McGill University

ECSE 325 - Lab1 Basic Mapping VHDL to FPGA Hardware

Group 2

Yi Zhu 260716006
Mai Zeng 260782174

1 Introduction

In this lab, the main task is to learn the basics of compiling synchronous circuit VHDL description to a target FPGA. The goal of this lab exercise is to become familiar with the Quartus tool, especially dealing with how compiler maps the design onto the FPGA hardware.

2 VHDL

2.1 VHDL Code

The VHDL code is shown below in Figure 2.1.1.

VHDL is a description code for describing the information and functionality for a hardware chip. It is important to understand the structure of the code so that the description of the hardware and the result of the real hardware would not have any mistakes. As a modern computer language, it is necessary to import some existing libraries so that the developer can directly use the functionalities in those libraries. The sub library STD_LOGIC_1164 in IEEE is used. This is a set of packages defining commonly used data types and operations (STD_LOGIC type and STD_LOGIC_VECTOR type are defined in the library).

The entity declaration describes the circuit as it appears from the "outside" only the hardware's inputs and outputs. The entity part can be treated as a declaration as being analogous to a block symbol on a chip schematic. In this lab, the inputs for this hardware are clk (clock), countbytwo, rst (reset), enable and the output is an 8-bit std_logic_vector.

The requirements indicated that the hardware is a basic 8 bit counter and has a counting by two feature so every time the chip detects a rising edge, the temporary counter that we set (line 14) for the hardware to the output should increase by 1 or 2 based on the value of the countbytwo.

The requirements indicated that the reset is synchronous but because that the hardware should only check the reset on the clock rising edge so the rst should not be in the sensitivity list. This is why we bring the check for rst in the process and check at the synthesized gate-level.

The requirements also indicated that the hardware should only have even outputs when the countbytwo is set to high. This brings out two conditions:
1. The previous result is even so the hardware should simply add two.
2. The previous result is odd so the hardware should add only one in order to have the even output.
In our code, line 27 to line 36 are the implementation. At every rising edge of the clock we will use mod to check whether the temporary counter is an odd number or an even number and based on the condition we add 1 or 2.

The requirements said that the counter should never over count. The code would never over count since we have 8 bit so the max value of the counter output is 11111111 which is 255. In the case that it meets the 255 no matter it adds by 1 or 2 it will overflow to 1 or 2 but will never over count. Because we only have 8 bit. The figure 2.1.2 shows the example that when it meets 254 (11111110) and the countbytwos is 1 so it should add 2 to meet 256 (100000000) but because the temporary counter we set is 8 bit so it can only output 0.

At the end of the process, simply cast the temporary counter to std_logic_vector and set it two output.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity g02_lab1 is
  Port(        clk          :in std_logic;
               countbytwo   :in std_logic;
               rst          :in std_logic;
               enable       :in std_logic;
               output       :out std_logic_vector(7 downto 0));
end g02_lab1;

architecture counter8 of g02_lab1 is
  signal count: unsigned(7 downto 0) := "00000000"; -- set it to unsigned integer and we have 8 bits

