EE 231 Lab 3

Decoders and Multiplexers

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 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).

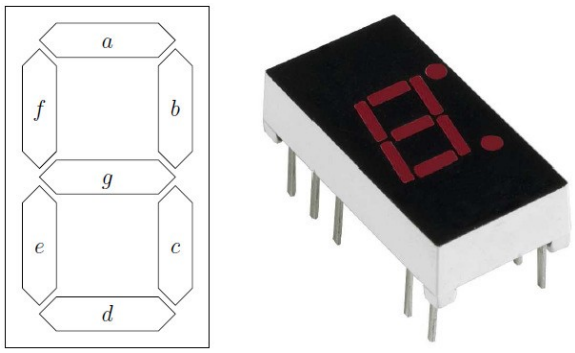

Figure 1: 7-Segment Display

1. Prelab

1.1. Wire-wrap the two 7-segment displays along with the pin header needed on a perf board. Schematics of MAN74 7-segment display.

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 1 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.

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.

1.3. Design a multiplexer with ADDR_SEL as the select signal, RST_ADDR (we will use address 0xFF), P, MAR, and X as 8-bit input signals.

1.4. Design a Verilog program to implement this multiplexer.

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							

Table 1: Truth Table for 7-Segment-Display Decoder
("1" LED is on, and "0" it is off)

2. Lab

2.1. Place a block of 8 DIP switches on your breadboard, Figure 2.

2.2. Connect each lead on one side to VCC. You can use an external power or the VCC_SYS provided on your board.

2.3. Put a 1k resistor from each of the leads on the other side to ground. Also on this side, place a row of 8 pin headers so that you have outputs from all of these switches.

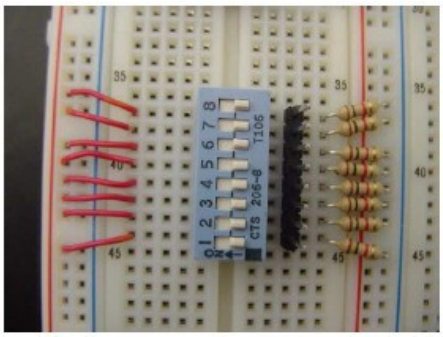

Figure 2: Dip Switches

2.4. In order to be able to connect to the board you will need to assign pins to the proper expansion header on your board. The easiest to use is the GPIO-0 at the top of the board (Figure 3).

