GREATEST COMMON DIVISOR

Electronic Systems' Project

Abstract

Design and development of a GCD hardware using Xilinx FPGA Zync platform

CLAUDIO DAKA – A.A. 2022-2023

Index

Introduction

Functional Requirements

The Greatest Common Divisor that we want to design is a device that, given two numbers on the input, A and B, it can calculate the MCD using the Euclidean method. Since this algorithm must be implemented in hardware and the division operation would be too complex to be implemented, we reformulate to use only subtraction operations.

If A and B are both non-zero, the algorithm operates as follows:

If A is greater than or equal to B, the algorithm subtracts B from A and assigns the result to A; it repeats until A becomes less than B (at the end of this operation, the value of A coincides with the remainder of the division A/B). At this point, it swaps the two numbers, so that A is again greater than B, and repeats the operations described. It exits the loop when B (i.e. the remainder of the last division performed) becomes zero, assigning to the output Y the value of A (the last divisor used), which is the MCD sought.

```
1. function GCD(A, B){
2.      if (A == 0)
3.          return B;
4.      else if (B == 0)
5.          return A;
6.      while A ≠ B {
7.          if A > B
8.              A = A – B;
9.          else
10.             B = B – A;
11.     }
12.     return A;
13. }
```

Non-Functional Requirements

The device is synchronized by an external clock signal and a reset signal must be used to initialize it.

A, ***B*** and ***Y*** width is n (generic value).

We use ***load*** for the input data valid, and ***done*** for the output data valid

For a coherence purpose, we want the two outputs of the system (Y and done) to respond to the input at the same time.

No time constraints are requested for this application; in fact, when inputs are given, we cannot expect the output to immediately respond.

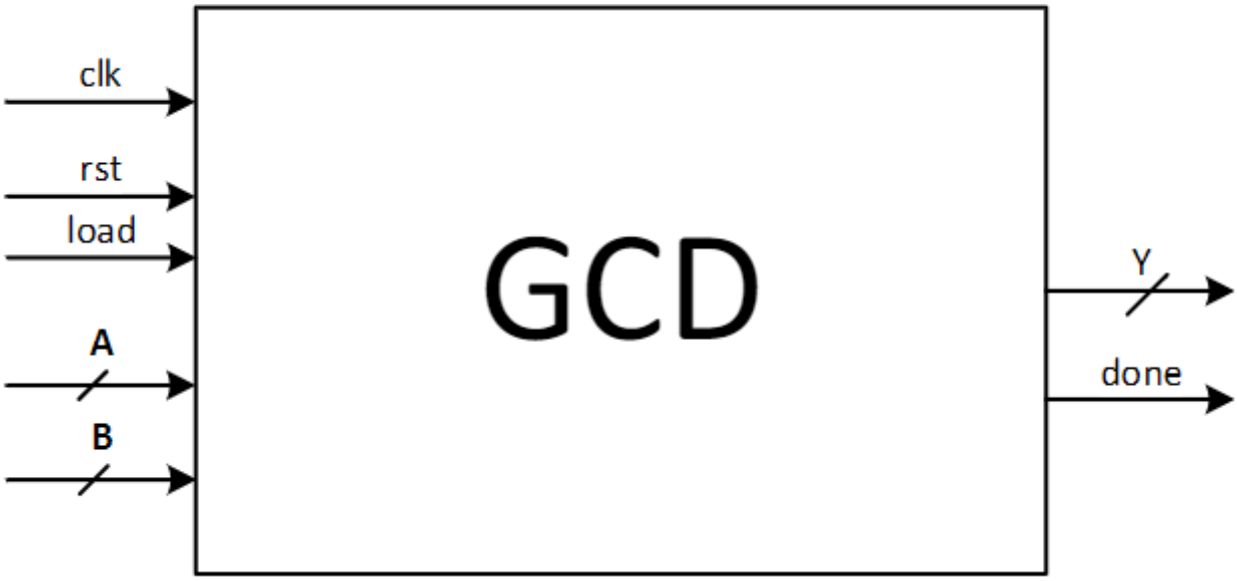
Possible Employments

The GCD is used for a number of applications both simple and complex. In fact, you've likely implicitly calculated GCDs without recognizing it when simplifying fractions: reducing a fraction is a matter of dividing both the numerator and denominator by their GCD.

The GCD is also used in the extended Euclidean algorithm to compute modular inverses, which are of extreme importance in encryption schemes such as RSA. It is additionally rather important when considering the order of an element, particularly in Lagrange's theorem especially as applied to modular arithmetic. This makes it a common topic in competitions and Olympiads.

Architectural Design

Interface



Input and Output ports

Input ports:

- **clk**: external clock signal to synchronize the device;
- **rst**: external reset signal to initialize/reset the device;
- **load**: external signal for the input data valid;
- **A**: external signal for the 1st number;
- **B**: external signal for the 2nd number.

Output ports:

- **Y**: it is the main output of the device, it is the MCD of the 2 numbers;
- **done**: external signal for the output data valid.