begin
   process(clk) -- rst is asyn so it should not be in the sensitivity list
   begin
      if rising_edge(clk) then -- check every rising edge
         if rst = '1' then -- rst is asyn so should belong to this if
            count <= "00000000"; -- count goes back to 0 after reset
         elsif enable='1' then
            -- if countbytwo is active and count is currently odd so only add by one to make an even output
            -- if the count is even so add 2
            -- we do not need to check whether the count is smaller than 255 or not
            -- since if the count is overflow, the least significant 8 bits will go back to 0
            if countbytwo = '1' then
               if (count mod 2) = 0 then
                  count <= count + 2;
               else
                  count <= count + 1;
               end if;
            -- if countbytwo is not active then just add 1
            else
               count <= count + 1;
            end if;
         end if;
      end if;
   end process;
   output <= std_logic_vector(count);
end counter8;
```

Figure 2.1.1 VHDL Code

Figure 2.1.2 Overflow but not Overcount

3 Compilation Result

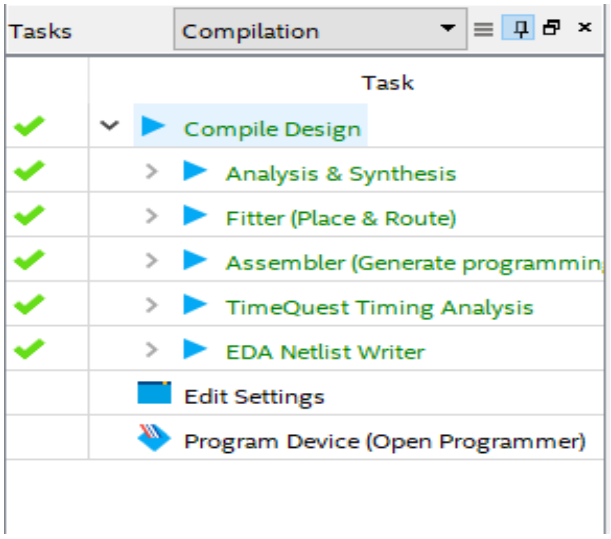
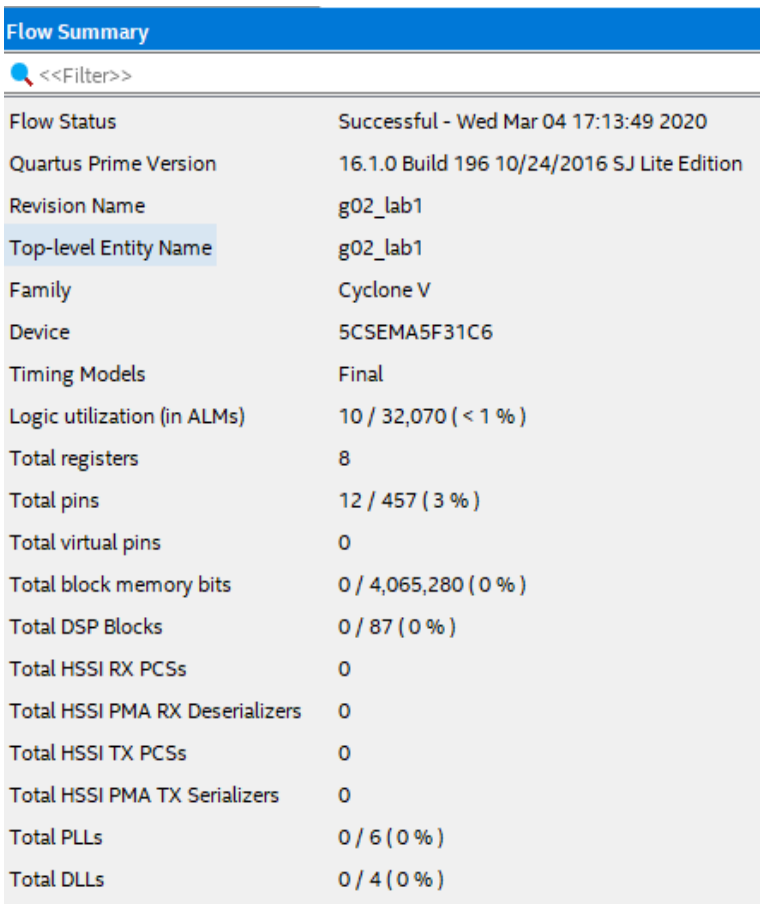


Figure 3.1 Compilation Result



Flow Summary

<<Filter>>

Flow Status	Successful - Wed Mar 04 17:13:49 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g02_lab1
Top-level Entity Name	g02_lab1
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	10 / 32,070 (< 1 %)
Total registers	8
Total pins	12 / 457 (3 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
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 %)

Figure 3.2 Flow Summary

Figure 3.1 and Figure 3.2 show the compilation results and also the summary.

4 Resource Utilization

4.1 RTL View

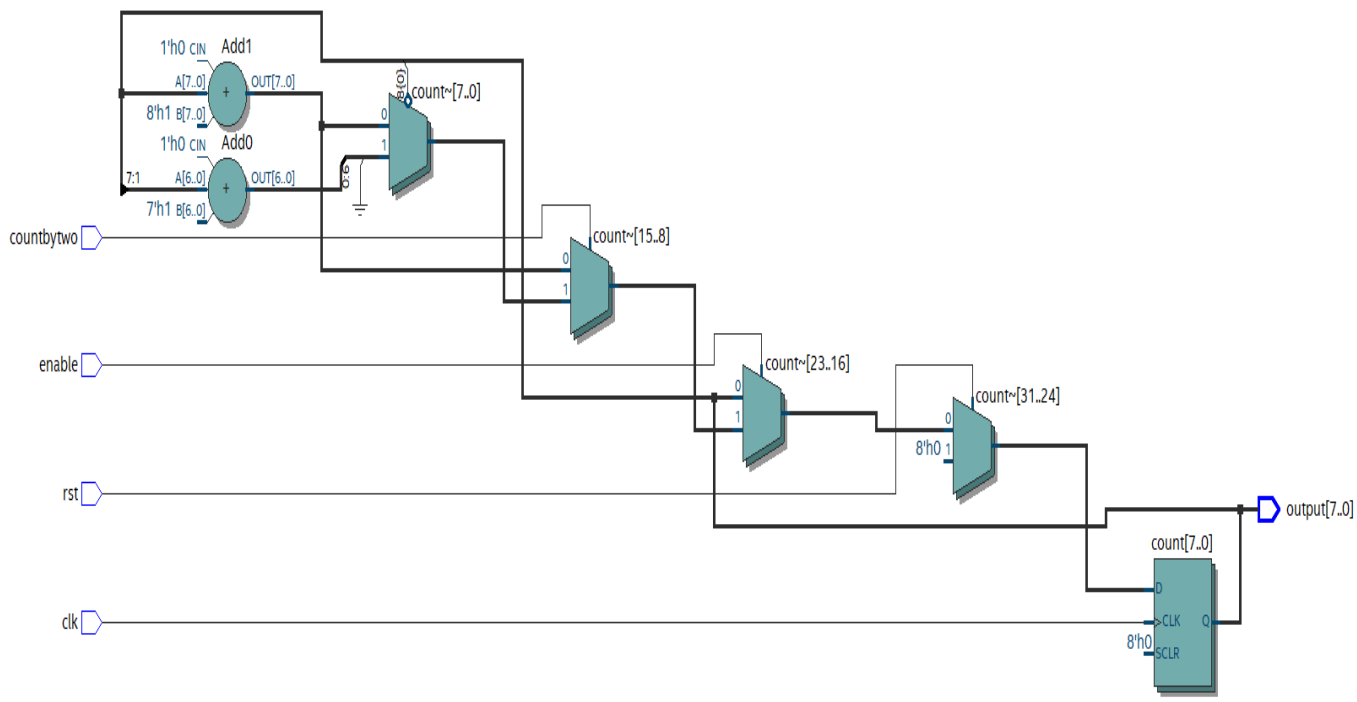


Figure 2.1.1 RTL Viewer 1

The RTL viewer showed that there are 4 2:1 Mux and 1 D flip-flop needed. The 4 Mux are used to select the condition of reset, enable, count by two and the adder. In the flow summary we can see that the logic utilization (in ALMs) are 10/32070 which is the estimation of how full the device is. The Adaptive Logic Module (ALM) is the basic building block of supported device families and is designed to maximize performance and resource usage. We need 8 registers and 12 pins for the device according to the flow summary.

We need the 8 registers to for the synchronous 8-bit counter. If we want to increase the bit we need to increase the number of registers. For example, if we want 10 bit counter we need 10 registers.

4.2 Chip Planner

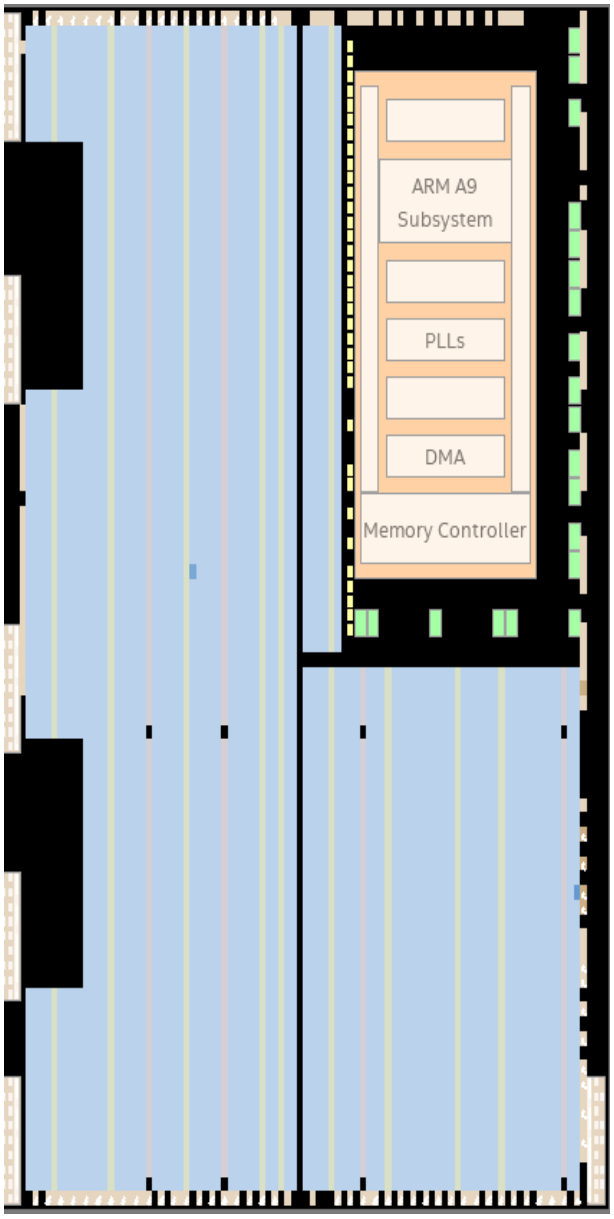


Figure 2.2.1 Chip Planner

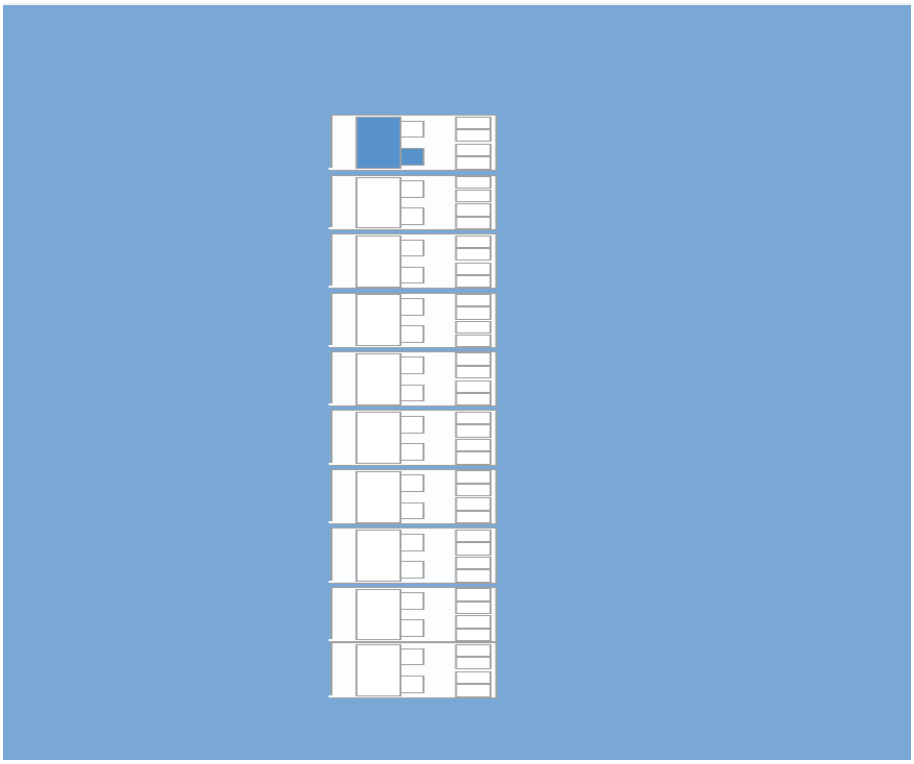


Figure 2.2.2 Used Resources Highlight

5 Conclusion

In this lab, we learned the basic building blocks of using FPGA coding software and the mapping techniques of hardware description. The total resources are only used in a very small portion because an 8-bit counter is a fairly small hardware but we can gain a better understanding of the edge-triggering event handling and get prepared for more complicated design.