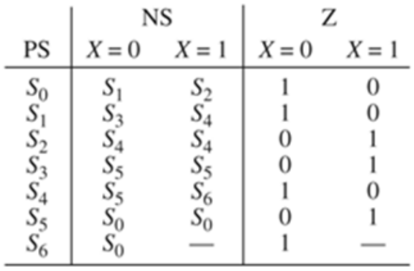
More on Finite State Machines (FSMs)

1. Objective

To study several different ways of specifying and implementing finite state machines (FSMs). We also discuss finite state machines with datapath (FSMD).

2. Introduction

There are two basic types of sequential circuits: Mealy and Moore. Because these circuits transit among a finite number of internal states, they are referred to as finite state machines (FSMs). In a Mealy circuit, the outputs depend on both the present inputs and state. In a more circuit, the outputs depend only on the present state. The most common way of schematically representing a Mealy sequential circuit is shown in Fig.1.



PS	NS		Z	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

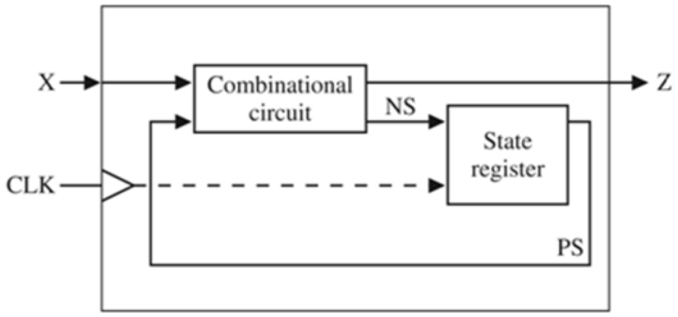


Figure 1 State transition table and block diagram of a Mealy type seq. circuit (BCD to excess-3 converter)

The state register normally consists of D flip-flops (DFFs). However, other types of flip-flops can be utilized, such as JKFFs. The normal sequence of events is: (1) inputs X change to a new value, (2) after a clock period delay, outputs Z and next state NS become stable at the output of the combinational circuit, (3) the next state signals NS are stored in the state register; that is, next state NS replace present state PS at the output of the state register, which feeds back into the combinational circuit. At this time, a new cycle is ready to start. These operational cycles are synchronized with the clock signal CLK.

It is worth mentioning that some authors further classify sequential circuits into two categories. The first category, referred to as “regular sequential circuits”, includes circuits like (shift) registers, FIFOs, and binary counters and variants. The second category, referred to as “finite state machines” (FSMs), include circuits that typically do not exhibit a simple, repetitive pattern.

3. Example 1: Mealy Machine Design – BCD to Excess-3 Code Converter

In this example, we’ll design a serial converter that converts a **binary coded decimal (BCD) digit** to an **excess-3-coded decimal digit**. Excess-3 binary-coded decimal (XS-3) code, also called biased representation or Excess-N, is a complementary BCD code and numeral system. It was used on some older computers with a pre-specified number N as a biasing value. It is a way to represent values with a balanced number of positive and negative numbers. In our example, the XS-3 code is formed by adding 0011 to the BCD digit. The table and state graph in Fig.2 describe the functionality of our design.

X Input (BCD)				Z Output (excess-3)			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

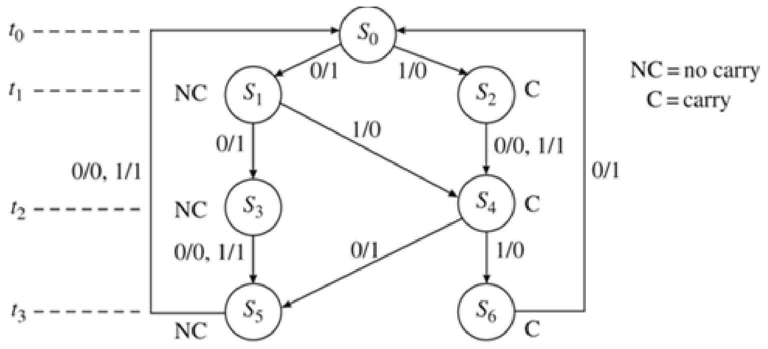


Figure 2 Code converter: table and state graph.