VHDL Code

gcd.vhd – Greatest Common Divisor VHDL description

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;

entity gcd is
    generic (
        Nbit : positive := 8
    );
    port(
        clk : in std_logic;
        rst : in std_logic;
        load : in std_logic;
        a_in : in std_logic_vector (Nbit-1 downto 0);
        b_in : in std_logic_vector (Nbit-1 downto 0);
        done : out std_logic;
        gcd_out : out std_logic_vector (Nbit-1 downto 0)
    );
end entity;

architecture arch of gcd is
    signal a : std_logic_vector (Nbit-1 downto 0 );
    signal b : std_logic_vector (Nbit-1 downto 0 );
    constant all_zeros : std_logic_vector(Nbit-1 downto 0) := (others => '0');

    begin


GCD_PROCESS: process(clk, rst)
        variable calc: std_logic;
        variable done_var: std_logic;
        begin
            if rst = '0' then
                a <= (others => '0');
                b <= (others => '0');
                gcd_out <= (others => '0');
                done_var := '0';
                calc := '0';
            elsif rising_edge(clk) then
                done_var := '0';
                if load = '1' and calc = '0' then
                    a <= a_in;
```

```
                    b <= b_in;
                    calc := '1';
                elsif calc = '1' then
                    if a = b then
                        gcd_out <= a;
                        done_var := '1';
                        calc := '0';
                    elsif a = all_zeros then
                        gcd_out <= b;
                        done_var := '1';
                        calc := '0';
                    elsif b = all_zeros then
                        gcd_out <= a;
                        done_var := '1';
                        calc := '0';
                    elsif a < b then
                        b <= b - a;
                    else
                        a <= a - b;
                    end if;
                end if;
            end if;
            done <= done_var;
    end process GCD_PROCESS;
end arch;
```

Testing

Test Plan

The system must respond correctly to different use cases:

- A and B are not co-prime;
- B is 0;
- A is 0;
- Both A and B are 0;
- A and B are co-prime;

#	Input	Expected response
1	A <= 00000110 B <= 00100100	Y <= 00000110
2	A <= 00100100 B <= 00000000	Y <= 00100100
3	A <= 00000000 B <= 00000110	Y <= 00000110
4	A <= 00000000 B <= 00000000	Y <= 00000000
5	A <= 00000111 B <= 00100100	Y <= 00000001

Tested Cases

In the test bench written for testing the GCD, all the cases of the test plan will be implemented.

gcd_tb.vhd – Testbench Description

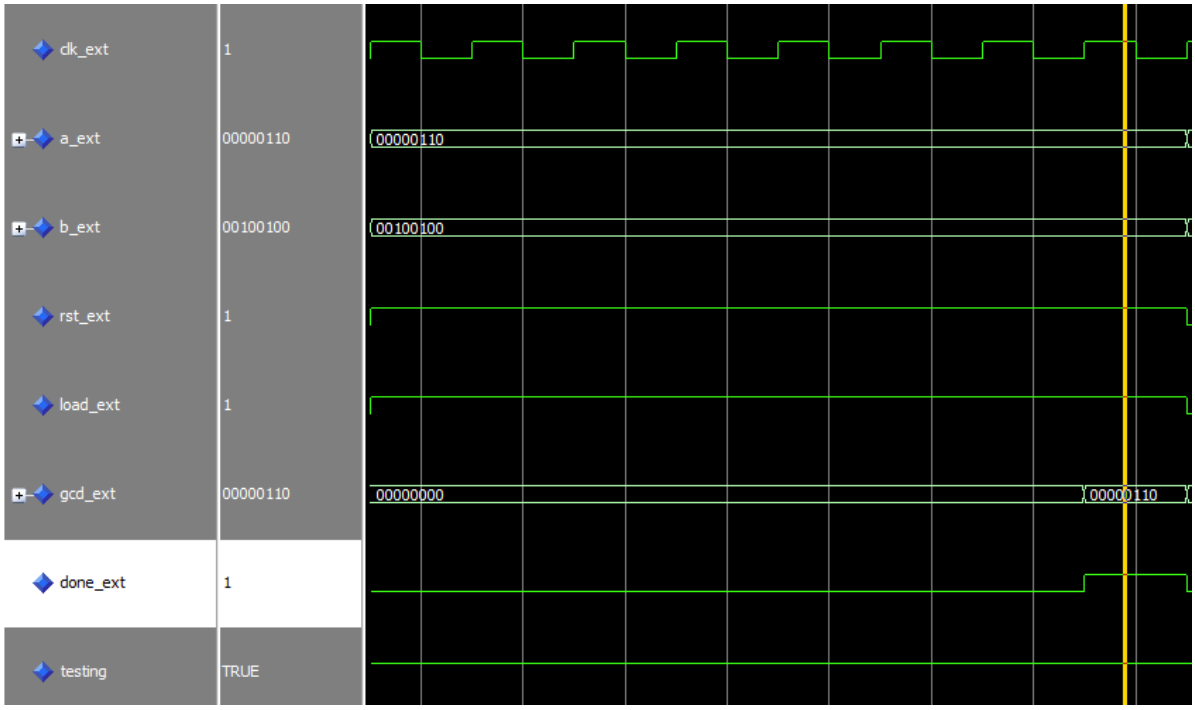
```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity gcd_tb is
end entity;

architecture tb of gcd_tb is

  constant clk_period : time     := 100 ns;
  constant N          : positive := 8;

  component gcd is
    generic (
      Nbit : positive := 8
    );
    port (
        clk : in std_logic;
        rst : in std_logic;
        load : in std_logic;
        a_in : in std_logic_vector (Nbit-1 downto 0);
        b_in : in std_logic_vector (Nbit-1 downto 0);
        done : out std_logic;
        gcd_out : out std_logic_vector (Nbit-1 downto 0)
    );
  end component;

  signal clk_ext    : std_logic := '0';                                    --
clock signal, intialized to '0'
  signal a_ext      : std_logic_vector(N-1 downto 0) := (others => '0');   --
a signal to connect to the a_in port of the component
  signal b_ext      : std_logic_vector(N-1 downto 0) := (others => '0');   --
b signal to connect to the b_in port of the component
  signal rst_ext    : std_logic := '1';                                    --
reset signal
  signal load_ext   : std_logic := '0';                                    --
load signal to connect to the load port of the component: it is the input data
valid
  signal gcd_ext    : std_logic_vector(N-1 downto 0);                      --
gcd signal to connect to the gcd_out port of the component
  signal done_ext   : std_logic := '0';                                    --
done signal to connect to the done port of the component: it is the output data
valid
```

