Thuc Tran

HW2 Write Up

1. Full Adder

Idea: The idea behind this is to create a 1 Bit Full Adder. The main key here was to apply the use of XOR gates in lieu of alternatives in order to have a more condensed circuit for sums.

Implementation Code:

```
module structuralFullAdder(out, carryout, a, b, carryin);
output out, carryout;
input a, b, carryin;
wire aXorb,bOrC, aAndBorC, bAndc;

//Two xors used for determining the output sum.
`XOR aXorbgate(aXorb, a, b);
`XOR aXorbXorcgate(out, aXorb, carryin);

//Used for determining the Carryout value.
`OR bOrcgate(bOrC, b, carryin);
`AND aAndbOrCgate(aAndBorC, bOrC, a);
`AND bAndcgate(bAndc, b,carryin);
`OR cout(carryout, bAndc, aAndBorC);

Endmodule
```

Do File:

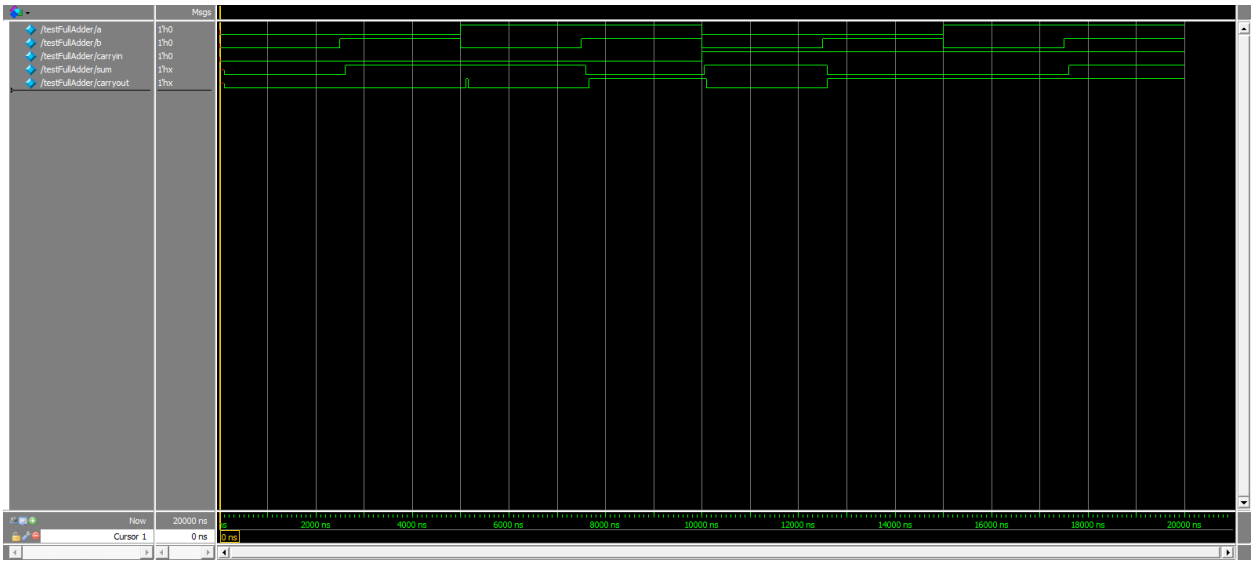
```
vlog -reportprogress 300 -work work adder.v
vsim -voptargs="+acc" testFullAdder
add wave -position insertpoint  \
sim:/testFullAdder/a \
sim:/testFullAdder/b \
sim:/testFullAdder/carryin \
sim:/testFullAdder/sum \
sim:/testFullAdder/carryout
run -all
wave zoom full
```

Bench Results:

The design was validated with an exhaustive testing of all combinations of input.

```
# Cin  a   b |  Sum  Cout | Exp Sum  Exp Cout
#  0   0   0 |   0    0   |    0        0
#  0   1   0 |   1    0   |    1        0
#  0   0   1 |   1    0   |    1        0
#  0   1   1 |   0    1   |    0        1
#  1   0   0 |   1    0   |    1        0
#  1   1   0 |   0    1   |    0        1
#  1   0   1 |   0    1   |    0        1
#  1   1   1 |   1    1   |    1        1
```

Waveform:



For a larger image, see Adder_graph.bmp in repo.

