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.

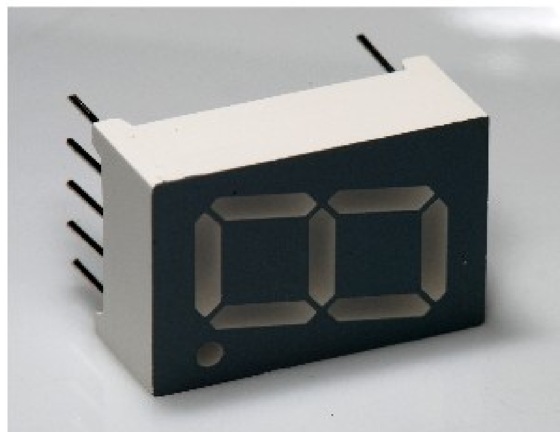

Figure 1: 7-Segment Display; Photo Credit: Peter Halasz

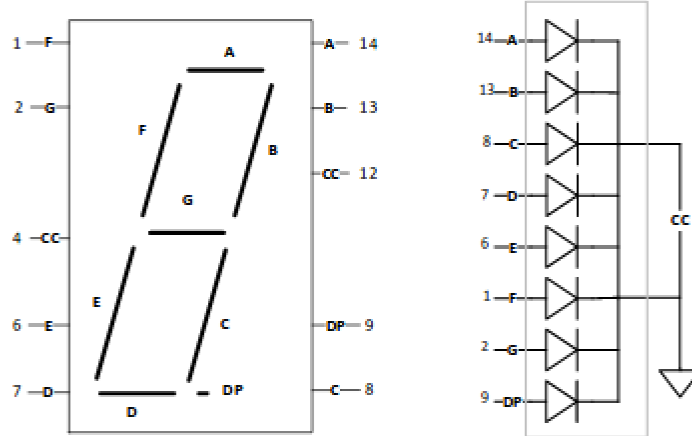

Figure 2: MAN 74 7-Segment Display

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. See Figure 2 for reference).

For inputs `0xA` through `0xF`, naturally they don't correspond to any number in the 0-9 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-seg 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 shown in Figure 3.

2.1.2. Connect each lead on one side to VCC. You will need an external power of 3.5 V from the Proto-board. **Do NOT use more than 4 V.**

2.1.3. Put a 1kΩ 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 expansion header on your board. Familiarize yourself with the orientation and numbering convention of the board, shown in Figure 4. Consult Tables 3 and 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.

