Comp E 470L

Digital Systems

Lab 4: GCD

Student: DAVIT BARBAKADZE

Red ID #: 822164584

10.03.2020

Contents

[Task Description 2](#_Toc22460835)

[Solution 2](#_Toc22460836)

[Simulation & Verification 7](#_Toc22460837)

[Comparison 9](#_Toc22460838)

[Conclusion 10](#_Toc22460839)

# Task Description

In this laboratory assignment we had to implement logic for finding GCD of two 8bit 2’s complement numbers. Our code **must** generate Finite State Machine (FSM) either Moore or Mealy, and for each button press, it executes specific state code, take inputs, calculate value of GCD, etc. We had to show the result on LEDs or seven segment display. We had to use two input numbers (signed 2’s complement), use buttons to initialize them and find logic to calculate GCD of them. The project also had requirement that the maximum possible clock frequency should be more than 100MHz. We had to write code, simulate, check if requirements are satisfied and implement it on the board.

# Solution

At first I came up with the logic of finding GCD, at first I tried to use modulus operation but as it turned out modulus (%) is not supported in implementation so I used logic which uses minus sign instead. See logic in Figure 1.

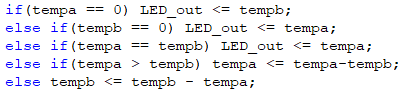


Figure 1 GCD logic

Figure 1 shows the logic behind calculating the GCD of two numbers, tempa and tempb. If one of the two numbers is 0, we know that nonzero number is the GCD of those two numbers, so I took it into account. In C code, we need while loop to calculate the GCD with same logic but in our case I used always block so the machine does one instruction per clock cycle, so it requires multiple clock cycles to find GCD, we could also use task and recursion in it but, in that case combinational path delay would be higher and the requirement of the project would not be satisfied.

Next, as a requirement of our project, I learned how to make the machine instantiate FSM in the hardware, so I used case statement with following states: take, calculate and hold. I will go through the code step by step now.

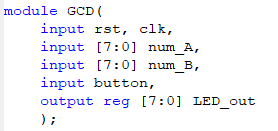


Figure 2 module instantiation

At first I instantiated module GCD and initialized all the I/Os in it. Rst for the reset signal, clk for the clock signal, num\_A and num\_B for inputs, button to change states of FSM and output LED\_out which would be connected to board’s leds in implementation.

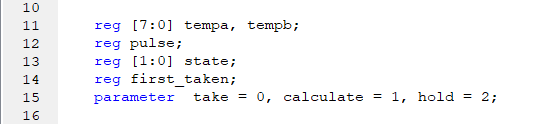


Figure 3 helper variables

Then I created helper variables to use in code, tempa and tempb are registers that will hold the value of inputs, as inputs needed to be changed in our code logic, I used registers where changes are acceptable, also as input numbers are in two’s complement, we had to change the sign of inputs with two’s complement conversion and use them to find GCD. How I achieved conversion will be shown later. Also, pulse register is used to get button signal only once even if user presses for a long time, this logic was also used in previous labs. State register is used to control FSM states, first\_taken register checks if first input is already inserted, if so, it inserts second input. Parameters are just used to visualize state names.

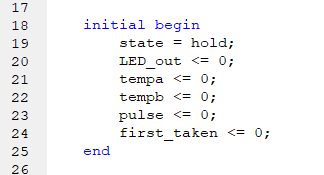


Figure 4 initialization

In this part of code I initialized values of all the registers, initial state is hold state as the machine should wait for button click to take inputs from switches. LEDs are initially off, tempa and tempb 0, as well as pulse Is 0 as button has not been clicked yet and finally first\_taken is 0 as first input has not been sent to first register tempa.

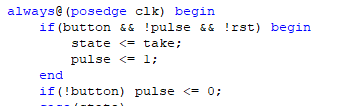


Figure 5 button signal pulse generator

This logic is used in previous labs as well, which prevents button from executing part of a code multiple times caused by long-time press. Pulse becomes 0 only when push button is released (== 0). When button is pressed, the machine gets the value of input from switches and stores it to tempa.

