Multiplexers — Two Types + Verilog

ENEE 245: Digital Circuits and Systems Laboratory
Lab 7

Objectives

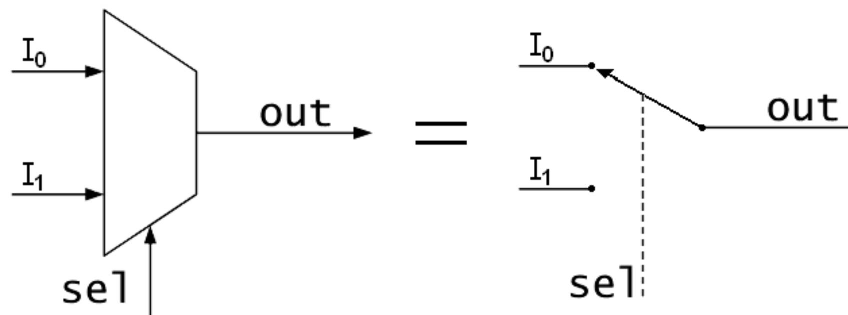
The objectives of this laboratory are the following:

- To become familiar with continuous assignments and procedural programming in Verilog
- To design various multiplexer configurations in Verilog
- To implement a 4-bit wide 4:1multiplexer in the Nexys2 FPGA prototyping board
- To implement an analog MUX on breadboard

A multiplexer is a device that selects one of several input signals and forwards the selected input to the output. Typical multiplexers come in 2:1, 4:1, 8:1, and 16:1 forms. A multiplexer of 2n inputs has n select lines. A TTL series 8:1 MUX is 74151. It has three select lines S2, S1, S0. Each of the 8 possible combinations of S2, S1, S0 selects In this lab, you will design a several MUXes using Verilog, on a Nexys2 board (from Digilent), which contains a Spartan 3E FPGA (from Xilinx).

Multiplexers

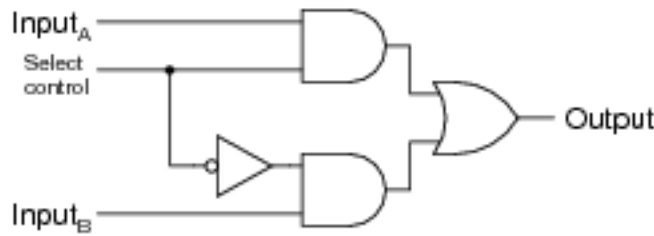
The MUX is generally shown as follows, as a high-level abstraction:



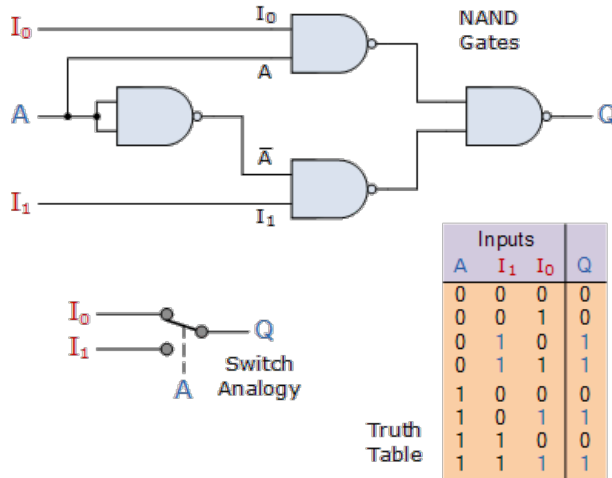
It selects between multiple input signals, but though the picture above suggests that the chosen signal is passed from input to output, that depends on the implementation of the MUX. There is a difference between passing an input signal to the output, and the value of the input signal to the output.

In general, there are two main MUX designs. The first is a digital MUX—it passes on either “1” or “0” depending on the logic values given to it; the second is an analog MUX—it passes on the desired signals themselves. Now that you have been exposed to analog signals as they appear in the digital world, this should make sense.