2. 3-8 Decoder w/ Enable

Idea: The idea of this is to design a device that will output a high signal to the line that the address points to upon enable. This was done via looking at the use of three input AND gates.

Implementation Code:

```
module structuralDecoder(out0,out1,out2,out3, address0,address1, enable);
output out0, out1, out2, out3;
input address0, address1;
input enable;
wire nAddress0, nAddress1;

//Creates negations of the input addresses.
`NOT a0inv(nAddress0, address0);
`NOT a1inv(nAddress1, address1);

//Associates the proper bit conditions for each output line.
`AND o0(out0, enable, nAddress0, nAddress1);
`AND o1(out1, enable, address0, nAddress1);
`AND o2(out2, enable, nAddress0, address1);
`AND o3(out3, enable, address0, address1);

endmodule
```

Do File:

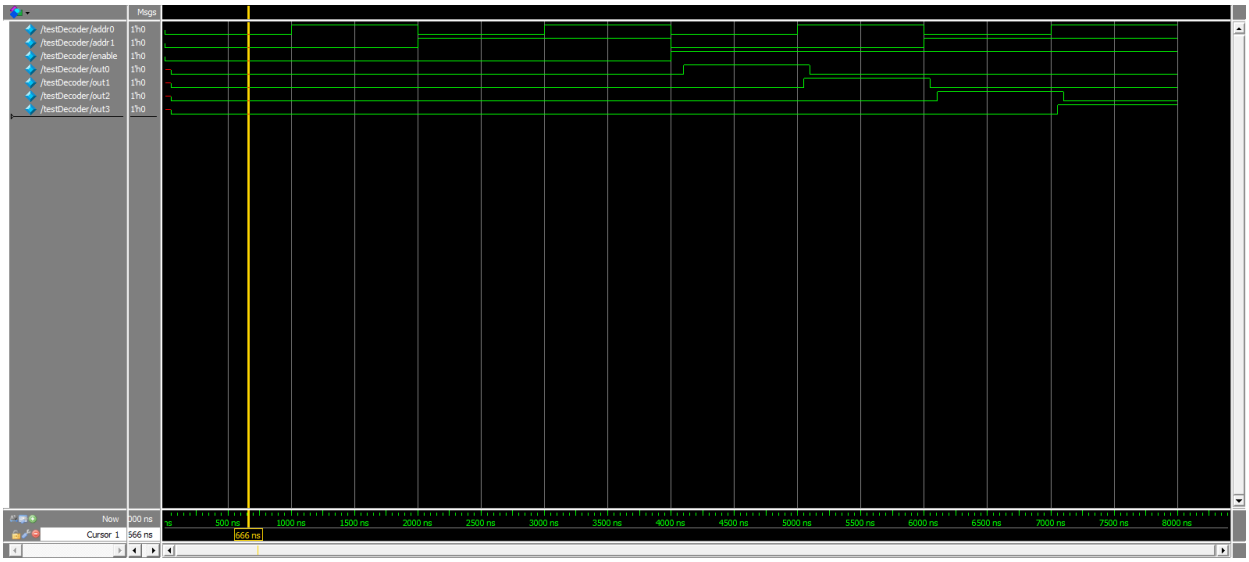
```
vlog -reportprogress 300 -work work decoder.v
vsim -voptargs="+acc" testDecoder
add wave -position insertpoint  \
sim:/testDecoder/addr0 \
sim:/testDecoder/addr1 \
sim:/testDecoder/enable \
sim:/testDecoder/out0 \
sim:/testDecoder/out1 \
sim:/testDecoder/out2 \
sim:/testDecoder/out3
run -all
wave zoom full
```

Bench Results:

This design was validated by looking at the expected input and output of signal. This was done exhaustively, by looking at all combinations of the address bits and enable bit.

```
# En A0 A1| O0 O1 O2 O3 | Expected Output
# 0  0  0 |  0  0  0  0 | All false
# 0  1  0 |  0  0  0  0 | All false
# 0  0  1 |  0  0  0  0 | All false
# 0  1  1 |  0  0  0  0 | All false
# 1  0  0 |  1  0  0  0 | O0 Only
# 1  1  0 |  0  1  0  0 | O1 Only
# 1  0  1 |  0  0  1  0 | O2 Only
# 1  1  1 |  0  0  0  1 | O3 Only
```

Waveform:



For a larger image, see Decoder_graph.bmp in repo.

