Lab 3: Decoders and Multiplexers

October 10, 2008

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).

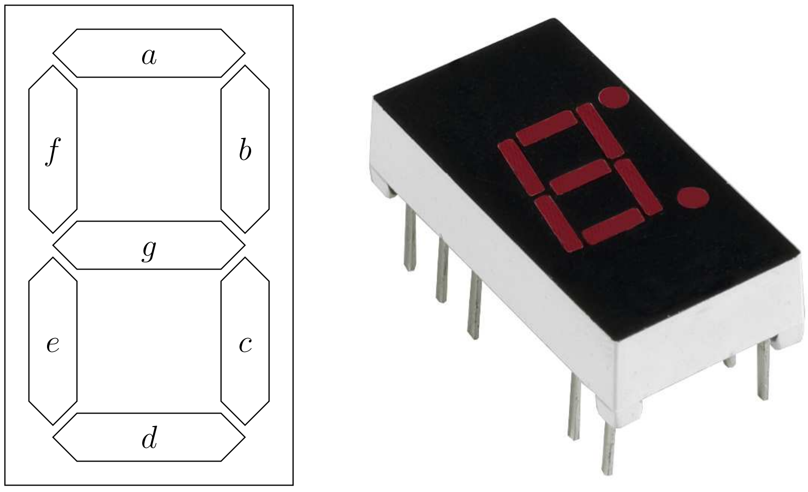

Figure 1: 7-segment display

Contents

1 Prelab

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

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 inputs `0xA` through `0xF`, naturally they don't correspond to any number in the range 0-9, therefore output the corresponding hex value instead, i.e., for `0xA` the display should show the letter `A`.

3. 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 gives you some prospective on the different components that you will be building and what they may be used for.

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

(a) Design a multiplexer with `MEM_SEL` as the select signal, `PROG_ADDR`, `PC`, `MAR` and `X` as 8-bit input signals.

(b) Design a Verilog program to implement this multiplexer.

Digit	Binary	a	b	c	d	e	f	g
0								
1								
2								
3	0011	1	1	1	1	0	0	1
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Table 1: Truth table for 7-segment-display decoder (1 means that LED is on and 0 means off).

2 Lab

1. Place a block of 8 DIP switches on your breadboard, see Figure 2.
2. Connect each lead on one side to VCC.
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.
4. In order to be able to connect to the board you will need to solder pin headers to one of the prototyping areas (see Figure 3). Use area *A*. For now solder the first two rows (Figure 4).
5. Use Figure 5 for the pin labels.
6. Now that you have all the hardware setup. Design the BCD-to-seven-segment decoder and test it using different inputs using the dip switches.
7. Implement your multiplexer program that you developed in the prelab.