Digital MUXes of various flavors

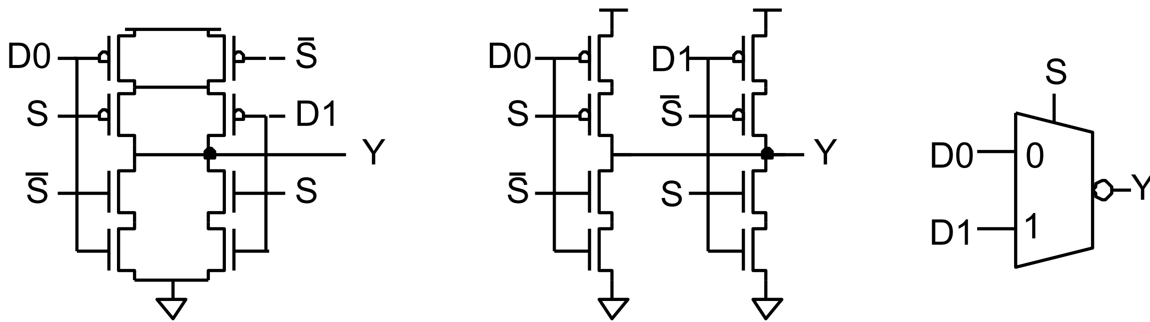


AND gates feeding into an n-way OR gate



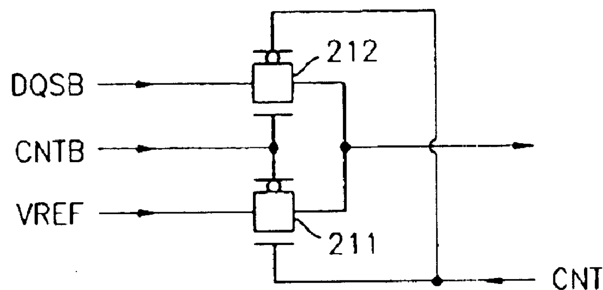
Inputs			
A	I_1	I_0	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

NAND gates feeding into an n-way NAND gate (note the left-most NAND could be a simple inverter)



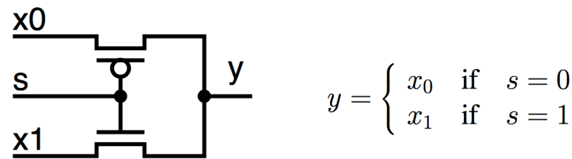
CMOS inverting MUXes (a non-inverting MUX requires an additional inverter at the output)

Analog MUX using transmission gates



CNT and CNTB are the select/select-bar signals to enable only one transmission gate

An even cheaper analog MUX using simple pass transistors



$$y = \begin{cases} x_0 & \text{if} \quad s = 0 \\ x_1 & \text{if} \quad s = 1 \end{cases}$$

These are all MUXes. The “digital” ones on the top imply a CMOS implementation, in which case each AND/NAND/OR/NOR would be at least four transistors (the non-inverting ones would require six), and each inverter would be two transistors. So the top design would require 20 transistors, and the second one would require 16 (could be reduced to 14 if an inverter were used on the select signal). The third one down, the inverting implementation, reduces the transistor count to 8, or 10 if a non-inverting MUX is desired.

So it should be extremely clear why the analog ones are the most commonly used in industrial designs: they are much smaller—the transmission-gate implementation requires only **six** transistors (which includes an inverter to get the CNTB signal), and the last implementation only requires **two**.

Note that the analog implementations are bi-directional—the selected signal can pass both ways, as the pass transistors are all agnostic about signal direction, whereas the information is strictly one-way in CMOS designs.

Verilog Implementation

The most important thing in writing behavioral Verilog is to ensure that what you want is what the software actually delivers to you. This is called “writing synthesizable code,” and when generating MUXes, the thing to keep in mind is that you ensure *every* possible value for the select signal is accounted for. If you fail to do this, the synthesis software will create odd little state machines to try to account for any values you neglected to mention.

