Lab 3
Multiplexers, Encoders and Behaviour Modelling

Module ID : CX-203

Digital System Design

Instructor : Dr. Abid Rafique

Version 1.2

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Dec, 2023	Qamar Moavia
1.1	Modified with new exercises	03 September, 2024	Qamar Moavia

Table of Contents

Objectives

By the end of this lab, students will :

- Describe combinational circuits using behavioral modeling in SystemVerilog.
- Understand Multiplexers and Encoders.

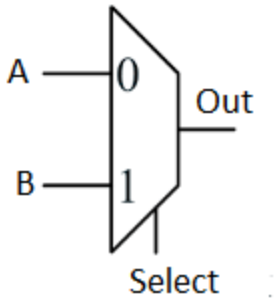
Deliverables

- Circuit Diagram for each task.
- Source code (code or screenshots)
- Simulation results (waveforms) screenshots
- Link to git repository
- FPGA Implementation results (multiple images for each task)

1. Multiplexer

A multiplexer is a combinational circuit that selects one input from multiple inputs and forwards it to a single output line. This selection is controlled by a set of selection lines. MUX is widely used in digital systems to manage data routing, signal selection, and more.

Multiplexers can be described in various ways. In this section we will describe a 2x1 multiplexer in data flow modeling and then we will describe it using two methods in behavior coding.



To design a 2-to-1 line multiplexer as shown above we use the dataflow method. We use a conditional (? :) operator. The conditional operator in Verilog HDL takes three operands:

Condition ? true-expression: false-expression;

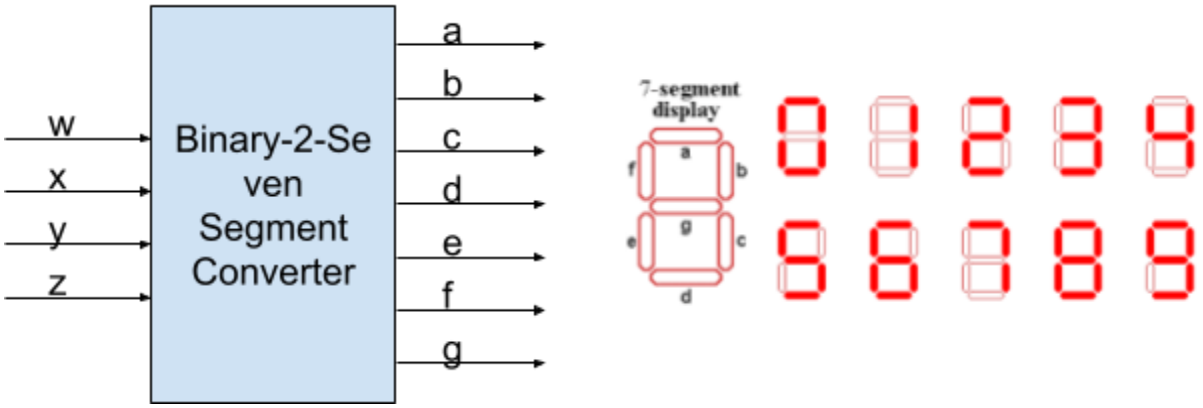
This operator is equivalent to an if-else condition. HDL given below shows the description of a 2-to-1 line multiplexer using a conditional operator.

```
// Dataflow description of 2-to-1 line multiplexer
module mux2x1_df (input A,B,select,output OUT);
   assign OUT=select ? B : A;
endmodule
```

In case select is high Out will be assigned value of B otherwise Out will be assigned value of A.

TASK 1 : Binary-to-Seven Segment Decoder Using data-flow Modeling

In this task, you will design, simulate, and implement a Binary-to-Seven Segment Decoder using data-flow modeling. Modify the `sev_seg_decoder.sv` file within the provided task1 directory. This directory contains several files, including `counter_n_bit.sv`, `pin-assignment.xdc`, `sev_seg_decoder.sv`, `decoder.sv`, `sev_seg_controller.sv`, and `sev_seg.sv`, but you will only work within the `sev_seg_decoder.sv` file for this task. The objective is to create a SystemVerilog code to control a seven-segment display based on binary inputs.



