Hands-on Sessions on Krypton CPLD board

(Designed at Wadhwani Electronics Lab)

Digital Circuits & Systems Lab Manual

EE Department, IIT Bombay

Digital Circuits & Systems Lab

Pre-Requisite

Pre-Requisite :

1. Digital Circuits & System Course
2. Basic VHDL
3. Familiarity with Quartus Prime (Software)

Pre-Requisite by Topic for Digital Circuits System:

1. Number System
2. Basic Gates
3. Combinational Circuits
4. Sequential Circuits
5. Finite State Machine(FSM)

Pre-Requisite by Topic for VHDL:

1. Design Flow for logic circuit in VHDL
2. Structural Modelling
3. Dataflow Modelling
4. Behavioral Modelling

All required references are attached in the Supporting Document on page-2.

Supporting Documents

Experiment - 0

Introductory Lab

Instructions:

1. The objective of this experiment is to get familiar with Quartus design tool, VHDL description and verification of design using testbench.
2. Follow this video step-wise to perform this experiment - Quartus Design Flow.
3. Have a look at these videos to get familiar with VHDL and Structural description - Introduction to VHDL, Structural Descritpion in VHDL.
4. Link of these videos are also given in the supporting document at the beginning of the manual.
5. The files required to perform RTL and Gate level simulation simulation using testbench are available here - Resource Files.
6. Do pen paper design, use proper labeling for each wire and use the same labels in the VHDL code.
7. Perform RTL simulation and Gate level simulation using the provided testbench and tracefile.

Problem Statement:

1. Design of Full Adder -
   - Design a Full Adder circuit (pen-paper design) using basic gates.
   - Now describe the same Full Adder design in VHDL using basic gates provided in Gates.vhdl (which is present in Resource Files).
   - Verify the working of your design by performing RTL and Gate level simulation using the given tracefile and testbench.

   NOTE:

   TRACEFILE format for Full Adder -

   Input{X2 X1 X0} Output{Sum Carry} MASK{1 1}

   Click on Full Adder to get the tracefile of Full Adder.

2. Design of XOR gate using NAND gates-
   - Design XOR gate using NAND gates (pen-paper design)
   - Now describe the same XOR gate using NAND gate in VHDL using NAND gate provided in Gates.vhdl (which is present in Resource Files).
   - Verify the working of your design by performing RTL and Gate level simulation using the given tracefile and testbench.

   NOTE:

   TRACEFILE format for XOR gate -

   Input{X1 X0} Output{Y0} MASK{1}

   Click on XOR to get the tracefile of XOR gate.

Experiment - 1

Full Adder Design using NAND gate

Instructions:

1. NAND is a universal gate.
2. For writing VHDL description use the NAND gate provided in Gates.vhdl(which has been provided in Resource Files).
3. Do pen paper design, use proper labeling for each wire and use the same labels in the VHDL code.
4. Perform RTL simulation and Gate level simulation using the provided testbench and tracefile.

Problem Statement:

1. Design Full Adder using NAND gates (pen-paper design).
2. Describe Full Adder using NAND gates in VHDL using Structural modeling.
3. Verify the working of your design by performing RTL and Gate level simulation using the given tracefile and testbench.

   NOTE:

   TRACEFILE format for Full Adder -

   Input{X2 X1 X0} Output{Sum Carry} MASK{1 1}

   Click on Full Adder to get the tracefile of Full Adder.

Experiment 2
4-Bit Adder-Subtractor

Instructions:

1. Use structural modelling for this experiment, that is, instantiate components and use port mapping to connect them.
2. Perform RTL and Gate-level simulation using the provided testbench and tracefile.
3. Link to Resource Files

Problem Statement: 4-bit Adder-Subtractor:

1. VHDL Description: Using the full adder and XOR gate as a component, describe a 4-bit ripple carry adder-subtractor in VHDL.

   INFO: 4-bit ripple carry adder-subtractor has following parts
   (a) Two 4-bit inputs : A (A3 A2 A1 A0) , B (B3 B2 B1 B0)
   (b) One 1-bit input: M
   (c) One 4-bit output : S (S3 S2 S1 S0)
   (d) One 1-bit output: Cout

   NOTE: It is a simple binary adder-subtractor that can be implemented by cascading four full adders such that the carry generated by the addition of lower significant bits forms the incoming carry for addition of the next significant bits.