The following are all reasonable implementations:

```
wire        in0, in1, sel, out;
assign      out = (~sel & in0) | (sel & in1);

wire        in0, in1, sel, out;
assign      out = (sel) ? in1 : in0;

wire        in0, in1, sel, out;
wire [1:0]  d = {in0, in1};
assign      out = d[sel];

wire        in0, in1, sel, out;
assign      out = (sel==1'b0) ? in0 : 1'bz,
            out = (sel==1'b1) ? in1 : 1'bz;
```

The main issue is when you go to larger numbers of inputs, like 4, 8, 16, 32 … In the case of, say, a 4-way MUX, you get the following Verilog code if you use the ternary operator:

Example of what NOT to do:

```
wire        in0, in1, in2, in3, out;
wire  [1:0] sel;
assign      out = (sel==2'b00) ? in0 :
                  (sel==2'b01) ? in1 :
                  (sel==2'b10) ? in2 :
                  in3 ;
```

What this actually produces is a priority encoder, just like it would if you use a long list of if-then-else statements. So, while it might be functionally equivalent, it is in fact a bit more expensive in both resources (die area, gates, power) and time. The following tri-state buffer implementation is a better solution (similar to the transmission-gate implementations above):

```
wire        in0, in1, in2, in3, out;
wire  [1:0] sel;
assign      out = (sel==2'b0) ? in0 : 1'bz,
            out = (sel==2'b1) ? in1 : 1'bz,
            out = (sel==2'b2) ? in2 : 1'bz,
            out = (sel==2'b3) ? in3 : 1'bz;
```

Note that this scales to wide data widths. For instance, the following is a 4-way MUX for selecting between four different 64-bit data values:

```
wire [63:0] in0, in1, in2, in3, out;
wire  [1:0] sel;
assign      out = (sel==2'd0) ? in0 : 64'bz,
            out = (sel==2'd1) ? in1 : 64'bz,
            out = (sel==2'd2) ? in2 : 64'bz,
            out = (sel==2'd3) ? in3 : 64'bz;
```

Another good option is the *case* statement, which is extremely useful in module format:

```
module (in0, in1, in2, in3, in4, in5, in6, in7, sel, out);

input  [63:0] in0, in1, in2, in3, in4, in5, in6, in7;
input  [2:0]  sel;
output [63:0] out;

always @(in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7 or sel)
      case (sel)
            3'b000 : out = in0;
            3'b001 : out = in1;
            3'b010 : out = in2;
            3'b011 : out = in3;
            3'b100 : out = in4;
            3'b101 : out = in5;
            3'b110 : out = in6;
            3'b111 : out = in7;
      endcase
endmodule
```

Here is the corresponding tri-state buffer implementation:

```
module (in0, in1, in2, in3, in4, in5, in6, in7, sel, out);

input  [63:0] in0, in1, in2, in3, in4, in5, in6, in7;
input  [2:0]  sel;
output [63:0] out;

assign      out = (sel==3'd0) ? in0 : 64'bz,
            out = (sel==3'd1) ? in1 : 64'bz,
            out = (sel==3'd2) ? in2 : 64'bz,
            out = (sel==3'd3) ? in3 : 64'bz;
            out = (sel==3'd4) ? in4 : 64'bz,
            out = (sel==3'd5) ? in5 : 64'bz,
            out = (sel==3'd6) ? in6 : 64'bz,
            out = (sel==3'd7) ? in7 : 64'bz;

endmodule
```

