Islamic University of Technology

Department of Electrical and Electronic Engineering

EEE 4484 – (Digital Electronics and Pulse Technique Lab)

Lab - #4

Synthesizable Delays of Digital Pulses using VHDL

We will go through various supplementary topics before jump into actual delay generation.

Finite State Machine

A *Finite State Machine* is a model of computation based on a hypothetical machine made of one or more states. Only one single state of this machine can be active at the same time. It means the machine has to transition from one state to another in to perform different actions.

A *Finite State Machine* is any device storing the state of something at a given time. The state will change based on inputs, providing the resulting output for the implemented changes. Finite State Machines come from a branch of Computer Science called **automata theory**. The family of data structure belonging to this domain also includes the Turing Machine. Finite state machines can be used to model problems in many fields including mathematics, artificial intelligence, games, and linguistics.

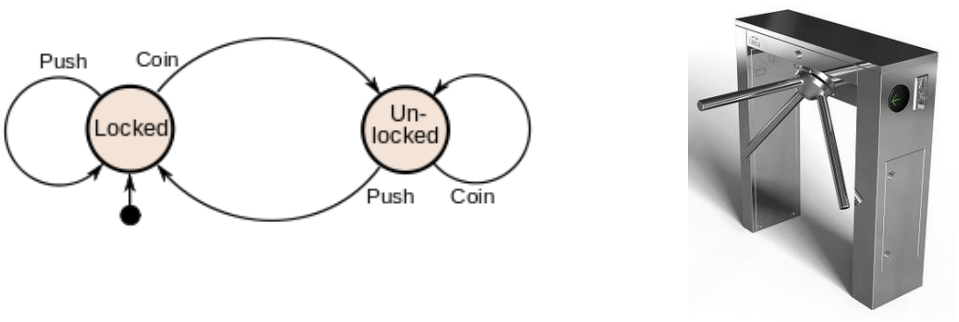

Fig 1: (a) Example of a *finite state machine* (b) A turnstile

A system where particular inputs cause particular changes in state can be represented using *finite state machines*. This example describes the various states of a turnstile. Inserting a coin into a turnstile will unlock it, and after the turnstile has been pushed, it locks again. Inserting a coin into an unlocked turnstile, or pushing against a locked turnstile will not change its state.

More on ***Finite State Machine*** ➵ link.

FPGA

The FPGA is *Field Programmable Gate Array*. It is a type of device that is widely used in electronic circuits. FPGAs are semiconductor devices which contain programmable logic blocks and interconnection circuits. It can be programmed or reprogrammed to the required functionality after manufacturing.

With a microcontroller, like an Arduino, the chip is already designed for you. You simply write some software, usually in C or C++, and compile it to a hex file that you load onto the microcontroller. The microcontroller stores the program in flash memory and will store it until it is erased or replaced. With microcontrollers you have control over the software.