There are several ways to model this sequential machine. One popular/common approach is to use **two processes** to represent the two parts of the circuit: the combinational part and the state register. For clarity and flexibility, we use VHDL’s *enumerated data type* to represent the FSM’s states. The following VHDL code describes the converter (file **code_conv_2processes.vhd**):

```
-- Behavioral model of a Mealy state machine: code converter w/ 2 processes
-- It is based on its state table. The output (Z) and next state are
-- computed before the active edge of the clock. The state change
-- occurs on the rising edge of the clock.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Code_Converter is
  port(
       enable: in std_logic;
       X, CLK: in std_logic;
       Z: out std_logic);
end Code_Converter;

architecture Behavioral of Code_Converter is

type state_type is (S0, S1, S2, S3, S4, S5, S6);
signal State, Nextstate: state_type;
-- a different way: represent states as integer signals:
-- signal State, Nextstate: integer range 0 to 6;

begin

-- Combinational Circuit
process(State, X)
begin
 case State is
       when S0 =>
         if X = '0' then Z <= '1'; Nextstate <= S1;
         else Z <= '0'; Nextstate <= S2; end if;
       when S1 =>
         if X = '0' then Z <= '1'; Nextstate <= S3;
         else Z <= '0'; Nextstate <= S4; end if;
       when S2 =>
         if X = '0' then Z <= '0'; Nextstate <= S4;
         else Z <= '1'; Nextstate <= S4; end if;
```

```
      when S3 =>
        if X = '0' then Z <= '0'; Nextstate <= S5;
        else Z <= '1'; Nextstate <= S5; end if;
      when S4 =>
        if X = '0' then Z <= '1'; Nextstate <= S5;
        else Z <= '0'; Nextstate <= S6; end if;
      when S5 =>
        if X = '0' then Z <= '0'; Nextstate <= S0;
        else Z <= '1'; Nextstate <= S0; end if;
      when S6 =>
        if X = '0' then Z <= '1'; Nextstate <= S0;
        else Z <= '0'; Nextstate <= S0; end if;
      when others => null;          -- should not occur
 end case;
end process;

-- State Register
process (enable, CLK)
begin
  if enable = '0' then
    State <= S0;
  elsif rising_edge (CLK) then
    State <= Nextstate;
  end if;
end process;

end Behavioral;
```

Note that in each branch of the case statement, the output **Z** and **Nextstate** are assigned values. The second process represents the state register, which is updated on the rising edge of the **CLK** signal.

To test this converter on the board, we'll design a test circuit that uses two shift-registers, the converter, and a clock divider, as shown in the diagram of Fig.3. The input is provided parallel as four bits via four slide switches while the output is displayed on four LEDs. We use a clock divider to generate a slower clock signal (about 1 Hz) to make it easier to monitor the operation of the whole system.

Create a new project (call it **bcd_to_excess3_test**) and add to it the following VHDL files: **code_conv_2processes.vhd, ck_divider.vhd, shift_register.vhd, and top_level.vhd**. These files contain the declaration and description of all necessary entities to implement the system from Fig.3. These files are included in the provided .zip file for this example.Read **top_level.vhd** and figure out what exactly the "control" block in Fig.3 does.

Do necessary pin assignment, synthesize the design, place and route (i.e., fitter); generate the programming file and program the FPGA. Verify the operation of your design. Observe and comment.