Note that it is very important, when using either the tri-state implementation or the *case* statement, that you have a line for every possible value of the select signal. If you, the designer, happen to know that, for instance, only 6 out of 8 possible bit patterns will be used by the hardware while running, you would be tempted to write code accounting for only those six possibilities, leaving the other two out entirely. This will cause problems with the *case* statement, because Verilog does not know that only six of the 8 states are possible, and it will try to account for the missing two values by inserting a lot of inefficient circuitry that you might find puzzling to debug. In your *case* statement, you should have a "default" for all unexpected values, for instance, as in the following code, which assumes that select values "6" and "7" will never happen:

```
always @(in0 or in1 or in2 or in3 or in4 or in5 or sel)
      case (sel)
            3'b000 : out = in0;
            3'b001 : out = in1;
            3'b010 : out = in2;
            3'b011 : out = in3;
            3'b100 : out = in4;
            3'b101 : out = in5;
            default : out = 64'bz;
      endcase
```

When using the tri-state buffer implementation, which requires a separate comparator for each possible input, it is not necessary to account for any "unused" bit patterns. However, for debugging purposes (e.g., for other engineers looking at your code), you might want to account for them anyway.

Pre-Lab Preparation

Please make sure to complete the prelab before you attend your lab section. The lab will be long and frustrating if you do not do the prelab ahead of time. In this lab, you will use Verilog to implement several multiplexer configurations on the Xilinx Spartan 3E FPGA. The Verilog used to describe the multiplexers will be in structural as well as procedural.

Part 1 — 2:1 Multiplexer

1. Create a top-level design called mux21_top that connects inputs a and b to the rightmost two slide switches of Nexys2, connects input s to btn[0] of Nexys2, and connects output y to ld[0].
2. Perform a functional simulation of your design.
3. Create a symbol for the multiplexer.

Part 2 — 4-Bit Wide 2:1 MUX Using if Statement

1. Create an N-bit wide 2:1 MUX using the parameter statement. Create a symbol for it.
2. Create a 4-bit wide 2:1 MUX by instantiating the N-bit wide MUX designed above.
3. Functional Simulation. Perform a functional simulation of the circuit to verify that it is working correctly.
4. Create Symbol. Create a symbol for the multiplexer to use in the graphical editor. This creates a symbol file that is a Graphic File and can be viewed and edited by opening it.

Part 3 — 4-Bit Wide 4:1 MUX

1. Design a 4-bit wide 4:1 multiplexer from three 4-bit wide 2:1 multiplexers.
2. Perform a functional simulation of the circuit. Paste the results in your prelab report.

3. Create a symbol for the 4-bit wide 4:1 MUX to use in the graphical editor.
4. Design a 4:1 multiplexer using the Verilog case statement.
5. Simulate the design. Paste the results in your prelab report.
6. Bring your Verilog codes in a flash drive.

Pre-Lab Report

In your prelab report, include circuit schematics, Verilog programs, and simulation results for all multiplexers discussed above.

Incorrect or incomplete designs and Verilog programs will not receive full credit. If you have any problems with Verilog syntax and other pre-lab related issues, please resolve them before coming to the lab. Your TA may not be able to help you with these issues during the lab session.

In-Lab Procedure

Bring flash drives to store your data.

Ask the TA questions regarding any procedures about which you are uncertain.

Set your 4-bit 4:1 MUX Verilog file (the hierarchical one built from three 2:1 multiplexers) as the top module of your design and implement the complete design (*synthesize*, *map*, and *Place & Route*) using the Xilinx ISE tools. Program the FPGA using the bit-stream file which is generated in the process. For checkoff, you will show the TA the following:

1. Use hard-wired values on the expansion board to drive your inputs; e.g., have eight inputs connected directly to GND and eight inputs connected directly to VDD, and use these to form your inputs that your MUX will select between. Do not hard-code values into your Verilog, or the synthesis tools will optimize the circuitry into no-ops.
2. Show the multiplexer working on the board.
3. Demonstrate at least one selection operation from the patterns you have used in the test bench. Show that your simulation results above match the observed waveforms on the DLA.

Post-Lab Report

Write up your code, schematics, and lab procedures. Demonstrate the correctness of your designs through your DLA output and note any differences between what you simulated and how the circuits behaved in the lab.

Along with your post-lab report, submit the observed waveforms (in separate sheets) for the 4-bit wide 4:1 multiplexer.