

Lab 2: Decoders and Multiplexers

Introduction

Decoders and multiplexers are important combinational circuits in many logic designs. Decoders convert n inputs to a maximum of unique 2^n outputs. A special case is the binary coded decimal (BCD)-to-seven-segment decoder, where a four-bit decimal digit (represented in BCD) is decoded into the corresponding seven-segment code used as an input to the seven-segment display (Figure 1). In this lab an understanding of both multiplexers and how to wire a DIP switch will be fostered. These skills will be used in tandem to create a multiplexer circuit and controlled by external input stimuli.

A simple computer has several main blocks, e.g.:

- Arithmetic Logic Unit (ALU): performs arithmetic operations on numbers.
- Memory: where the program is stored.
- Multiplexers: select which piece of information to be passed on.
- Decoders: to determine, based on the input, whether to read from memory or input/output lines.
- Computer Control Unit: outputs the control signals that direct the operation of the rest of the computer.

Even though we are not building a computer, this information give you some perspective on the different components that you will be building and what they may be used for.

In this lab we will focus on the multiplexer that chooses either a reset address (`Rst_Addr`), program counter (PC), memory address register (MAR), or index register X (IRX). These signals are used to determine the information required to enter the arithmetic logic unit (ALU) component of the computer.

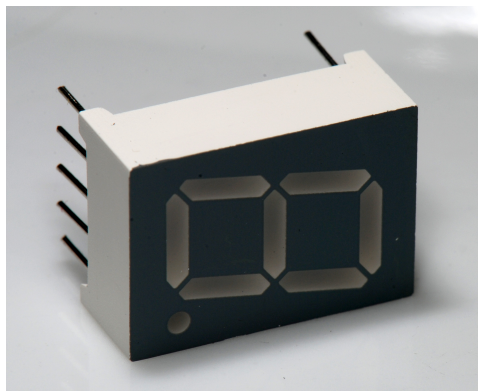


Figure 1: 7-Segment Display; Photo Credit: Peter Halasz

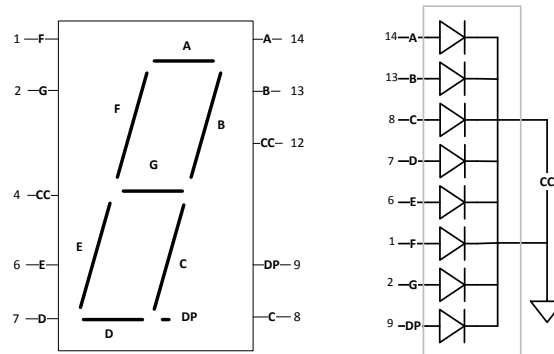


Figure 2: MAN 74 7-Segment Display

1 Prelab

- 1.1. Connect the two 7-segment displays along with the pin header needed on a perf board.
- 1.2. Fill in the truth table for the BCD-to-7-segment decoder shown in Table 1, e.g., if the input is 0011, LEDs a, b, c, d, and g should be on while LEDs f and e will be off (“1” is LED on, and “0” is LED off). See Figure 2 for reference.

For inputs 0xA through 0xF, naturally they don’t correspond to any number in the decimal range, therefore output the corresponding hex value instead, i.e., for 0xA the display should show the letter A.

Table 1: Truth Table for Hexadecimal to 7-Segment Decoder

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

- 1.3. Design a multiplexer with `Addr_Sel` as the select signal, `Rst_Addr` (we will use address `0xFF`), `PC`, `MAR`, and `IRX` as 8-bit input signals.
- 1.4. Design a Verilog program to implement the multiplexer/decoder from Table 1.

2 Lab

- 2.1. Place a block of 8 DIP switches on your proto-board. Wire each pin according to the circuit show in Figure 3.
 - 2.1.1. Connect each lead on one side to VCC. You will need an external power of 3.5 V from the Protoboard. **Do NOT use more than 4 V.**
 - 2.1.2. Put a $1\text{k}\Omega$ resistor from each of the leads on the other side to ground. Also on this side, place a row of pin header to access the outputs from the switches.



Figure 3: DIP Switch Circuit

- 2.2. In order to be able to connect to the board you will need to assign pins to the proper header on your board. Familiarize yourself with the orientation and numbering convention of the board, shown in Figure 4. Consult Table 4 for pin numbering and names.
- 2.3. Now that the hardware is setup, design the binary coded decimal (BCD)-to-seven-segment decoder and then test it using different inputs from the dip switches.
- 2.4. Implement the multiplexer program that you made in the Prelab, as shown in Figure 5. To test the multiplexer we need to hard wire in Verilog `Rst_Addr` to `0xFF` `PC` to the address `0x0A`, and `MAR` to `0x10`. Connect `IRX` to the 8 DIP switches, and `Mem_Sel` to the 2 push-button switches on the board.

3 Supplement: Verilog (2)

3.1 Verilog Logic Levels

Within Verilog there exist four logic levels, listed in Table 2.