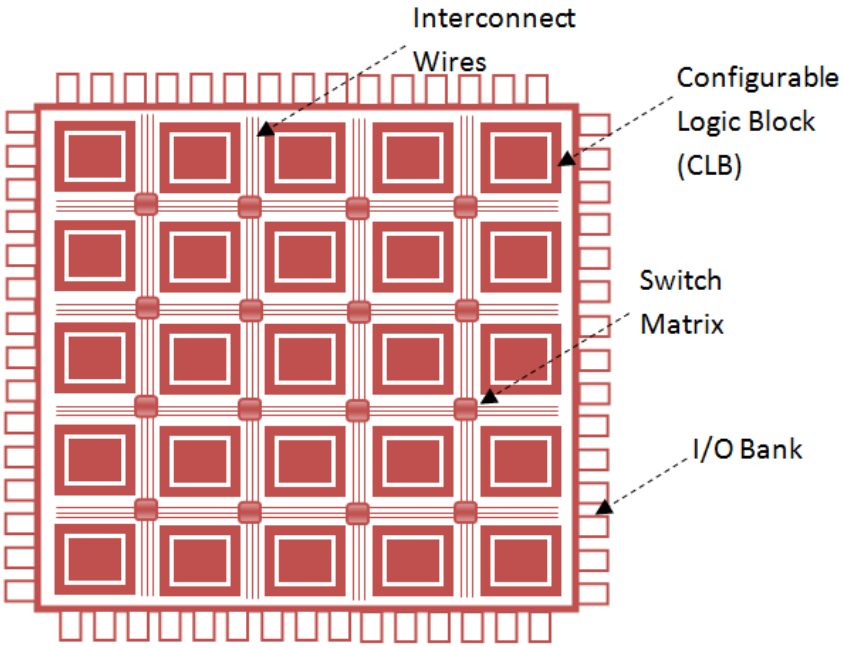

Fig. 2: General FPGA Architecture

FPGAs are different. You are the one designing the circuit. There is no processor to run software on, at least until you design one! You can configure an FPGA to be something as simple as an and gate, or something as complex as a multi-core processor. To create your design, you write some HDL code (VHDL/Verilog/SystemVerilog). You then **synthesize** your HDL into a *bit file* which you can use to configure the FPGA. With FPGAs you have control over the hardware.

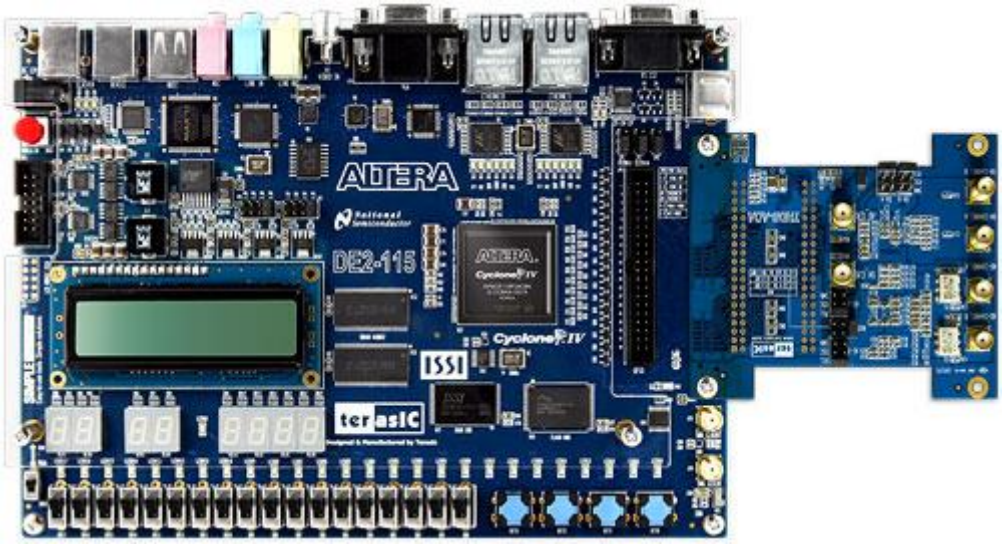

Fig. 3: Altera DE2-115 FPGA Development Board

With a typical microprocessor, you have dedicated pins for specific features. For example, there will be only two pins on some microprocessors that are used as a serial port. If you want more than one serial port, or you want to use some other pins, your only solution besides getting a different chip is to use software to emulate a serial port. That works fine except you are wasting valuable processor time with the very basic task of sending out bits. If you want to emulate more than one port then you end up using all your processor time. With an FPGA you are able to create the actual circuit, so it is up to you to decide what pins the serial port connects to. That also means you can create as many serial ports as you want. The only limitations you really have are the number of physical I/O pins and the size of the FPGA. Just like microcontrollers that have a set amount of memory for your program, FPGAs can only emulate a circuit so large.

One of the very interesting things about FPGAs is that while you are designing the hardware, you can design the hardware to be a processor that you then can write software for! In fact, companies that design digital circuits, like **Intel** or **nVidia**, often use FPGAs to prototype their chips before creating them.

