Aristotle University of Thessaloniki

Computer Science Department

Semester assignment in NNA-07-08

Digital Electronic Systems in VHDL

Alexandros Korkos
alexkork@csd.auth.gr
3870

Thessaloniki, January 16, 2024

Abstract

Digital design of logic circuits is a fundamental part of computer engineering. The design of logic circuits is done using Hardware Description Languages (HDLs). The most common HDLs are Verilog and VHDL. This project focuses on the design of logic circuits using VHDL. The project is divided into two parts. The first part focuses on the design of combinational circuits, while the second part focuses on the design of sequential circuits. All the circuits designed and implemented are fundamental circuits, that are used in the design of more complex circuits.

Besides the design of the circuits, the project also focuses on the simulation and validation of the circuits. All the above are done using free and open source software. The software used for the design and simulation of the circuits are GHDL and GTKWave (see Appendix B). The VHDL codes for the circuits are following the version 2008 of IEEE standard 1076.

All VHDL codes are available at GitHub.

Contents

Part I
Sequential circuits

A sequential circuit, consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs.

1 D flip-flop

A D flip-flop, is a type of digital storage element used in digital circuits, particularly in sequential logic systems like registers and memories. The output of the D flip-flop follows the value of the D input when the appropriate synchronization pulses are applied.

In this implementation, the D flip-flop is triggered on the rising edge of the clock signal. Its operation is shown in the following truth table.

Table 1: D flip-flop truth table

clk	**D**	$\mathbf{Q_{t+1}}$
↑	0	0
↑	1	1
other	X	Q_t

In the implementation seen in listing 1 the negated Q is not configured, because it was not required by the instructor. Also, the listing 1 uses an asynchronous reset.

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

entity d_flip_flop is
  port (
    clk            : in  std_logic;
    rst_n, preset  : in  std_logic;
    D_in           : in  std_logic;
    Q_out          : out std_logic
  );
end entity;

-- asynchronous reset
architecture behavior of d_flip_flop is
begin
  DUT: process(clk, rst_n, preset)
  begin
    if (rst_n = '0') then
      Q_out <= '0';
    elsif (preset = '0') then
      Q_out <= '1';
    elsif (clk'event and clk = '1') then
      Q_out <= D_in;
    end if;
  end process;
end architecture;
```

Listing 1: D flip-flop

For the testing of the circuit the test bench created as shown here listing 2 finishes in 90ns without any errors. Additionally, it uses the asserted instruction, to check instantly the value of the output. The clock period simulated is 50ns long, where each 25ns edge is rising/falling.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tb_d_flip_flop is
end entity;

architecture bench of tb_d_flip_flop is
  signal D_tb_in: std_logic := '0';
  signal clk_tb: std_logic := '0';
  signal rst_n_tb: std_logic := '0';
  signal preset_tb: std_logic := '0';
  signal Q_tb_out: std_logic;

  constant clk_period : time := 50 ns;

begin
  UUT: entity work.d_flip_flop port map (clk_tb, rst_n_tb, preset_tb, D_tb_in, Q_tb_out
  ↪  );

  clk_process: process
  begin
    clk_tb <= '0';
    wait for clk_period/2;
    clk_tb <= '1';
    wait for clk_period/2;
  end process;

  stimulus : process
  begin
    rst_n_tb <= '0';
    wait for 30 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    rst_n_tb   <= '1';
    preset_tb  <= '1';
    D_tb_in    <= '0';
    wait for 15 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    rst_n_tb   <= '1';
    preset_tb  <= '1';
    wait for 15 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    rst_n_tb   <= '1';
    preset_tb  <= '0';
    wait for 10 ns;
    assert(Q_tb_out = '1') report ”Error in q” severity failure;

    rst_n_tb   <= '1';
    preset_tb  <= '1';
    wait for 20 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    finish;
  end process;
end architecture;
```

Listing 2: Testbench for D flip-flop

The waveform of this circuit simulation is shown in the fig. 1 following.

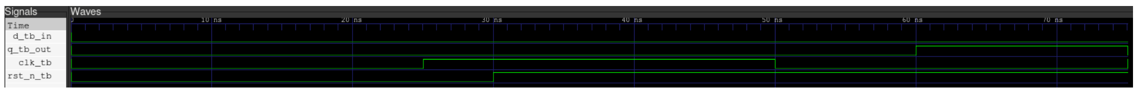

Figure 1: Waveform of D flip-flop

2 JK flip-flop

A JK flip-flop is a type of digital storage element that has the ability to toggle its output between two states. It is a synchronous device, meaning that it only responds to input signals at transitions of a clock signal. The JK flip- flop is a refinement of the SR flip-flop, where S and R inputs are combined into a single input J (set) with the addition of a clock input and a second input K (reset). The output of the JK flip-flop toggles between two states according the inputs.

In this implementation, the JK flip-flop is triggered on the rising edge of the clock signal. Its operation is shown in the following truth table 2.

Table 2: JK flip-flop truth table

clk	**J**	**K**	$\mathbf{Q_{t+1}}$
↑	0	0	Q_t
↑	0	1	0
↑	1	0	1
↑	1	1	$\overline{Q_t}$
other	X	X	Q_t

In the implementation seen in listing 3 the negated Q output is like in the D flip-flop implementation, not configured. Also, the listing 3 uses an asynchronous reset.

