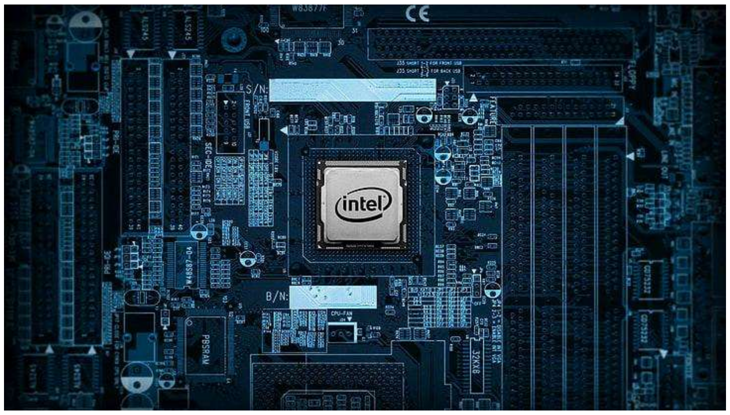
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

# DIGITAL DESIGN & COA

*INTEGRATED LABORATORY*

**REFERENCE GUIDE**

***BCS302***



Prepared by:
Mr. Akhilesh S Narayan, B.E, MSE, DiMAP
Asst. Professor,
**Dept. of CSE**

★ **PROGRAM 1**: Given a 4-variable logic expression, simplify it using the appropriate technique and simulate the same using basic gates.

```
module four_var_logic (
  input A,
  input B,
  input C,
  input D,
  output Y
);

  assign Y = (C & D) | (~A & D); // Y = CD + A'D

endmodule
```

- *Need to select any sample 4 variable expression, simplify it using K-Maps and then implement the simplified expression using Verilog.*
- *Using a test bench, you also need to test all possible combinations for the given variables.*

★ **PROGRAM 2**: Design a 4-bit full adder and subtractor and simulate the same using basic gates.

a)
```
module FullAdder4bit (
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] Sum,
    output Cout
);
wire C1, C2, C3;

// 1st bit
FullAdder FA1(A[0], B[0], Cin, Sum[0], C1);
// 2nd bit
```

```
FullAdder FA2(A[1], B[1], C1, Sum[1], C2);
// 3rd bit
FullAdder FA3(A[2], B[2], C2, Sum[2], C3);
// 4th bit
FullAdder FA4(A[3], B[3], C3, Sum[3], Cout);

endmodule

module FullAdder (
    input A,
    input B,
    input Cin,
    output Sum,
    output Cout
);

assign Sum = A ^ B ^ Cin;
assign Cout = (A & B) | (A & Cin) | (B & Cin);
endmodule
```

b)
```
module FullSubtractor4bit (
    input [3:0] A,
    input [3:0] B,
    input Bin,
    output [3:0] Diff,
    output Bout
);

wire B1, B2, B3;

// 1st bit
FullSubtractor FS1(A[0], B[0], Bin, Diff[0], B1);
// 2nd bit
FullSubtractor FS2(A[1], B[1], B1, Diff[1], B2);
// 3rd bit
```

```
FullSubtractor FS3(A[2], B[2], B2, Diff[2], B3);
// 4th bit
FullSubtractor FS4(A[3], B[3], B3, Diff[3], Bout);

endmodule

module FullSubtractor (
    input A,
    input B,
    input Bin,
    output Diff,
    output Bout
);

assign Diff = A ^ B ^ Bin;
assign Bout = (~A & B & Bin) | (A & ~B & Bin);
endmodule
```

★ **PROGRAM 3**: Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural models.

a. 2-input AND Gate, Structural Verilog:

```
module and_gate (
  input a,
  input b,
  output y
);

  assign y = a & b;

endmodule
```

Dataflow Verilog:

```
module and_gate (
  input a,
  input b,
  output y
);

  and (y, a, b);

endmodule
```

Behavioural Verilog:

```
module and_gate (
  input a,
  input b,
  output y
);

  assign y = (a == 1 && b == 1);

endmodule
```

**b. 2-input XOR Gate, Structural Verilog:**

```
module xor_gate (
  input a,
  input b,
  output y
);

  assign y = (a ^ b);

endmodule
```

Dataflow Verilog:

```
module xor_gate (
  input a,
  input b,
  output y
);

  xor (y, a, b);

endmodule
```

