LAB REPORT 2

2X4 DECODER, 3X8 DECODER
GROUP 13

GROUP MEMBERS: HASSAN AHMED

NEHA CHAUDHARY

HAMZA SHABBIR

SUBMITTED TO: Sir Asim

2.1 Objectives

The objectives of this lab are:

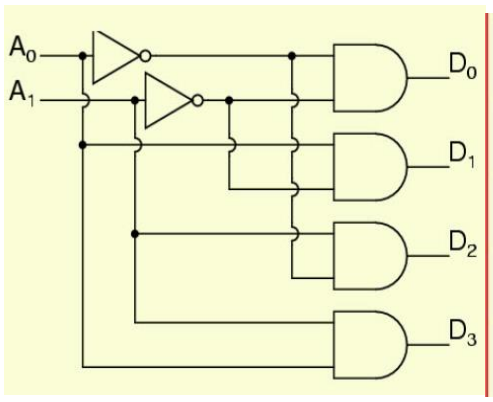
- We will be designing a 2 x 4 decoder schematic with the help of gate primitives
- We will be implementing the 3 x 8 decoder using

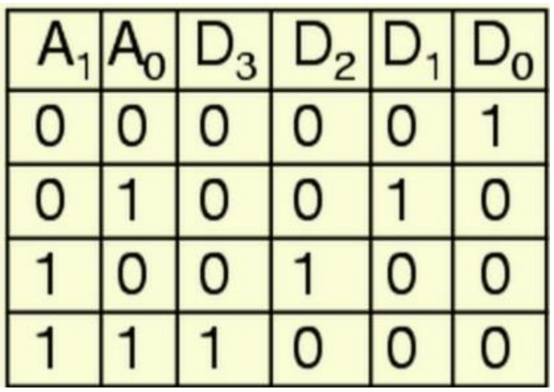
(i) instances using input and
(ii) output arrays

- Lastly, we will write the Verilog module to test 2 x 4 decoder & 3 x 8 decoder individually

2.2 Background

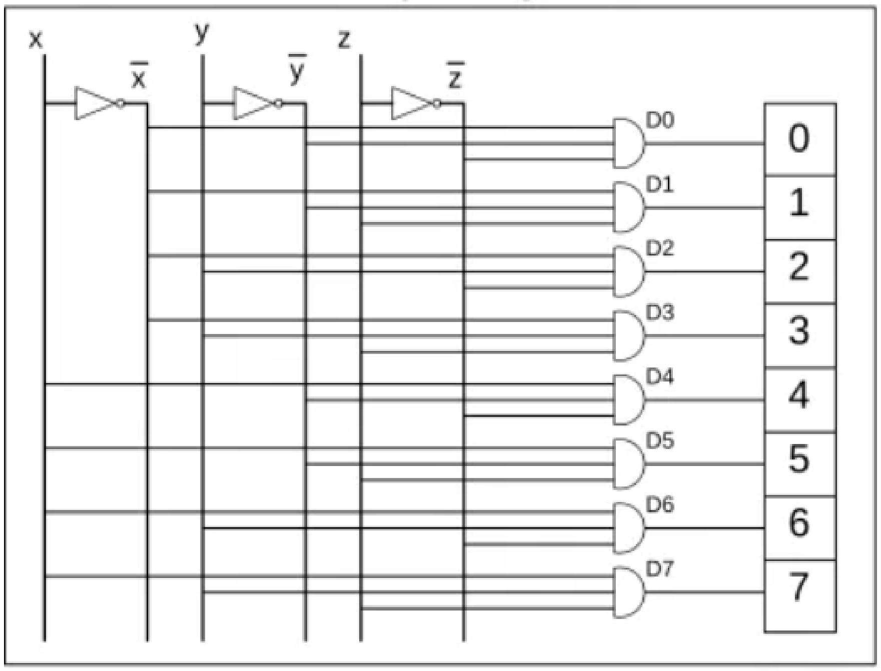
ModelSim is a great tool to implement different logic circuits. We are even able to write code at the gate level and then can move up the hierarchy to implement the bigger circuits at a modular level where we can call a module at a lower level to be used as our current module. This approach saves a lot of time and effort compared to the approach where we write the whole code at the gate level.
Decoders are used to send multiple different signals to different devices using fewer inputs. A decoder can be implemented using and gates. An enable signal can be added to the circuit which can be used when one decoder is used to implement another decoder. Verilog is an HDL language that is used for implementing digital systems at any level
Just like the mux circuit, the decoder can have multiple outputs (with two, three, or four address lines). The decoder circuit has the ability to decode a 2, 3, or 4-bit binary number, or can decode up to 4, 8, or 16 signals that have been multiplexed.
For this lab, we will implement a 2x4 decoder (with the help of gate primitives) and a 3x8 decoder (instances using input and output arrays).
Below are the schematic diagram and the truth table of a 2X4 decoder,