- ⇨ Please have a look on this interesting topic:

GPU and FPGA, Why They Are Important for Artificial Intelligence

Synthesizable VHDL code

Synthesis is the process of constructing a gate level netlist from a model of a circuit described in VHDL.

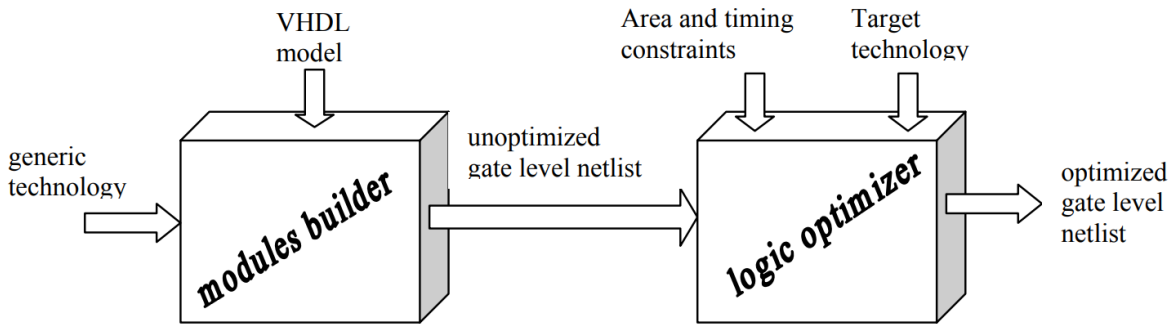

Fig. 4: Synthesis of VHDL program

When you write you Verilog or VHDL code, you are writing code that will be translated into gates, registers, RAMs, etc. The program that performs this task is known as a Synthesis Tool. It is the job of the Synthesis Tool to take your Verilog or VHDL code and turn it into something that the FPGA (rewritable chip, like you can re-write a CD-RW) can understand. However, there are some parts of Verilog and VHDL that the FPGA simply cannot implement. When you write code like this, it is called *non-synthesizable* code.

So why would you have a language that contains code that is non-synthesizable? The reason is that it makes your testbenches more powerful. When you write a testbench for simulation, often using *non-synthesizable* code constructs makes your testbench better and allows you to accomplish things easier. In simple term, *synthesizable* are those which will generate some hardware when implemented and *non-*

synthesizable are those which don't generate any kind of hardware, they are just the instructions for the compiler/assembler.

The most fundamental *non-synthesizable* piece of code is a delay statement. The FPGA has no concept of time, so it is impossible to tell the FPGA to `wait for 10 nanoseconds`. Instead, you need to use clocks and flip-flops to accomplish your goals.

Delay

Delay is a mechanism allowing introducing timing parameters of specified systems. The delay mechanism allows introducing propagation times of described systems. Delays are specified in signal assignment statements. It is not allowed to specify delays in variable assignments.

Synthesizable Delay

There are many situations in which you may need to activate a process after a certain delay or at fixed time intervals. If you want to do simulation alone for your design then you can simply use "`wait for`" statement to call a delay routine. But this keyword is not *synthesizable* (as discussed earlier). So, *what will you do in such situations?*

We will use a *counter* and *state machine* to introduce the delay. We have an input signal, and we want to assign it to the output only after (say) 20 clock cycles. In the following there is the state machine.