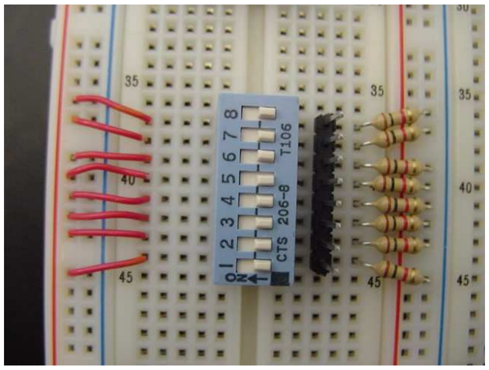

Figure 2: Dip switches

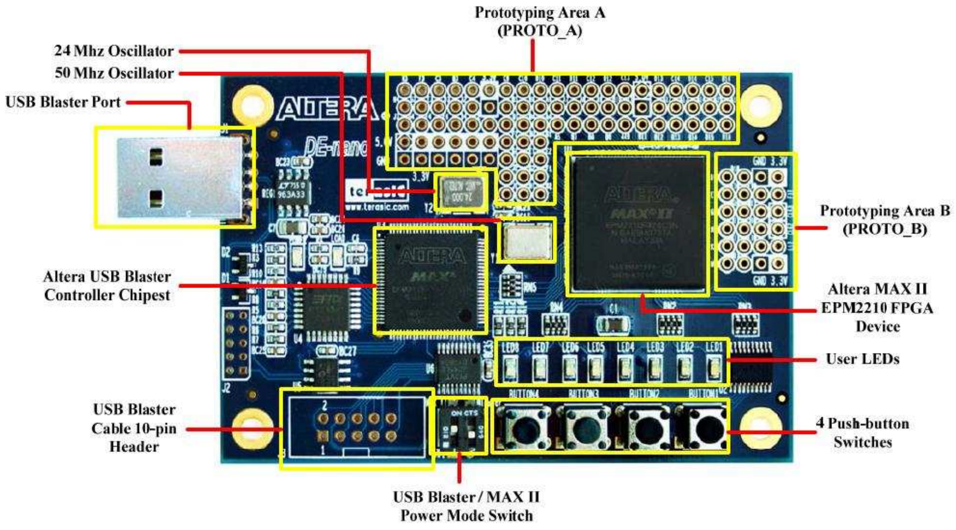

Figure 3: MAX II micro board

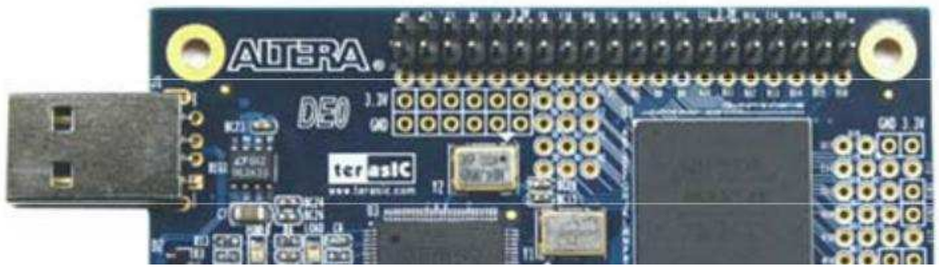

Figure 4: MAX II micro board with pin headers

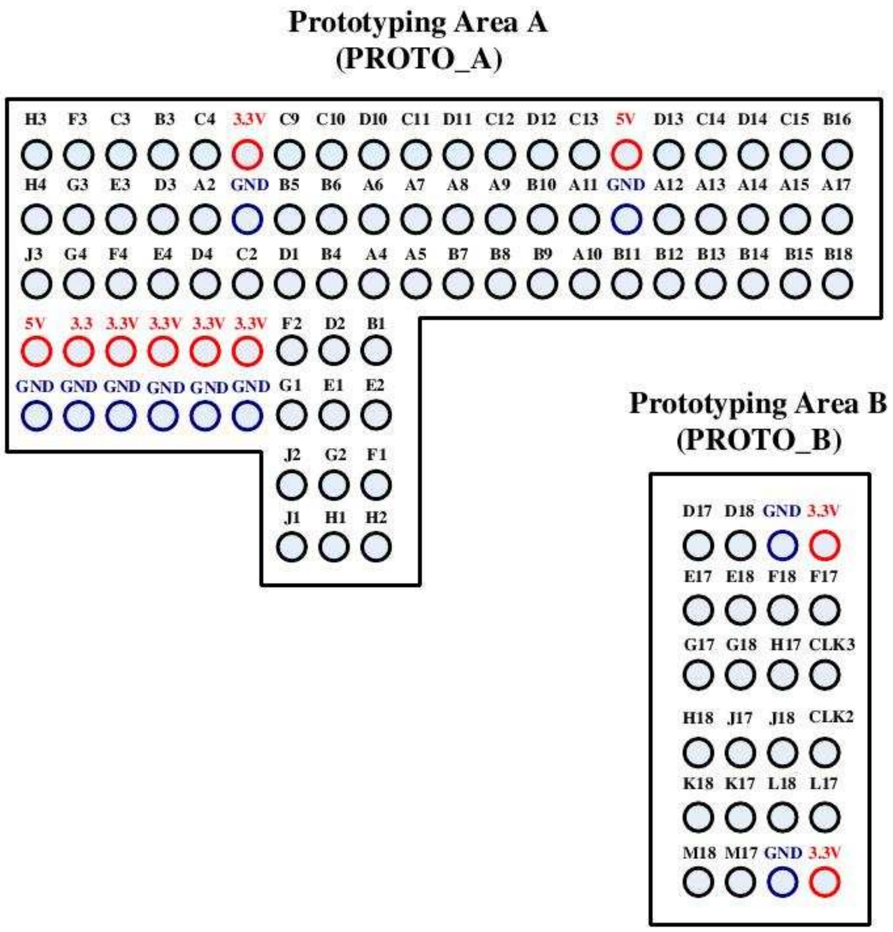
Prototyping Area A
(PROTO_A)
H3 F3 C3 B3 C4 3.3V C9 C10 D10 C11 D11 C12 D12 C13 5V D13 C14 D14 C15 B16
H4 G3 E3 D3 A2 GND B5 B6 A6 A7 A8 A9 B10 A11 GND A12 A13 A14 A15 A17
J3 G4 F4 E4 D4 C2 D1 B4 A4 A5 B7 B8 B9 A10 B11 B12 B13 B14 B15 B18
5V 3.3 3.3V 3.3V 3.3V 3.3V F2 D2 B1
GND GND GND GND GND GND G1 E1 E2
J2 G2 F1
J1 H1 H2
Prototyping Area B
(PROTO_B)
D17 D18 GND 3.3V
E17 E18 F18 F17
G17 G18 H17 CLK3
H18 J17 J18 CLK2
K18 K17 L18 L17
M18 M17 GND 3.3V

Figure 5: I/O map of prototyping areas

3 Supplementary Material

3.1 More on Verilog

3.1.1 Three-State Gates

If the control signal is 1, the output is enabled, and if the control signal is 0, the output is disabled and the gate is in high-impedance mode. There are three types of 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 use the keyword `always`.

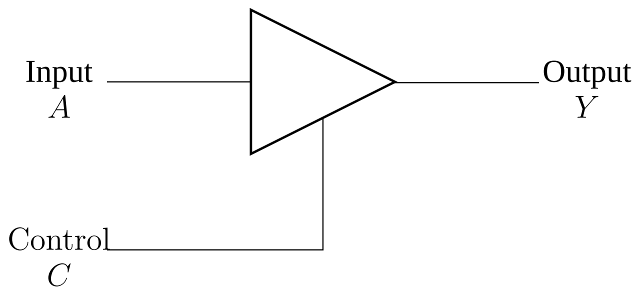

Figure 6: Three-state gate

2. Target output is of 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 @ 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 means for a conditional output based on the arguments of the `if` statement.

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.

```
        .
        .
        .

        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

        .
        .
        .
```

Program 1 four-to-one line multiplexer

```
    module mux_4x1_example(
        output reg  out,
        input [1:0]  s,        // select is 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)               //no semi-colon
            2'b00:   out = in_0; //if s is 00 then output is in_0
            2'b01:   out = in_1; //else if s is 01 then out is in_1
            2'b10:   out = in_2; //else if s is 10 then out is in_2
            2'b11:   out = in_3; //else if s is 11 then out is in_3
        endcase                 //case statement ends with endcase
    endmodule
```