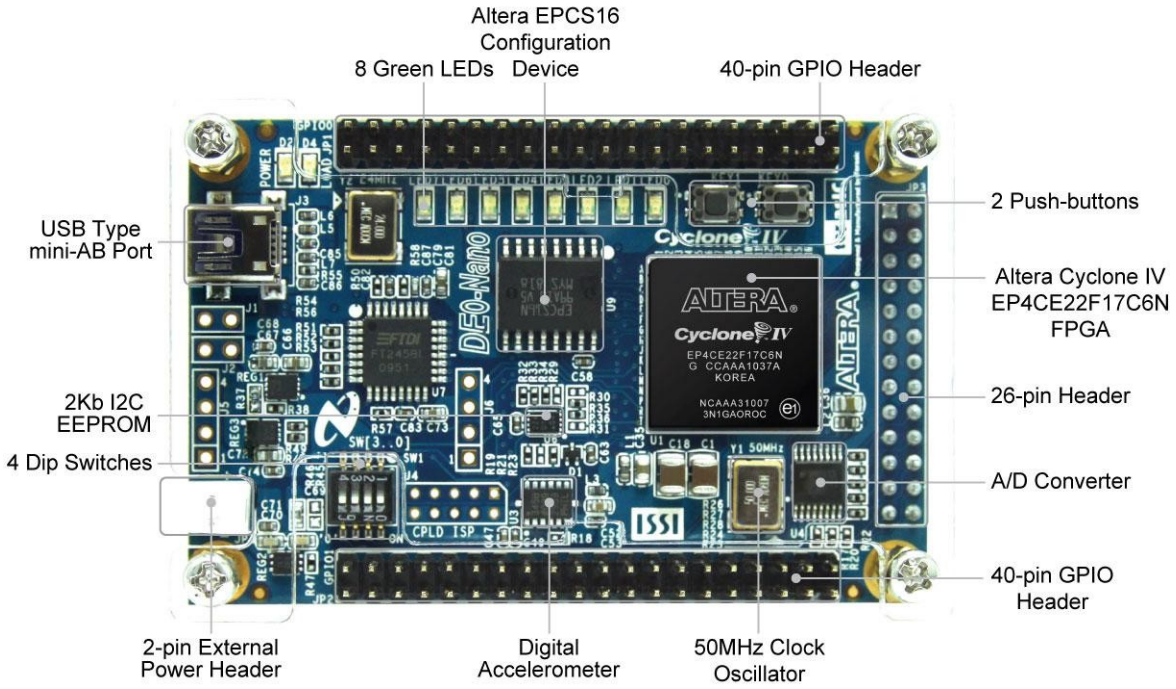

Figure 3: DE0-NANO Development Board

2.5. Use Figure 4 for the pin labels for the GPIO-0. Table 2 shows the pins assignments for the GPIO-0.

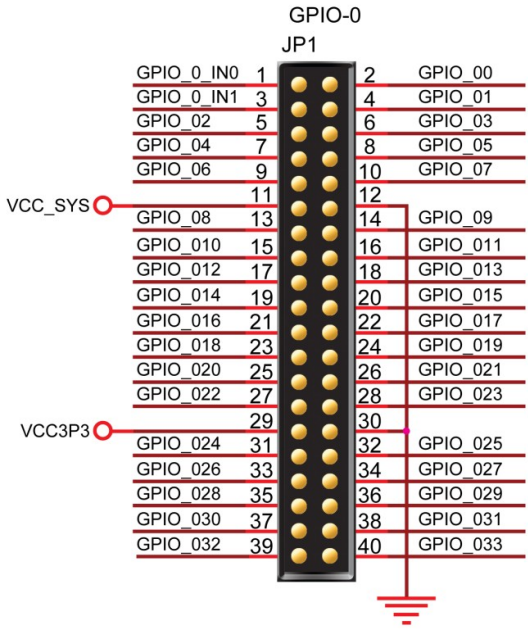

Figure 4: Pin Distribution of the GPIO-0 Expansion Header

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

Table 2: Pin Assignments for GPIO-0

2.6. Now that the hardware is setup, design the BCD-to-seven-segment decoder and test it using different inputs using the DIP switches.

2.7. Implement the multiplexer program that you developed 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 IR to the 8 DIP switches, and MEM_SEL to the 2 push-button switches on the board.

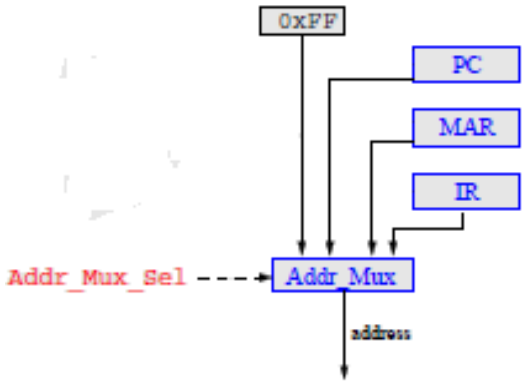

Figure 5: Implementation of a Simple Multiplexer

3. Supplementary Material

3.1. More on Verilog

3.1.1. Three-State Gates

1. bufif1(*output, input, control*): output equals the input if the control signal is 1, and high-impedance state, z, if the control signal is 0.

2. bufif0(*output, input, control*): the control signal Is the complement of bufif1.

3. notif1(*output, input, control*): same as bufif1 except the output is the complement of the input if the control signal is 1.

4. notif0(*output, input, control*): same as bufif0 except the output is the complement of the input if the control signal is 0.

3.1.2. Logic Levels

- 0 | Logic zero, false condition
- 1 | Logic one, true condition
- x | Unknown logic value

- z | High impedance

3.2. Verilog – Behavioral Modeling

3.2.1. Always and Reg

1. Behavioral modeling uses the keywords *always*.

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. *Always* may be followed by an event control expression.

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.

5. There is no semicolon at the end of the *always* block.

6. The list of variables are separated by logical operator or and not bitwise OR operator "—".

7. Below is an example of an always block:

```
always @(A or B)
…
…
…
```

3.2.2. if-else Statements

if-else statements provide a means for conditional outputs based on the arguments of the if statement.

```
…
output      out;
input       s, A, B;
reg         out;
…
…
if(s)       out = A;        // if select is 1 then out is A
else        out = B;        // else output is B
…
…
```

3.2.3. case Statements

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

1. The first statement that makes a match is executed
2. Unspecified bit patterns could be treated using default keyword.

Program 1 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
```