```
  signal testing    : boolean := true;                                    --
signal to use to start/stop the simulation when there is nothing else to test

begin

  clk_ext <= not clk_ext after clk_period/2 when testing else '0';  -- The clock
toggles after clk_period / 2 when testing is high.
                                                                    -- When testing
is forced low, the clock stops toggling and the simulation ends.

DUT: gcd
    generic map (
      Nbit => N
    )
    port map (
        clk => clk_ext,
        rst => rst_ext,
        load => load_ext,
        a_in => a_ext,
        b_in => b_ext,
        done => done_ext,
        gcd_out => gcd_ext
    );

    stimuli: process
    begin
      --Initialization
      wait until rising_edge(clk_ext);
      rst_ext <= '0';
      wait until rising_edge(clk_ext);
      --1st test: MCD between 6 and 36 --> Desired output: 00000110 (6)
      rst_ext <= '1';
      a_ext    <= "00000110";
      b_ext    <= "00100100";
      load_ext <= '1';
      wait until falling_edge(done_ext);
      --Reset the device
      rst_ext <= '0';
      a_ext <= (others => '0');
      b_ext <= (others => '0');
      load_ext <= '0';
      wait until rising_edge(clk_ext);
      --2nd test: MCD between 32 and 0 --> Desired output: 00100000 (32)
      rst_ext <= '1';
```