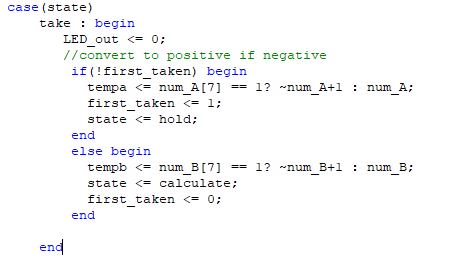


Figure 6 take state

When button is pressed and take state becomes active, at first LEDs are off so we know another inputs are about to be inserted. Then the program checks whether the input is first input or the second, also it checks if the sign of the number is negative (if leftmost bit is 1) and converts to positive - using two’s complement conversion (~A + 1). When first input is initialized, the FSM goes to hold state to wait for next button press to insert second input, if second is inserted, it goes to calculate state which we have already shown in figure 1.

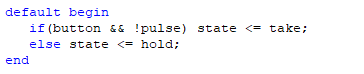


Figure 7 hold state

The hold state Is the default state of FSM, which does nothing but wait for button click to go on the take stage or reset signal to reset the program.

Full code can be seen in attached .V files.

------------------------------- TESTBENCH CODE ----------------------------------------

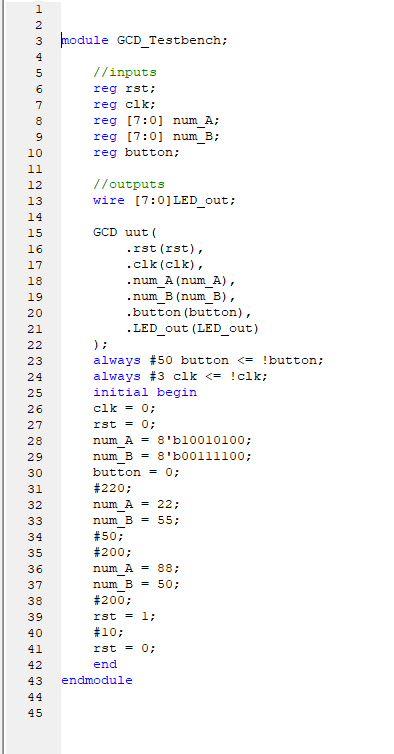
] 

Figure 8 GCD test bench code

Test bench code shows module and signal instantiation for simulation. We have changed input signals for test purposes.

---------------------------------- CONSTRAINTS CODE ----------------------------------------

set\_property PACKAGE\_PIN T22 [get\_ports {LED\_out[0]}]; # "LD0"

set\_property PACKAGE\_PIN T21 [get\_ports {LED\_out[1]}]; # "LD1"

set\_property PACKAGE\_PIN U22 [get\_ports {LED\_out[2]}]; # "LD2"

set\_property PACKAGE\_PIN U21 [get\_ports {LED\_out[3]}]; # "LD3"

set\_property PACKAGE\_PIN V22 [get\_ports {LED\_out[4]}]; # "LD4"

set\_property PACKAGE\_PIN W22 [get\_ports {LED\_out[5]}]; # "LD5"

set\_property PACKAGE\_PIN U19 [get\_ports {LED\_out[6]}]; # "LD6"

set\_property PACKAGE\_PIN U14 [get\_ports {LED\_out[7]}]; # "LD7"

## User DIP Switches - Bank 35

## ----------------------------------------------------------------------------

set\_property PACKAGE\_PIN F22 [get\_ports {num\_A[0]}]; # "SW0"

set\_property PACKAGE\_PIN G22 [get\_ports {num\_A[1]}]; # "SW1"

set\_property PACKAGE\_PIN H22 [get\_ports {num\_A[2]}]; # "SW2"

set\_property PACKAGE\_PIN F21 [get\_ports {num\_A[3]}]; # "SW3"

set\_property PACKAGE\_PIN H19 [get\_ports {num\_A[4]}]; # "SW4"

set\_property PACKAGE\_PIN H18 [get\_ports {num\_A[5]}]; # "SW5"

set\_property PACKAGE\_PIN H17 [get\_ports {num\_A[6]}]; # "SW6"

set\_property PACKAGE\_PIN M15 [get\_ports {num\_A[7]}]; # "SW7"

# ----------------------------------------------------------------------------

# User Push Buttons - Bank 34

# ----------------------------------------------------------------------------

set\_property PACKAGE\_PIN P16 [get\_ports {rst}]; # "BTNC"

set\_property PACKAGE\_PIN R16 [get\_ports {button}]; # "BTND"

set\_property PACKAGE\_PIN Y9 [get\_ports {clk}]; # "GCLK"