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

entity jk_flip_flop is
  port (
    J_in: in  std_logic;
    K_in: in  std_logic;
    rst, clk: in  std_logic;
    Q_out: inout std_logic
  );
end entity;

architecture behavior of jk_flip_flop is
begin
  DUT: process (rst, clk) is
  begin
    if rst = '1' then
      Q_out <= '0';
    elsif(rising_edge(clk)) then
      if (J_in = '0' and K_in = '0') then
        Q_out <= Q_out;
      elsif (J_in = '0' and K_in = '1') then
        Q_out <= '0';
      elsif (J_in = '1' and K_in = '0') then
        Q_out <= '1';
      elsif (J_in = '1' and K_in = '1') then
        Q_out <= not (Q_out);
      end if;
    end if;
  end process;
end behavior;
```

Listing 3: JK flip-flop

For the testing of this JK flip-flop, the test bench created as shown here listing 4 finishes in 100ns without any errors. Additionally, it uses the asserted instruction, to check instantly the value of the output. Here the clock period simulated is 10ns long, where each 5ns edge is rising/falling.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tb_jk_flip_flop is
end entity;

architecture bench of tb_jk_flip_flop is
  signal J_tb_in: std_logic := '0';
  signal K_tb_in: std_logic := '0';
  signal rst_n_tb: std_logic := '0';
  signal clk_tb: std_logic := '0';
  signal Q_tb_out: std_logic;

  constant clk_period : time := 10 ns;

begin
  UUT: entity work.jk_flip_flop port map (J_tb_in, K_tb_in, rst_n_tb, clk_tb, Q_tb_out
  ↪  );

  clk_process : process
  begin
    clk_tb <= '0';
    wait for clk_period/2;
    clk_tb <= '1';
    wait for clk_period/2;
  end process;

  stimulus: process
  begin
    J_tb_in <= '1';
    K_tb_in <= '0';
    rst_n_tb <= '0';
    wait for 20 ns;
    assert(Q_tb_out = '1') report ”Error in q” severity failure;

    J_tb_in <= '0';
    K_tb_in <= '0';
    rst_n_tb <= '0';
    wait for 20 ns;
    assert(Q_tb_out = '1') report ”Error in q” severity failure;

    J_tb_in <= '0';
    K_tb_in <= '1';
    rst_n_tb <= '0';
    wait for 20 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    J_tb_in <= '1';
    K_tb_in <= '1';
    rst_n_tb <= '0';
    wait for 20 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    J_tb_in <= '1';
    K_tb_in <= '1';
    rst_n_tb <= '1';
    wait for 20 ns;
    assert(Q_tb_out = '0') report ”Error in q” severity failure;

    finish;
  end process;
end architecture;
```

Listing 4: Testbench for JK flip-flop

The results of the simulation are shown in the fig. 2 following.

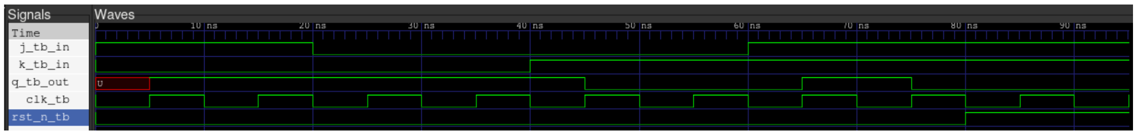

Figure 2: Waveform of JK flip-flop

For the first 5ns, the Q output is Undefined, because all the inputs are set to 0 and the clock is in a falling edge which sets the flip-flop to a no-change state (default value of std_logic is ’U’).

3 Random Access Memory (RAM)

There are two types of memories that are used in digital systems: random access memory (RAM) and read only memory (ROM). A RAM stores new information for later use. The process of storing new information into memory is referred to as a memory write operation. The process of transferring the stored information out of memory is referred to as a memory read operation. RAM can perform both write and read operations.

The RAM implemented, is a 16 to 4. This means, 16 addresses to store 4 bits long data. An enable signal is used, to control the RAM. If enable is true, the system is operational, if not, the system is in an idle state. Besides that, a rw (read = 0, write = 1) is expected, to determine if the system is in a read or write state. Additionally, a vector of an address is expected, to determine which address to read or write. For the data input and output, a vector of 4 bits of inout is used, to combine in one. The sensitivity list only includes the clock signal.

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

entity ram is
  port(
    clk, enable, rw: in std_logic;
    address: in std_logic_vector (3 downto 0);
    data: inout std_logic_vector (3 downto 0)
  );
end ram;

architecture behavior of ram is
  type ram_t is array (0 to 15) of std_logic_vector(3 downto 0);
  signal tmp_ram: ram_t;
begin
  process (clk)
  begin
    if (clk = '1' and clk'event) then
      if enable = '1' then
        if rw = '1' then
          tmp_ram(to_integer(unsigned(address))) <= data;
        elsif rw = '0' then
          data <= tmp_ram(to_integer(unsigned(address)));
        else
          data <= (data'range => 'Z');
        end if;
      end if;
    end if;
  end process;
end architecture;
```

Listing 5: RAM

The test bench for the RAM is shown in listing 6. The clock period is 1ns and the raise and fall time is 0.25ns.

```
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tb_ram is
end tb_ram;

architecture behavior of tb_ram is
  signal clk: std_logic;
  signal enable: std_logic;
  signal rw: std_logic;
  signal address: std_logic_vector (3 downto 0);
  signal data: std_logic_vector (3 downto 0);
  constant clk_period: time := 1 ns;
begin
  RAM0: entity work.ram port map (clk, enable, rw, address, data);

  clock : process
  begin
    for i in 1 to 20 loop
      clk <= '1';
      wait for clk_period/4;
      clk <= '0';
      wait for clk_period/4;
    end loop;
  finish;
  end process;

  process
  begin
    wait for clk_period;
    enable <= '1';
    wait for clk_period;
    rw <= '1';
    address <= ”0011”;
    data <= ”1001”;
    wait for clk_period;
    rw <= '0';
    address <= ”0001”;
    data <= ”ZZZZ”;
    wait for clk_period;
    address <= ”0011”;
    data <= ”ZZZZ”;
    wait for clk_period;
    wait;
  end process;
end architecture;
```

Listing 6: Test bench for the RAM

The simulation of the RAM circuit, are presented with the following waveform image.

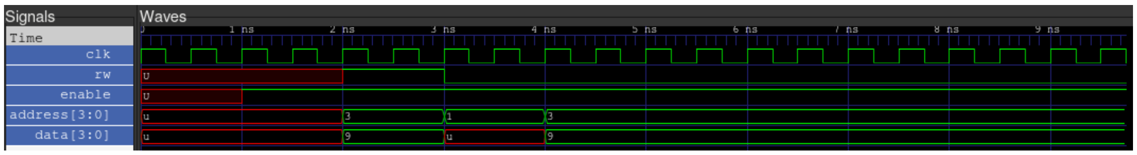

Figure 3: Waveform of RAM

4 Sequential multiplier

Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

The algorithm for the multiplication, is summed up by the pseudocode following.

Algorithm 1: Sequential multiplier