```
      a_ext    <= "00100000";
      b_ext    <= "00000000";
      load_ext <= '1';
      wait until falling_edge(done_ext);
      --Reset the device
      rst_ext <= '0';
      a_ext <= (others => '0');
      b_ext <= (others => '0');
      load_ext <= '0';
      wait until rising_edge(clk_ext);
      --3rd test: MCD between 0 and 6 --> Desired output: 00000110 (6)
      rst_ext <= '1';
      a_ext    <= "00000000";
      b_ext    <= "00000110";
      load_ext <= '1';
      wait until falling_edge(done_ext);
      --Reset the device
      rst_ext <= '0';
      a_ext <= (others => '0');
      b_ext <= (others => '0');
      load_ext <= '0';
      wait until rising_edge(clk_ext);
      --4th test: MCD between 0 and 0 --> Desired output: 00000000 (0)
      rst_ext <= '1';
      a_ext    <= "00000000";
      b_ext    <= "00000000";
      load_ext <= '1';
      wait until falling_edge(done_ext);
      --Reset the device
      rst_ext <= '0';
      a_ext <= (others => '0');
      b_ext <= (others => '0');
      load_ext <= '0';
      wait until rising_edge(clk_ext);
      --5th test: MCD between 7 and 36 --> Desired output: 00000001 (1)
      rst_ext <= '1';
      a_ext    <= "00000111";
      b_ext    <= "00100100";
      load_ext <= '1';
      wait until falling_edge(done_ext);
      testing <= false;
    end process;
end architecture;
```

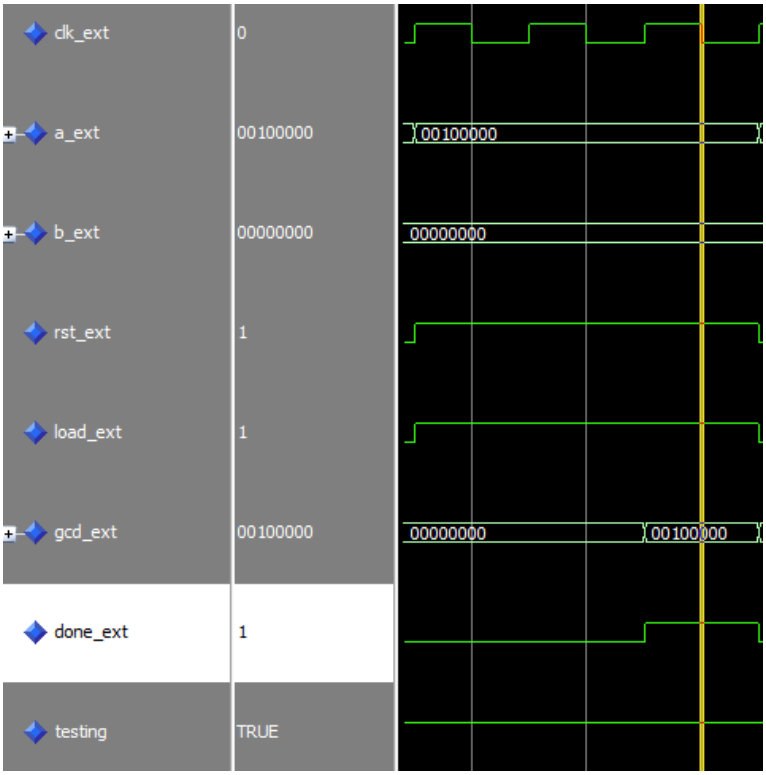
Results of the Tests

Case A <= 00000110, B <= 00100100



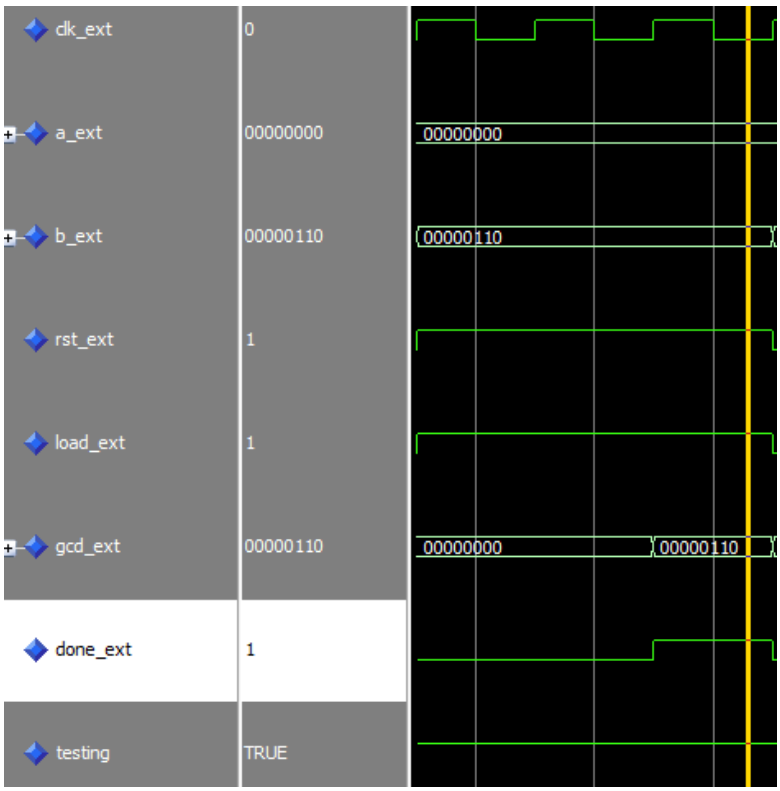
As expected, we have **00000110** as output.

Case A <= 00100000, B <= 00000000



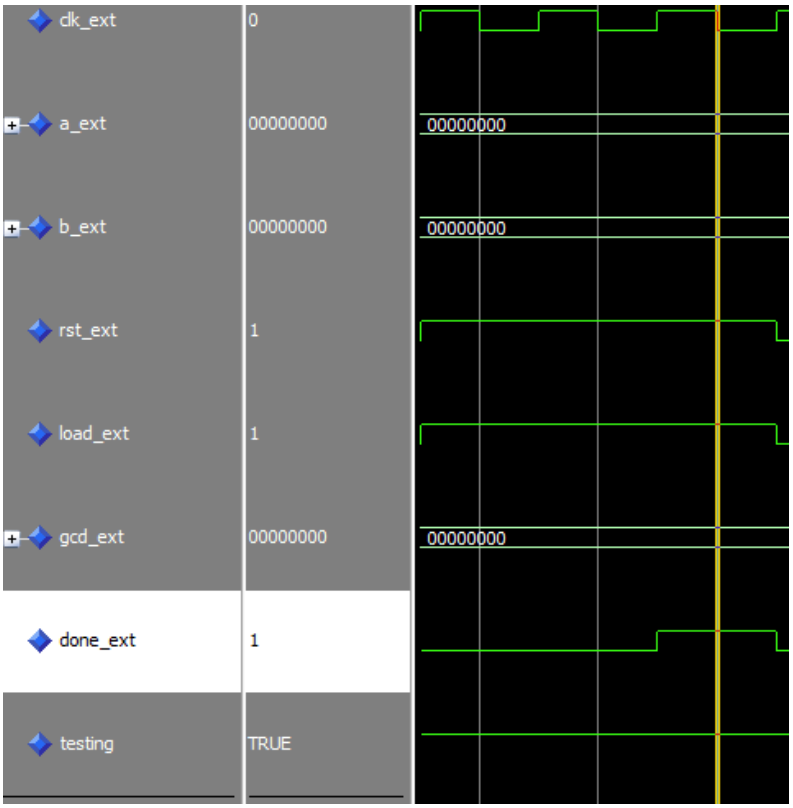
As expected, we have **00100000** as output.

Case A <= 00000000, B <= 00000110



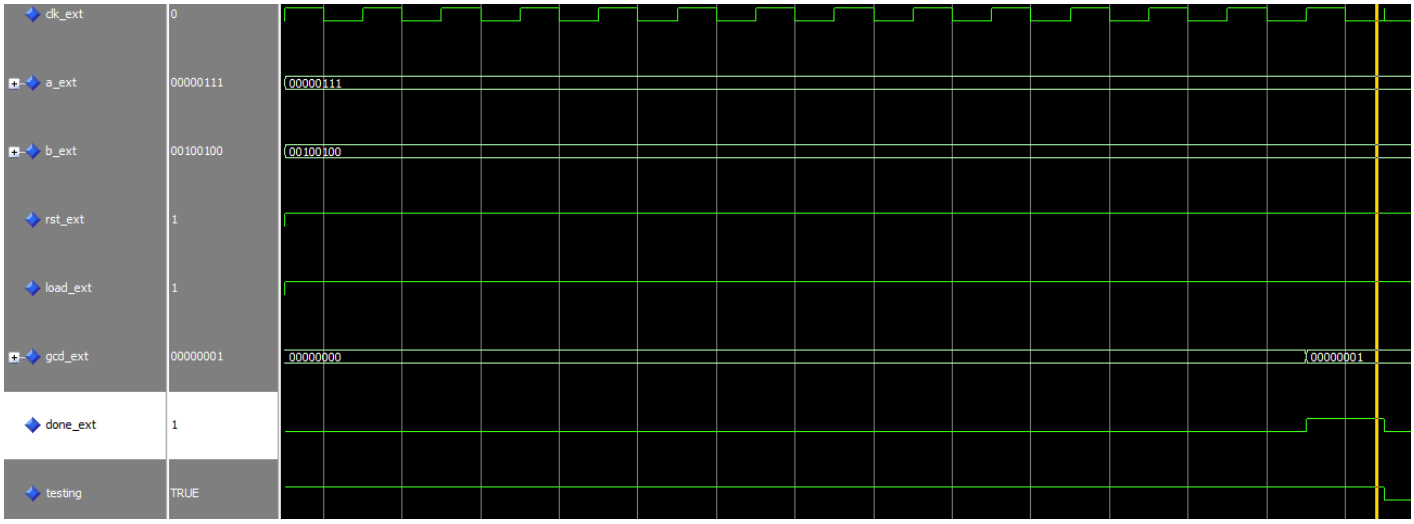
As expected, we have **00000110** as output.

Case A <= 00000000, B <= 00000000