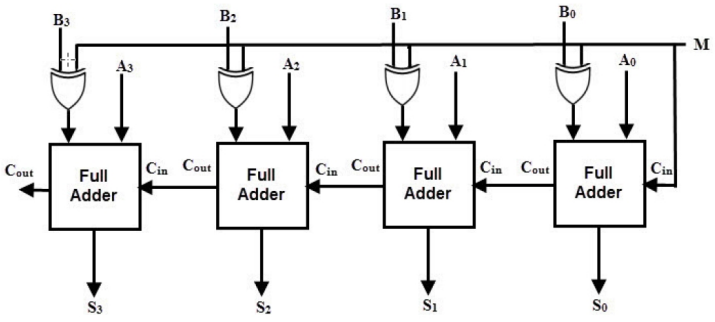


Figure 1: Design of 4 Bit Adder-Subtractor

2. Simulation: Simulate the design using the generic testbench to confirm the correctness of your description.
   NOTE: To do this, you need to use the given tracefile and modify the testbench given to you appropriately.
   Tracefile format: $(< a3\,a2\,a1\,a0 >< b3\,b2\,b1\,b0 >< M > \;< Cout >< S3\,S2\,S1\,S0 > \quad 11111)$
   Click on Tracefile to download the tracefile.

3. Expected Simulation Result of 4 Bit Adder-Subtractor:

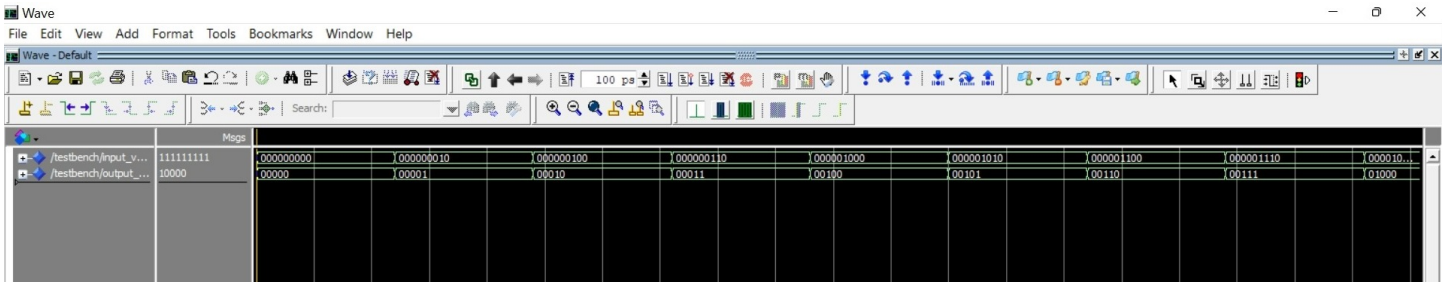


Figure 2: 4 Bit Adder-Subtractor Simulation Result

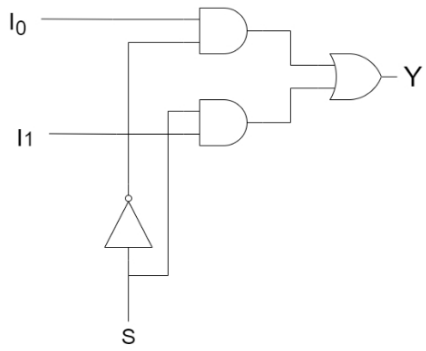
Experiment 3
Multiplexers

Instructions:

1. Use structural modelling for this experiment, that is, instantiate components and use port mapping to connect them.
2. Perform RTL and Gate-level simulation using the provided testbench and tracefile.
3. Link to Resource Files

Problem Statement:

1. **Part-A: 2x1 Mux**
   (a) VHDL description: Write the VHDL description of a 2x1 multiplexer as shown in figure below.
   INFO: Multiplexer is a combinational circuit that has maximum of 2^n data inputs, 'n' selection lines and single output lines.



   (b) Write Truth table and boolean expression for output Y.