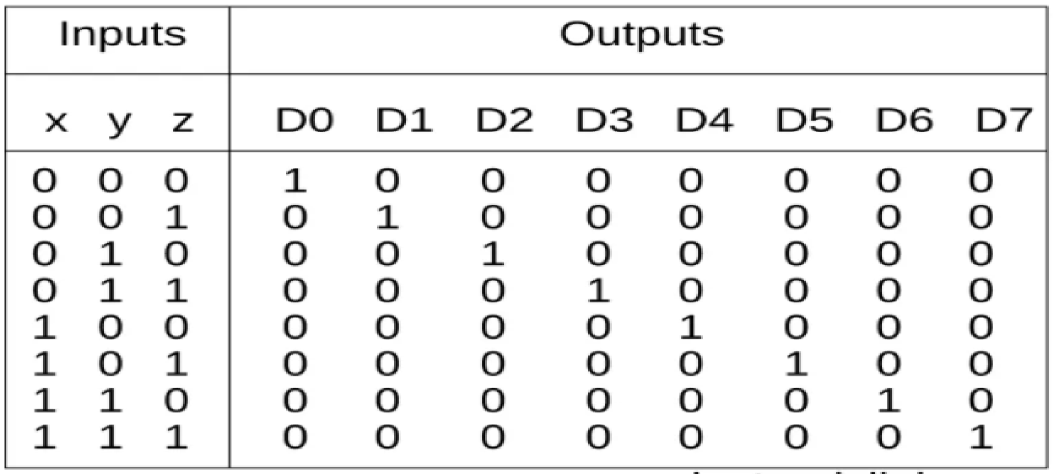


A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Below are the schematic diagram and the truth table of a 3X8 decoder, 3 x 8 line Decoder Logic Diagram



3 to 8 Line Decoder



Inputs			Outputs							
x	y	z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

2.3 Pre-Lab

2.3.1 Reading of ModelSim IDE tutorial to know Verilog language syntax

1. Lecture notes along with the reference material advised by the instructor should be available during the lab sessions
2. Knowledge of digital implementation of decoder and abstraction level syntax

2.4 Equipment

- Computer / laptop
- ModelSim / ISE IDE software

2.5 Procedure

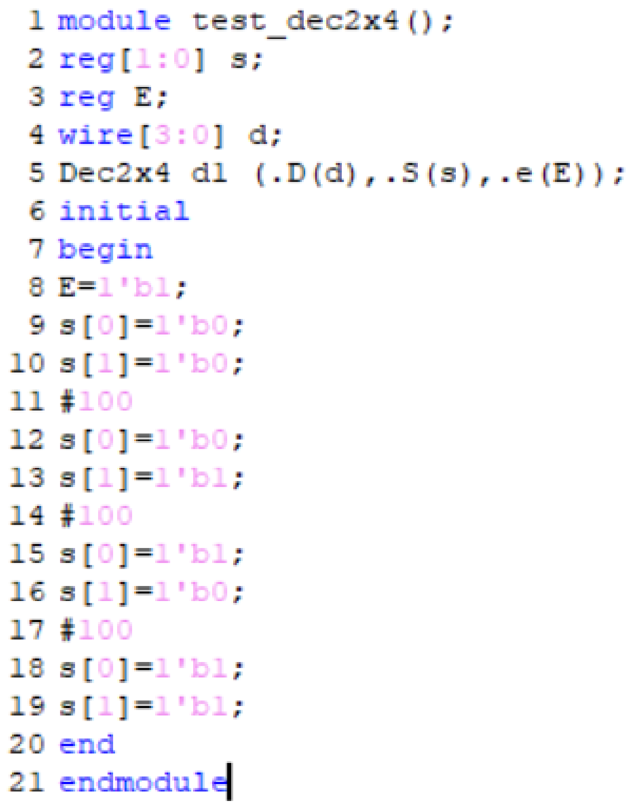
1. Firstly, we will create a project in ModelSim to implement the 2x4 decoder.
2. We will write the code for the 2x4 decoder using the gate primitives.
3. Next, we will write its test bench and then simulate and verify the results.
4. Then, we will write the Verilog code for the 3X8 decoder by using instances input and output arrays.
5. The function of both of the decoders will be confirmed by observing the waves produced by simulating the test bench and the waves will be provided in the next sections of the report.