```
Data: $A, B$ same $n$-bits binary numbers
Result: $A \times B$
1 $P \leftarrow 0$
2 Load $A, B$
3 while $B \neq 0$ do
4     if $b_0 = 1$ then
5         $P \leftarrow P + A$
6     left shift $A$
7     right shift $B$
```

The digital circuit for the sequential multiplier, implements the procedure shown from before and the implementation is a follows bellow.

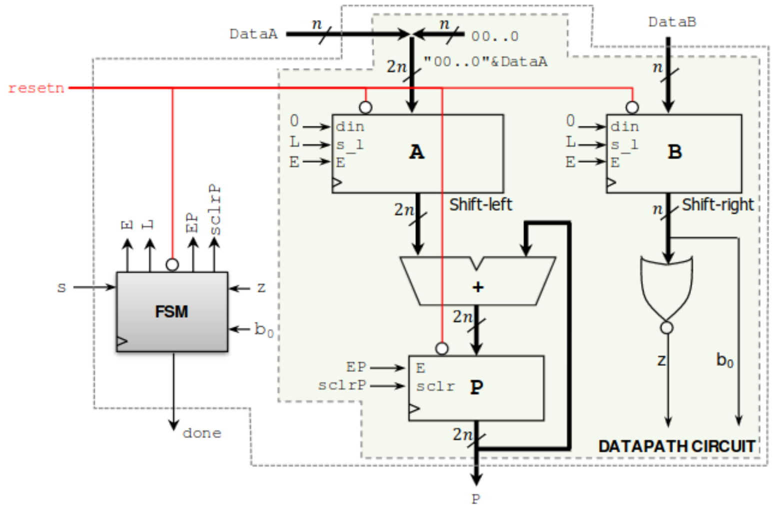

Figure 4: Circuit design of the sequential multiplier

Both the circuit and VHDL implementations, are developed in such way to calculate the multiplication of two n-bits numbers and not fixed length numbers.

The FSM following, shows the states that the sequential multiplier goes through to calculate the multiplication of two n-bits numbers. The states are represented by the circles and the arrows represent the transitions between the states. The FSM instructs to store the value of the partial product, in the p register. Finally, the p register is also the output of the sequential multiplier.

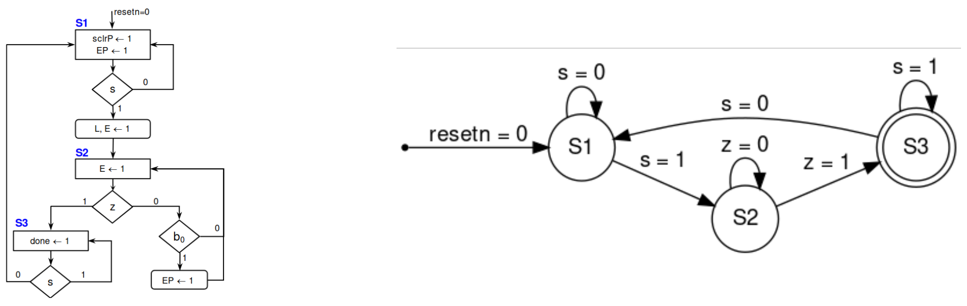

Figure 5: Fine state machine

The sequential multiplier system, uses VHDL implementations of n-bits register, n-bits parallel access shift register and adder/subtractor circuits. For space-saving reasons, the VHDL implementations are not shown here, but can be found in the appendix.

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

entity multiplier is
  generic (N: INTEGER  := 4);
  port (
    clk, reset, s: in std_logic;
    dA, dB: in std_logic_vector (N-1 downto 0);
    P: out std_logic_vector (2*N-1 downto 0);
    done: out std_logic
  );
end multiplier;

architecture Behavioral of multiplier is
  component parallel_shift_reg
    generic (
      N: INTEGER := 4;
      DIR: STRING := ”LEFT”
    );
    port (
      clk, reset: in std_logic;
      din, E, s_l: in std_logic; --din: shiftin input
      D: in std_logic_vector (N-1 downto 0);
      Q: out std_logic_vector (N-1 downto 0);
      shift_out: out std_logic
    );
  end component;

  component add_sub
    generic (N: INTEGER := 4);
    port(
      addsub: in std_logic;
      x, y: in std_logic_vector (N-1 downto 0);
      s: out std_logic_vector (N-1 downto 0);
      overflow, cout: out std_logic
    );
  end component;

  component n_register
    generic (N: INTEGER := 4);
    port (
      clk, reset: in std_logic;
      E, clear: in std_logic; -- clear: Synchronous clear
      D: in std_logic_vector (N-1 downto 0);
      Q: out std_logic_vector (N-1 downto 0)
    );
  end component;

  type state is (S1, S2, S3);
  signal y: state;

  signal L, E, EP, sclrP, z: std_logic;
  signal B: std_logic_vector (N-1 downto 0);
  signal A, dAx, dP, Pt: std_logic_vector (2*N -1 downto 0);

begin
rA: parallel_shift_reg generic map (N ⇒ 2*N, DIR ⇒ ”LEFT”)
    port map (
    clk ⇒ clk, reset ⇒ reset, din ⇒ '0',
    s_l ⇒ L, E ⇒ E, D ⇒ dAx, Q ⇒ A
  );
  dAx (2*N-1 downto N)⩽ (others ⇒'0');
  dAx (N-1 downto 0)   ⩽ dA;

rB: parallel_shift_reg generic map (N ⇒ N, DIR ⇒ ”RIGHT”)
    port map (
    clk ⇒ clk, reset ⇒ reset, din ⇒ '0',
    s_l ⇒ L, E ⇒ E, D ⇒ dB, Q ⇒ B
  );

-- n-bit NOR gate:
  process (B)
    variable result_or: std_logic;
  begin
    result_or := '0';
    for i in B'range loop -- 'range: iterates through all bits in 'A'
      result_or := result_or or B(i);
    end loop;
    z ⩽ not (result_or);
  end process;

--Adder:
ga: add_sub generic map (N ⇒ 2*N)
    port map (addsub ⇒ '0', x ⇒ A, y ⇒ Pt, s ⇒ dP);

--Register P:
rP: n_register generic map (N ⇒ 2*N)
  port map (
    clk ⇒ clk, reset ⇒ reset, E ⇒ EP,
      clear ⇒ sclrP, D ⇒ dP, Q ⇒ Pt
  );
  P ⩽ Pt;

--FSM:
  Transitions: process (reset, clk, s, z, B(0))
  begin
  if reset = '0' then -- asynchronous signal
    y ⩽ S1;     -- if reset asserted, go to initial state: S1
  elsif (clk'event and clk = '1') then
    case y is
      when S1 ⇒ if s = '1' then y ⩽ S2; else y ⩽ S1; end if;
      when S2 ⇒ if z = '1' then y ⩽ S3; else y ⩽ S2; end if;
      when S3 ⇒ if s = '1' then y ⩽ S3; else y ⩽ S1; end if;
    end case;
  end if;
  end process;

  Outputs: process (y, s, z, B(0))
  begin
--Initialization of output signals
  sclrP ⩽ '0'; EP ⩽ '0'; L ⩽ '0'; E ⩽ '0'; done ⩽ '0';
```

```vhdl
  case y is
    when S1 =>
      sclrP <= '1'; EP <= '1';
      if s = '1' then
        L <= '1'; E <= '1';
      end if;
    when S2 =>
      E <= '1';
      if z = '0' then
        if B(0) = '1' then EP <= '1'; end if;
      end if;
    when S3 =>
      done <= '1';
  end case;
  end process;
end Behavioral;
```