3. 4-1 Multiplexer

Idea: The idea here was to generate a circuit that would pass on the input at the address specified. The most useful feature here was taking advantage of multiple input gates again.

Implementation Code:

```
module structuralMultiplexer(out, address0,address1, in0,in1,in2,in3);
output out;
input address0, address1;
input in0, in1, in2, in3;
wire naddress0;
wire naddress1;
//Creates Negatives of addresses.
not naddr0(naddress0, address0);
not naddr1(naddress1, address1);

//Three input ANDS that look at passing along the input that the address is pointing to.
wire o0, o1, o2, o3;
`AND a0(o0, in0, naddress0, naddress1);
`AND a1(o1, in1, address0, naddress1);
`AND a2(o2, in2, naddress0, address1);
`AND a3(o3, in3, address0, address1);

//ORs all the other input together again.
//Probably replacable if you can implement o0-o3 such
//you can have them be high impedance on a line. Not relevant in Verilog at the moment.
`OR orAll(out, o0,o1,o2,o3);

Endmodule
```

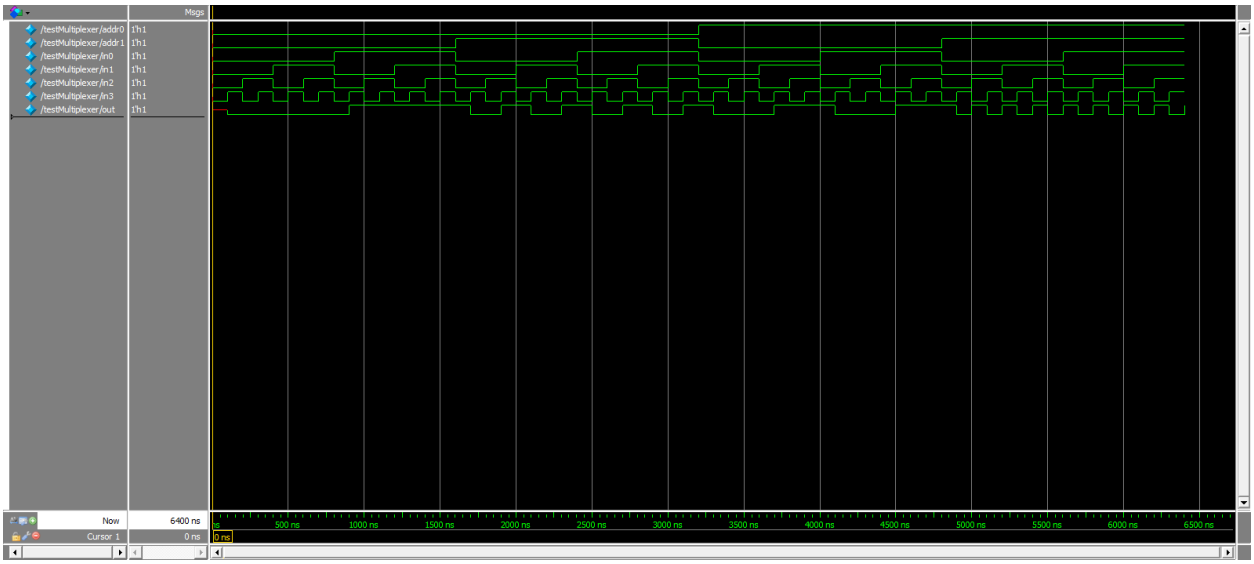
Do File:

```
vlog -reportprogress 300 -work work multiplexer.v
vsim -voptargs="+acc" testMultiplexer
add wave -position insertpoint  \
sim:/testMultiplexer/addr0 \
sim:/testMultiplexer/addr1 \
sim:/testMultiplexer/in0 \
sim:/testMultiplexer/in1 \
sim:/testMultiplexer/in2 \
sim:/testMultiplexer/in3 \
sim:/testMultiplexer/out
run -all
wave zoom full
```