# ----------------------------------------------------------------------------

# IOSTANDARD Constraints

#

# Note that these IOSTANDARD constraints are applied to all IOs currently

# assigned within an I/O bank. If these IOSTANDARD constraints are

# evaluated prior to other PACKAGE\_PIN constraints being applied, then

# the IOSTANDARD specified will likely not be applied properly to those

# pins. Therefore, bank wide IOSTANDARD constraints should be placed

# within the XDC file in a location that is evaluated AFTER all

# PACKAGE\_PIN constraints within the target bank have been evaluated.

#

# Un-comment one or more of the following IOSTANDARD constraints according to

# the bank pin assignments that are required within a design.

# ----------------------------------------------------------------------------

# Note that the bank voltage for IO Bank 33 is fixed to 3.3V on ZedBoard.

set\_property IOSTANDARD LVCMOS33 [get\_ports -of\_objects [get\_iobanks 33]];

# Set the bank voltage for IO Bank 34 to 1.8V by default.

# set\_property IOSTANDARD LVCMOS33 [get\_ports -of\_objects [get\_iobanks 34]];

# set\_property IOSTANDARD LVCMOS25 [get\_ports -of\_objects [get\_iobanks 34]];

set\_property IOSTANDARD LVCMOS18 [get\_ports -of\_objects [get\_iobanks 34]];

# Set the bank voltage for IO Bank 35 to 1.8V by default.

# set\_property IOSTANDARD LVCMOS33 [get\_ports -of\_objects [get\_iobanks 35]];

# set\_property IOSTANDARD LVCMOS25 [get\_ports -of\_objects [get\_iobanks 35]];

set\_property IOSTANDARD LVCMOS18 [get\_ports -of\_objects [get\_iobanks 35]];

# Note that the bank voltage for IO Bank 13 is fixed to 3.3V on ZedBoard.

set\_property IOSTANDARD LVCMOS33 [get\_ports -of\_objects [get\_iobanks 13]];

NOTE that since we have only 8 switches on board, we had to insert both num\_A and num\_B from the same input, so in implementation we only insert imput values from num\_A, and in the calculating process, first input goes to tempa, and another goes to tempb.

# Simulation & Verification

Verilog code and test bench is already shown in above sections, in this section I will show simulation result and try to explain how it works.