Listing 7: n-bits multiplier

Point of interest of the above implementation listing 7 is the finite state machine (FSM) that is used to control the system. The FSM is implemented using a case statement, and the sensitivity list includes the clock signal. It follows exactly the FSM diagram is shown in fig. 5 and the logic presented in that figure.

The test bench for the multiplier is shown in listing 8. It has to be pointed out that, in worst case scenario, the multiplication of two n-bits numbers, needs $n+1$ clock cycles to complete. For that reason, on each multiplication the test bench waits $n+3$ clock cycles, to ensure that the multiplication is completed.

```vhdl
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.all;
use std.env.finish;

entity tb_multiplier is
  generic (N: INTEGER := 4);
end tb_multiplier;

architecture behavior of tb_multiplier is
   --Inputs
   signal clk: std_logic := '0';
   signal reset: std_logic := '0';
   signal s: std_logic := '0';
   signal dA: std_logic_vector (N-1 downto 0) := (others => '0');
   signal dB: std_logic_vector (N-1 downto 0) := (others => '0');

   --Outputs
   signal P: std_logic_vector (2*N - 1 downto 0);
   signal done: std_logic;

   -- Clock period definitions
   constant clock_period: time := 10 ns;
begin
  -- Instantiate the Unit Under Test (UUT)
  uut: entity work.multiplier port map (clk, reset, s, dA, dB, P, done);

  -- Clock process definitions
  clock_process: process
  begin
  clk <= '0';
    wait for clock_period/2;
  clk <= '1';
    wait for clock_period/2;
  end process;

  -- Stimulus process
  stimulus: process
  begin
    -- hold reset state for 100 ns.
    wait for 100 ns; reset ⩽ '1';

    dA ⩽ conv_std_logic_vector (13, N); -- "1101";
    dB ⩽ conv_std_logic_vector (11, N); -- "1011";
    s ⩽ '1'; wait for clock_period;
    s ⩽ '0';
    wait for clock_period*(N+3); -- Multiplication takes at most N + 1 cycles

    dA ⩽ conv_std_logic_vector (9, N);    -- "1001";
    dB ⩽ conv_std_logic_vector (15, N);   -- "1111";
    s ⩽ '1'; wait for clock_period;
    s ⩽ '0';
    wait for clock_period*(N+3);
    finish;
  end process;
end architecture;
```

Listing 8: Test bench for the n-bits multiplier

The test bench in listing 8, generates the waveform shown in fig. 6. Before the end result is calculated, in the p register the partial multiplication result is stored.

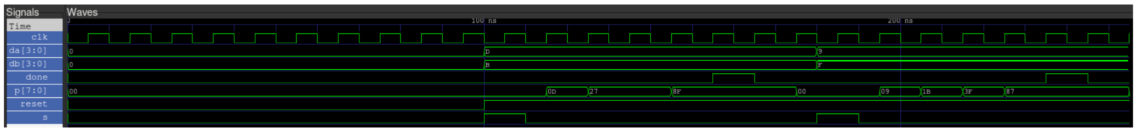

Figure 6: Waveform of the multiplier

Part II

Combinatorial circuits

A combinational circuit consists of an interconnection of logic gates. Combinationallogic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.

5 LED Counter

The concept of a counter is quite simple: it is a circuit that counts. In this particular case, the counter is a 4-bit counter, which means that it can count from 0 to 15.

A LED counter is a combination of a LED decoder and a counter (see Appendix A.1). The LED Counter takes in the clock and reset inputs from the user and outputs the 7-bit value to be displayed on the LEDs. The LED Counter is a hierarchical design, where the LED Decoder and the Counter are instantiated as components in the LED Counter.

This design is hierarchical; both the LED decoder and the counter are autonomous units connected to each other through another higher-level unit to create the final circuit.

The VHDL code for the LED Counter is shown here listing 9. The LED Counter has a clock and reset input, which are used to control the counter. The counter value is then passed to the LED Decoder, which returns the 7-bit value to be displayed on the LEDs. The LED Counter also has a 4-bit output, which is the counter value.

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

```

```vhdl
entity led_counter is
  port (
    clk, reset: in std_logic;
    Q_out: out std_logic_vector (6 downto 0)
  );
end entity;

architecture behavior of led_counter is
  signal counter_value : std_logic_vector (3 downto 0);
begin
  LED: entity work.led_decoder port map (clk, counter_value, Q_out);
  COUNT: entity work.counter port map (clk, reset, counter_value);
end architecture;
```

Listing 9: LED Counter

The test bench for the LED Counter is shown here listing 10. The test bench instantiates the LED Counter and provides the clock and reset inputs. The test bench also displays the counter value on the console. The test bench is run for 100ns, which is enough time for the counter to reach its maximum value and reset back to 0.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tb_led_counter is
end entity;

architecture behavior of tb_led_counter is
    constant clk_period: time := 10 ns;

    signal clk: std_logic := '0';
    signal reset: std_logic := '0';
    signal led: std_logic_vector(6 downto 0) := ”0000000”;
begin
  UUT: entity work.led_counter port map(clk, reset, led);

  clk_process : process
  begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
  end process;

  stimulus : process
  begin
    reset <= '1';
    wait for 100 ns;
    reset <= '0';
    wait for 100 ns;
    reset <= '1';
    wait for 50 ns;
    reset <= '0';
    wait for 450 ns;
    finish;
  end process;
end architecture;
```

Listing 10: Test bench for LED Counter

The waveform extracted form GTKWave is presented in fig. 7.