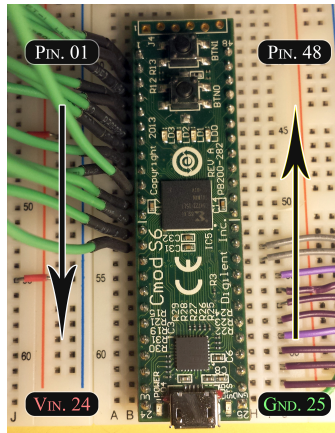


Figure 4: CMod-S6 Board Orientation and Numbering

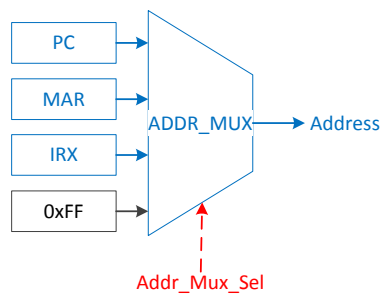


Figure 5: Simple Multiplexer (Mux)

3.2 Verilog Always and Reg Keywords

- 3.2.1. Behavioral modeling uses the keywords *always*.
- 3.2.2. Target output is a type *reg*. Unlike a wire, *reg* is updated only when a new value is assigned. In other words, it is not continuously updated as wire data types.
- 3.2.3. *[A]always* may be followed by an event control expression.
- 3.2.4. *[A]always* is followed by the symbol '@' which is followed by a list of variables. Each time there is a change in those variables, the *always* block is executed.
- 3.2.5. There is no semicolon at the end of the *always* block.

Table 2: Verilog Logic Levels

Logic	Description
0	Logic Zero; False Condition
1	Logic One; True Condition
X	Unknown Logic Value
Z	High Impedance

3.2.6. The list of variables are separated by logical operator or and not the bitwise OR operator.

3.2.7. Below is an example of an always block:

Listing 1: Example of an Always Block

```
1 always @(A or B)
2     //Do Stuff
```

3.3 Verilog if-else Statements

[I]f-else statements provide a means for conditional outputs based on the arguments of the if statement. An example is offered as Listing 2.

Listing 2: Example of if-else Statement

```
1 output out;
2 input s,A,B;
3 reg out;
4 if(s)
5     out = A; // if s is 1, then out is A
6 else
7     out = B; // else (s != 1), then out is B
```

3.4 Verilog case Statements

Case Statements provide an easy way to represent a multi-branch conditional statement.

3.4.1. The first statement that makes a match is executed.

3.4.2. Unspecified bit patterns should be treated using “default” as the keyword.

An example of a case statement is provided in Listing 3.

Listing 3: Four-to-one Line Multiplexer

```
1 module mux_4x1_example(
2     output reg out,
3     input [1:0] s, // Select Represented by 2 bits
4     input in_0, in_1, in_2, in_3);
5     always @(in_0,in_1,in_2,in_3,s)
6         case(s)
7             2'b00: out <= in_0; // if s is 00 then output is in_0
8             2'b01: out <= in_1; // if s is 01 then output is in_1
9             2'b10: out <= in_2; // ...
10            2'b11: out <= in_3;
11         endcase
12 endmodule
```

4 Supplement: CMOD-S6 Pin Names

Table 3: CMOD-S6 Feature Assignments

Special	Wire	FPGA Pin
Button 0	BTN0	P8
Button 1	BTN1	P9
1 Hz Clock	FPGA-LFC	N7
8 MHz Clock	FPGA-GCLK	N8
LED 0	LD0	N3
LED 1	LD1	P3
LED 2	LD2	N4
LED 3	LD3	P4

Table 4: CMOD-S6 DIP Assignments

DIP Pin	FPGA Pin	Wire	Wire	FPGA Pin	DIP Pin
1	P5	PIO01	PIO48	M2	48
2	N5	PIO02	PIO47	M1	47
3	N6	PIO03	PIO46	L2	46
4	P7	PIO04	PIO45	L1	45
5	P12	PIO05	PIO44	K2	44
6	N12	PIO06	PIO43	K1	43
7	L14	PIO07	PIO42	J2	42
8	L13	PIO08	PIO41	J1	41
9	K14	PIO09	PIO40	G2	40
10	K13	PIO10	PIO39	G1	39
11	J14	PIO11	PIO38	H2	38
12	J13	PIO12	PIO37	H1	37
13	H14	PIO13	PIO36	F2	36
14	H13	PIO14	PIO35	F1	35
15	F14	PIO15	PIO34	E2	34
16	F13	PIO16	PIO33	E1	33
17	G14	PIO17	PIO32	D2	32
18	G13	PIO18	PIO31	D1	31
19	E14	PIO19	PIO30	C1	30
20	E13	PIO20	PIO29	B1	29
21	D14	PIO21	PIO28	A2	28
22	D13	PIO22	PIO27	B3	27
23	C13	PIO23	PIO26	A3	26
24		VU	GND		25