As expected, we have **00000000** as output.

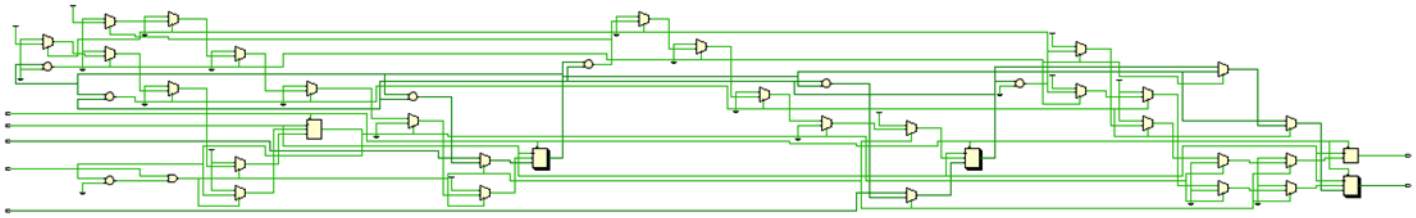
Case A <= 00000111, B <= 00100100



As expected, we have **00000001** as output.

Synthesis

Vivado's Elaborated Design



Timing Constraints

- Clock @ 125 MHz (period: 8 ns, rise at: 0 ns, fall at: 4 ns)

Setup and Hold Slacks

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,792 ns	Worst Hold Slack (WHS):	0,343 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	50	Total Number of Endpoints:	50	Total Number of Endpoints:	26

All user specified timing constraints are met.

As it's clear from the timing summary of the synthesis, since all the time slacks are positive, there are no setup nor hold time violations.

Critical Path

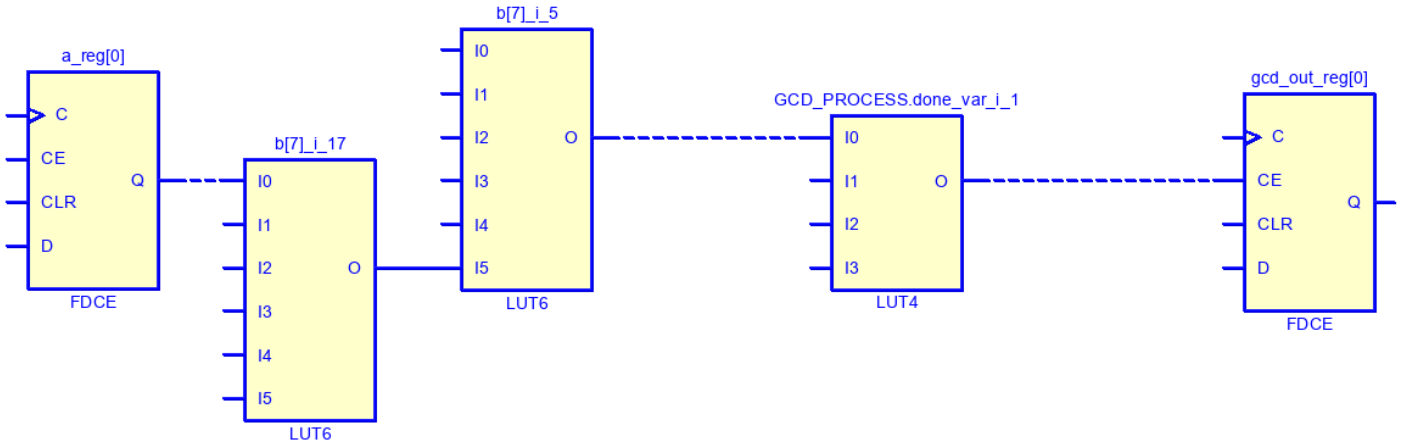
These ones are the register-logic-register paths sorted by t_{setup} slack time.

The critical ones are Path 1-8 and provide the global setup slack and the maximum clock frequency of the clock for this implementation:

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[0]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 2	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[1]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 3	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[2]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 4	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[3]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 5	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[4]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 6	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[5]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 7	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[6]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 8	3.703	3	4	12	a_reg[0]/C	gcd_out_reg[7]/CE	3.915	0.999	2.916	8.0	clock	clock		0.035
