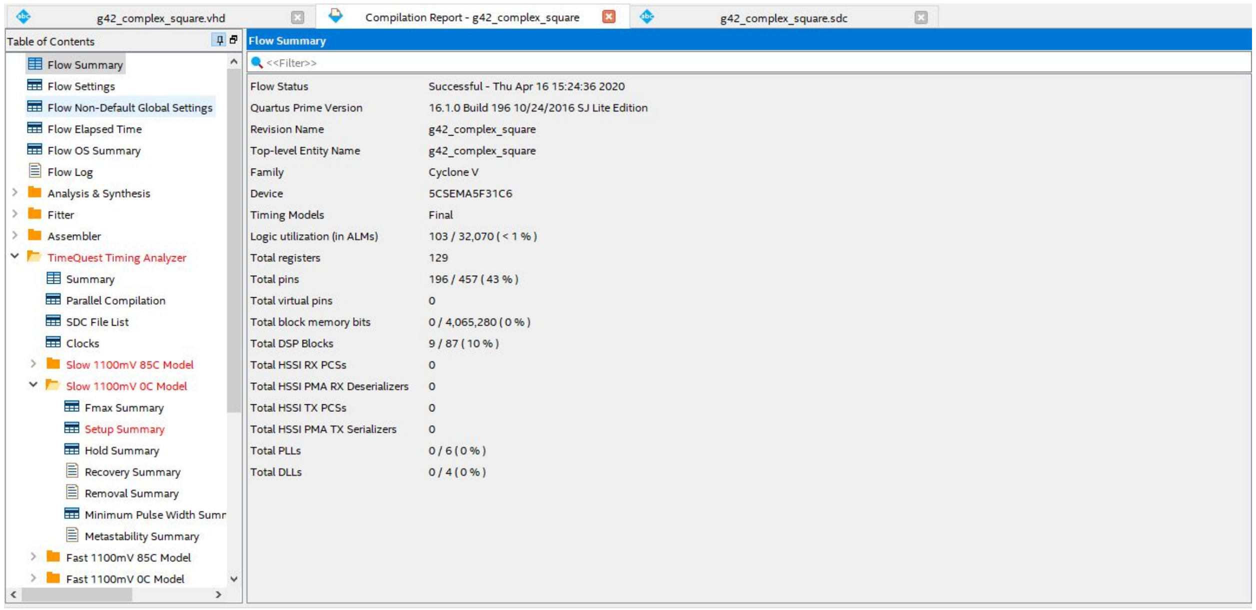
Compilation report



Flow Summary

Flow Status	Successful - Thu Apr 16 14:05:06 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g42_complex_square
Top-level Entity Name	g42_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 %)
Total DLLs	0 / 4 (0 %)

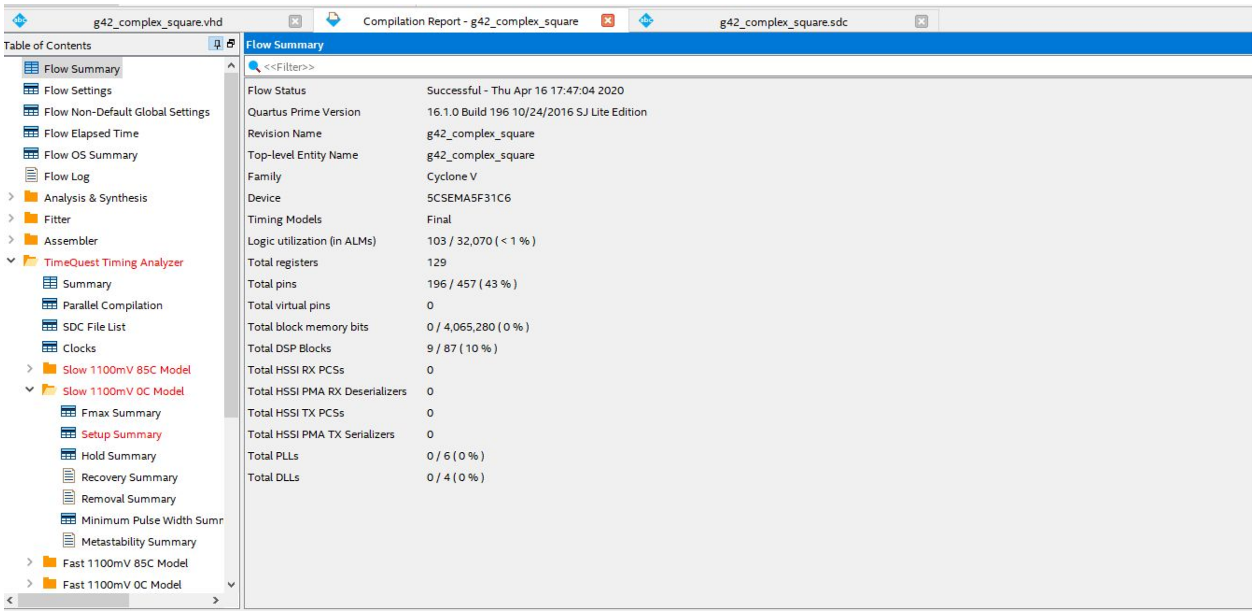
Image 6: Compilation report for the basic non pipelined code.



Flow Summary

Flow Status	Successful - Thu Apr 16 15:24:36 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g42_complex_square
Top-level Entity Name	g42_complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	103 / 32,070 (< 1 %)
Total registers	129
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 %)
Total DLLs	0 / 4 (0 %)

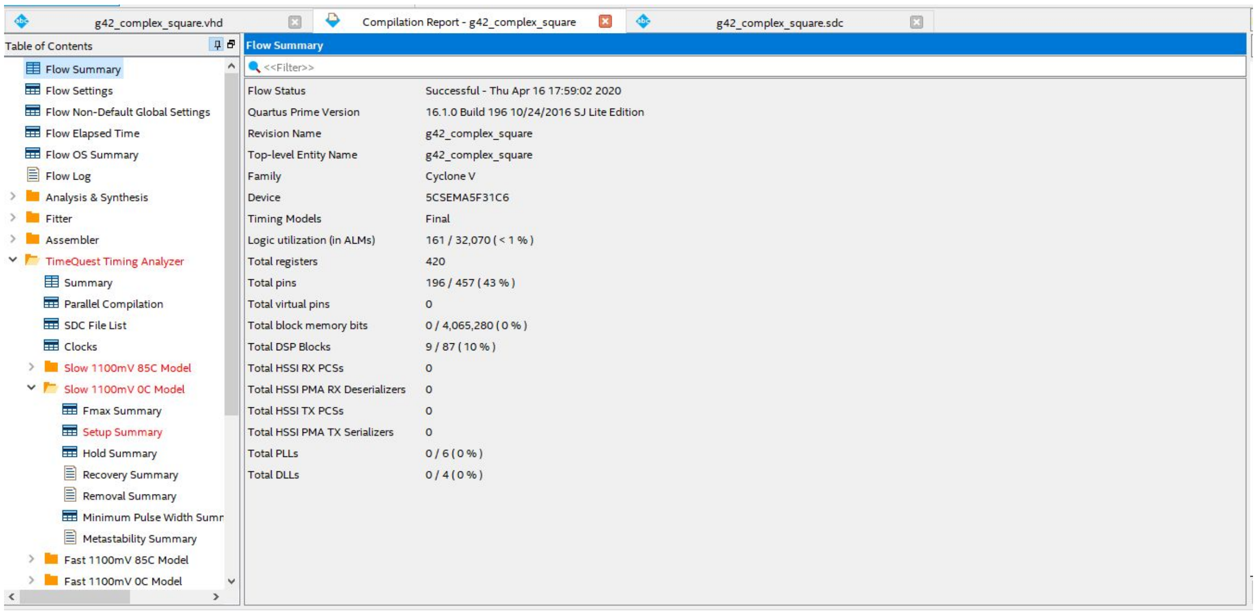
Image 7: Compilation report for pipelined code.



Flow Summary

Flow Status	Successful - Thu Apr 16 17:47:04 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g42_complex_square
Top-level Entity Name	g42_complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	103 / 32,070 (< 1 %)
Total registers	129
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 %)
Total DLLs	0 / 4 (0 %)

Image 8: Compilation report for 2 layer lpm multiplier pipelined code.



Flow Summary

Flow Status	Successful - Thu Apr 16 17:59:02 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g42_complex_square
Top-level Entity Name	g42_complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	161 / 32,070 (< 1 %)
Total registers	420
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 %)
Total DLLs	0 / 4 (0 %)

Image 9: Compilation report for 3 layer lpm multiplier pipelined code.

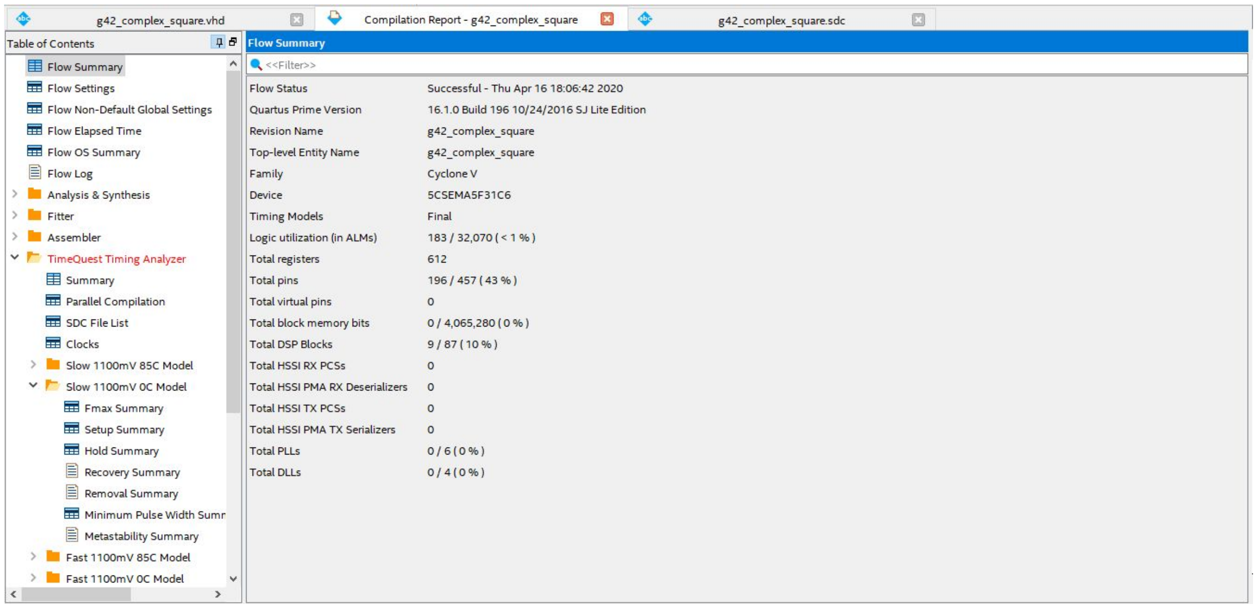

g42_complex_square.vhd
Compilation Report - g42_complex_square
g42_complex_square.sdc

Table of Contents
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Slow 1100mV 85C Model
  - Slow 1100mV 0C Model
    - Fmax Summary
    - Setup Summary
    - Hold Summary
    - Recovery Summary
    - Removal Summary
    - Minimum Pulse Width Sumr
    - Metastability Summary
  - Fast 1100mV 85C Model
  - Fast 1100mV 0C Model

Flow Summary

Flow Status	Successful - Thu Apr 16 18:06:42 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g42_complex_square
Top-level Entity Name	g42_complex_square
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	183 / 32,070 (< 1 %)
Total registers	612
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 %)
Total DLLs	0 / 4 (0 %)

Image 10: Compilation report for 4 layer lpm multiplier pipelined code.