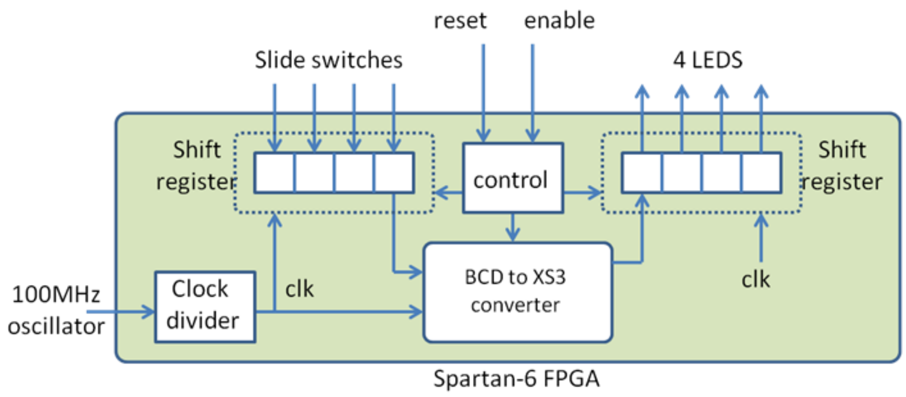


Figure 3 Block diagram of the top level design to test the BCD to XS3 converter.

4. Example 2: Finite state machine with datapath (FSMD) - Bit Difference Calculator

A finite state machine with datapath (FSMD) combines a FSM and regular sequential circuits. The FSM, sometimes referred to as a **control-path** or **controller**, examines the external commands and status and generates control signals to specify operations of the regular sequential circuits, which are known collectively as a **data-path** [1]. The FSMD is used to implement systems described by *RT (register transfer) methodology*, where the system's functionality is specified as data manipulation and transfer among a collection of registers.

Most realistic circuits combine a controller and a datapath to perform some computation. The use of the FSMD model is especially recommended whenever the structure of the datapath is important. For example, if you are creating a custom pipelined datapath for a specific application, specifying the structure of the pipeline is likely important.

The combination of a controller and datapath can be represented using several models in VHDL. In this example, we'll look at two different models. To do that, we'll design and simulate a simple example: a **bit difference calculator**. The design's description is as follows: Given an input of a generic width, the design entity calculates the difference between the number of 1s and 0s. If for example there are 3 more 1s than 0s, the output is 3. If there are 3 more 0s than 1s, the output is -3.

Implementation A: behavioral model using two processes

A simplified pseudocode description of the **bit difference calculator** is as follows:

```
Inputs: go, input (arbitrary width)
Outputs: output(arbitrary width), done (1 bit)

while (go == 0);
value = input;    // Store input in a register called value.
diff = 0;
for width iterations {
      if bit0 of value == 1
            diff++;
      else
            diff--;
      value = shiftRight(value,1);
}
output = diff;
done = 1;
```

One possible implementation as a FSMD is described by the state graph in Fig.4.
The VHDL file **top_level_bit_diff_impl_A.vhd** describes the entity bit_diff and its architecture the design.