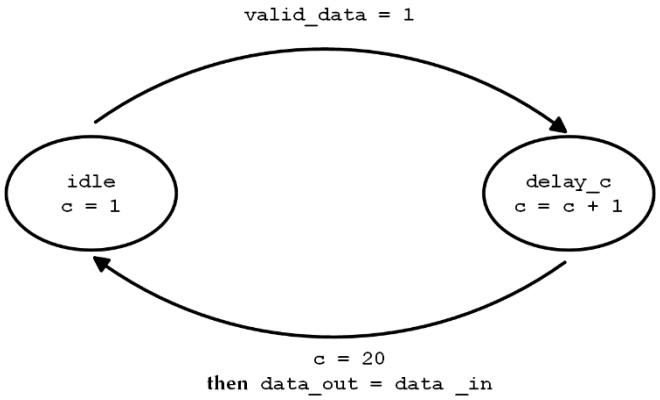

Fig. 5: FSM model to generate the synthesizable delay

There are two states in the *state machine*: **`idle`** and **`delay_c`**.

when the system is in idle state it waits for a valid input bit at the port **`data_in`**. whenever the data is valid, **`valid_data`** will go high. Upon receiving a valid data, the system moves to **`delay_c`** state where a counter, **`c`** is incremented every clock cycle till it reaches the maximum delay value (here it is 20). Once the max. count is reached system goes back to **`idle`** state and the input data will be assigned to output data. This process goes on and on.

Source Code

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

entity delay is
  port (
    clk :       in  std_logic;
    valid_data: in  std_logic;
    data_in:    in  std_logic;
    data_out:   out std_logic
);
end entity;


architecture behavioral of delay is

  signal c : integer := 0;
  constant d: integer := 20;
  signal data_temp : std_logic := '0';
  type state_type is (idle, delay_c); -- listing down the states of FSM
  signal next_s : state_type;         -- declare the state machine signal
begin

  process (clk)
  begin
    if (rising_edge(clk)) then
      case next_s is
        when idle =>
          if (valid_data = '1') then
            next_s <= delay_c;
            data_temp <= data_in;
            c <= 1;
          end if;
        when delay_c =>
          if (c = d) then
            c <= 1;
            data_out <= data_temp;
            next_s <= idle;
          else
            c <= c + 1;               -- counter increment
          end if;
        when others =>
          NULL;
      end case;
    end if;
  end process;


end architecture behavioral;
```

Testbench

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

entity tb_delay is
end entity;

architecture behavioral of tb_delay is

  component delay
     port (
       clk :       in  std_logic;
       valid_data: in  std_logic;
       data_in:    in  std_logic;
       data_out:   out std_logic
       );
  end component;

  signal clk    : std_logic := '0';
  signal tb_valid_data : std_logic := '0';
  signal tb_data_in : std_logic := '0';
  signal tb_data_out : std_logic := '0';

  constant clk_period : time := 4 ns;

begin
  -- Unit Under Test
  uut: delay port map (
    clk => clk,
    valid_data => tb_valid_data,
    data_in => tb_data_in,
    data_out => tb_data_out
    );

  -- clock process definition
  clk_proc : process
  begin
    clk <= '0';
    wait for clk_period / 2;
    clk <= '1';
    wait for clk_period / 2;
  end process;


  -- Stimulus process
  stim_proc : process
  begin

    wait for 25 ns;
    tb_data_in <= '1';
    wait for 25 ns;
    tb_valid_data <= '1';
    wait;
  end process;


end architecture behavioral;
```