Resource utilization

For the basic code, 65 registers were used for a complex square program that has two 32-bit inputs, two 64-bit output, a clock, and a reset. If we implement a two-level pipelining, we can see that the number of registers increases to 129. Furthermore, the number of registers required for a three or four level pipelining implementation would be 420 and 612, respectively. We can conclude that by increasing the number of levels of pipelining, we can expect the number of registers used to also increase. Additionally, since the number of registers used is directly proportional to the number of bits of the input and output, we can expect a linear increase or decrease in register utilization when adding or removing bits in the size input or the output.

Chip planner

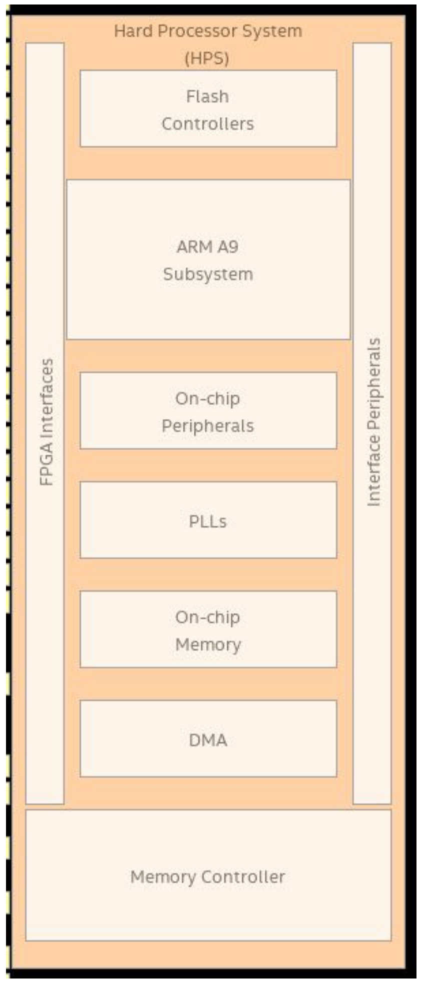


Image 11: Closeup of the processor of the FPGA board from the chip planner as it appears in all comples square designs

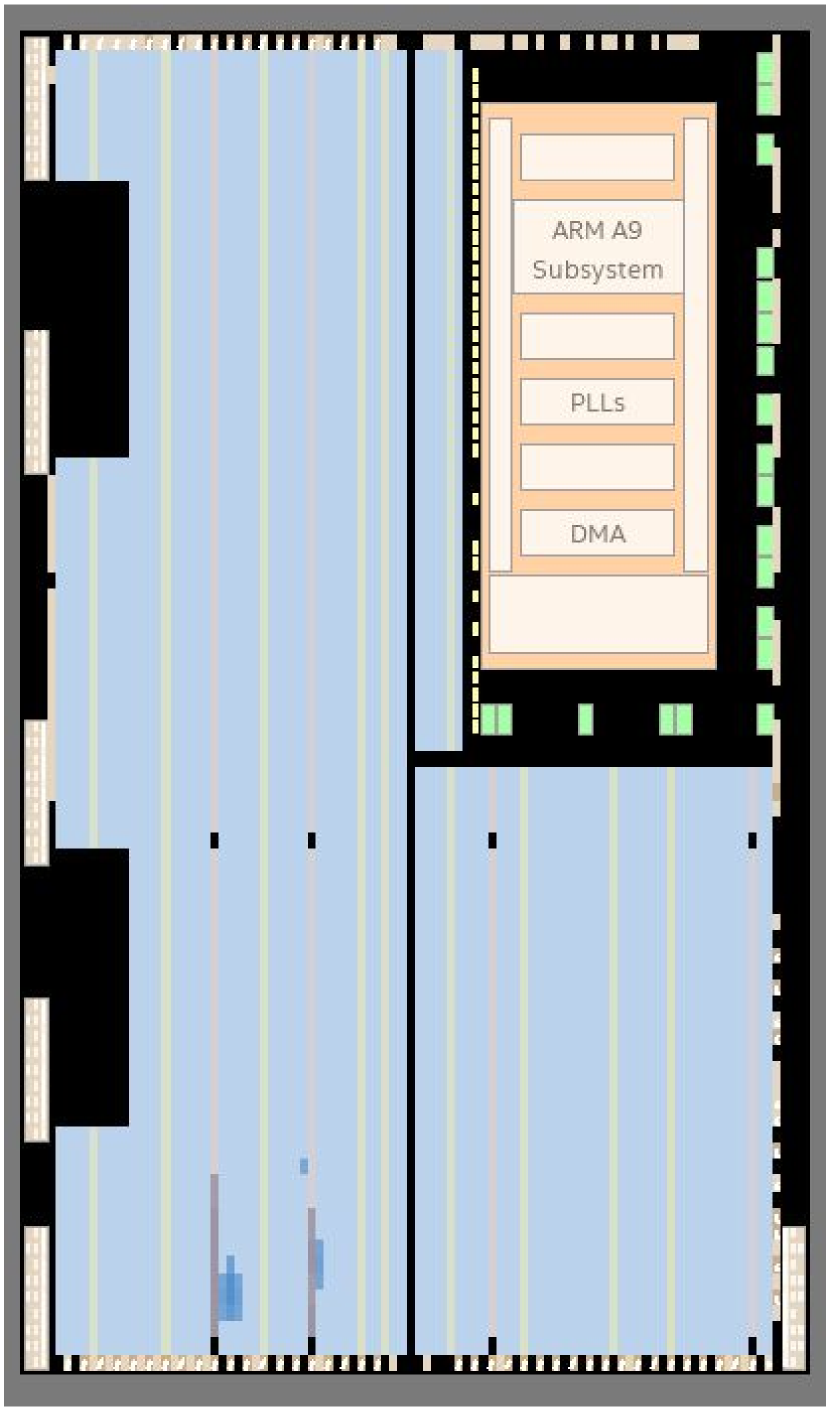


Image 12: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue and brown rectangles concentrated in the bottom left. (non-pipelined code)

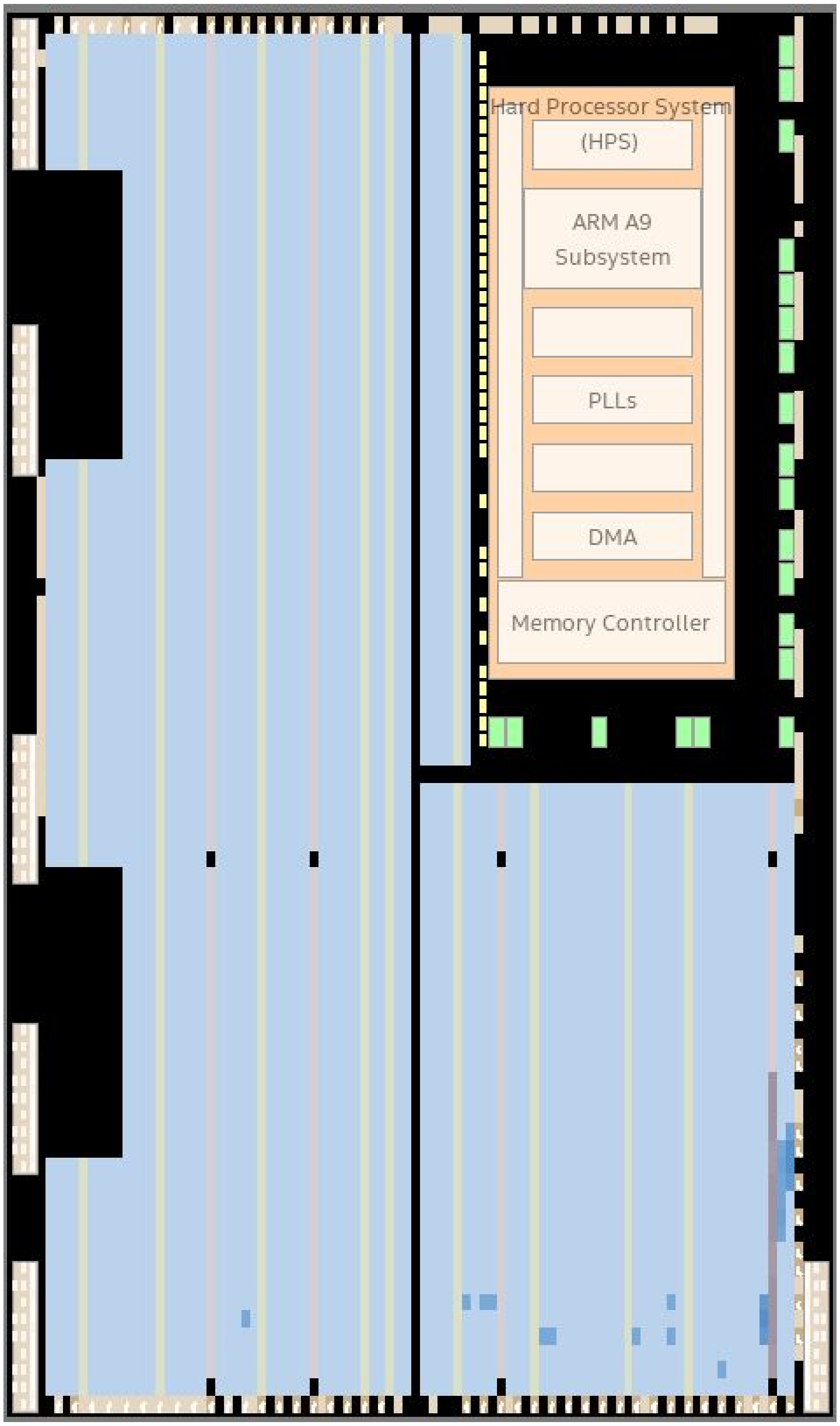


Image 13: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue and brown rectangles concentrated in the bottom right. (pipelined code)