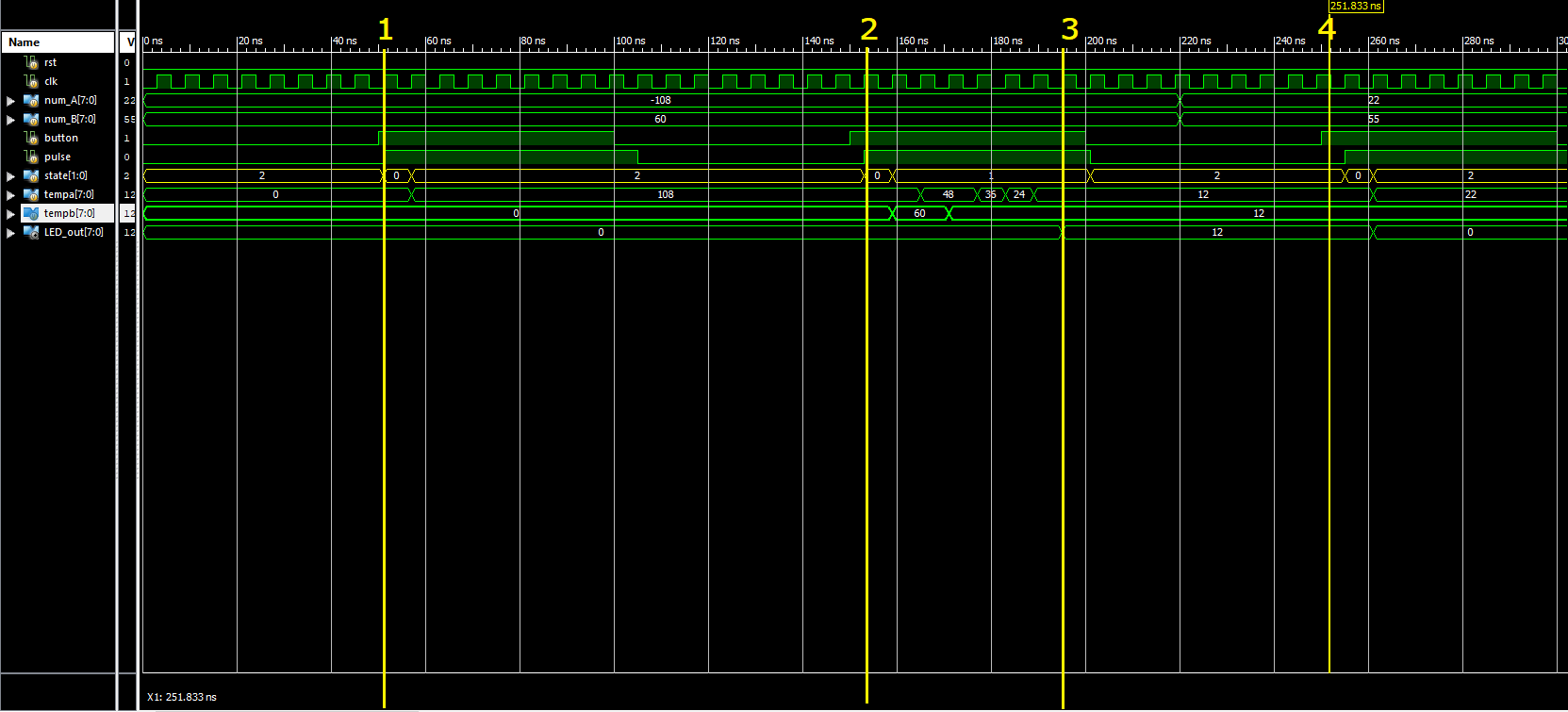


Figure 9 simulation result

This is just a part of simulation which shows the result partly, the full test will be shown in the implemented code, via video recording.

I have divided the simulation segments with yellow lines numbering from 1 to 4. At the start of the simulation as expected initial state is hold state and the program is waiting for a button to be pressed, when it Is pressed at line 1, pulse signal is activated and button signal is read only once, which send the program on the take stage and the value of first input is sent to tempa, as it is of a negative value, program also converts it to positive. When tempa gets the converted value of first input, the FSM again goes to state 2 – hold state and waits for the next push button. When it is pressed at yello line #2, program goes to take state and takes another input, as first register is already set, it goes to second register tempb. Take stage is over and the FSM goes to calculate stage. On third yellow line calculation is finished and the result goes to LEDs, and turns them on. When LEDs are on, the program again goes to hold state and waits for another button push to take other inputs from num\_A and num\_B.