   (c) Simulation: Simulate the 2x1 multiplexer using the generic testbench to confirm the correctness of your description.
   NOTE: To do this, use the tracefile given below and modify the testbench given to you appropriately.
   Tracefile format: $(< In1 >< In0 >< S > \ < Y > \ \ 1)$ Tracefile

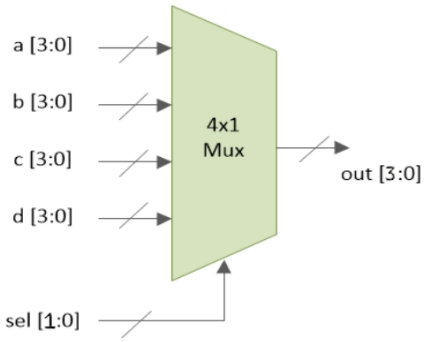
2. **Part-B: 4x1 Mux**
   (a) Design: Design 4x1 Mux using only 2x1 Mux
   INFO: $2^2 = \ 4$ data inputs, 2 select lines, 1 output line

   (b) VHDL description: Write the VHDL description of a 4x1 multiplexer designed using 2x1 Mux.

   (c) Simulation: Simulate the 4x1 multiplexer using the generic testbench to confirm the correctness of your description.
   NOTE: To do this, use the given tracefile and modify the testbench given to you appropriately.
   Tracefile format: $(< In4 >< In3 >< In2 >< In1 >< S2 >< S1 > \ < Y > \ \ 1)$ Tracefile

3. **Part-C: 4-bit 4x1 Mux**

(a) Design: Design 4-bit wide 4x1 Mux using only 4x1 Mux designed in part B.



(b) VHDL description: Write the VHDL description of a 4-bit wide 4x1 multiplexer designed using 4x1 Mux.

(c) Simulation: Simulate the 4-bit 4x1 multiplexer using the generic testbench to confirm the correctness of your description.
NOTE: To do this, use the tracefile given below and modify the testbench given to you appropriately.
Tracefile format:
($d3\ d2\ d1\ d0 >< c3\ c2\ c1\ c0 >< b3\ b2\ b1\ b0 >< a3\ a2\ a1\ a0 >< sel1\ sel0 >\ < Y3\ Y2\ Y1\ Y0 >\ \ 1111$)
Tracefile

Experiment 4
Design verification on Krypton Board

Instructions:

1. Go through Introduction to Krypton Board.
2. Installation files for UrJTAG can be downloaded from this link- UrJTAG installation Files
3. Krypton Board description and pin mapping (Page No. 6 of Krypton User Manual) of on-board peripherals can be found in this document - Krypton User Manual
4. Demo video to upload svf file on to the Krypton board - Krypton Board Demo Video
5. Download Krypton Board Test files and Driver from this link - Krypton Board Test files and Drivers

Problem Statement:

- Perform RTL and Gate-level simulation of all previous experiments.
- Generate svf file for all previous experiments and upload it onto the Krypton Board using UrJTAG.
- Ensure that all the test cases, which you had tried in the testbench, can be correctly observed on the board.

Experiment 5
Combinational Circuit

Instructions:

1. Use structural modelling for this experiment, that is, instantiate components and use port mapping to connect them.
2. Perform RTL and Gate-level simulation using the provided testbench and tracefile.
3. Verify the design on Krypton board.
4. Pin mapping of on-board peripherals can be found on Page No. 6 of the Krypton user manual - Krypton User Manual

Problem Statement: Scrabble

1. Scrabble is an extremely popular word game where players score points by placing tiles on a board that make correct words. Each letter used in the word has some points attached to it, based on how frequently it occurs in the English language. For example, the most frequent alphabet E has only 1 point whereas the least frequent letter Z has 10 points.

   The modified scoring system is as below.