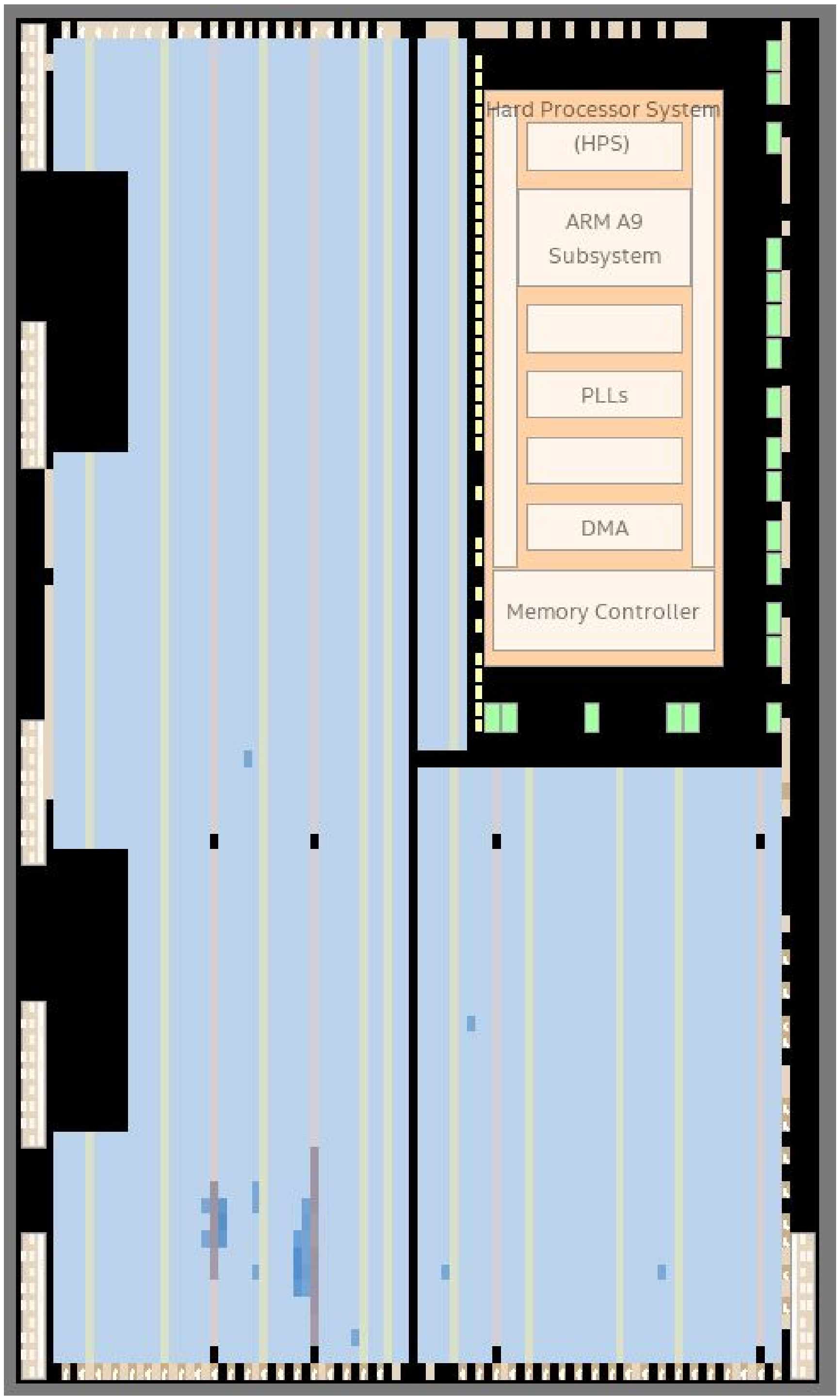


Image 14: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue and brown rectangles concentrated in the bottom left. (2 layer multiplier pipelined code)

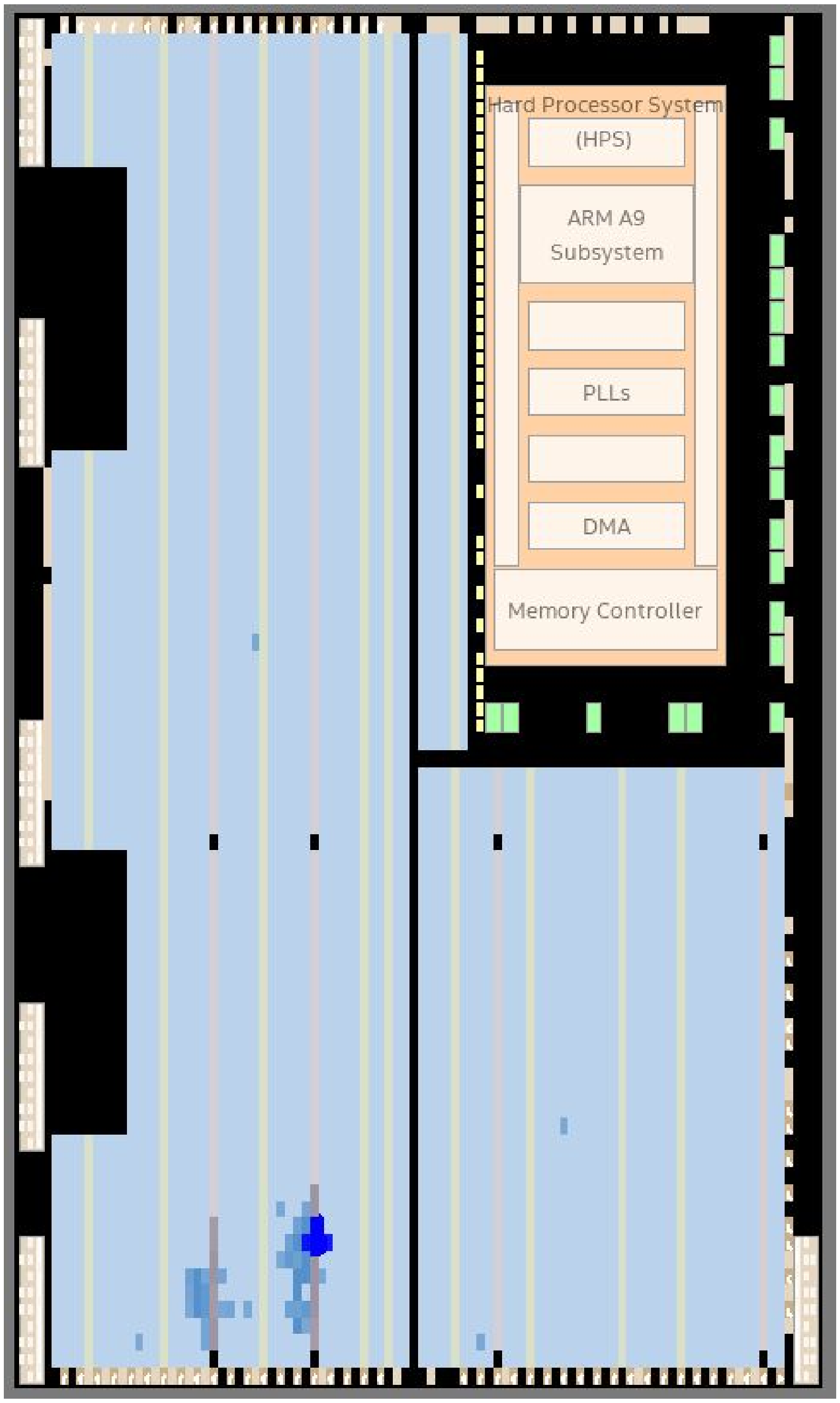


Image 15: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue and brown rectangles concentrated in the bottom left. (3 layer multiplier pipelined code)