2.6 Observations:

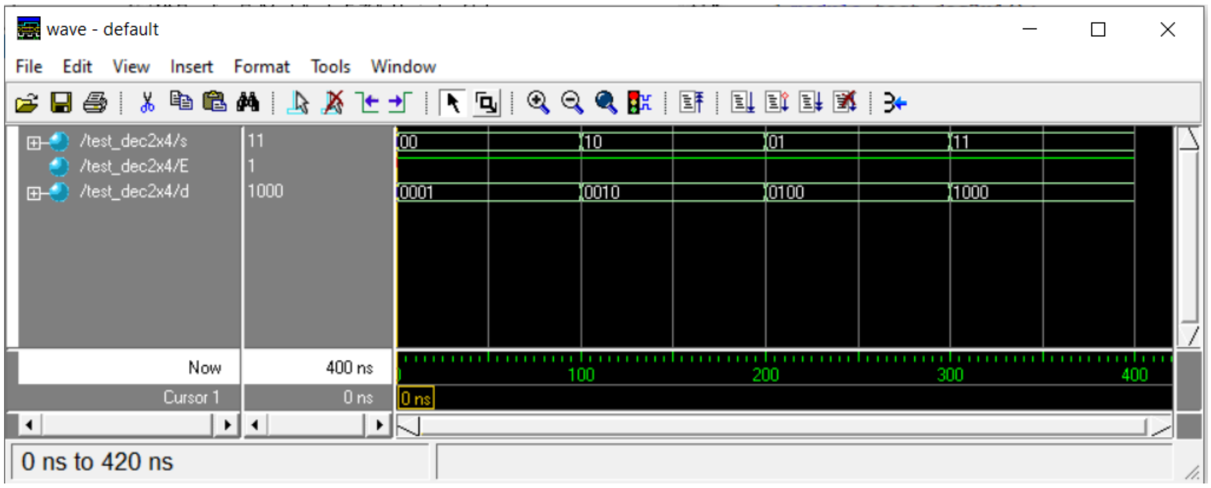
Firstly, 2x4 decoder was made using a gate-level approach and similarly, a 3x8 decoder was also made using the gate-level approach. Next, we implemented the same 3x8 decoder using a modular approach in which we used the previously made 2x4 decoder. At the last, we implemented a 3x8 decoder using dataflow abstraction.
The code for all the decoders is given below:

```
module Dec2x4 (D,S,e);
input[1:0] S;
input e;
output[3:0] D;
wire[1:0] w;
not (w[0],S[0]);
not (w[1],S[1]);
and (D[0],w[0],w[1],e);
and (D[1],w[0],S[1],e);
and (D[2],S[0],w[1],e);
and (D[3],S[0],S[1],e);
endmodule

module DEC3X8 (OUT,IN);
input[2:0] IN;
output[7:0] OUT;
wire[2:0] W;
not (W[0],IN[0]);
not (W[1],IN[1]);
not (W[2],IN[2]);
and (OUT[0],W[0],W[1],W[2]);
and (OUT[1],IN[0],W[1],W[2]);
and (OUT[2],W[0],IN[1],W[2]);
and (OUT[3],IN[0],IN[1],W[2]);
and (OUT[4],W[0],W[1],IN[2]);
and (OUT[5],IN[0],W[1],IN[2]);
and (OUT[6],W[0],IN[1],IN[2]);
and (OUT[7],IN[0],IN[1],IN[2]);
endmodule

module Dec3x8 (O,I);
input[2:0] I;
output[7:0] O;
wire w;
not (w,I[2]);
Dec2x4 D1(.D(O[3:0]),.S(I[1:0]),.e(w));
Dec2x4 D2(.D(O[7:4]),.S(I[1:0]),.e(I[2]));
endmodule

module dec3x8(out,in);
input[2:0] in;
output[7:0] out;
assign out=8'b00000001<<in;
endmodule
```