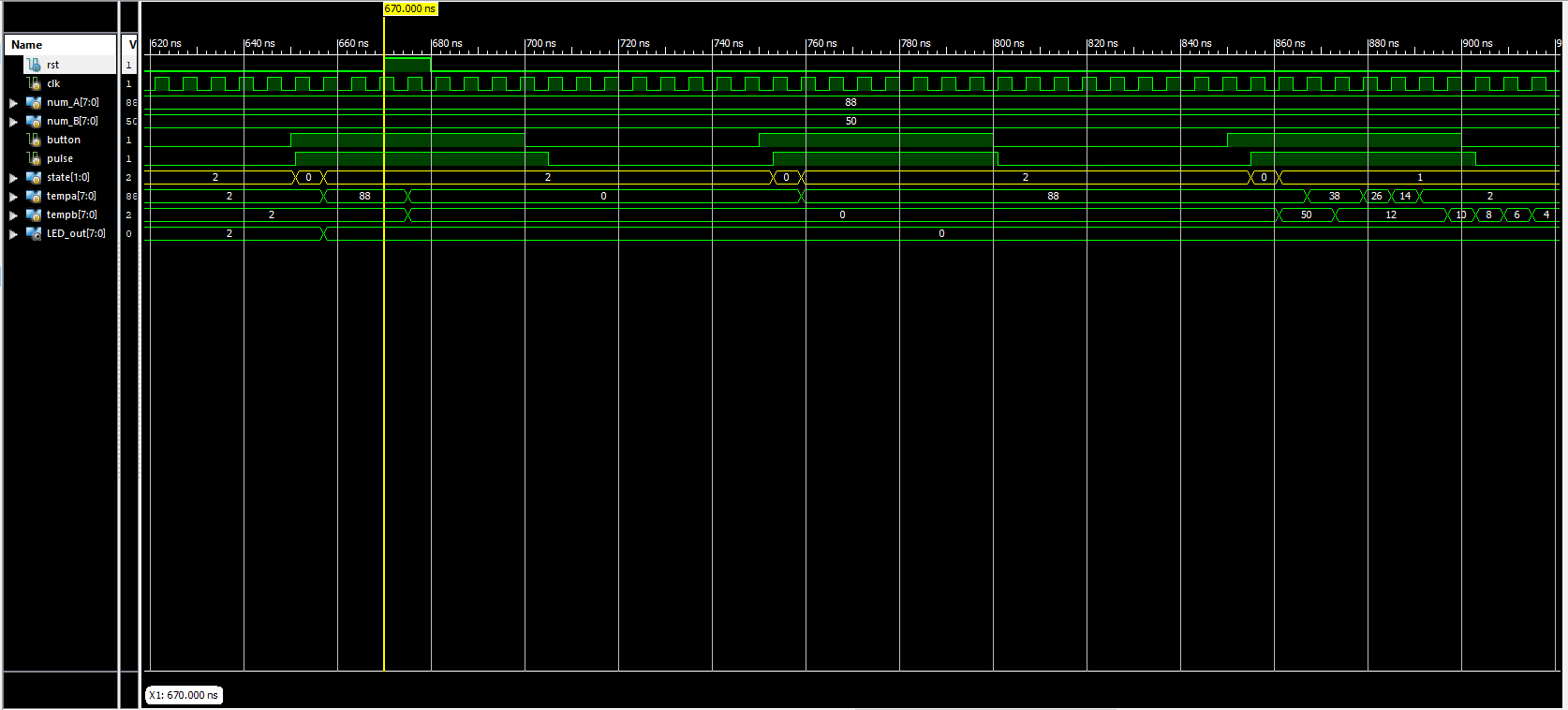


Figure 10 reset signal check

Figure 10 shows the reset signal at yellow line, as we see it resets the values of tempa and tempb, also sets the state to 2(hold) and turns LEDs off. When button is pressed, the program again starts working regularly.

