ECSE 325 – Digital Systems

Winter 2020

Take-Home Exam: Pipelined Complex Multiplier and Design Closure

Overview

In this lab you will explore the effects of pipelining on the performance of synchronous digital systems. You will implement a circuit for squaring a 32-bit complex number, and try to maximize its speed using pipelining.

Complex Squaring

You will create a system to compute the square of a complex number. A complex number, Z, can be represented by two signed signals, one holding the REAL part, X, of the complex number, and the other holding the IMAGINARY part, Y, of the complex number.

$$Z = X + iY$$

The square of the complex number Z is another complex number given by:

$$Z\hat{}2 = (X+iY)*(X+iY) = (X\hat{}2-Y\hat{}2)+i(2XY)$$

A VHDL description of a circuit to compute the complex square is shown on the next page. It creates registers r_x, r_y to hold the real and imaginary parts of the input complex number, and registers o_xx, o_yy to hold the output complex number. Make sure you understand the circuit. Note that the multiplication and addition operations are all done in one clock cycle.

VHDL Description

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity complex_square is
port (
  i_clk      : in  std_logic;
  i_rstb     : in  std_logic;
  i_x        : in  std_logic_vector(31 downto 0);
  i_y        : in  std_logic_vector(31 downto 0);
  o_xx, o_yy : out std_logic_vector(64 downto 0));
end complex_square;
architecture rtl of complex_square is
signal r_x, r_y : signed(31 downto 0);
begin
p_mult : process(i_clk,i_rstb)
begin
  if(i_rstb='0') then
    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);
             r_y <= signed(i_y);
             o_xx <= std_logic_vector(('0'&(r_x*r_x)) - r_y*r_y);
             o_yy <= std_logic_vector(r_x*r_y & '0');
      end if;
end process p_mult;
end rtl;
```

Compiling the Design

1. Copy the vhdl description on the previous page into a file named gNN_complex_square.vhd (where NN is your group number).

2. Create a timing constraints (.sdc) file with a single create_clock statement, constraining the clock period to 5 nsec (equivalent to a 200MHz clock). Name the file gNN_complex_square.sdc.

3. Making sure that the FPGA device is assigned to the 5CSEMA5F31C6, compile the gNN_complex_square description.

4. After compilation, check the Compilation Report and note the resource usage (# of ALMs used, # of registers used, # of DSP Blocks).

5. Next, look at the TimeQuest Timing Analyzer Report. Does the design pass the timing test? Look at the "Slow 1100mv 0C Model". Note the Fmax value in the Fmax summary and the *i_clk* slack in the Setup Summary.

6. Enter these values into the table provided at the end of this document.

Pipelining the Design

1. Modify the VHDL description to add a pipeline register between the multiplication and subtraction operation.

2. Re-run the compilation and note the results of the resource usage and timing analysis. Enter the values into the table.

3. You should see increased resource usage, but also see an improvement in the speed of the circuit (increases in Fmax and the setup time slack).

4. Include your VHDL code and compilation report in the report.

Using the LPM_MULT Component

The Quartus compiler will make use of specific structures found on the target FPGA device to optimize the speed and size of the design. The Cyclone V family (and newer devices) contain special purpose embedded DSP Blocks that include two 18x19 bit multipliers and some addition circuits. The 5CSEMA5F31C6 device has 87 of these DSP Blocks that can be used to implement multiplication operations directly.

The 18x19 bit multipliers in this device are rated at a maximum clock speed of 287 MHz. Thus, we should be able to approach this speed with our circuit, provided we have adequate pipelining. At least we should be able to meet our target of a 200MHz clock rate.

Since we need 32x32 bit multiplications and each DPS Block only has 18x19 bit multipliers, these operations will need to combine multiple DSP Blocks. The DSP Blocks include within them programmable pipeline registers.

However, when you just use the "*" multiplication operator in VHDL, the compiler will not make use of the pipeline registers within the DSP Blocks. To enable the use of the pipeline registers in the DSP Blocks, we must use the LPM_MULT component.

To use the LPM_MULT component, you need to have 3 items in your VHDL code:

- Load and use the LPM library
- Declare the LPM_MULT component
- Instantiate the LPM_MULT component for each multiplication operation

Examples of the VHDL code for each of these steps are shown below. The *Generic* statements tell the compiler how to configure the various parameters of the component. For example, the **LPM_WIDTHA** parameter specifies the number of bits in the A input. You specify the particular value using the GENERIC MAP list as part of the component instantiation.

All data taken from the Cyclone V Datasheet. Feel free to consult it further at:
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v1.pdf

LPM_MULT Component Usage

```
LIBRARY lpm;
USE lpm.lpm_components.all;
--------------------- COMPONENT DECLARATION------------------------------------
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;
------------------------COMPONENT INSTANTIATION---------------------------------
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 );
```

Pipeline the LPM Design

1. Modify your VHDL description to replace each of your multiplication operations with the LPM_MULT components. Set the LPM_PIPELINE value to 2 (LPM_PIPELINE => 2)

2. Observe the two levels of pipelining for implementing a 32x32 bit multiplication operation.

3. Re-run the compilation and note the results of the resource usage and timing analysis. Enter the values into the table.

4. You should see increased resource usage, but also see an improvement in the speed of the circuit (increases in Fmax and the setup time slack).

5. Show your VHDL code and compilation report to the TA.

6. Repeat the compilation with the LPM_PIPELINE value set to 3, then again with the value set to 4.

7. Note the results of the resource usage and timing analysis. Enter the values into the table.

8. By doing the pipelining with the LPM_MULT component, can you achieve the timing goals of a 200 MHz clock?

Lab Report

The report must include the following items:

- A header listing the group number, the names and student numbers of each group member.
- A description of the circuit function, listing the inputs and outputs.
- The VHDL description of all circuits designed in this lab (also include the .vhd files in the assignment submission zip file).
- Fully populated resource and timing table from the next page in the report. Provide a discussion of results obtained and the design tradeoffs.

The lab report, and all associated design files must be submitted, as an assignment to the myCourses site. Only one submission need be made per group.

Combine all of the files that you are submitting into one *zip* file, and name the zip file *gNN_LAB_4.zip* (where NN is your group number).

FPGA Resource Usage and Timing Analysis Summary Table

	No Pipelining	Pipelined	lpm_mult P = 2	lpm_mult P = 3	Lpm_mult P=4
# ALMs					
# Registers					
# DSP Blocks					
Fmax					
Slack					

Bonus: Connecting FPGA Logic to the Processor

Any group demonstrating the use of the above FPGA circuitry as a complex number co-processor by the ARM processor inside the chip will get the bonus points, depending on the complexity of the connection. Of all the ways to attached the FPGA logic to the processor shown in the class, one (which one?) should be the most useful for attaching this logic.

Specifically, if a connection is realized in one direction (processor providing the data to the complex multiplier), there will be 10% bonus points.

If both the inputs and outputs of the FPGA circuit are accessed by the processor, there will be 15% bonus points. In both cases, simple processing on the processor side (e.g., sending the data, receiving, validating the results) should suffice – the emphasis is not on complex embedded software, but on demonstrating the ability to access the FPGA logic by Avalon interfaces.

If the group manages to apply the more complex software processing, such as the Mandelbrot fractal from the last lecture, additional 5% bonus points will be added.

Grading Sheet

Group Number:
Name 1:
Name 2:

Task	Grade	/Total	Comments
VHDL non-pipelined		/25	
VHDL pipelined		/25	
VHDL with LPM		/25	
Resource usage, timing table		/25	