Bench Results:
The testing was done exhaustively, with a comparison of the output `out` to the expected value `exp`.

```
# A0  A1  |  in0  in1  in2  in3 | out | exp
# 0   0   |   0    0    0    0  |  0  |  0
# 0   0   |   0    0    0    1  |  0  |  0
# 0   0   |   0    0    1    0  |  0  |  0
# 0   0   |   0    0    1    1  |  0  |  0
# 0   0   |   0    1    0    0  |  0  |  0
# 0   0   |   0    1    0    1  |  0  |  0
# 0   0   |   0    1    1    0  |  0  |  0
# 0   0   |   0    1    1    1  |  0  |  0
# 0   0   |   1    0    0    0  |  1  |  1
# 0   0   |   1    0    0    1  |  1  |  1
# 0   0   |   1    0    1    0  |  1  |  1
# 0   0   |   1    0    1    1  |  1  |  1
# 0   0   |   1    1    0    0  |  1  |  1
# 0   0   |   1    1    0    1  |  1  |  1
# 0   0   |   1    1    1    0  |  1  |  1
# 0   0   |   1    1    1    1  |  1  |  1
# 0   1   |   0    0    0    0  |  0  |  0
# 0   1   |   0    0    0    1  |  0  |  0
# 0   1   |   0    0    1    0  |  1  |  1
# 0   1   |   0    0    1    1  |  1  |  1
# 0   1   |   0    1    0    0  |  0  |  0
# 0   1   |   0    1    0    1  |  0  |  0
# 0   1   |   0    1    1    0  |  1  |  1
# 0   1   |   0    1    1    1  |  1  |  1
# 0   1   |   1    0    0    0  |  0  |  0
# 0   1   |   1    0    0    1  |  0  |  0
# 0   1   |   1    0    1    0  |  1  |  1
# 0   1   |   1    0    1    1  |  1  |  1
# 0   1   |   1    1    0    0  |  0  |  0
# 0   1   |   1    1    0    1  |  0  |  0
# 0   1   |   1    1    1    0  |  1  |  1
# 0   1   |   1    1    1    1  |  1  |  1
# 1   0   |   0    0    0    0  |  0  |  0
# 1   0   |   0    0    0    1  |  0  |  0
# 1   0   |   0    0    1    0  |  0  |  0
# 1   0   |   0    0    1    1  |  0  |  0
# 1   0   |   0    1    0    0  |  1  |  1
# 1   0   |   0    1    0    1  |  1  |  1
# 1   0   |   0    1    1    0  |  1  |  1
# 1   0   |   0    1    1    1  |  1  |  1
# 1   0   |   1    0    0    0  |  0  |  0
# 1   0   |   1    0    0    1  |  0  |  0
# 1   0   |   1    0    1    0  |  0  |  0
# 1   0   |   1    0    1    1  |  0  |  0
# 1   0   |   1    1    0    0  |  1  |  1
# 1   0   |   1    1    0    1  |  1  |  1
# 1   0   |   1    1    1    0  |  1  |  1
# 1   0   |   1    1    1    1  |  1  |  1
# 1   1   |   0    0    0    0  |  0  |  0
# 1   1   |   0    0    0    1  |  1  |  1
# 1   1   |   0    0    1    0  |  0  |  0
# 1   1   |   0    0    1    1  |  1  |  1
# 1   1   |   0    1    0    0  |  0  |  0
# 1   1   |   0    1    0    1  |  1  |  1
# 1   1   |   0    1    1    0  |  0  |  0
# 1   1   |   0    1    1    1  |  1  |  1
# 1   1   |   1    0    0    0  |  0  |  0
# 1   1   |   1    0    0    1  |  1  |  1
# 1   1   |   1    0    1    0  |  0  |  0
# 1   1   |   1    0    1    1  |  1  |  1
# 1   1   |   1    1    0    0  |  0  |  0
# 1   1   |   1    1    0    1  |  1  |  1
# 1   1   |   1    1    1    0  |  0  |  0
# 1   1   |   1    1    1    1  |  1  |  1
```

Waveform:



For a larger image, see Multiplexer_graph.bmp in repo.