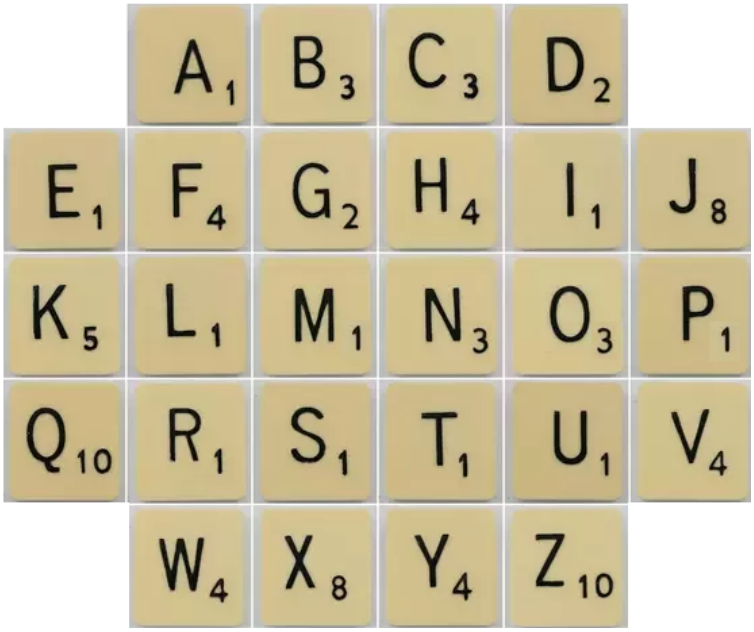


Figure 1: Score Board

2. Let us assume that we have only the first 16 letters of the English alphabets (i.e. A to P). Let 0000 represent A, 0001 represent B and so on.
3. Design a system that gives output as '1' when a given alphabet has 3 points, and '0' in other cases.
4. Verify working of your design by performing RTL and Gate-level simulation.
5. Simulate the above designs in Modelsim and validate its functionality using the given Tracefile.
   NOTE: TRACEFILE format $< X3\,X2\,X1\,X0 >< Y >\ 1$
6. Perform the experiment in the Krypton board. Inputs can be given using four switches corresponding to binary representation of alphabets (for example to give input B, four switch combination will be 0001). One LED will turn ON when given input alphabet has three points.

Experiment 6

BCD Number Addition

Instructions:

1. Do pen paper design of the circuit using proper labeling for each wire. And use same labels for the VHDL code.
2. Perform RTL simulation using the provided testbench and tracefile.
3. Brush up the concept of BCD addition before performing this experiment.
4. Verify the design on Krypton board.
5. Pin mapping of on-board peripherals can be found on Page No. 6 of the Krypton user manual - Krypton User Manual

Problem Statement:

1. Design a circuit on pen paper for addition of two BCD numbers using two 4 bit binary adder-subtractor and additional logic gates from Gates.vhdl. Input numbers are BCD(0 to 9) format only.

   Hint:

   (a) Use 4-bit binary adder for initial addition.
   (b) Design a logic circuit to detect sum greater than 9.
   (c) One more 4-bit adder to add $(0110)_2$ in the sum if sum is greater than 9 or carry is 1.

2. Write a VHDL description for the same.
3. Simulate the BCD Adder using the generic testbench and given tracefile to confirm the correctness of your design.
4. Pin plan switches S8 to S1 as input and LEDs as output.

Tracefile format: ($< A3\ A2\ A1\ A0\ B3\ B2\ B1\ B0 >\ < Y4\ Y3\ Y2\ Y1\ Y0 >\ < 1\ 1\ 1\ 1\ 1 >$) Tracefile

Experiment 7
Design Verification using Scanchain

Objective: We follow scanchain based testing when on-board peripherals are not sufficient for testing the design.

Instructions:

1. Download Scanchain files from this link - Scanchain files.
2. Follow this document for python installation and additional python libraries for scanchain- Python and pip installation guide
3. Refer this document for performing Scanchain based testing - Scanchain testing procedure

Problem Statement:

- Perform RTL and Gate-level simulation of all previous experiments.
- Ensure that all test cases of the testbench have passed.
- Note that pin planning is not required for scanchain based testing.
- Generate svf file for all previous experiments and follow this document - Scanchain testing procedure for performing Scanchain based testing.

Experiment 8
Multiplier

Multiplier

Instructions:

1. Use structural modelling for this experiment, that is, instantiate components and use port mapping to connect them.
2. Do pen paper design of the circuit using proper labeling for each wire and use same labels for the VHDL code.
3. Perform RTL simulation using the given testbench and tracefile.
4. Perform scan-chain based testing on krypton board using the given tracefile.
5. Refer this document for performing Scanchain based testing - Scanchain testing procedure

Problem Statement:

1. **Design**

   Design a multiplier circuit with one 4-bit input and one 3-bit input.

