EE 231 Lab 2

Decoders and Multiplexers

1.Lab

1.1. Place a block of 8 DIP switches on your proto-board, Figure 1.

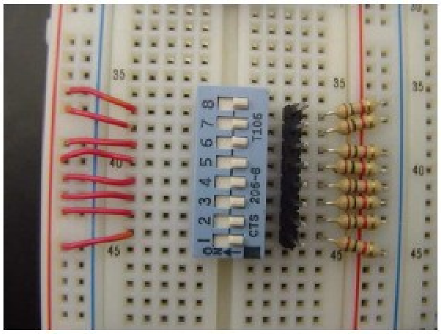

Figure 1: Dip Switches

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

1.3. 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 header so that you have the outputs from all of these switches.

1.4. 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 2.

1.5. 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.

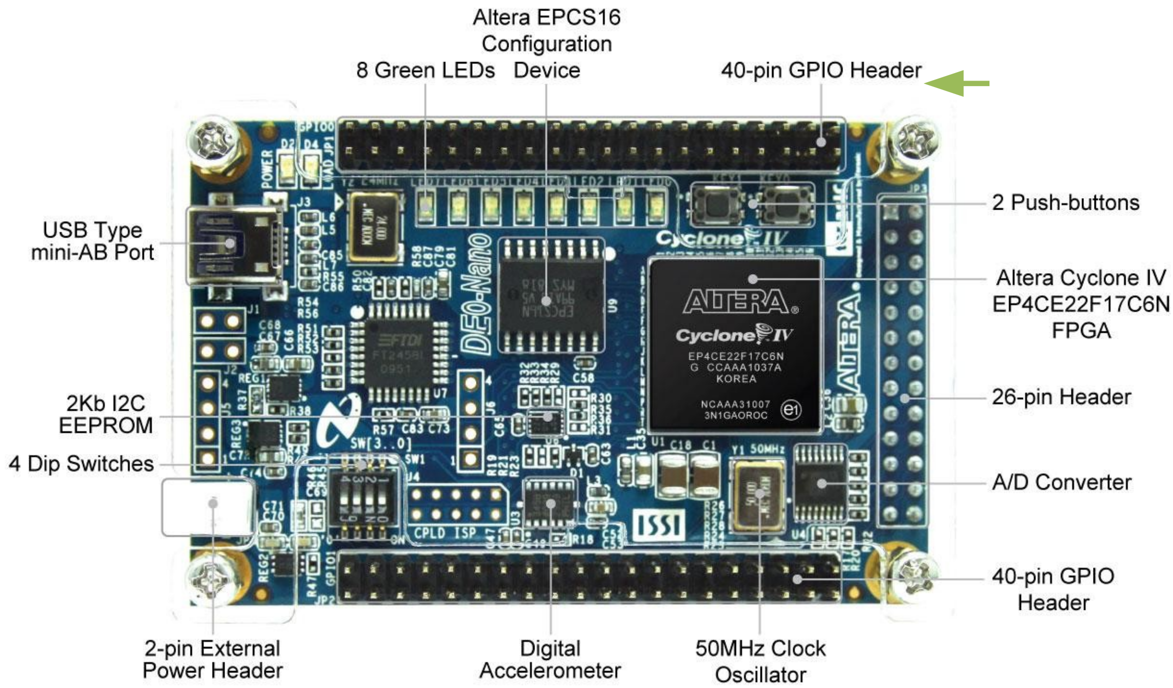

Figure 2: DE0-NANO Development Board

Use Figure 3 for the pin labels for the GPIO-0. Table 2 shows the pins assignments for the GPIO-0.

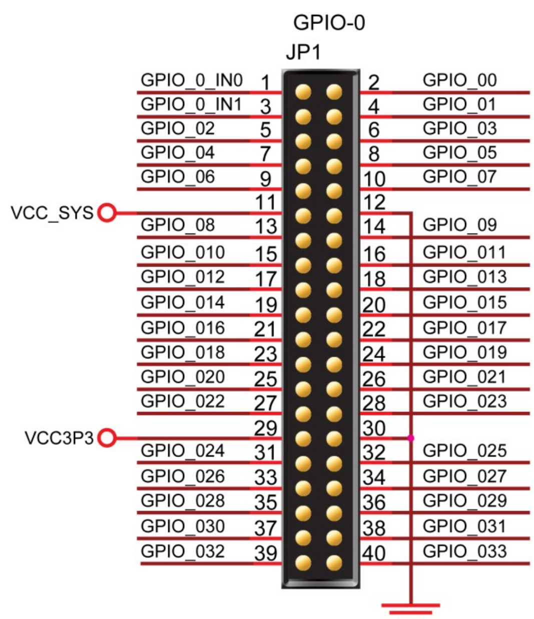

Figure 3: Pin Distribution of the GPIO-0 Expansion header

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 1: Pin Assignments for GPIO-0

1.6. Implement the multiplexer program that you made in the Prelab, as shown in Figure 4. 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.

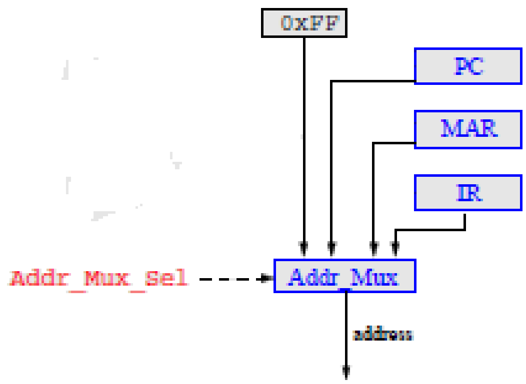

Figure 4: Implementation of a Simple Multiplexer

2.Supplementary Material

2.1.More on Verilog

2.1.1.Logic Levels

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

1.1.Verilog – Behavioral Modeling

1.1.1.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 the bitwise OR operator "—".

7.Below is an example of an always block:

```
always @(A or B)
…
…
…
```

1.1.1.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
…
…
```

1.1.2.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 should be treated using “default” as the 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
```