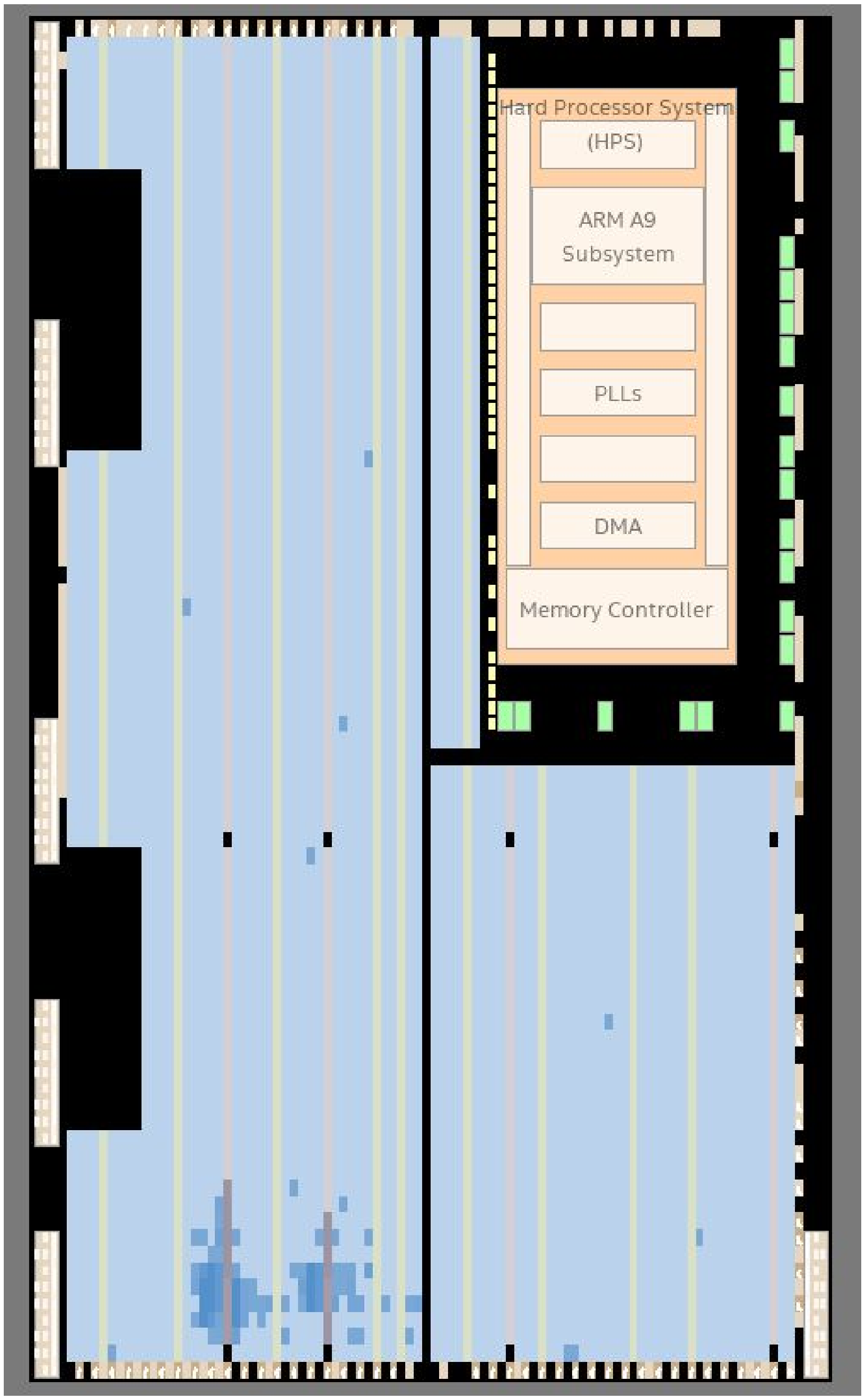


Image 16: Overview of the FPGA chip from the chip planner. Used resources/look-up-tables are represented by darkened blue and brown rectangles concentrated in the bottom left. (4 layer multiplier pipelined code)

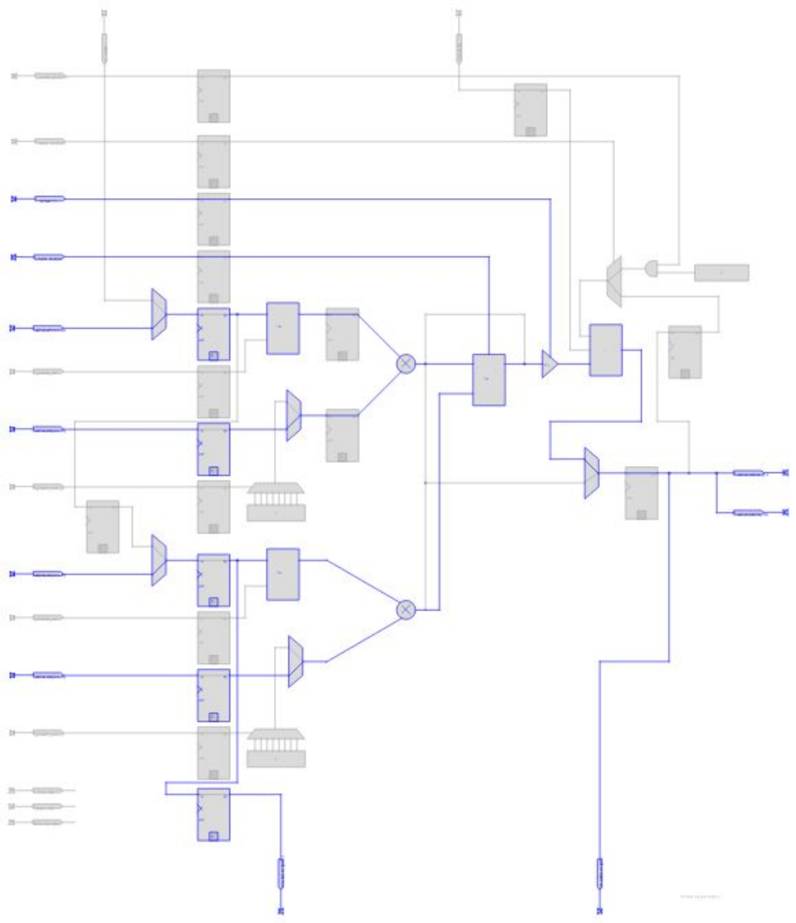


Image 17: Example of a logic circuit layout of a look-up-table being used by the FPGA board. This image is from the chip planner of the non pipelining design. All designs have similar look up tables but enable different paths through the components.

RTL view

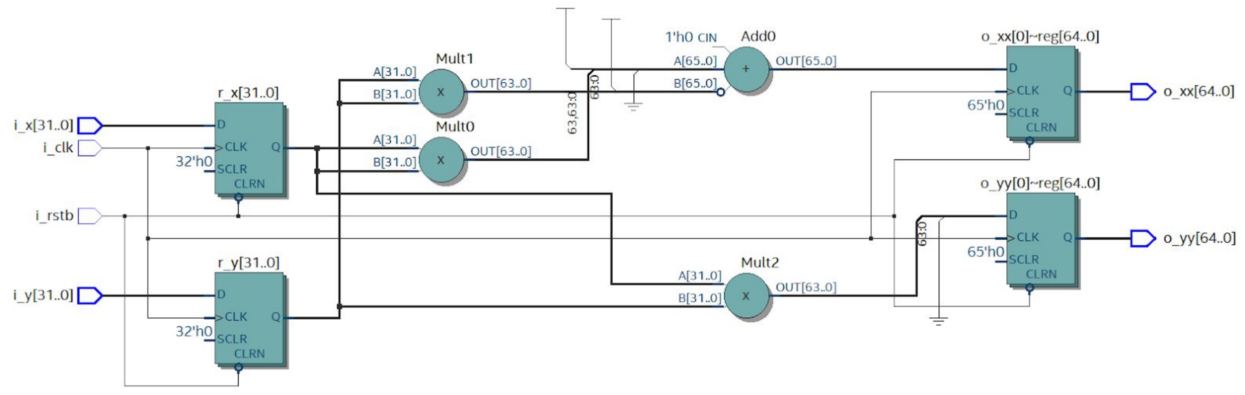


Image 18: RTL view for basic code

Basic code: From the RTL viewer screenshot, it can be seen that we are using 1 adder, 3 multipliers, and 4 registers. The control signals on the registers are used to implement the clock and reset.The two rightmost registers at the end are used to store the two 32-bit outputs of the complex square output.

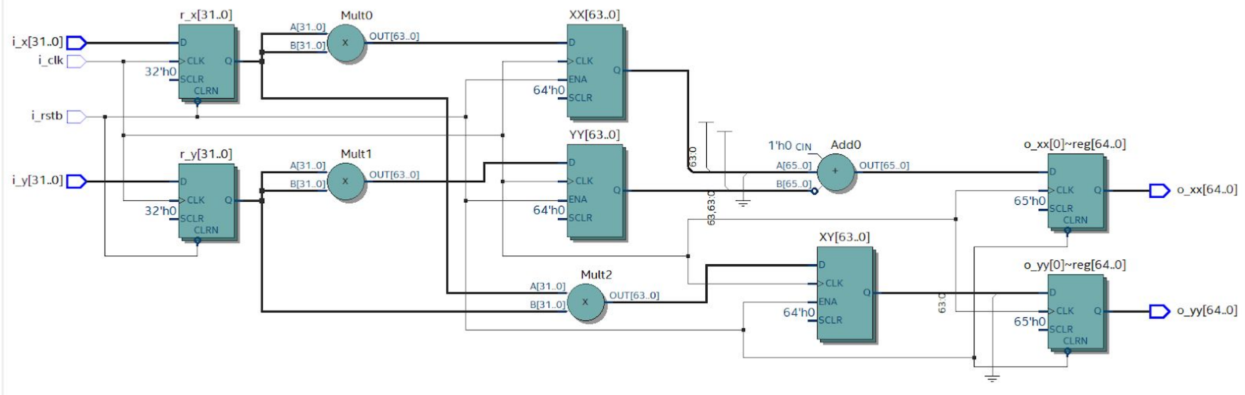


Image 19: RTL view for pipelined code

Pipelined code: From the RTL viewer screenshot, it can be seen that we are using 1 adder, 3 multipliers, and 7 registers. The two rightmost registers at the end are used to store the two 32-bit outputs of the complex square output. We can also observe that there is an increase in registers used compared to the basic code RTL, which suggests that pipelining was correctly implemented in our code.