Behavioural Verilog:

```
module xor_gate (
  input a,
  input b,
  output y
);

  assign y = (a != b);

endmodule
```

**c. 4-input Multiplexer, Structural Verilog:**

```
module mux4to1 (
  input sel,
  input d0,
  input d1,
  input d2,
  input d3,
  output y
);
```

```
  assign y = (sel == 0) ? d0 : ((sel == 1) ? d1 : ((sel == 2) ? d2 : d3));

endmodule
```

Dataflow Verilog:

```
module mux4to1 (
  input sel,
  input d0,
  input d1,
  input d2,
  input d3,
  output y
);

  assign y = sel ? ((sel == 1) ? d1 : ((sel == 2) ? d2 : d3)) : d0;

endmodule
```

Behavioural Verilog:

```
module mux4to1 (
  input sel,
  input d0,
  input d1,
  input d2,
  input d3,
  output y
);

  case (sel)
    0: assign y = d0;
    1: assign y = d1;
    2: assign y = d2;
    default: assign y = d3;
  endcase

endmodule
```

★ **PROGRAM 4**: Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.

```
module half_adder(
    input A,
    input B,
    output sum,
    output carry
);

assign sum = A ^ B;
assign carry = A & B;

endmodule

module full_adder(
    input A,
    input B,
    input Cin,
    output sum,
    output Cout
);

wire sum1, carry1, carry2;

half_adder HA1(A, B, sum1, carry1);
half_adder HA2(sum1, Cin, sum, carry2);
assign Cout = carry1 | carry2;

endmodule
```

```
module half_subtractor(
    input A,
    input B,
    output difference,
    output borrow
);

assign difference = A ^ B;
assign borrow = (~A & B);

endmodule

module full_subtractor(
    input A,
    input B,
    input Bin,
    output difference,
    output Bout
);

wire diff1, bor1, bor2;

half_subtractor HS1(A, B, diff1, bor1);
half_subtractor HS2(diff1, Bin, difference, bor2);
assign Bout = bor1 | bor2;

endmodule

module binary_adder_subtractor(
    input [3:0] A,
    input [3:0] B,
    input SUB,
    output [3:0] result,
    output overflow
);
```

```
wire [3:0] sum;
wire [3:0] difference;
wire carry_in, borrow_in, carry_out, borrow_out;

// Select adder or subtractor based on SUB signal
assign carry_in = ~SUB;
assign borrow_in = SUB;

// Instantiate full adders
full_adder FA0(A[0], B[0], carry_in, sum[0], carry_out);
full_adder FA1(A[1], B[1], carry_out, sum[1], carry_out);
full_adder FA2(A[2], B[2], carry_out, sum[2], carry_out);
full_adder FA3(A[3], B[3], carry_out, sum[3], carry_out);

// Instantiate full subtractors
full_subtractor FS0(A[0], B[0], borrow_in, difference[0], borrow_out);
full_subtractor FS1(A[1], B[1], borrow_out, difference[1], borrow_out);
full_subtractor FS2(A[2], B[2], borrow_out, difference[2], borrow_out);
full_subtractor FS3(A[3], B[3], borrow_out, difference[3], borrow_out);

// Assign result based on SUB signal
assign result = (SUB) ? difference : sum;

// Overflow detection
assign overflow = carry_out ^ borrow_out;

endmodule
```

★ **PROGRAM 5**: Design Verilog HDL to implement Decimal adder.

```
module DecimalAdder(
   input [3:0] A, // 4-bit input representing a decimal digit
   input [3:0] B,
   output reg [3:0] Sum
);

// Define constants for decimal to binary conversion
parameter [3:0] DECIMAL_TO_BINARY[9:0] = {4'b0000, 4'b0001, 4'b0010, 4'b0011,
4'b0100, 4'b0101, 4'b0110, 4'b0111, 4'b1000, 4'b1001};

// Define constants for binary to decimal conversion
parameter [3:0] BINARY_TO_DECIMAL[15:0] = {4'b0000, 4'b0001, 4'b0010,
4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111, 4'b1000, 4'b1001, 4'b0000, 4'b0000,
4'b0000, 4'b0000, 4'b0000, 4'b0000};

// Convert A and B to binary
reg [3:0] A_binary;
reg [3:0] B_binary;
always @(*) begin
   A_binary = DECIMAL_TO_BINARY[A];
   B_binary = DECIMAL_TO_BINARY[B];
end

// Perform binary addition
reg [4:0] binary_sum;
always @(*) begin
   binary_sum = A_binary + B_binary;
end

// Convert binary sum back to decimal
always @(*) begin
   Sum = BINARY_TO_DECIMAL[binary_sum];
end

endmodule
```

★ **PROGRAM 6**: Design a Verilog program to implement Different types of multiplexers like 2:1, 4:1 and 8:1.