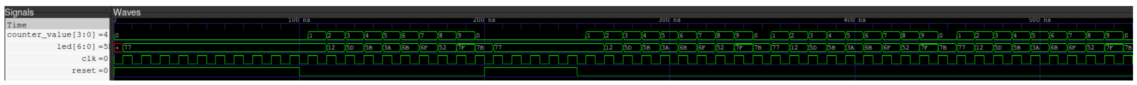

Figure 7: Waveform of LED Counter

6 Multiplexer

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

Here the n value is 2, so this means that the MUX implemented is a 4 to 1 circuit. The truth table for the multiplexer is shown in table 3.

Table 3: Truth table for the 4 to 1 MUX

S_0	S_1	Q_{out}
0	0	A
0	1	B
1	0	C
1	1	D

The circuit design, of a 4 to 1 MUX using digital logic gates, is shown in fig. 8.

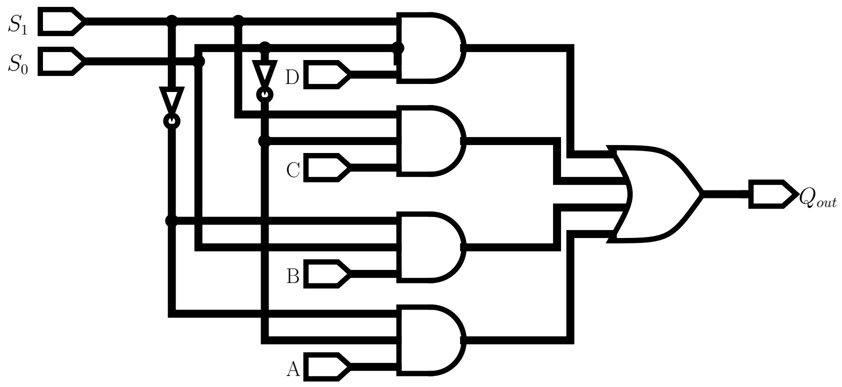

Figure 8: Circuit design of MUX 4 to 1

The developed code seen in listing 11, uses a vector to store the input values, and then uses the selection lines, also a vector to select the output value. The sensitivity list includes the selection and the input vectors, that means if the values of those two vectors the output gets recalculated. A small but important detail is that the output gets the value Uninitialized ('X') as a default value in the case statement, this is to avoid latches.

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

entity mux_4to1 is
  port (
    ABCD_in : in  std_logic_vector(3 downto 0);
    Sel_in  : in  std_logic_vector(1 downto 0);
    Q_out   : out std_logic
  );
end entity;
```

```vhdl

architecture behavior of mux_4to1 is
begin
  DUT: process(ABCD_in, Sel_in)
  begin
    case Sel_in is
      when ”00”   => Q_out <= ABCD_in(0);
      when ”01”   => Q_out <= ABCD_in(1);
      when ”10”   => Q_out <= ABCD_in(2);
      when ”11”   => Q_out <= ABCD_in(3);
      when others => Q_out <= 'X';
    end case;
  end process;
end architecture;
```

Listing 11: MUX 4 to 1

The test bench for the listing 11 is shown in listing 12. To avoid long and repeated case statements, the test bench uses a for loop to iterate through all the possible combinations of the selection, input and output values. These values are stored in a separate file, and then read by the test bench. For this to work, the library STD is used.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;

library STD;
use STD.textio.all;

entity tb_mux_4to1 is
end entity;

architecture bench of tb_mux_4to1 is

  component mux_4to1
    port (
      ABCD_in: in std_logic_vector(3 downto 0);
      Sel_in: in std_logic_vector(1 downto 0);
      Q_out: out std_logic
    );
  end component;

  signal ACBD_tb_in: std_logic_vector(3 downto 0);
  signal Sel_tb_in: std_logic_vector(1 downto 0);
  signal Q_tb_in: std_logic;

begin
  DUT: mux_4to1 port map(ABCD_in => ACBD_tb_in, Sel_in => Sel_tb_in, Q_out => Q_tb_in);
  process
    file Fin : text open read_mode is ”mux_4to1_tests.txt”;

    variable current_read_line   : line;
    variable current_read_field1 : std_logic_vector(0 to 3);
    variable current_read_field2 : std_logic_vector(0 to 1);
    variable current_read_field3 : std_logic;

  begin
    while (not endFile(Fin)) loop

      readline(Fin, current_read_line);
      read(current_read_line, current_read_field1);
      read(current_read_line, current_read_field2);
      read(current_read_line, current_read_field3);

      ACBD_tb_in <= current_read_field1;
      Sel_tb_in  <= current_read_field2;
      wait for 50 ns;
```

```

            assert(Q_tb_in = current_read_field3);

        end loop;
        wait;
    end process;
end architecture;
```

Listing 12: Testbench for MUX 4 to 1

The simulation of the listing 11 creates the waveform shown in fig. 9.

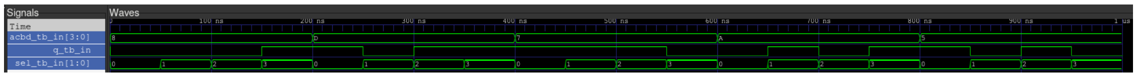

Figure 9: Waveform of MUX 4 to 1

Which is the expected result, because the output value is the same as the input value that is selected by the selection lines.

7 Full adder

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous stage. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary representation of 2 or 3 needs two bits. The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry formed by adding the input carry and the bits of the words. The truth table of the full adder is listed in table 4.

Table 4: Truth table for the 3 bit full adder

x	**y**	**c_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

The circuit design, of a full adder using digital logic gates, is shown in fig. 10.