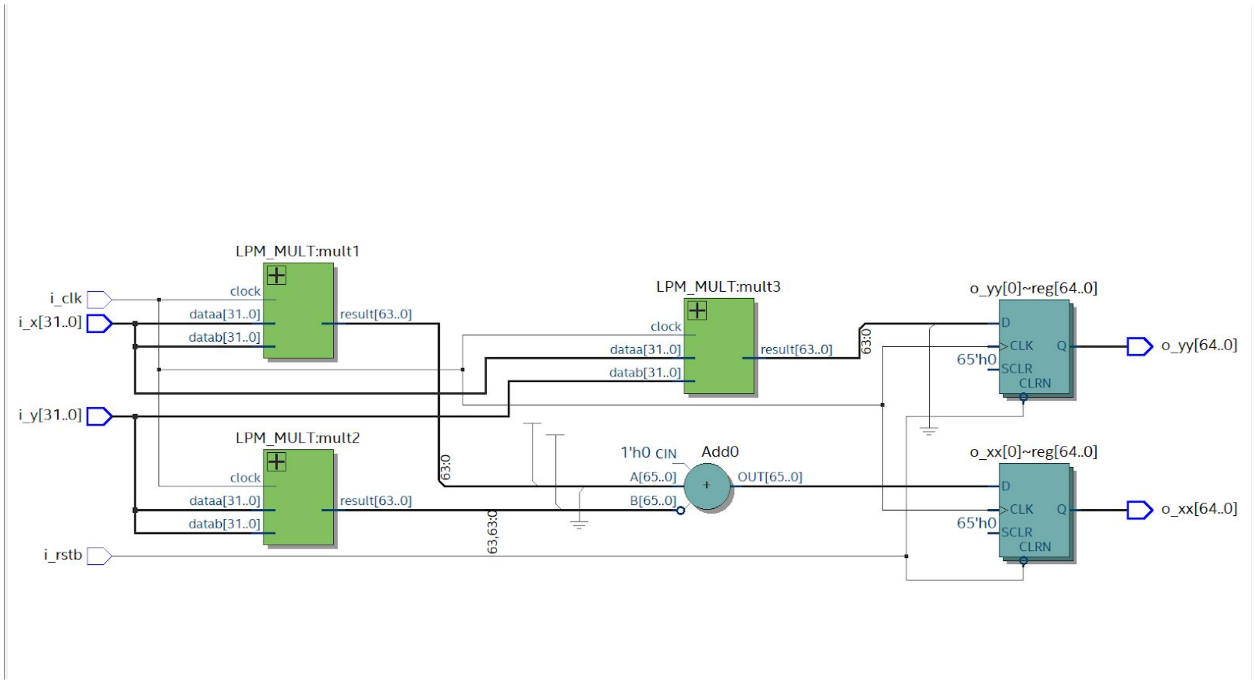


Image 20: RTL view for 2 layer multiplier pipelined code

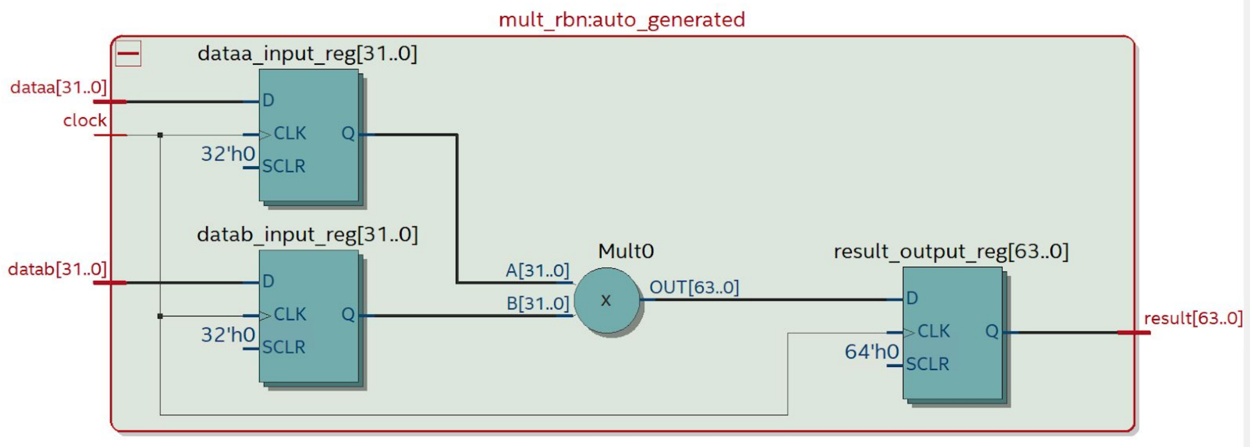


Image 21: RTL view for the built in multiplier as taken from the P=2 multiplier

Two layer multiplier design: From the RTL viewer screenshot, it can be seen that we are using 1 adder, 3 built in multipliers, and 2 registers. The two rightmost registers at the end are used to store the two 32-bit outputs of the complex square output. The built in multipliers consists of 3 registers and a multiplier and are visibly arranged in two distinct layers.

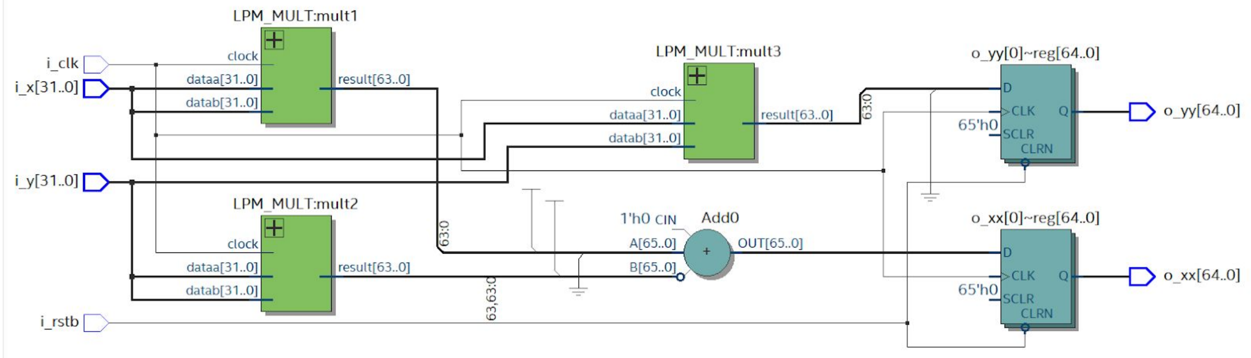


Image 22: RTL view for 3 layer multiplier pipelined code

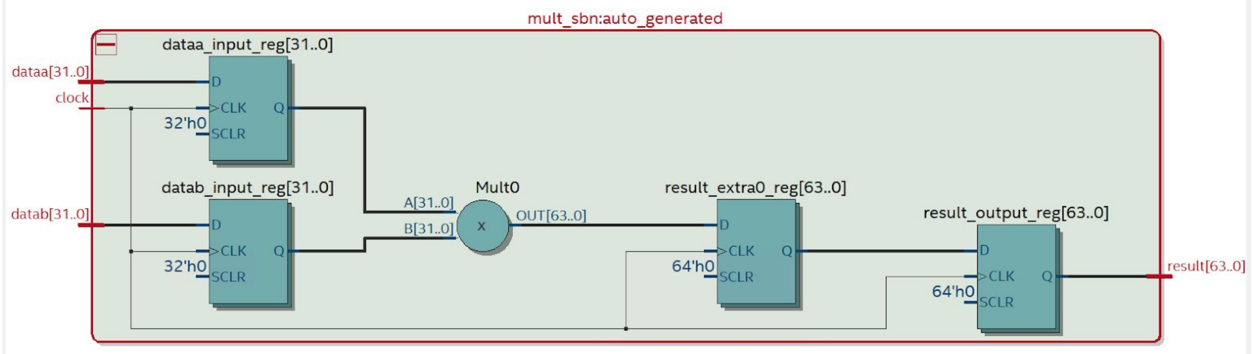


Image 23: RTL view for the built in multiplier as taken from the P=3 multiplier

Three layer multiplier design: From the RTL viewer screenshot, it can be seen that we are using 1 adder, 3 built in multipliers, and 2 registers. The two rightmost registers at the end are used to store the two 32-bit outputs of the complex square output. The built in multiplier consists of 4 registers and a multiplier. We can notice that the built in register has one more register in order to add a extra layer of pipelining.