To complete the task, follow these steps:

1. Begin by creating a truth table that maps binary inputs to the segments required to display each digit in **hexadecimal form (0-F)**. Remember that for each segment (a - g) on the display to light up, it must be driven to zero. For example, to display the number 8, all segments (a - g) should be driven to zero. To turn off the display completely, all segments should be set to one.

2. Using the truth table, write SystemVerilog code in `sev_seg_decoder.sv` to implement the decoder logic using data-flow modeling.
3. Simulate your code to verify that each binary input correctly lights up the corresponding segments on the seven-segment display.
4. Test the code to ensure all digits from **0** to **F** display accurately on the seven-segment, with segments lighting up according to your truth table definitions.

Complete these steps in `sev_seg_decoder.sv`, and confirm that the decoder functions correctly in simulation and implementation.

TASK 2 : Arithmetic and Logic Unit (ALU)

Design, Simulate and Implement Arithmetic and Logic Unit that performs the following operations.

1. Addition (alu_out = a + b)
2. Subtraction (alu_out = a - b)
3. Shift left (alu_out = a << b)
4. AND operation (alu_out = a & b)

You can't use more than two 2x1 multiplexers. Take **2-bit alu_op** from the switches along with the **4 bit value of a and b** and display the **4bit result, along with a and b** on the seven seg display.
Note: You can reuse the **sev_seg_controller** that we provided in the task1.
This **sev_seg_controller** takes eight 4-bit numbers,

```
module sev_seg_controller #(
   parameter display_speed = 20
)(
   input logic clk,
   input logic resetn,
   input logic [3:0] digits [0:7],      // eight 4-bit binary inputs
   output logic  [6:0] Seg,          // 7-bit output for segments a-g
```

```
    output logic  [7:0] AN
);
```

Follow these steps:

1. Implement your 2-bit ALU logic using no more than two 2x1 multiplexers. Use the alu_op value to determine the operation performed on a and b, which could include basic operations such as addition, subtraction, or bitwise operations.
2. Once the ALU logic is defined, create the digits array input for sev_seg_controller. Map a, b, and the 4-bit ALU result to different elements of this array, ensuring that all values are displayed on the seven-segment display.
3. Verify that sev_seg_controller displays the 4-bit values of a, b, and the ALU result correctly, updating according to the alu_op selected by the switches.

2. Behavioral Modeling

Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can be used to describe combinational circuits as well. Here the behavioral modeling concept will be presented for combinational circuits.

Behavioral description uses the keyword always followed by a list of procedural assignment statements. The target output of the procedural assignment statement must be of the reg data type.

The behavioral description of 2-to-1 line multiplexer using if else statements in HDL is given below.

```
// Behavioral description of 2-to-1 line multiplexer
module mux2x1_bh (input A,B,select,output OUT);
reg OUT;
always @(*)
begin
        if (select==1) OUT=A;
        else OUT=B;
end
endmodule
```

In Verilog, always @(*) is a sensitivity list used in an always block to create a combinational logic block. This sensitivity list indicates that the block should execute whenever any of its input signals change. The (*) symbol is a wildcard that represents all possible input signals within the block.

The procedural assignment statements inside the always block are executed every time there is a change in any of the variables listed after the @ symbol. In this case, they are the input variables A, B, and select. The condition statement if-else provides a decision based upon the value of the select input.

Alternatively we can also describe the behavior of a circuit using case statements as well. The behavioral description of 2-to-1 line multiplexer using if else statements in HDL is given below.

```
// Behavioral description of 2-to-1 line multiplexer using case statement
module mux2x1_bh (input A,B,select,output OUT);
reg OUT;
always @(*)
begin
        case(select)
                0: OUT=A;
                1: OUT=B;
        endcase
end
endmodule
```

The always statement has a sequential block enclosed between the keywords case and endcase. The block is executed whenever any of the inputs listed after the @ symbol changes in value. The case statement is a multi way conditional branch condition. The case expression (select) is evaluated and compared with the values in the list of statements that follow. The first value that matches the true condition is executed.