Path 9	3.706	3	4	12	a_reg[0]/C	a_reg[0]/CE	3.912	0.999	2.913	8.0	clock	clock		0.035
Path 10	3.706	3	4	12	a_reg[0]/C	a_reg[1]/CE	3.912	0.999	2.913	8.0	clock	clock		0.035

Since $\boldsymbol{Slack = T_{ck} - T_{c-q} - T_{prop} - T_{setup} = 3.792\ ns}$, and $\boldsymbol{T_{ck} = 8\ ns}$, it's possible to speed up the clock period to reach $\boldsymbol{T_{\max ck} = T_{ck} - Slack = 8\ ns - 3.792\ ns = 4.208\ ns}$, getting a maximum frequency of $\frac{1}{T_{\max ck}} = \frac{1}{4.208\ ns} \cong 237\ MHz$. This is an estimation, an upper boundary of the real maximum frequency value, because in this part of the synthesis we are not considering the delay due to the capacitance of the wires that interconnect the inner blocks of the device.

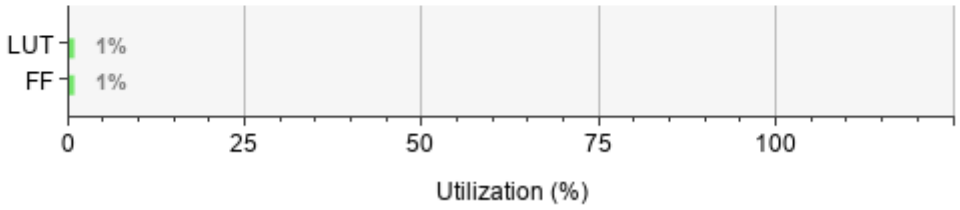
The following screenshot shows the synthetized registers involved in Path 1:



Utilization Report

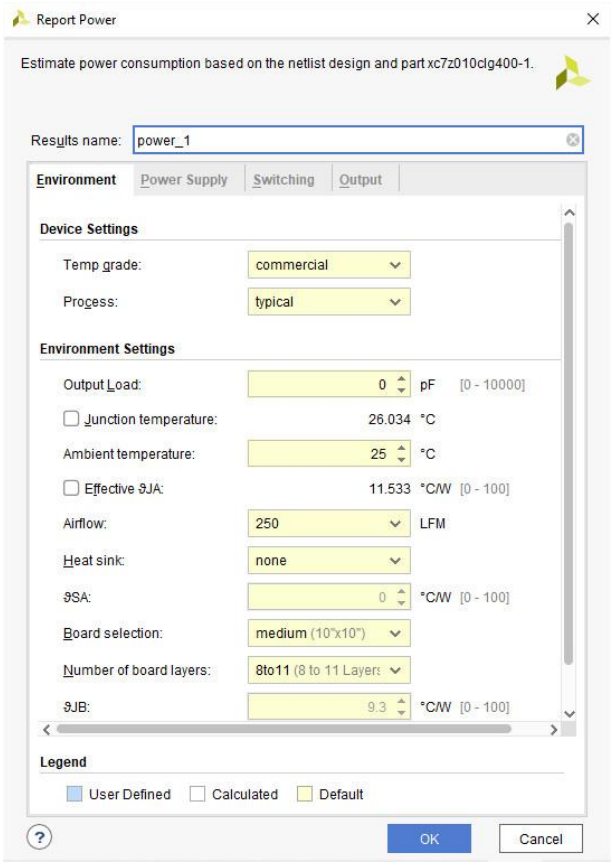
Summary

Resource	Utilization	Available	Utilization %
LUT	41	17600	0.23
FF	26	35200	0.07



Power Report

Given the default parameters for the environment:



The synthesis gives back the following power report:

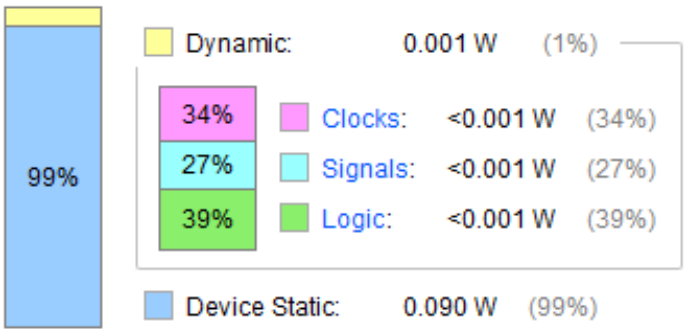
Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.091 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,0°C**
Thermal Margin:	59,0°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

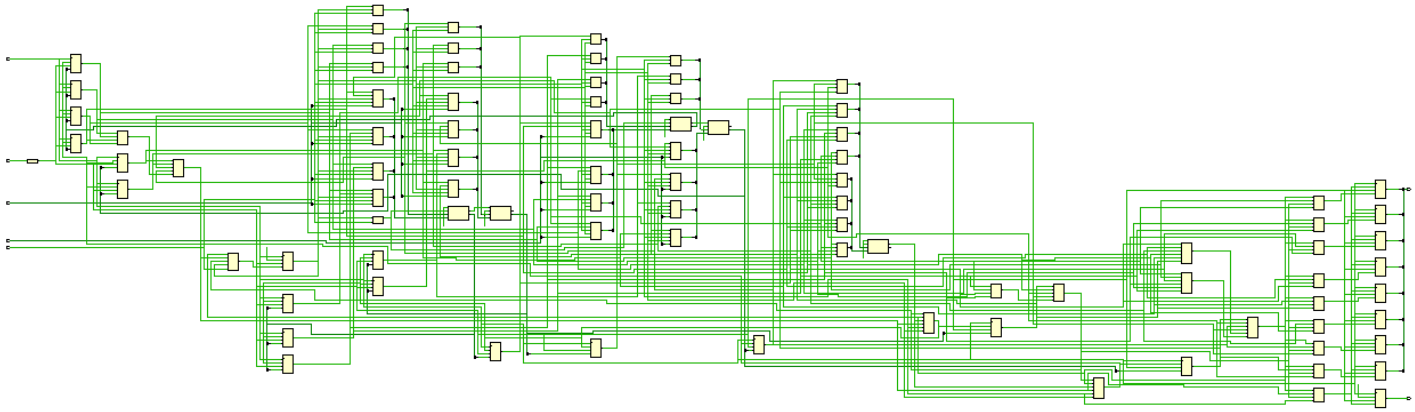
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power



Implementation

Vivado's Elaborated Design



Warnings

[Timing 38-242] The property HD.CLK_SRC of clock port "clk" is not set. In out-of-context mode, this prevents timing estimation for clock delay/skew