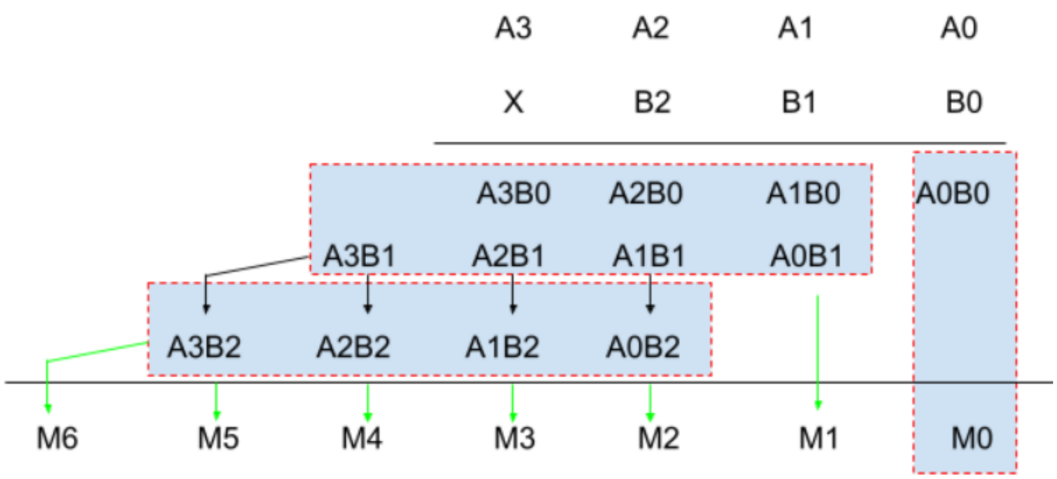


Figure 1: Block diagram

2. **VHDL description**

   Describe your designed circuit in VHDL.

3. **Simulation**
   Simulate your design using the generic testbench to confirm the correctness of your description.
   To do this, use the tracefile given below and modify the testbench given to you appropriately.

   Tracefile format: $(< A3\ A2\ A1\ A0\ B2\ B1\ B0 >\ \ < M6\ M5\ M4\ M3\ M2\ M1\ M0 >\ \ 1111111)$ Tracefile

4. **Scanchain**
   Test the correctness of your design implemented on the krypton board using scanchain technique.

Experiment 9

ALU

Instructions:

1. Use **Behavioral-Dataflow** modelling for writing VHDL description
2. Perform RTL simulation and gate level simulation using the provided testbench and tracefile.
3. Perform **Scanchain** based testing on the Krypton board.
4. Refer this document for performing Scanchain based testing - Scanchain testing procedure

Problem Statement:

1. Describe the given ALU in VHDL. This ALU circuit performs various functions based on select lines.

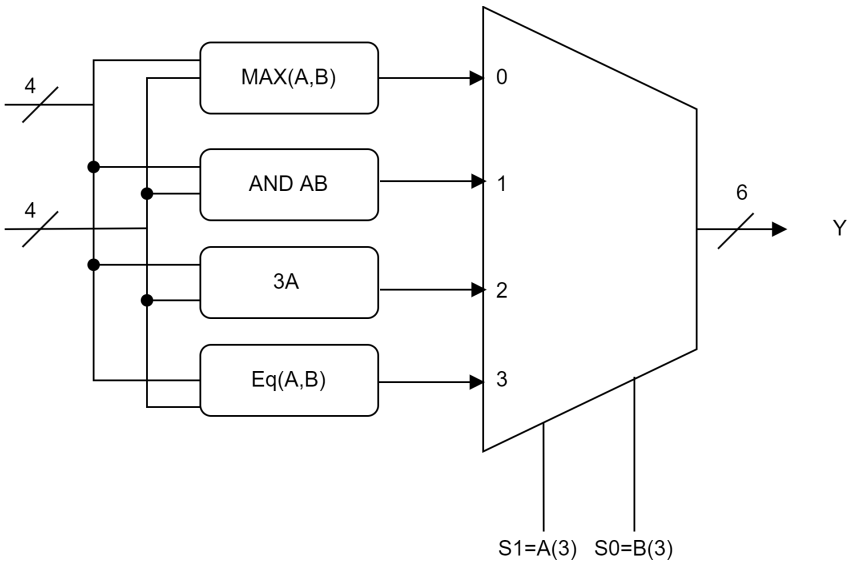


Figure 1: ALU with 4 functions

S1 S0	ALU Output
0 0	MAX(A,B): This block outputs larger number between A and B else outputs 0000.
0 1	AND A B: This block performs bitwise AND operation between A , B.
1 0	3*A: This block Produces output as 3*A
1 1	Eq(A,B): This block outputs the number whenever A=B else it should output 0000.