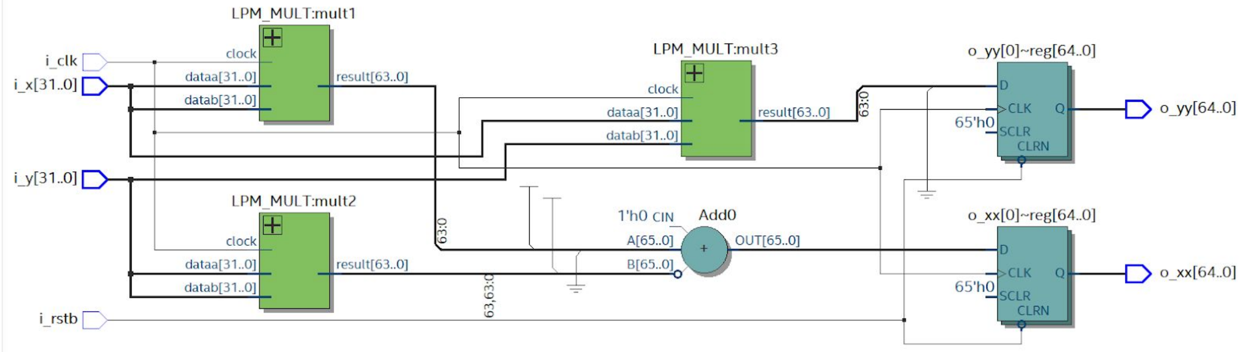


Image 24: RTL view for 4 layer multiplier pipelined code

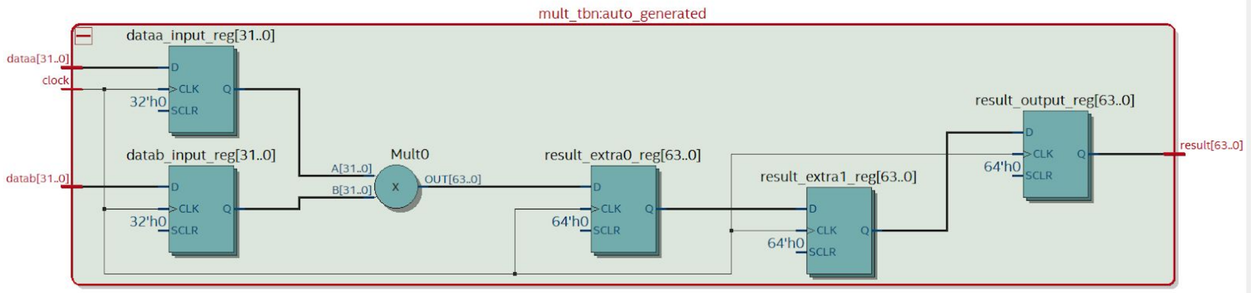
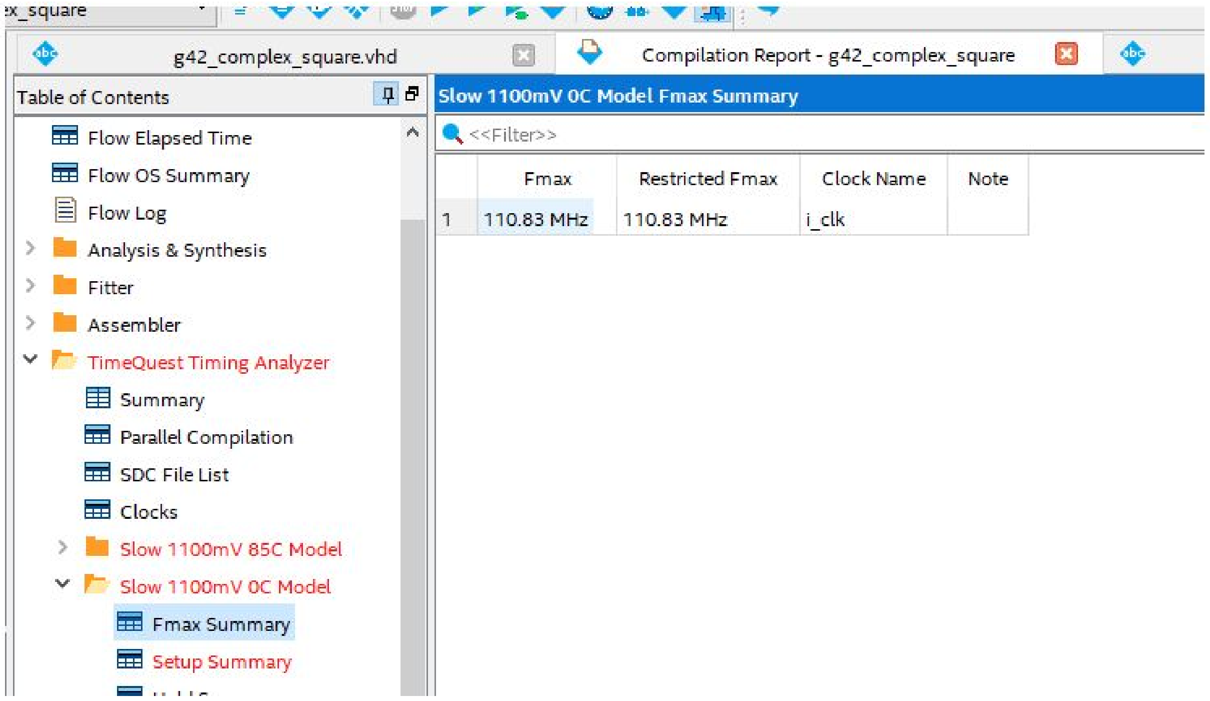


Image 25: RTL view for the built in multiplier as taken from the P=4 multiplier

Four layer multiplier design: From the RTL viewer screenshot, it can be seen that we are using 1 adder, 3 built in multipliers, and 2 registers. The two rightmost registers at the end are used to store the two 32-bit outputs of the complex square output. The built in multiplier consists of 5 registers and a multiplier. Again it can be seen that the built in register has more registers, in this case two more, compared to the 2 layer pipeline, in order to add 2 more layers of pipelining.

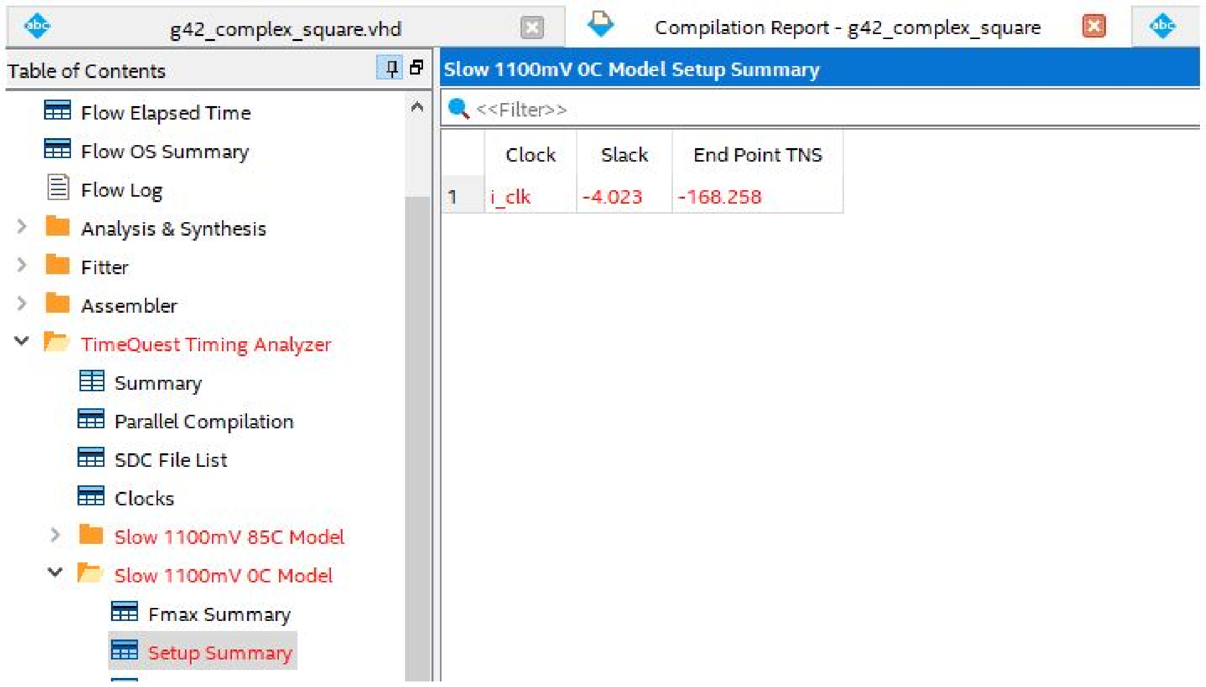
SDC



Slow 1100mV 0C Model Fmax Summary

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

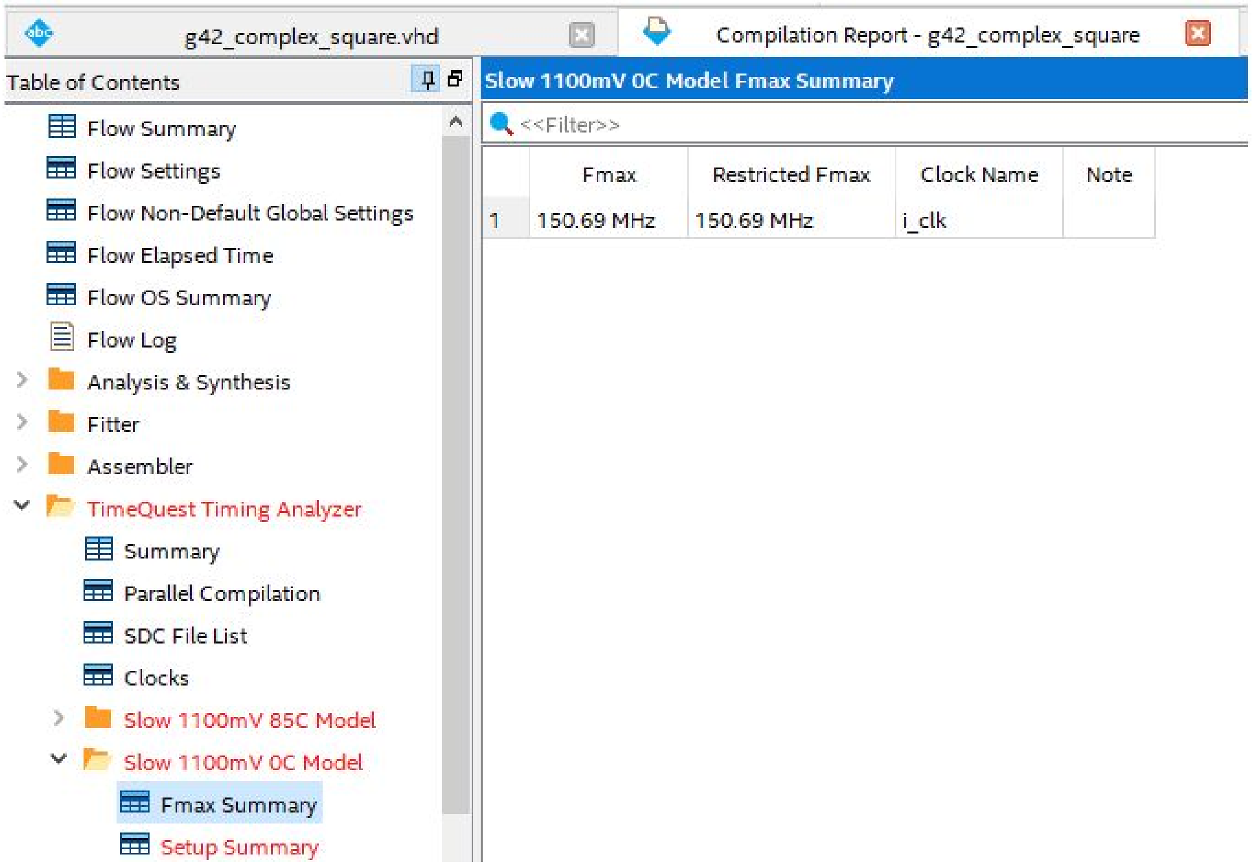
Image 26: F_{max} of basic code.



