Tutorial: Multiplexor, Encoder, Decoder

Create a New Project

Create a Vivado project as you have done before.

4:1 1-Bit Multiplexor

This project starts with designing a **4:1 1-bit** multiplexer. A 4:1 1-bit multiplexer (Mux) is a digital circuit that takes four 1-bit inputs and a 2-bit select signal as input and produces a single 1-bit output. The select signal determines which input is selected and passed to the output. **Four** on-board slide **switches** will be used to provide the data **inputs**, **2 push buttons** will be used as **select** signals, and **LED 0** will be used to show the **output** of the multiplexer. The most common way to define a 4:1 mux in Verilog is to use a **case statement inside an always block**.

Note: we renamed led[0] to Y in our constraints (.xdc) file, so the output port name is Y.

Create a new source file named mux_4_1.v and enter the code as follows.

```
module mux_4_1 (
    input [3:0] data,
    input [1:0] sel,
    output Y
);

// we can only assign values to registers inside an always block

reg tmp;

always @(data, sel) begin
    case (sel)
    2'b00:   tmp <= data[0];
    2'b01:   tmp <= data[1];
    2'b10:   tmp <= data[2];
    2'b11:   tmp <= data[3];
    default: tmp <= 1'b0;
    endcase
end
assign Y = tmp;
endmodule
```

We defined our mux to have 4 data inputs, 2 select inputs, and one output signal. $log_2(\text{number of data inputs})$= number of select inputs (i.e, $log_2(4)$=2).

Since we are using an **always** block, we created a temporary 1-bit register called tmp.

The always block has the sensitivity list `@(data, sel)`. This means that **tmp will be updated whenever data or sel change.**

We wrap the contents of the always block with begin...end. This is analogous to wrapping functions with { } in C or Java.

The case statement checks the current value of sel, and then sets tmp to the corresponding data bit. The **default** keyword tells what should **tmp** be if sel doesn't equal 00, 01, 10, or 11. Every case statement should include a default case. (Each bit of sel can be equal to 0, 1, x, or z. In this example, the default case covers all cases where one ofthe bits of sel is an **x or z.**)

We assign Y to tmp outside the always block. This completes our mux.

In the **constraints** (.xdc) file rename **led[0] to Y**, **sw[3:0] to data[3:0],** and**btn[1:0] to sel[1:0]**. Generate a bitstream and upload it to your Boolean board. Verify the mux works as expected.

To **simulate** the mux, add a new simulation source file. Paste the following code, save the file, and then press "**Run Simulation**." Youmay need to right-click on the file in the Sources window and then select "**Set as Top**" if you have other simulation source files in the project. Become familiar with zooming and panning in the simulation window.
Understand how the simulation works and see ifyou can verify the mux is working correctly from the output waveforms alone.

```
// simulation file
`timescale 1ns / 1ps

module mux_tb;

// inputs
reg [3:0] data;
reg [1:0] sel;

// outputs
wire Y;

// connect test signals to our mux
mux_4_1 CUT (
    .data(data),
    .sel(sel),
    .Y(Y)
);

integer k;
initial begin
    sel = 2'b00;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10; // wait 10ns
    end

    sel = 2'b01;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10;
    end

    sel = 2'b10;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10;
    end

    sel = 2'b11;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10;
    end

    sel = 2'b1z;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10;
    end

    sel = 2'b1x;
    for(k=0; k < 16; k=k+1) begin
        data = k;
        #10;
    end
    $finish;
end

endmodule
```

Since the mux is such a common digital circuit element, it shouldn't be too surprising that there are **other** ways tocreate a mux. Two common implementations are shown below.

Using the`?:` Selection Operator

The first way to code a mux behaviorally is to use the `?:` selection operator. This method is most analogous to the **if statement**. You can think of this statement as follows: **assign data[0] to Y if the statement in the parenthesis is true,** else **assign whatever is after the colon to Y** (and so on). This method is most commonly used with **2-input muxes** in practice.

```
assign Y = (sel == 2'd0) ? data[0] : (
                (sel == 2'd1) ? data[1] : (
                    (sel == 2'd2) ? data[2] : data[3]
                )
            );
```

Using an `always` Block with an `if` Statement

The second way to code a mux is by using an **always** block together with an if-else statement. Note that because `tmp` is assigned in an always block, it must be declared as register type `reg` (**assign** statements cannot be used inside an always block). It is important that there is a closing else statement in Verilog, unlike C or Java.

```
reg tmp;

always @ (sel, data)
begin
    if (sel == 2'd0)
        tmp <= data[0];
    else if (sel == 2'd1)
        tmp <= data[1];
    else if (sel == 2'd2)
        tmp <= data[2];
    else
        tmp <= data[3];
end
assign Y = tmp;
```

4:1 2-Bit Multiplexor

Now let's design a 4:1 **2-bit** bus multiplexer (that is, a multiplexor with **four 2-bit bus inputs** and a 2-bit bus output). **Eight** on-board slide **switches** will be used to provide the data **inputs** (organized as four 2-bit inputs: I0, I1, I2, I3), **two push buttons will be used as select signals**, while **LED 0 and LED 1 will be used to show the 2-bit outputof the bus multiplexer.**

We will still need 2 (i.e., $log_2(4)$=2) select signals, as this lets us choose between 4 different input signals.