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

entity bit_diff is
  generic (
    width : positive := 16);
  port (
    clk    : in  std_logic;
    rst    : in  std_logic;
    go     : in  std_logic;
    input  : in  std_logic_vector(width-1 downto 0);
    output : out std_logic_vector(width-1 downto 0);
    done   : out std_logic);
end bit_diff;
```

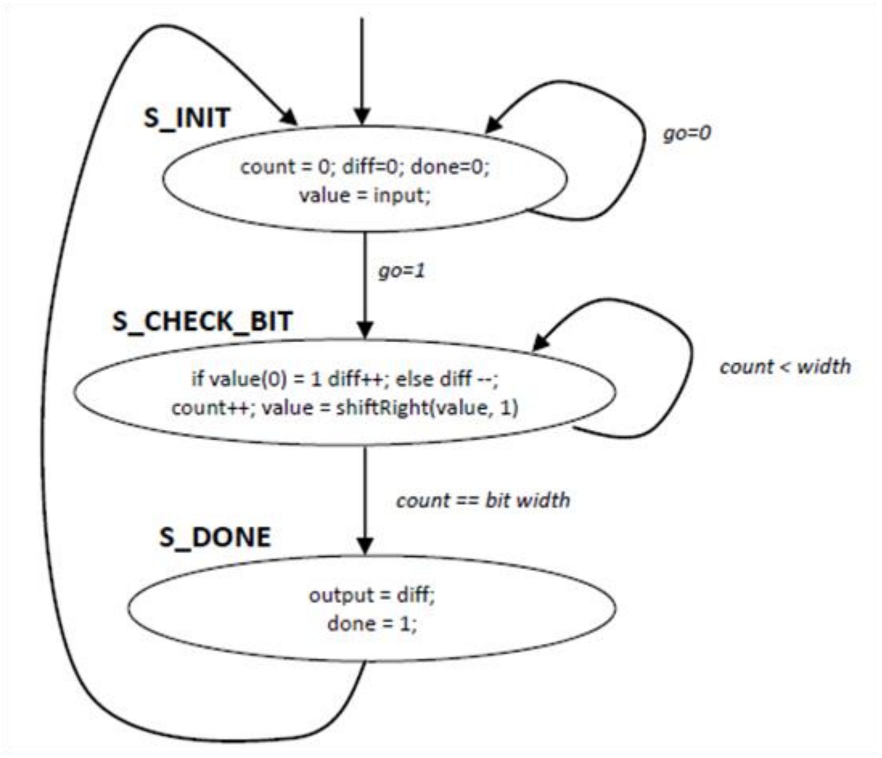

S_INIT
count = 0; diff=0; done=0;
value = input;
go=0
go=1
S_CHECK_BIT
if value(0) = 1 diff++; else diff --;
count++; value = shiftRight(value, 1)
count < width
count == bit width
S_DONE
output = diff;
done = 1;

Figure 4 State graph of FSMD implementation.

```
architecture FSMD_2P of bit_diff is

  type STATE_TYPE is (S_INIT, S_CHECK_BIT, S_STORE_OUTPUT, S_DONE);

  signal state, next_state : STATE_TYPE;
  signal value, next_value : std_logic_vector(width-1 downto 0);
  signal diff, next_diff   : signed(width-1 downto 0);
  signal count, next_count : integer range 0 to width;
  signal output_s, next_output : std_logic_vector(width-1 downto 0);
begin

  -- this process defines all registers used in the FSMD
  process(clk, rst)
  begin
    if (rst = '1') then
      value    <= (others => '0');
      count    <= 0;
      diff     <= (others => '0');
      output_s <= (others => '0');
      state    <= S_INIT;
    elsif (clk'event and clk = '1') then
      -- these are the only registers used by the 2-process FSMD
      value    <= next_value;
      count    <= next_count;
      diff     <= next_diff;
      output_s <= next_output;
      state    <= next_state;
    end if;
  end process;

  -- combinational logic
  process(go, input, value, count, diff, output_s, state)
    variable temp : integer range 0 to width;
  begin
    next_count  <= count;
    next_value  <= value;
    next_diff   <= diff;
    next_output <= output_s;
    next_state  <= state;
    done <= '0';

    case state is
      when S_INIT =>
        next_count <= 0;
        next_diff  <= (others => '0');
        next_value <= input;
        if (go = '1') then
          next_state <= S_CHECK_BIT;
        end if;

      when S_CHECK_BIT =>
        if (value(0) = '0') then
          next_diff <= diff - 1;
        elsif (value(0) = '1') then
          next_diff <= diff + 1;
        end if;
        next_value <= std_logic_vector(shift_right(unsigned(value), 1));
        temp := count + 1;
        next_count <= temp;
        if (temp = width) then
          next_state <= S_STORE_OUTPUT;
        end if;

      when S_STORE_OUTPUT =>
```

```
        next_output <= std_logic_vector(diff);
        next_state  <= S_DONE;