Slow 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	i_clk	-4.023	-168.258

Image 27: Slack of basic code.


g42_complex_square.vhd

Compilation Report - g42_complex_square

Table of Contents

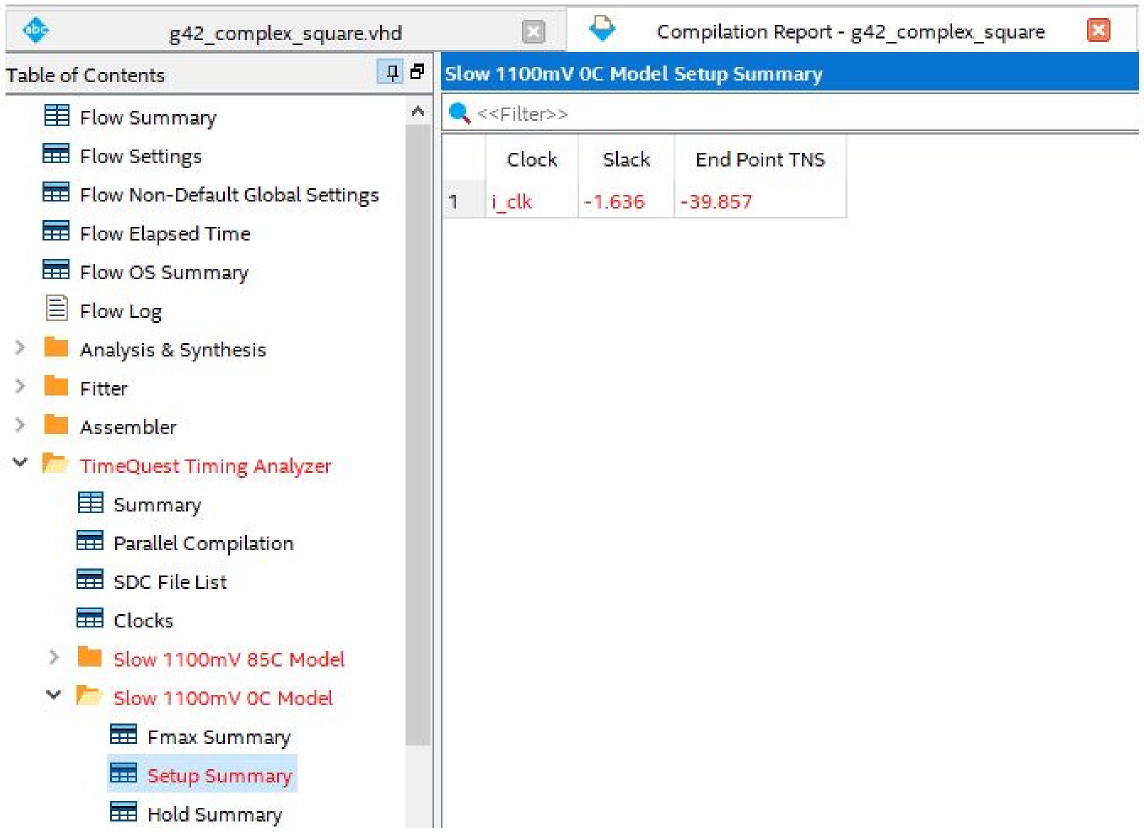
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Slow 1100mV 85C Model
  - Slow 1100mV 0C Model
    - Fmax Summary
    - Setup Summary

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	150.69 MHz	150.69 MHz	i_clk	

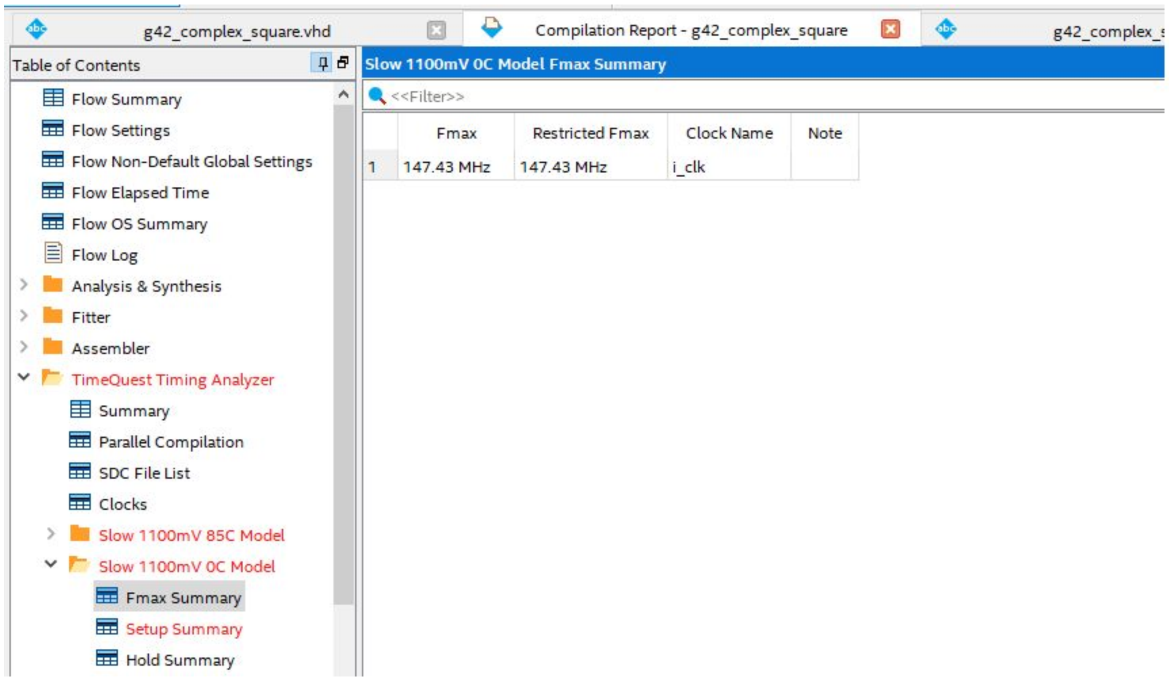
Image 28: F_{max} of pipelined code.



Slow 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	i_clk	-1.636	-39.857

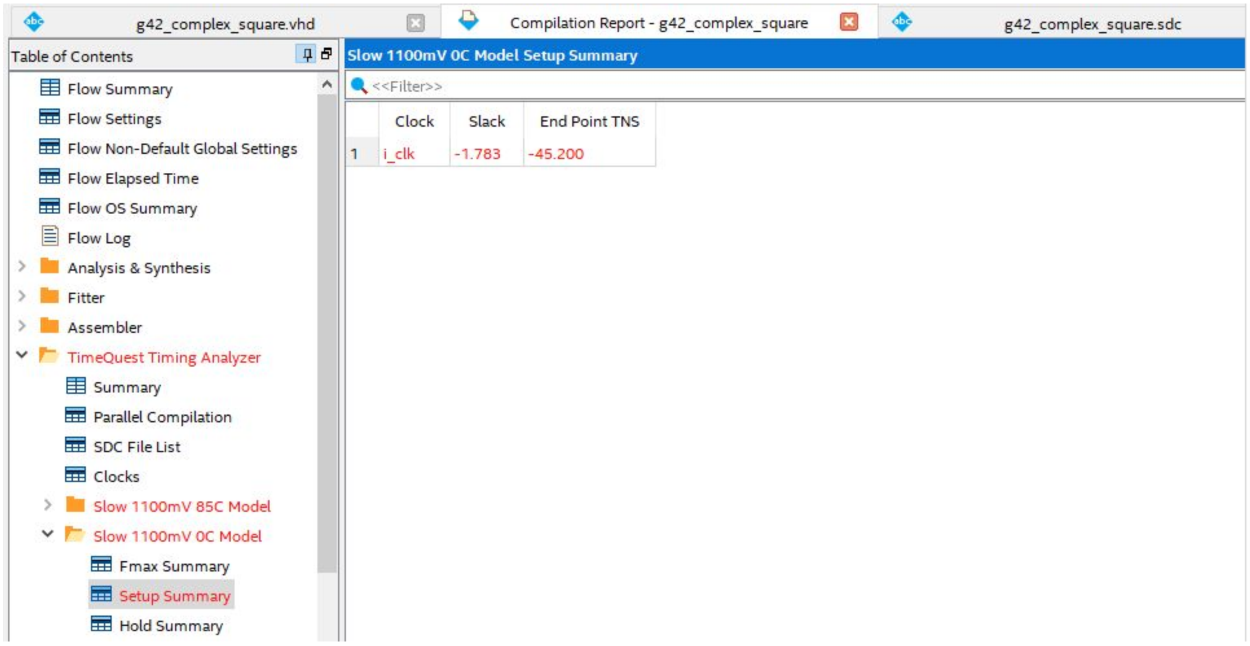
Image 29: Slack of pipelined code.



Slow 1100mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	147.43 MHz	147.43 MHz	i_clk	

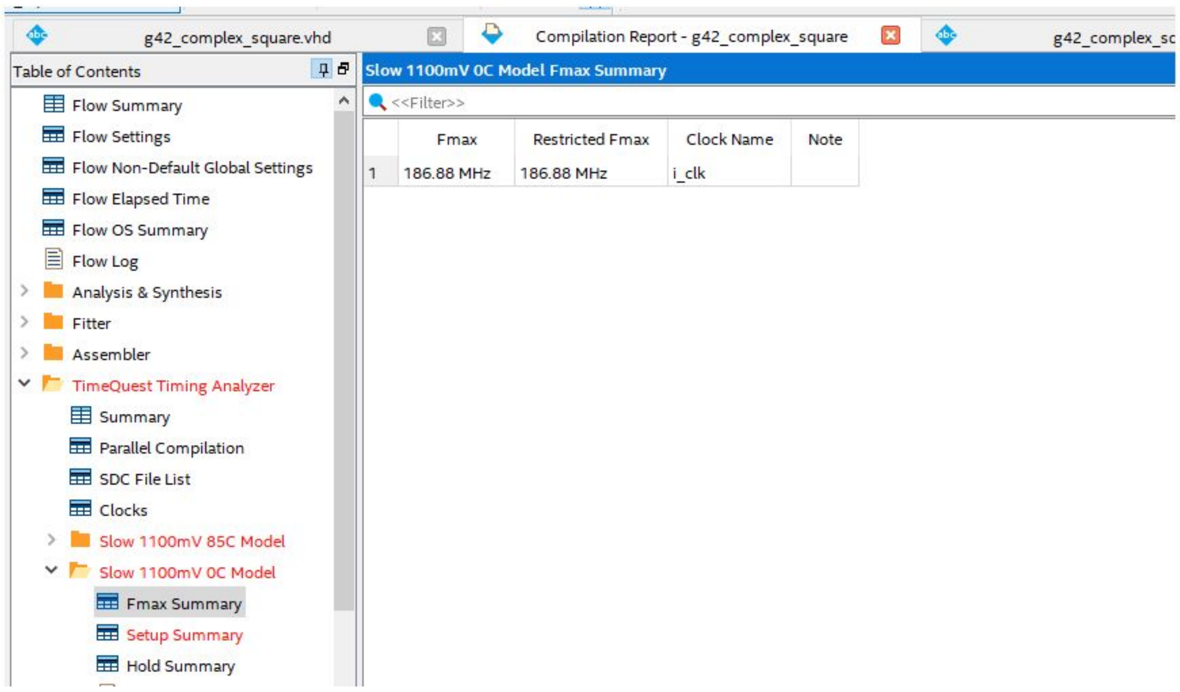
Image 30: F_{max} of 2 layer multiplier pipelined code.