Overall, the following code implements a 4:1 multiplexer where one of the four 2-bit inputs (I0, I1, I2, I3) is selected based on the 2-bit select signal (sel). The selected input is then assigned to the 2-bit output signal Y.

```
module mux_4_2 (
    input [1:0] I0, I1, I2, I3, // four 2-bit input
    input [1:0] sel,
    output [1:0] Y
);

reg [1:0] tmp; //2-bit reg

always @(I0, I1, I2, I3, sel) begin
    case (sel)
    2'b00:   tmp <= I0;
    2'b01:   tmp <= I1;
    2'b10:   tmp <= I2;
    2'b11:   tmp <= I3;
    default: tmp <= 2'b00;
    endcase
end

assign Y = tmp;

endmodule
```

Create a new **simulation** file and run the code. Make sure you right-click this simulation file and select "Set as Top".

Simulate and implement the 4:1 2-bit mux on your Boolean Board. To create a bitstream, you will need to modify the constraints file from step 2.

3:8 Binary Decoder

In this step, you are going to design and implement a **3:8 binary decoder**.

A decoder is a combinational logic circuit that takes an **input** signal and **activates one of the output** lines based on the **binary value of the input.**

The example presented here uses a 3-bit bus `I[2:0]`for input signals, and an 8-bit bus `Y[7:0]` for output signals. Three individual input wires and eight individualoutput wires could have been used instead, but then the Verilog code would be less compact.

Create a Verilog module called `decoder_3_8` with inputs `I` and outputs `Y` as follows.

Perhaps the most readable way to describe the behavior of a decoder is to use a **case statement in an always** blockas shown.

```
module decoder_3_8 (
input [2:0] I,
output reg [7:0] Y
);
always @ (I)
begin
    case (I)

        3'd0:    Y <= 8'd1;
        3'd1:    Y <= 8'd2;
        3'd2:    Y <= 8'd4;
        3'd3:    Y <= 8'd8;
        3'd4:    Y <= 8'd16;
        3'd5:    Y <= 8'd32;
        3'd6:    Y <= 8'd64;
        3'd7:    Y <= 8'd128;

       default: Y <= 8'd0;
    endcase
end
endmodule
```

4-Input Priority Encoder

In this section, you are going to design a 4-input priority encoder.

A priority encoder is a digital circuit that takes multiple input signals and produces an output indicating the highest priority active input.

A 4-bit bus `I[3:0]` will be used as **data inputs**, and `Ein`, will act as the "**Enable**" signal. A **2-bit output** bus `Y[1:0]` will show the **encoded** value of the inputs, and two one-bit **outputs** `GS` and `Eout` will show the group signal and enable output, respectively.

Create another Verilog module called `encoder` with inputs `I`, `Ein`, and outputs `Eout`, `GS`, and `Y`.

The most efficient way to describe the behavior of a priority encoder is to use `if-else` statement in an always block. The priority encoder has three outputs, so three always blocks are needed to define the output signals.

In the following code, the first always block is sensitive to changes in I and Ein signals. If Ein is enabled (Ein equals 1), it checks the highest priority input by examining each bit of the I signal. The priority order is from the most significant bit to the least significant bit. **If any of the input bits is active (equal to 1), the corresponding output Y is assigned the priority value (3, 2, 1) using 2-bit representation**. If none of the inputs are active, Y is assigned 2'd0. If Ein is not enabled, Y is set to 2'd0.

The second always block checks if Ein is enabled and all the bits of I are 0. If both conditions are met, the Eout signal is set to 1, indicating an enabled output. Otherwise, Eout is set to 0.

The third always block checks if Ein is enabled and at least one of the bits in I is not 0. If both conditions are true, the GS signal is set to 1, indicating the gate status. Otherwise, GS is set to 0.

Overall, this priority encoder module evaluates the input signals and determines the highest priority input. It provides the priority value through the **2-bit output Y** and also indicates the enable and gate status through the Eout and GS outputs, respectively.

```
module priority_encoder(
        input [3:0] I,
        input Ein,
        output reg [1:0] Y,
        output reg GS,
        output reg Eout
);

always @ (I, Ein)
begin
    if(Ein == 1) begin
        if (I[3] == 1)
            Y <= 2'd3;
        else if (I[2] == 1)
            Y <= 2'd2;
        else if (I[1] == 1)
            Y <= 2'd1;
        else
            Y <= 2'd0;
    end
    else
        Y = 2'd0;
end

always @ (I, Ein)
begin
    if (Ein == 1 && I == 0)
        Eout <= 1'b1;
    else
        Eout <= 1'b0;
end

always @ (I, Ein)
begin
    if (Ein == 1 && I != 0)
        GS <= 1'b1;
    else
        GS <= 1'b0;
end
endmodule
```