      when S_DONE =>
        done       <= '1';
        next_state <= S_INIT;

      when others => null;
    end case;
  end process;

  output <= output_s;

end FSMD_2P;
```

At this time, you should create a simple testbench VHDL file (you can do it by modifying **testbench_top_level.vhd** file from Example 1) and simulate the above entity. Verify its operation and comment.

Implementation B: structural model using component instantiations for registers, muxes, adders, subtracters, etc.

The structural implementation is recommended when the exact structure of the datapath is important. In this model, we separate the controller and datapath from each other. Then, typically, we define the datapath structurally and then combine it with a corresponding controller (FSM) described using any of the possible models discussed in Example 1.

For example, assume that we really wanted to implement the datapath described in Fig.5. Then, the following files: **top_level_bit_diff_impl_B.vhd, datapath.vhd, fsm.vhd, add.vhd, sub.vhd, reg.vhd, mux2x1.vhd, comp.vhd** describe all the entities required for implementing the design. Read these files to understand the description. Then, use the same testbench that you created to simulate the previous implementation (implementation A) of this design to verify also the operation of this description too.

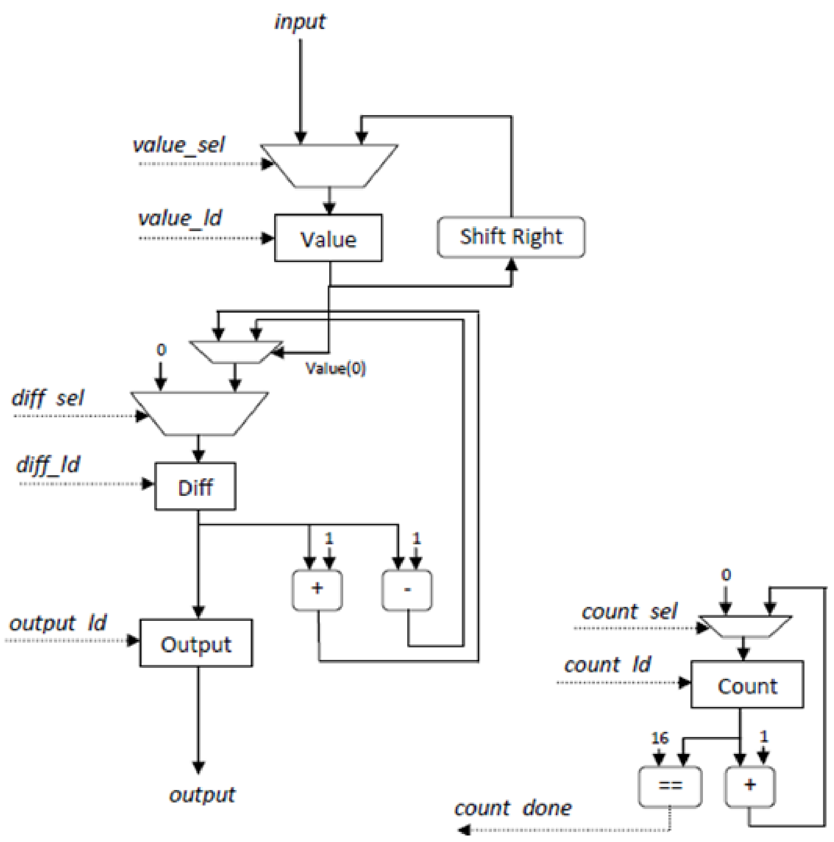


Figure 5 Block diagram of datapath.

5. Assignment - Optional

Design and code in VHDL the converter from Example 1 but as a Moore machine. Verify its operation both on the FPGA board and in simulation using a testbench. Write a report. The report should include the state diagram, VHDL code, discussion of operation, and waveforms.

6. References

[1] P.P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability and Scalability, Wiley-Interscience, 2006.