- In this problem MSB of inputs A and B are also working as selection lines. S0 is connected to MSB of input B [B(3)] and S1 is connected to MSB of input A [A(3)].
- Don't use multiply operation directly.
- Don't use numeric_std library.
- Simulate your design using the generic testbench to confirm the correctness of your description.

- Tracefile format $< A3\,A2\,A1\,A0\,B3\,B2\,B1\,B0 >< Y5\,Y4\,Y3\,Y2\,Y1\,Y0 >\ 1\,1\,1\,1\,1\,1$
- Perform Scanchain based testing on the Krypton board.

Download the code snippet shown below from this link - Code Snippet

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

entity alu_beh is
    generic(
        operand_width : integer:=4);
    port (
        A: in std_logic_vector(operand_width-1 downto 0);
        B: in std_logic_vector(operand_width-1 downto 0);
       op: out std_logic_vector(5 downto 0)) ;
end alu_beh;

architecture a1 of alu_beh is
    function add(A: in std_logic_vector(operand_width-1 downto 0);
    B: in std_logic_vector(operand_width-1 downto 0))
        return std_logic_vector is
            -- Declare "sum" and "carry" variable
            -- you can use aggregate to initialize the variables as shown below
            --    variable variable_name : std_logic_vector(3 downto 0) := (others => '0');
        begin
            -- write logic for addition
            -- Hint: Use for loop
            return sum;  --according to your logic you can change what you want to return
    end add;


begin
alu : process( A, B)
variable sel  : std_logic_vector(1 downto 0);