TASK 3: Incrementer Circuit using ROM

Create an incrementer circuit that takes a **4-bit input** value and outputs that value incremented by 1. The output is also 4-bit. Design this circuit using a ROM:

- If the input is 0, the output should be 1.
- For an input of 1, the output should be 2, and so on.

Ensure that the ROM provides the correct incremented output for each possible 4-bit input.

TASK 4: Rotating word on four displays

Design a module that takes a 2-bit input from switches (`[1:0]`) and displays a specific pattern of characters on a seven-segment display, as follows:

SW[1:0]	Display Pattern
00	C0dE
01	E0Cd
10	dEC0
11	0dCE

Use the switch input to select and display the correct character pattern, ensuring that each unique input shows the corresponding pattern on the display.

TASK 5 : 4-bit Priority Encoder Design

Design a circuit that checks 4-bit windows from 8 switches and outputs the index of the first match with the value in sw[11:8]. The switches are divided into 4-bit groups: sw[11:8] and sw[7:0]. The circuit slides a 4-bit window across sw[7:0]: sw[3:0], sw[4:1], sw[5:2], sw[6:3], sw[7:4]. For each window, the circuit compares the 4-bit value to the value in sw[11:8]. If a match is found, it outputs the corresponding 3-bit index of that window (e.g., 3'b000 for sw[3:0]). If no match is found, it outputs 3'b111.

For example:
If sw[11:8] = 1010 and sw[7:0] = 10101010, the first match of 1010 occurs at sw[3:0], so the output will be 3'b000.
If sw[11:8] = 1010 and sw[7:0] = 10101000, the first match occurs at sw[5:2], so the output will be 3'b010.
If none of the windows match sw[11:8], the output will be 3'b111 (-1).

The task involves creating the block diagram, writing the SystemVerilog code, and providing simulation results to verify the correct output. The circuit will detect the first occurrence of the 4-bit chunk that matches the value in sw[11:8] and output the base index of the first match. If no match is found, the output will be 3'b111.

3. Double Dabble Method

The Double Dabble method is an algorithm used to convert binary numbers into their Binary Coded Decimal (BCD) equivalent. The algorithm works as follows:

Suppose the original number to be converted is placed into a buffer that is n bits wide. A scratch space is reserved to hold both the original number and its BCD representation. The size of this space is `n + 4×ceil(n/3)` bits, as each decimal digit requires 4 bits in binary.

For example, if the original number is 243, its binary representation is `11110011`. The scratch space is initialized to all zeros, and the original number is copied into the "original register" space on the right.

The algorithm then iterates over each bit in the binary number. On each iteration, any BCD digit that is at least 5 (binary `0101`) is incremented by 3 (binary `0011`), and the entire scratch space is shifted left by one bit. This increment ensures that values of 5 or greater correctly carry over when shifted, as 5 plus 3 equals 8, and the left shift causes a carry into the next BCD digit.

Here's how the algorithm works step by step on the number 243 (binary `11110011`):

First start with the number 11110011 on the right along with three 4 bit digital initialized to zero on the left the do the following.

0000 0000 0000	11110011	Initialization
0000 0000 0001	11100110	Shift
0000 0000 0011	11001100	Shift
0000 0000 0111	10011000	Shift
0000 0000 1010	10011000	Add 3 to ONES, since it was 7

0000 0001 0101	00110000	Shift
0000 0001 1000	00110000	Add 3 to ONES, since it was 5
0000 0011 0000	01100000	Shift
0000 0110 0000	11000000	Shift
0000 1001 0000	11000000	Add 3 to TENS, since it was 6
0001 0010 0001	10000000	Shift
0010 0100 0011	00000000	Shift

2 4 3 <- Final Result

Once eight shifts have been performed, the algorithm terminates. The BCD digits to the left of the "original register" space display the BCD encoding of the original value 243.

TASK 6 : Binary to Decimal(BCD) Converter

Design and Simulate a circuit that converts an 8-bit binary number into its equivalent BCD number using the double dabble algorithm.

Implement your design on the FPGA board by applying switches and observing outputs on the seven segment display.

Good Luck 🙂