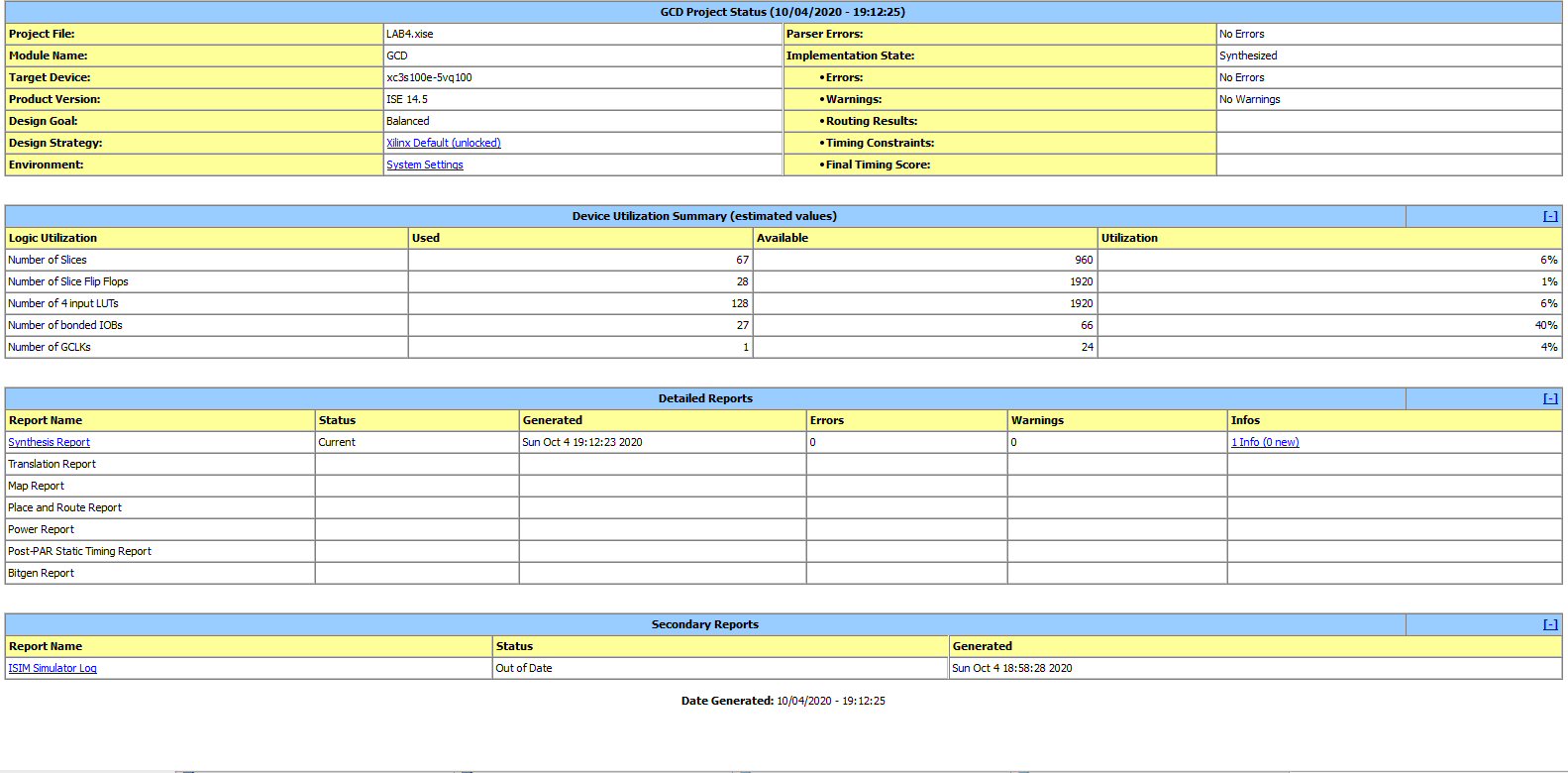


Figure 11 Project status

Project status show that there are no errors or warnings in this program.

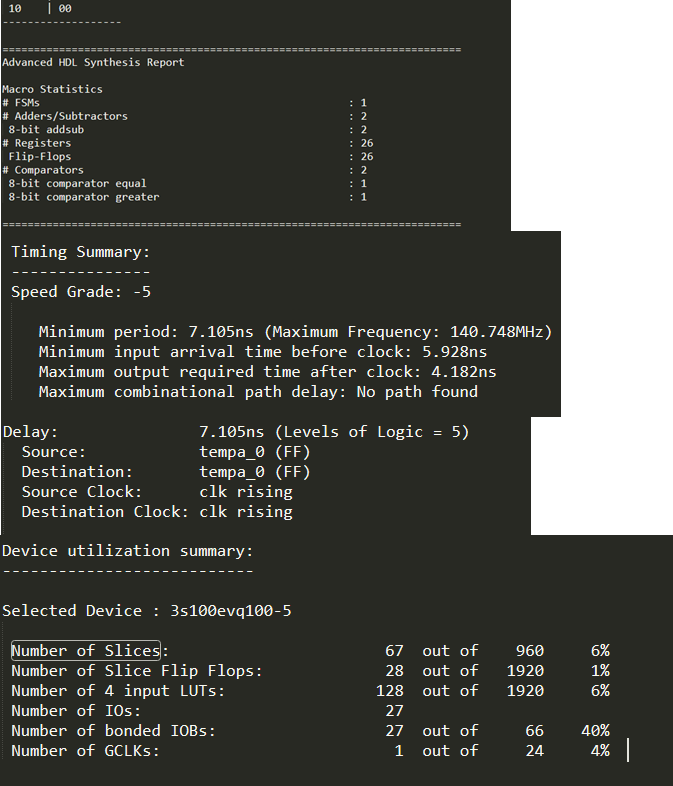


Figure synthesis report

As we see on figure 12, FSM is correctly identified and maximum frequency is higher than 100MHz. so requirements are satisfied. All other information about report can be seen in attached .syr file.

Video recording of implemented code can be seen on the following link:

https://youtu.be/KgbdMehkZ9A

# Comparison

The code is efficient in terms of maximum clock frequency, we could also use task or function with recursion, but, to my mind, it would rise up the combinational path delay and requirement would not be satisfied. In our code case, each clock cycle does fewer operations and maximum clock frequency is higher.

# Conclusion

In this laboratory we learned how to think about the mathematical equation logic, translate it to Verilog code and implement it on hardware. The lab was quite challenging as we had to make the program identify FSM, but from this lab we gathered enough knowledge to use FSM as we like in the following assignments. Finally, using hardware to make completes mathematical equations was hard but interesting.