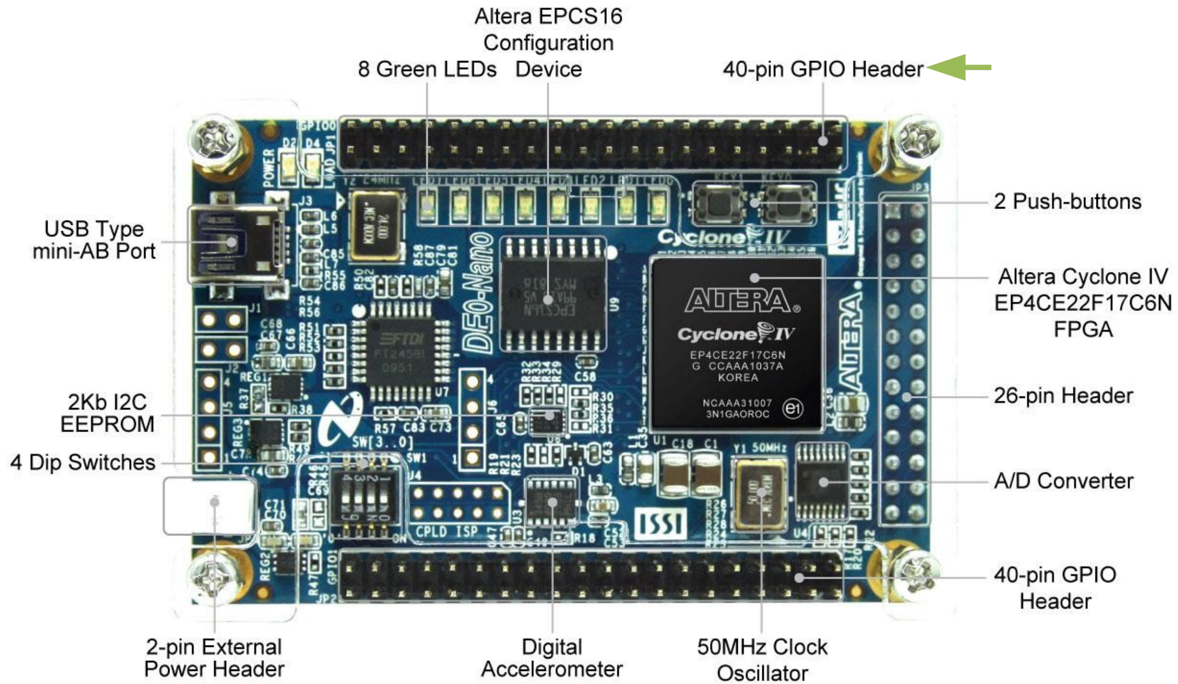

Figure 4: DE0-NANO Board Orientation and Numbering

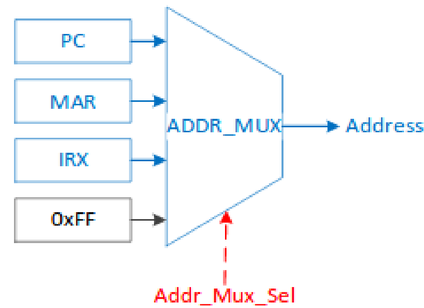

Figure 5: Simple Multiplexer (Mux)

3 Supplement: Verilog(2)

3.1. Verilog Logic Levels

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

3.2 Verilog Always and Reg Keywords

3.2.1. Behavioral modeling uses the keywords *always.*

3.2.2. Target output in a type *reg*. Unlike *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. *Always* may be followed by an event control expression.

3.2.4. *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 logicl 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

The if-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 Statements

```
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

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

4 Supplement: Pin names and Assignments

Table 3: Pin Assignments for GPIO-0

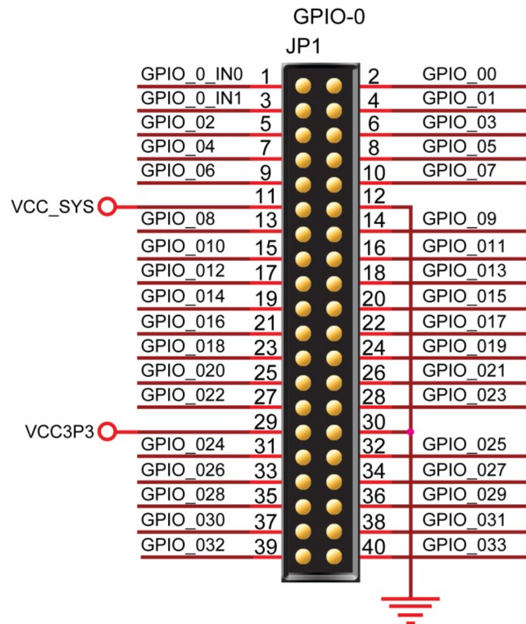

Table 4: Detail of Pin Assignments for GPIO-0

GPIO-0 Left Column	CPLD Pin #	GPIO-0 Right Column	CPLD Pin #
1	A8	2	D3
3	B8	4	C3
5	A2	6	A3
7	B3	8	B4
9	A4	10	B5
11	--	12	--
13	A5	14	D5
15	B6	16	A6
17	B7	18	D6
19	A7	20	C6
21	C8	22	E6
23	E7	24	D8
25	E8	26	F8
27	F9	28	E9
29	--	30	--
31	C9	32	D9
33	E11	34	E10
35	C11	36	B11
37	A12	38	D11
39	D12	40	B12