[Route 35-198] Port "load" does not have an associated HD.PARTPIN_LOCS, which will prevent the partial routing of the signal "load". Without this partial route, timing analysis to/from this port will not be accurate, and no routing information for this port can be exported. (17 more like this)

[Route 35-197] Clock port "clk" does not have an associated HD.CLK_SRC. Without this constraint, timing analysis may not be accurate and upstream checks cannot be done to ensure correct clock placement.

These are warnings due to the Out of Context mode, in fact they are related to inputs and outputs that have no information on the placement. They can be ignored.

Setup and Hold Slacks

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,073 ns	Worst Hold Slack (WHS):	0,290 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	50	Total Number of Endpoints:	50	Total Number of Endpoints:	26

All user specified timing constraints are met.

As it's clear from the timing summary of the implementation, since all the time slacks are positive, there is no setup nor hold time violations. As we can see the Worst Negative Slack is lower than the synthesis's one.

Critical Path

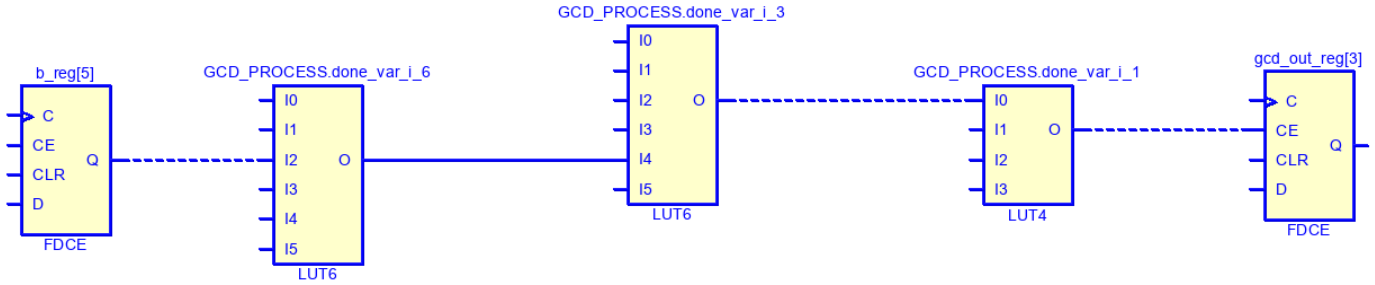
These ones are the register-logic-register paths sorted by t_{setup} slack time.

The critical ones are Path 1-3 and provide the global setup slack and the maximum clock frequency of the clock for this implementation:

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	3.073	3	12	b_reg[5]/C	gcd_out_reg[3]/CE	4.638	0.828	3.810	8.0	clock	clock		0.035
Path 2	3.073	3	12	b_reg[5]/C	gcd_out_reg[4]/CE	4.638	0.828	3.810	8.0	clock	clock		0.035
Path 3	3.073	3	12	b_reg[5]/C	gcd_out_reg[5]/CE	4.638	0.828	3.810	8.0	clock	clock		0.035
Path 4	3.081	3	12	b_reg[5]/C	gcd_out_reg[0]/CE	4.630	0.828	3.802	8.0	clock	clock		0.035
Path 5	3.215	3	12	b_reg[5]/C	gcd_out_reg[6]/CE	4.496	0.828	3.668	8.0	clock	clock		0.035