--declare other variables
begin
   -- complete VHDL code for various outputs of ALU based on select lines
   sel := ---;
   case sel is
      when "00" =>
   -- Hint: use if/else statement
   --
   -- add function usage :
   --   signal_name <= add(A,B)
   --   variable_name := add(A,B)
   --
   -- concatenate operator usage:
   --    "0000"&A
end process ; -- alu
end a1 ; -- a1
```

Experiment 10
String Detector

Lab Task:

- Describe behavioral model of the string detector Mealy type FSM in VHDL.
- Perform RTL and Gate-level simulation using the provided testbench and tracefile.
- Perform scan-chain based testing of the design.
- Refer this document for performing Scanchain based testing - Scanchain testing procedure

Problem Statement:

Design a string detector using a Mealy type FSM which will detect the occurrence of ***students*** word in a string of letters. The design accepts a sequence of letters coded in binary and outputs 1 if the required word is detected. The letters of **students** can be present anywhere in the string but in sequence.

For this experiment, letter a is encoded as 00001, b is encoded as 00010 and so on.

For instance, l**s**lp**t**l**u**lk**d**le**l**i**n**l**t**l**s**lh is the input text then the output sequence would be 00000000000000100.
The state diagram below shows the flow of the FSM.

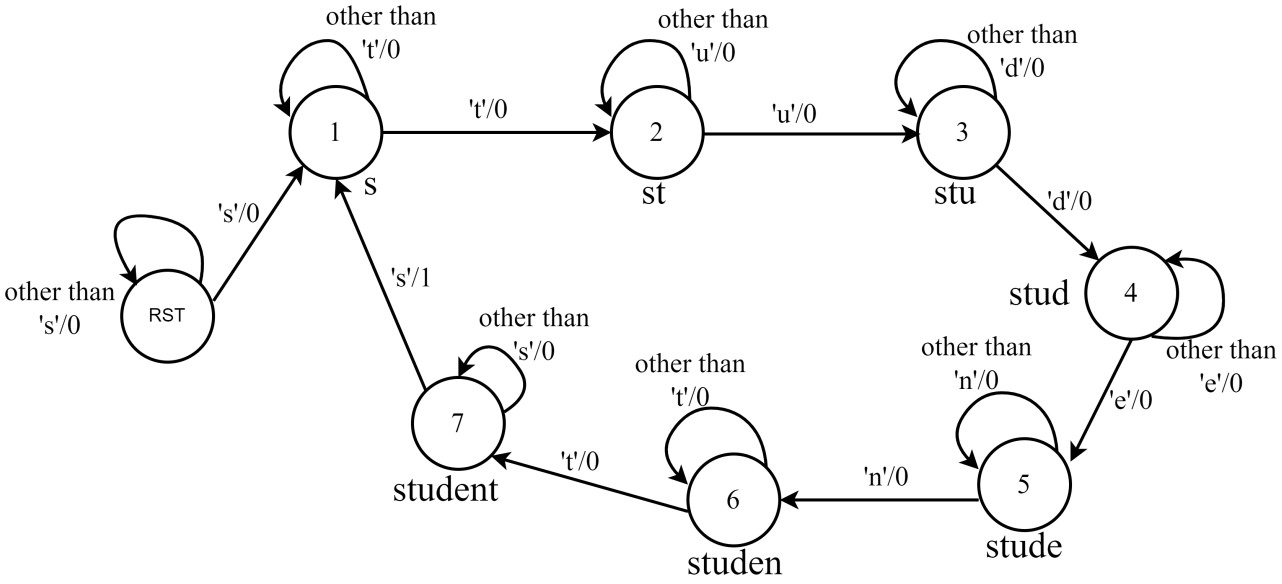


Figure 1: State diagram for detecting word "*students*" in the given string of alphabets.

In the state diagram, 8 states with one of its state being *RST* state is shown above. Here Reset is synchronous. You have to fill the state table from the above state diagram.

Reset	Input	Present state	Next state	Output
1	X	XXX	*RST*	0
0	's'	*RST*	1	
0	't'			
0	'u'			
0	'd'			
0	'e'			
0	'n'			
0	't'			
0	's'			1

Table 1: State transition table

Each state remembers the letters encountered so far of the word to be detected. If some other letter appears, it remains in the same state and waits for the correct input to arrive. For example, as shown in the figure(1) state-2 remembers "*st*" letters from the word "*students*". For any input other than "*u*", the next state is chosen as state-2. Similarly, state-7 remembers "*student*" letters from the word "*students*". For an input "*s*" when in state-7, the next state is chosen as state-1. And also outputs 1 as the entire string "*students*" is detected. For any other input other than "*s*", the next state is chosen as state-7 itself. This design shows an overlapping string detector as explained in the above example.

Design Specification:

- Input: 5-bit input signal encodes blank-space and 26 lower-case characters (from a to z and where a = 1 to z = 26) , Reset, Clock
- TRACEFILE format $< 5\ bit\ input >< Reset >< Clock > space < Output > space < Mask\ bit >$
- Output: 1-bit output

Download the code snippet shown below from this link - Code Snippet

Code Snippet:

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

entity students is
port(   inp:in std_logic_vector(4 downto 0);
        reset,clock:in std_logic;
        outp: out std_logic);
end students;

architecture bhv of students is

---------------Define state type here------------------------------
type state is (rst,s1,s2.....); -- Fill other states here
---------------Define signals of state type-----------------------
signal y_present,y_next: state:=rst;

begin
clock_proc:process(clock,reset)
begin
        if(clock='1' and clock' event) then
                if(reset='1') then
                        y_present<=
                        --Here Reset is synchronous
                        -- Fill the code here
                else
                        -- Fill the code here
                end if;
        end if;
```

```
  end process;
  state_transition_proc:process(inp,y_present)
  begin
          case y_present is
                  when rst=>
                          if(unsigned(inp)=19) then         --s has been detected
                                  y_next<=     -- Fill the code here
                          else
                              -------------------------
          ---------Fill rest of the code here---------
          -------Similarly define output process after this which will give
          -------the output based on the present state and input (Mealy machine)
  output_proc:process(y_present, inp)
  begin
     case y_present is
                when rst=>
                       outp<='0';
                    -------------
                    -----fill----
                    -------------
                    when s7=>
                      if (unsigned(inp)=19) then
                        outp<=---
                      -------------
                      ----fill------
     end case;
end process;
end bhv;
```

Experiment 11

Sequence Generator

Instructions:

1. Use **Dataflow/Behavioral** modeling for writing VHDL description
2. Perform RTL and gate level simulation using the provided testbench and tracefile.
3. Perform **Scanchain** based testing on the krypton board.
4. Refer this document for performing Scanchain based testing - Scanchain testing procedure

Problem Statement:

- Write VHDL description in **Structural-Dataflow** modeling to generate the sequence **1 1 0 0 1 1**.
- Use **structural-dataflow** modeling only.
- Reset is asynchronous in nature i.e. reset effects the output sequence irrespective of the input clock arrival.
- On Reset, sequence should start from the first '1'.
- Unused states should be mapped to one of the known state which is reset state.

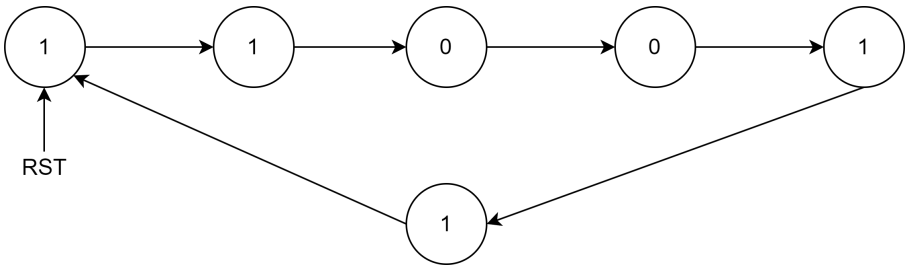


Figure 1: Sequence Generator

- Determine number of bits required to distinguish the states individually.
- Draw a state diagram to generate the states so that LSB of the states will generate the required sequence.
- Draw a state table consisting of Present State, Nest State and D FlipFlop inputs. [5 Marks]

Present State(Qn..Q1 Q0)	Next State(N_Qn..N_Q1 N_Q0)	Dn...D1 D0
one state	next state	DFF inputs
–	–	–

- From the state table with the help of K-Maps generate equations for DFF inputs in terms of present state and reset. Take LSB of the states as your output. [5 Marks]
- Tracefile format: ($< reset\ clock >$ $< output >$ $< Maskbit >$)
  Tracefile
- Perform Scanchain based testing on the Krypton Board.

Download the code snippet shown below from this link - Code Snippet

```
library ieee;
use ieee.std_logic_1164.all;
package Flipflops is
component dff_set is port(D,clock,set:in std_logic;Q:out std_logic);
end component dff_set;
component dff_reset is port(D,clock,reset:in std_logic;Q:out std_logic);
end component dff_reset;
end package Flipflops;
--D flip flop with set
library ieee;
use ieee.std_logic_1164.all;
entity dff_set is port(D,clock,set:in std_logic;Q:out std_logic);
end entity dff_set;
architecture behav of dff_set is
begin
dff_set_proc: process (clock,set)
begin
if(set='1')then -- set implies flip flip output logic high
Q <= ; -- write the flip flop output when set
elsif (clock'event and (clock='1')) then
Q <= ; -- write flip flop output when not set
end if ;
end process dff_set_proc;
end behav;
--D flip flop with reset
library ieee;
use ieee.std_logic_1164.all;
entity dff_reset is port(D,clock,reset:in std_logic;Q:out std_logic);
end entity dff_reset;
architecture behav of dff_reset is
begin
dff_reset_proc: process (clock,reset)
begin
if(reset='1')then -- reset implies flip flip output logic low
Q <= ; -- write the flip flop output when reset
elsif (clock'event and (clock='1')) then
Q <= ; -- write flip flop output when not reset
end if ;
end process dff_reset_proc;
end behav;


library ieee;
use ieee.std_logic_1164.all;
-- write the Flipflops packege declaration
entity Sequence_generator_stru_dataflow is
port (reset,clock: in std_logic;
y:out std_logic);
end entity Sequence_generator_stru_dataflow;
architecture struct of Sequence_generator_stru_dataflow is
signal D :std_logic_vector(...);
signal Q:std_logic_vector(...);
begin
-- write the equations in dataflow e.g z=a+bc written as z <= a or (b and c)
-- for DFF inputs which was derived and also for y.
-- Instantiate components dff_reset
-- and dff_set appropriately using port map statements.
end struct;
```