Slow 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	i_clk	-1.783	-45.200

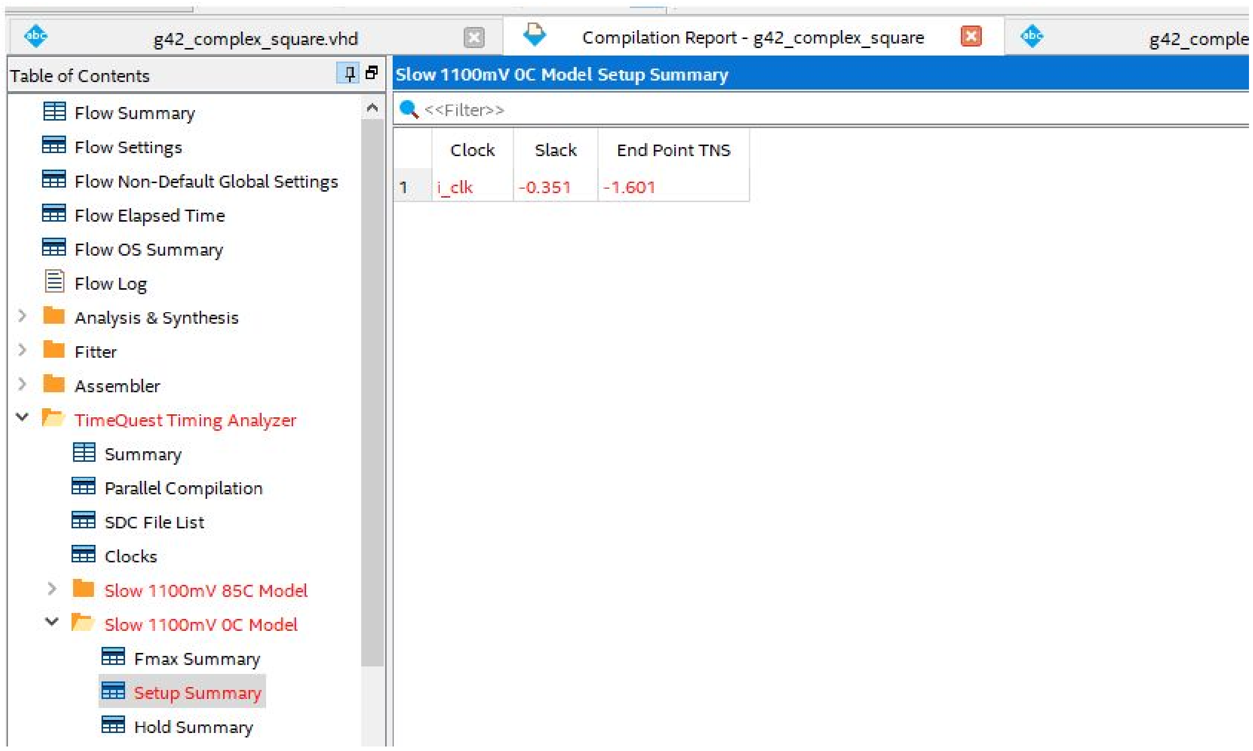
Image 31: Slack of 2 layer multiplier pipelined code.



Slow 1100mV 0C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	186.88 MHz	186.88 MHz	i_clk	

Image 32: F_{max} of 3 layer multiplier pipelined code.



g42_complex_square.vhd | Compilation Report - g42_complex_square | g42_comple

Table of Contents

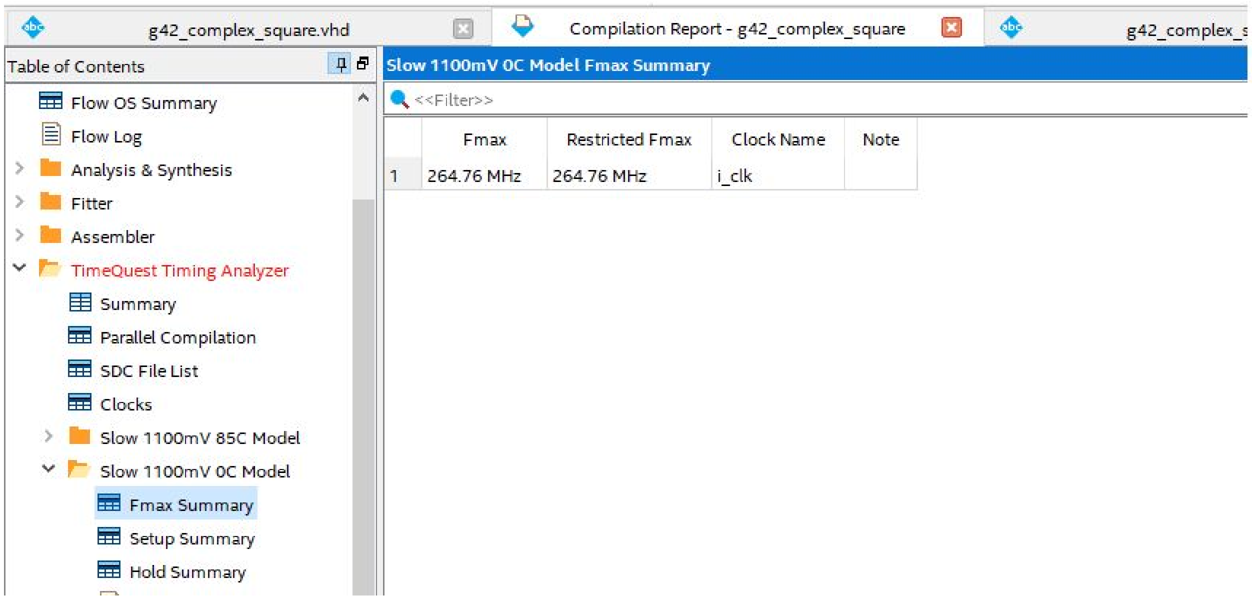
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Slow 1100mV 85C Model
  - Slow 1100mV 0C Model
    - Fmax Summary
    - Setup Summary
    - Hold Summary

Slow 1100mV 0C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	i_clk	-0.351	-1.601

Image 33: Slack of 3 layer multiplier pipelined code.



g42_complex_square.vhd | Compilation Report - g42_complex_square | g42_complex_s

Table of Contents

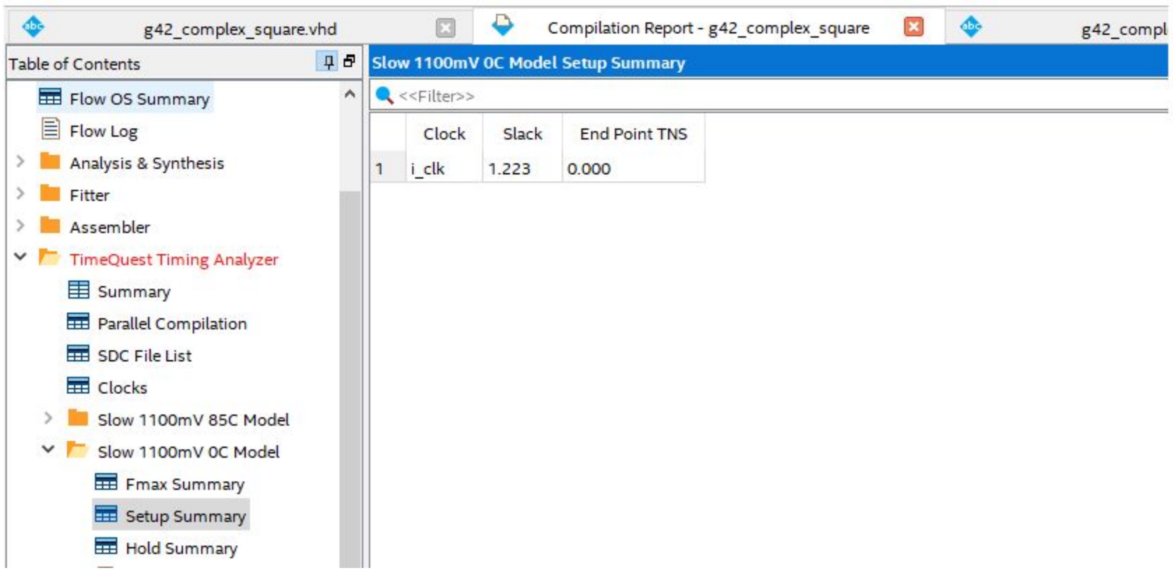
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
  - Summary
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Slow 1100mV 85C Model
  - Slow 1100mV 0C Model
    - Fmax Summary
    - Setup Summary
    - Hold Summary

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	264.76 MHz	264.76 MHz	i_clk	

Image 34: F_{max} of 4 layer multiplier pipelined code.



Slow 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	i_clk	1.223	0.000

Image 35: Slack of 4 layer multiplier pipelined code.

FPGA resource usage and timing analysis summary table

	No Pipelining	**Pipelined**	**Lpm_mult P=2**	**Lpm_mult P=3**	**Lpm_mult P=4**
# ALMs	103/32070	103/32070	103/32070	161/32070	183/32070
# Registers	65	129	129	420	612
# DSP Blocks	9/87	9/87	9/87	9/87	9/87
F_{max}	110.83MHz	150.69MHz	147.43MHz	186.88MHz	264.76MHz
Slack	-4.023ns	-1.636ns	-1.783ns	-0.351ns	1.223ns
Timing Test (Pass/Fail)	Fail	Fail	Fail	Fail	Pass

Table 1: FPGA resource usage and timing analysis summary

As predicted in our introduction a non-pipelined design will use less resources but will perform poorly in timing tests whereas designs that are pipelined to varying degrees will see improved timing at the cost of more resource usage. This also aligns with the information presented during lectures. Universally, any form of pipelining greatly reduced slack (i.e. setup time violations) but only the design which used a four layer multiplier passed the timing violation test and surpassed the desired frequency. The most surprising results were those of the manually pipelined design and the two layered multiplier. Both of the designs used the same amount of resources even with their different designs with the manually pipelined design performing slightly better on timing despite the multiplier design having its pipelined layout automatimatically layed out and, to an extent, optimized by the VHDL software.

Conclusion

As stated in the analysis of our resource and timing table in the section above, the various pipelined designs for the comlex square calculator aligned with our expectations. The non-pipelined control design used few resources but exhibited the largest timing violations but by adding more layers of logic, whether a few extra intermediate signal variables or components, the design’s timing improved at the cost of more FPGA resources.