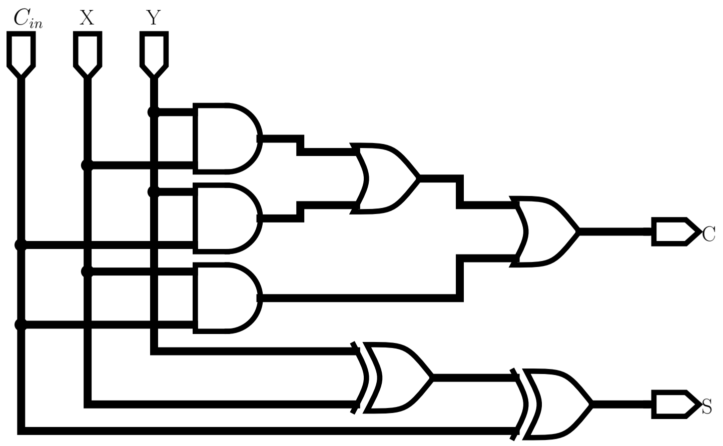

Figure 10: Circuit design of a full adder

The VHDL code for the full adder is shown in listing 13. This implementation is quite simple and follows the basic description from the design shown in fig. 10. The sensitivity list includes all the inputs, so if any of them changes the output gets recalculated.

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

entity full_adder is
  port (
    x_in, y_in, c_in : in  std_logic;
    s_out, c_out     : out std_logic
  );
end entity;

architecture behavior of full_adder is
begin
  DUT: process(x_in, y_in, c_in)
  begin
    s_out <= x_in xor y_in xor c_in;
    c_out <= (x_in and y_in) or (x_in and c_in) or (y_in and c_in);
  end process;
end architecture;
```

Listing 13: Full adder

For the same reasons as before mentioned, the test bench for the full adder uses a for loop to iterate through all the possible combinations of the inputs and outputs. If an unexpected value is presented, to assert instruction used, will underline the error. The test bench implantation is shown in below.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;

library STD;
use STD.textio.all;

entity tb_full_adder is
end entity;

architecture bench of tb_full_adder is
  component full_adder
    port (
      x_in, y_in, c_in : in std_logic;
      s_out, c_out     : out std_logic
  );
  end component;

  signal x_tb_in  : std_logic;
  signal y_tb_in  : std_logic;
  signal c_tb_in  : std_logic;
  signal s_tb_out : std_logic;
  signal c_tb_out : std_logic;

begin
  DUT: full_adder port map(x_in => x_tb_in, y_in => y_tb_in, c_in => c_tb_in, s_out => s_tb_out, c_out => c_tb_out);
  process

    file Fin : text open read_mode is "full_adder_input.txt";

    variable current_read_line   : line;
    variable current_read_field1 : std_logic;
    variable current_read_field2 : std_logic;
    variable current_read_field3 : std_logic;
    variable current_read_field4 : std_logic;
    variable current_read_field5 : std_logic;

  begin
    while (not endFile(Fin)) loop

      readline(Fin, current_read_line);
      read(current_read_line, current_read_field1);
      read(current_read_line, current_read_field2);
      read(current_read_line, current_read_field3);
      read(current_read_line, current_read_field4);
      read(current_read_line, current_read_field5);

      x_tb_in <= current_read_field1;
      y_tb_in <= current_read_field2;
      c_tb_in <= current_read_field3;

      wait for 50 ns;

      assert(c_tb_out = current_read_field4) report "Error in carry" severity failure;
      assert(s_tb_out = current_read_field5) report "Error in sum" severity failure;
    end loop;
    wait;
  end process;
end architecture;
```

Listing 14: Test bench for the full adder

8 Comparator

A (magnitude) comparator is a combinational circuit that compares two numbers A and B and determines their relative magnitudes. The outcome of the comparison is specified by a variable which indicates, if the numbers are equal or not.

The VHDL implementation following in listing 15, uses generic value for the size of the numbers, this means that the comparator can be used for any size of numbers.

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

entity comparator is
  generic(N: integer:= 4);
  port (
    x_in, y_in: in  std_logic_vector(N-1 downto 0);
    is_eq: out std_logic
  );
end entity;

architecture behavior of comparator is
begin
  DUT: process(x_in, y_in)
  begin
    is_eq <= '1' when (x_in = y_in)
    else '0';
  end process;
end architecture;
```

Listing 15: Comparator

For the testing of the comparator, the N value was set to 4, i.e. 4 bits numbers. The test bench for the comparator is shown in listing 16.

```
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tb_comparator is
end tb_comparator;

architecture behavior of tb_comparator is
  signal x_tb_in, y_tb_in: std_logic_vector(3 downto 0) := "0000";
  signal is_eq_tb: std_logic;
  constant clk: time := 1 ns;
begin
DUT: entity work.comparator port map (x_tb_in, y_tb_in, is_eq_tb);
  process
  begin
    wait for clk;
    x_tb_in <= "0001";
    y_tb_in <= "0000";
    wait for clk;
    x_tb_in <= "0110";
    y_tb_in <= "0110";
    wait for clk;
    wait;
  finish;
  end process;
end architecture;
```

Listing 16: Test bench for Comparator

The simulation results of the above test bench are summed up in the following waveform image.

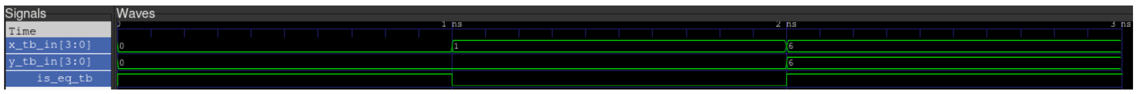

Figure 11: Waveform of Comparator

9 Encoder

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value.

In this implementation, the encoder is a 16 to 4 circuit. The table for how the systems switches between the given values is shown below table 5.

Table 5: Truth table for the 16 to 4 encoder

Input (16-bits)	**Output (4-bits)**
0000000000000001	0000
0000000000000010	0001
0000000000000100	0010
0000000000001000	0011
0000000000010000	0100
0000000000100000	0101
0000000001000000	0110
0000000010000000	0111
0000000100000000	1000
0000001000000000	1001
0000010000000000	1010
0000100000000000	1011
0001000000000000	1100
0010000000000000	1101
0100000000000000	1110
1000000000000000	1111

The implemented encoder, shown in listing 17, reads the input vector and then uses a case statement to select the output value. The sensitivity list only consists of the input vector. A default value of Undefined ('XXXX') is also used to avoid latches.

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

entity encoder_16to4 is
  port (
    D_in: in std_logic_vector(15 downto 0);
    Q_out: out std_logic_vector(3 downto 0)
  );
end entity;

architecture behavior of encoder_16to4 is
begin
  DUT: process(D_in)
  begin
    case D_in is
      when "0000000000000001" => Q_out <= "0000";
      when "0000000000000010" => Q_out <= "0001";
      when "0000000000000100" => Q_out <= "0010";
      when "0000000000001000" => Q_out <= "0011";
      when "0000000000010000" => Q_out <= "0100";
      when "0000000000100000" => Q_out <= "0101";
      when "0000000001000000" => Q_out <= "0110";
      when "0000000010000000" => Q_out <= "0111";
      when "0000000100000000" => Q_out <= "1000";
      when "0000001000000000" => Q_out <= "1001";
      when "0000010000000000" => Q_out <= "1010";
      when "0000100000000000" => Q_out <= "1011";
      when "0001000000000000" => Q_out <= "1100";
      when "0010000000000000" => Q_out <= "1101";
      when "0100000000000000" => Q_out <= "1110";
      when "1000000000000000" => Q_out <= "1111";
      when others => Q_out <= "XXXX";
    end case;
  end process;
end architecture;
```