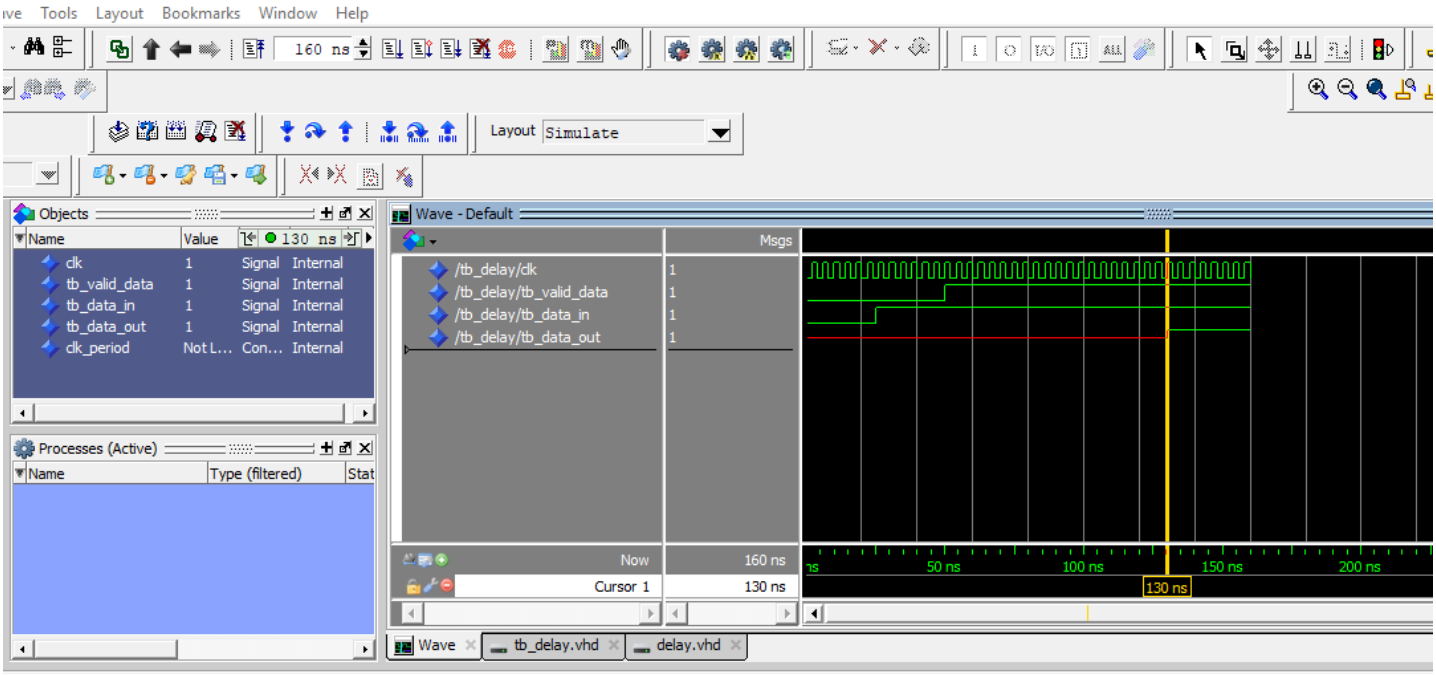

Fig. 6: Synthesizable Delay is generated after 130 ns

From Fig. 6 we can see that, input is applied after 25 ns. Then **`valid_data`** made HIGH at 50 ns. Therefore, counting starts from 51 ns. After 20 clock cycles the output will be available at the **`data_out`** port. The total time of 20 clock cycle is 20 × 4 = 80 ns. If we make the proper calculation output will be available after (80 + 50) ns or 130 ns. The total *synthesizable digital delay pulse* generated after inserting the input is (130 -25) ns = 105 ns.

Tasks to complete

1. Change the program in a way that input will be applied at 20 ns. And the synthesizable delay will bd 150 ns.
2. If you alter the insetting sequence of **`valid_data`** and **`data_in`** what will be the outcome, inset what you observe in the Modelsim simulation scope.
3. Insert a reset input in the main source code, which will override the **`valid_data`** input to suppress the output.

Appendix (Helpful to understand the main Delay program)

Counter Implementation

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

entity counter is
  port (
    clk         : in  std_logic;
    reset       : in  std_logic;
    enable      : in  std_logic;
    cout        : out std_logic_vector (7 downto 0); -- counter out
);
end entity;


architecture behavioral of counter is
  signal temp_count : std_logic_vector (7 downto 0);
begin
  process (clk, reset)
  begin
    if (reset = '1') then
      temp_count <= (others => '0');
    elsif (clk'event and clk = '1') then
      if (enable = '1') then
        temp_count <= temp_count + 1;        -- incrementing the counter
      end if;
    end if;
  end process;

  cout <= temp_count;

end architecture behavioral;
```