Path 6	3.215	3	12	b_reg[5]/C	gcd_out_reg[7]/CE	4.496	0.828	3.668	8.0	clock	clock		0.035
Path 7	3.356	3	12	b_reg[5]/C	gcd_out_reg[1]/CE	4.355	0.828	3.527	8.0	clock	clock		0.035
Path 8	3.356	3	12	b_reg[5]/C	gcd_out_reg[2]/CE	4.355	0.828	3.527	8.0	clock	clock		0.035
Path 9	3.357	3	12	b_reg[5]/C	b_reg[4]/CE	4.354	0.828	3.526	8.0	clock	clock		0.035
Path 10	3.357	3	12	b_reg[5]/C	b_reg[5]/CE	4.354	0.828	3.526	8.0	clock	clock		0.035

Since $\boldsymbol{Slack = T_{ck} - T_{c-q} - T_{prop} - T_{setup} = 3.073\ ns}$, and $\boldsymbol{T_{ck} = 8\ ns}$, it's possible to speed up the clock period to reach $\boldsymbol{T_{\max ck} = T_{ck} - Slack = 8\ ns - 3.073\ ns = 4.927\ ns}$, getting a maximum frequency of $\frac{1}{T_{\max ck}} = \frac{1}{4.927\ ns} \cong 203\ MHz$.

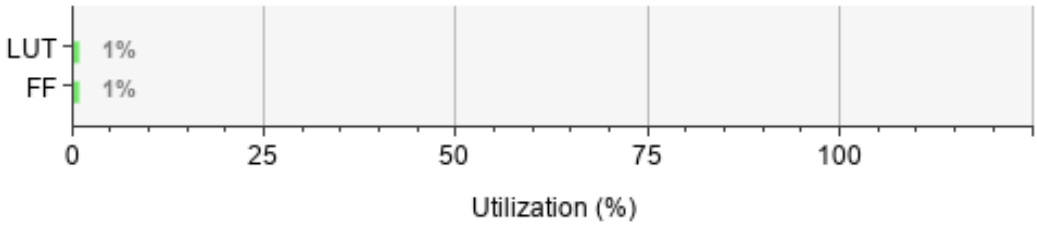
The following screenshot shows the implemented registers involved in Path 1:



Utilization Report

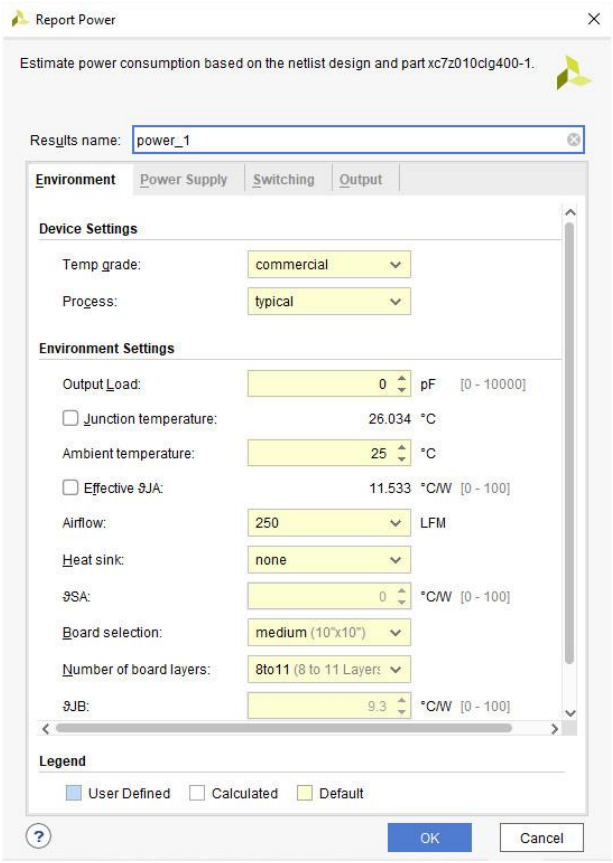
Summary

Resource	Utilization	Available	Utilization %
LUT	40	17600	0.23
FF	26	35200	0.07



Power Report

Given the default parameters for the environment:



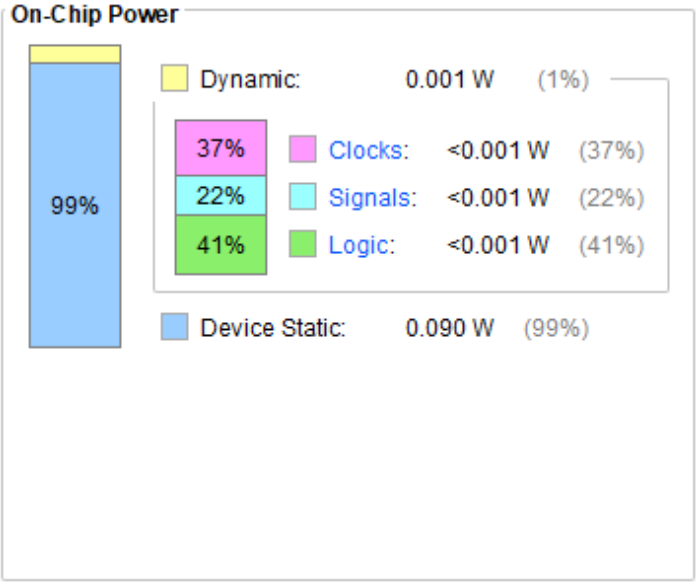
The implementation gives back the following power report:

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.091 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,0°C**
Thermal Margin:	59,0°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



Conclusions

The GCD during this work meets both the functional and the non-functional requirements.
The simulation of the planned test cases proved that the VHDL code of the device works correctly, and its responses are deterministic and reflect the desired behavior of the system.
After the automatic implementation stage, we can assert that the final product is a robust, low-power consuming device, and it can be used at a clock frequency, up to 0.20 GHz.