Listing 17: Encoder 16 to 4

For the testing of the encoder, a text file was used to store the input and output values and like shown before, the contents of the file were read and compared with the values that the encoder produced.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;

library STD;
use STD.textio.all;

entity tb_encoder_16to4 is
end entity;

architecture behavior of tb_encoder_16to4 is
  signal D_tb_in: std_logic_vector(15 downto 0);
  signal Q_tb_out: std_logic_vector(3 downto 0);
begin
  DUT: entity work.encoder_16to4 port map (D_tb_in, Q_tb_out);
  process
    file f_in: text open read_mode is "encoder_16to4_test.txt";

    variable current_read_line: line;
    variable current_read_field1: std_logic_vector(0 to 15);
    variable current_read_field2: std_logic_vector(0 to 3);
  begin
    while (not endFile(f_in)) loop
      readline(f_in, current_read_line);
      read(current_read_line, current_read_field1);
      read(current_read_line, current_read_field2);

      D_tb_in <= current_read_field1;
      wait for 50 ns;

      assert(Q_tb_out = current_read_field2);
    end loop;
    wait;
  end process;
end architecture;
```

Listing 18: Encoder 16 to 4

The simulation results of the test bench in listing 18, are shown in fig. 12. The easy-to-understand functionality of the circuit, makes it possible to verify the results by looking in the waveform.

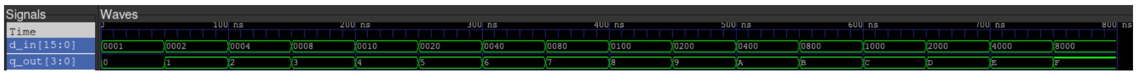

Figure 12: Waveform of the encoder

A Supplementary code for LED counter & multiplier

A.1 LED counter

A.1.1 LED Decoder

The LED Decoder is a combination of LEDs that should display the value of the decoder at any given moment. The decoder takes in 4-bits at its input and outputs 7 bits. Essentially, the LED will represent the hexadecimal value of the 4-bits as shown in the table below.

Table 6: LED Decoder

Binary value (4-bits)	LED value
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The VHDL code for the LED Decoder is shown here listing 19.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.ALL;

entity led_decoder is
  port (
    clk: in std_logic;
    d: in  std_logic_vector (3 downto 0);
    s: out std_logic_vector (6 downto 0)
  );
end entity;

architecture behavior of led_decoder is
begin
  DUT: process(clk, d)
  begin
    if rising_edge(clk) then
      s<="1110111" when d="0000" else
      "0010010" when d="0001" else
      "1011101" when d="0010" else
      "1011011" when d="0011" else
      "0111010" when d="0100" else
      "1101011" when d="0101" else
      "1101111" when d="0110" else
      "1010010" when d="0111" else
      "1111111" when d="1000" else
      "1111011" when d="1001" else
      "1111110" when d="1010" else
      "0101111" when d="1011" else
      "1100101" when d="1100" else
      "0011111" when d="1101" else
      "1101101" when d="1110" else
      "1101100";
    end if;
  end process;
end architecture;
```

Listing 19: LED Decoder

A.1.2 Counter

The counter implemented, is a 4-bit counter that counts from 0 to 15. The increment of the counter is a synchronous manner, i.e. the counter value is updated only when the clock signal is high. When the counter reaches its maximum value (15), it resets back to 0 (wrap around).

The VHDL code for the counter is shown here listing 20. The reset input is used to reset the counter back to 0. Besides that, the counter also has a clock input, which is used to increment the counter value. The counter value returned is a 4-bit value, using a vector.

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

entity counter is
  port (
    clk, reset: in std_logic;
    count: out std_logic_vector (3 downto 0)
  );
end entity;

architecture behavior of counter is
begin
  process(clk, reset)
  begin
    if reset = '1' then
      count <= "0000";
    elsif rising_edge(clk) then
      if count = "1001" then
        count <= "0000";
      else
        count <= count + 1;
      end if;
    end if;
  end process;
end architecture;
```

Listing 20: Counter

A.2 Multiplier

A.2.1 Adder/Subtractor

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

entity add_sub is
  generic (N: INTEGER:= 4);
  port(
    addsub: in std_logic; -- addsub = 0 (add), addsub = 1 (sub)
    x, y: in std_logic_vector (N-1 downto 0);
    s: out std_logic_vector (N-1 downto 0);
    overflow: out std_logic;
    cout: out std_logic);
end add_sub;

architecture structure of add_sub is
  component full_adder
    port(
      x_in, y_in, c_in : in  std_logic;
        s_out, c_out: out std_logic
    );
  end component;

  signal c: std_logic_vector (N downto 0);
  signal yx: std_logic_vector (N-1 downto 0);
begin
     c(0) <= addsub;
  cout <= c(N);
  overflow <= c(N) xor c(N-1);

  gi: for i in 0 to N-1 generate
      yx(i) <= y(i) xor addsub;
      fi: full_adder port map (
        c_in => c(i), x_in => x(i), y_in => yx(i), s_out => s(i),
        c_out => c(i+1)
      );
      end generate;
end structure;
```