**a. 2:1 Multiplexer:**

```
module mux2to1(
  input a,
  input b,
  input sel,
  output y
);
  assign y = (sel) ? b : a;

endmodule
```

**b. 4:1 Multiplexer:**

```
module mux4to1(
  input d0,
  input d1,
  input d2,
  input d3,
  input sel1,
  input sel0,
  output y
);
  assign y = (sel1 == 1'b1) ? ((sel0 == 1'b1) ? d3 : d1) : ((sel0 == 1'b1) ? d2 : d0);

endmodule
```

**c. 8:1 Multiplexer:**

```
module mux8to1(
  input d0,
  input d1,
  input d2,
  input d3,
  input d4,
  input d5,
  input d6,
  input d7,
  input sel2,
  input sel1,
  input sel0,
  output y
);
  assign y = (sel2 == 1'b1) ? ((sel1 == 1'b1) ? ((sel0 == 1'b1) ? d7 : d5) : ((sel0 == 1'b1)
? d6 : d4)) : ((sel1 == 1'b1) ? ((sel0 == 1'b1) ? d3 : d1) : ((sel0 == 1'b1) ? d2 : d0));

endmodule
```

★ **PROGRAM 7**: Design a Verilog program to implement types of De-Multiplexer.

**a. 1:2 De-Multiplexer:**

```
module demux1to2(
  input d,
  input sel,
  output y0,
  output y1
);

  assign y0 = ~sel & d;
  assign y1 = sel & d;

endmodule
```

**b. 2:4 De-Multiplexer:**

```
module demux2to4(
  input d0,
  input d1,
  output y0,
  output y1,
  output y2,
  output y3
);

  assign y0 = ~d1 & ~d0;
  assign y1 = ~d1 & d0;
  assign y2 = d1 & ~d0;
  assign y3 = d1 & d0;

endmodule
```

**<u>c. 3:8 De-Multiplexer:</u>**

```
module demux3to8(
  input d0,
  input d1,
  input d2,
  output y0,
  output y1,
  output y2,
  output y3,
  output y4,
  output y5,
  output y6,
  output y7
);

  assign y0 = ~d2 & ~d1 & ~d0;
  assign y1 = ~d2 & ~d1 & d0;
  assign y2 = ~d2 & d1 & ~d0;
  assign y3 = ~d2 & d1 & d0;
  assign y4 = d2 & ~d1 & ~d0;
  assign y5 = d2 & ~d1 & d0;
  assign y6 = d2 & d1 & ~d0;
  assign y7 = d2 & d1 & d0;

endmodule
```

★ **PROGRAM 8**: Design a Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

**1. SR Flip-Flop:**

```
module sr_ff(
  input clk,
  input S,
  input R,
  output reg Q
);

  always @(posedge clk) begin
    if (S && ~R) begin
      Q <= 1'b1;
    end else if (~S && R) begin
      Q <= 1'b0;
    end
  end

endmodule
```

**2. JK Flip-Flop:**

```
module jk_ff(
  input clk,
  input J,
  input K,
  output reg Q
);
```

```
  always @(posedge clk) begin
    if (J && ~K) begin
      Q <= 1;
    end else if (~J && K) begin
      Q <= 0;
    end
  end

endmodule
```

**3. D Flip-Flop:**

```
module d_ff(
  input clk,
  input D,
  output reg Q
);

  always @(posedge clk) begin
    Q <= D;
  end

endmodule
```

***NOTE:***

- *Practice all the experiments giving a variety of inputs to get familiar with the IDE.*
- *Make the habit of reading the errors and debugging them. Scroll through the output window to get details of the errors.*
- *Understand combinational and sequential logic properly to gain insights into the problem statement.*