Listing 21: n-bits adder/subtractor

A.2.2 Register

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

-- N-bit Register
-- E = '1', clear = '0' --> Input data 'D' is copied on Q
-- E = '1', clear = '1' --> Q is cleared (0)
entity n_register is
     generic (N: INTEGER:= 4);
  port (
    clk, reset   : in std_logic;
      E, clear   : in std_logic; -- clear: Synchronous clear
    D       : in std_logic_vector (N-1 downto 0);
      Q       : out std_logic_vector (N-1 downto 0));
end n_register;

architecture Behavioral of n_register is
  signal Qt    : std_logic_vector (N-1 downto 0);
begin
  process (reset, clk)
  begin
    if reset = '0' then
      Qt <= (others => '0');
    elsif (clk'event and clk = '1') then
      if E = '1' then
        if clear = '1' then
          Qt <= (others => '0');
        else
          Qt <= D;
        end if;
      end if;
    end if;
  end process;
  Q <= Qt;
```

```
end Behavioral;

```

Listing 22: n-bits register

A.2.3 Parallel Shift Register

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

-- N-bit Parallel access Shift register:
-- DIR = ”LEFT” --> Shift to the left: from LSB to MSB
-- DIR = ”RIGHT” --> Shift to the right: from MSB to LSB
-- s_l = 0 --> Shift operation (MSB to LSB)
-- s_l = 1 -> Parallel load
entity parallel_shift_reg is
     generic (
    N   : INTEGER  := 4;
      DIR  : STRING  := ”LEFT”
  );

  port (
    clk, reset  : in std_logic;
      din, E, s_l  : in std_logic; -- din: shiftin input
    D      : in std_logic_vector (N-1 downto 0);
      Q      : out std_logic_vector (N-1 downto 0);
        shift_out  : out std_logic
  );
end parallel_shift_reg;

architecture Behavioral of parallel_shift_reg is
  signal Qt: std_logic_vector (N-1 downto 0);
begin

a0: assert (DIR = ”LEFT” or DIR = ”RIGHT”)
    report ”DIR can only be LEFT or RIGHT” severity error;

  process (reset, clk)
  begin
    if reset = '0' then
      Qt <= (others => '0');
    elsif (clk'event and clk = '1') then
      if E = '1' then
        if s_l = '1' then
          Qt <= D;
        else
          if DIR = ”LEFT” then
            Qt(0) <= din;
            for i in 1 to N-1 loop
              Qt(i) <= Qt(i-1);
            end loop;
          elsif DIR = ”RIGHT” then
            Qt(N-1) <= din;
            for i in 0 to N-2 loop
              Qt(i) <= Qt(i+1);
            end loop;
          end if;
        end if;
      end if;
    end if;

  end process;

  Q <= Qt;

  gl: if DIR = ”LEFT” generate
```

```
      shift_out <= Qt(N-1);
    end generate;
  gr: if DIR = "RIGHT" generate
      shift_out <= Qt(0);
    end generate;
end Behavioral;
```

Listing 23: n-bits parallel shift register

B Tools used for digital design and VHDL compilation

There are a lot of development enviroments suites for analyzing, compiling, simulating and synthesizing VHDL code. Those products come with a hefty price for licensing and the proprietary. For this project, a vocal point was to use free and open source software.

For the analyzing, compiling and simulating of the VHDL code, the open source software GHDL was used. Unlike some other simulators, GHDL is a compiler: it directly translates a VHDL file to machine code, without using an intermediary language such as C/C++.

If synthesizing was needed, the open source software Yosys is an excellent tool for that.

To validate the behavior of the designs, the test benches simulations were exported as vcd files and then imported into GTKWave, which is an external waveform viewer.

The design of some circuits with logic gates, was done using the open source software Logisim-Evolution. This software is a digital logic designer and simulator. It allows the user to design and simulate digital logic circuits.

Finally, the IDE of choice for this project was VSCodium with the extension of ghdl-ls, that acts as a language server for VHDL. This extension provides syntax highlighting, code completion, linting and other features. Additionally, for each design, a Makefile was created to automate the process of compiling and simulating the VHDL code.

How to use GHDL and GTKWave

The GHDL framework is available through various package managers. The most common way to install GHDL is by building yourself from source. The instructions for building GHDL from source are available here. For the GTKWave viewer, visit their official website and follow from there the instructions for your operating system.

For each design, a source file of the particular design is needed, which can consists of one or more VHDL files in a hierarchical way. Furthermore, a test bench file is needed to simulate the design. The test bench file is usually named with the prefix tb_ followed by the name of the design.

For each design the following instructions have to be used:

- Analyze the source file(s):

```
ghdl -a -std=08 -ieee=synopsys <design>.vhd
```

- Analyze the testbench file(s):

```
ghdl -a -std=08 -ieee=synopsys tb_<design>.vhd
```

- Generate executable file:

```
ghdl -e -std=08 -ieee=synopsys tb_<design>.vhd
```

- Run the simulation:

```
ghdl -r -std=08 -ieee=synopsys tb_<design>.vhd
        -vcd=tb_<design>.vcd
```

- View the waveform:

```
gtkwave tb_<design>.vcd
```

Where `<design>` is the name of the design. If all tests are passed, then the no errors will be shown on the terminal. Alternatively, for each design the Makefile can be used to automate the process.

References

[1] M. M. Mano and M. D. Ciletti, *Digital Design (4th Edition).* USA: Prentice-Hall, Inc., 2006.

[2] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design with CD-ROM.* USA: McGraw-Hill, Inc., 3 ed., 2008.

[3] D. Llamocca, “Unit 7 - introduction to digital system design.” https://www.secs.oakland.edu/ llamocca/, 2023.

[4] V. Angelov, “Vhdl vorlesung ss2009.” https://www.physi.uni-heidelberg.de/ angelov/VHDL/VHDL_SS09_Teil05.pdf, 2009.