Now each module was tested separately. The test bench and simulation for the 2x4 decoder is given below:

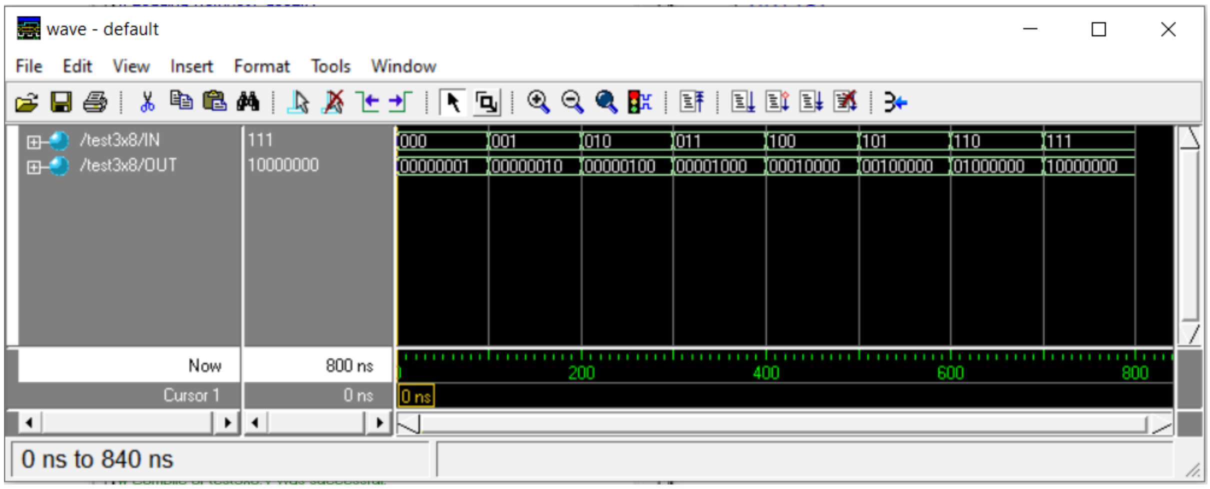


```
module test_dec2x4();
reg[1:0] s;
reg E;
wire[3:0] d;
Dec2x4 d1 (.D(d),.S(s),.e(E));
initial
begin
E=1'b1;
s[0]=1'b0;
s[1]=1'b0;
#100
s[0]=1'b0;
s[1]=1'b1;
#100
s[0]=1'b1;
s[1]=1'b0;
#100
s[0]=1'b1;
s[1]=1'b1;
end
endmodule
```



Now the test bench and simulation of 3x8 decoder using a gate-level approach:

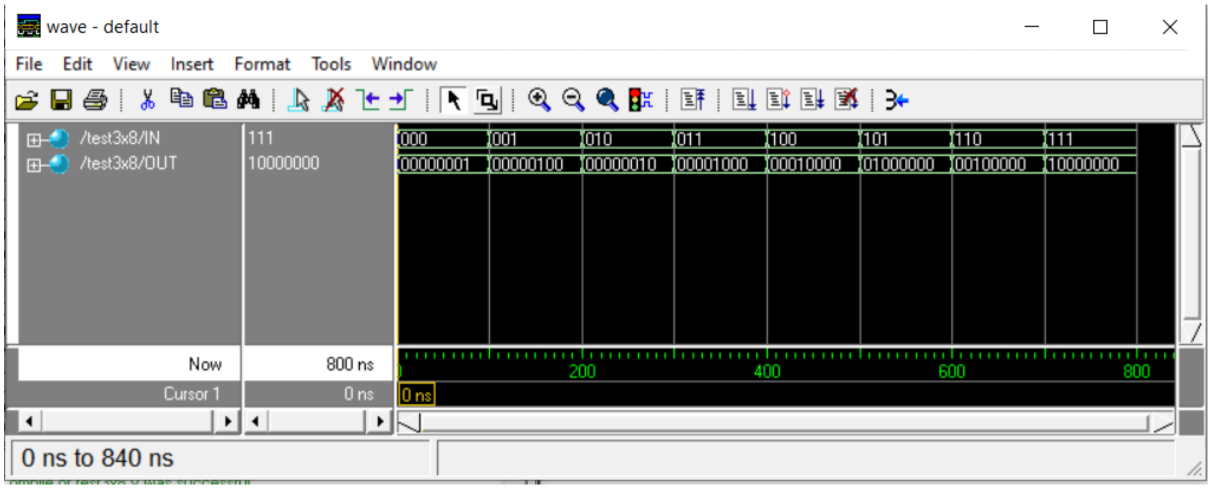
```
module test3x8();
reg[2:0] IN;
wire[7:0] OUT;
DEC3X8 D1(.OUT(OUT),.IN(IN));
initial
begin
IN=3'b000;
#100
IN=3'b001;
#100
IN=3'b010;
#100
IN=3'b011;
#100
IN=3'b100;
#100
IN=3'b101;
#100
IN=3'b110;
#100
IN=3'b111;
end
endmodule
```



```
/test3x8/IN    111        000       001       010       011       100       101       110       111
/test3x8/OUT   10000000   00000001  00000010  00000100  00001000  00010000  00100000  01000000  10000000
Now            800 ns               200                 400                 600                 800
Cursor 1       0 ns       0 ns
0 ns to 840 ns
```

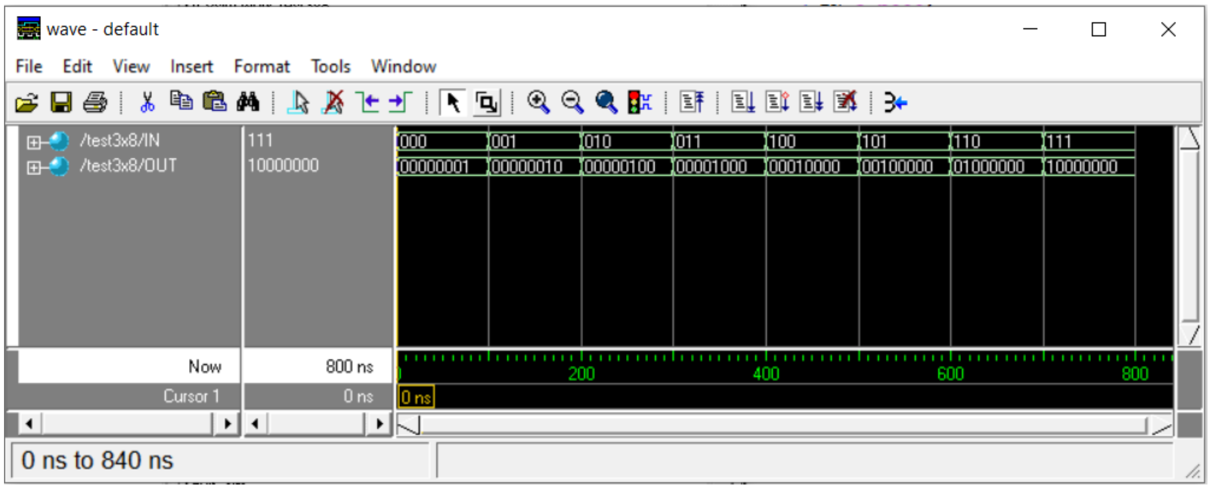
Simulation and test bench for 3x8 decoder using a modular approach:

```
module test3x8();
reg[2:0] IN;
wire[7:0] OUT;
Dec3x8 D1(.O(OUT),.I(IN));
initial
begin
IN=3'b000;
#100
IN=3'b001;
#100
IN=3'b010;
#100
IN=3'b011;
#100
IN=3'b100;
#100
IN=3'b101;
#100
IN=3'b110;
#100
IN=3'b111;
end
endmodule
```



Simulation and test bench for 3x8 decoder using dataflow abstraction.

```
module test3x8();
reg[2:0] IN;
wire[7:0] OUT;
dec3x8 D1(.out(OUT),.in(IN));
initial
begin
IN=3'b000;
#100
IN=3'b001;
#100
IN=3'b010;
#100
IN=3'b011;
#100
IN=3'b100;
#100
IN=3'b101;
#100
IN=3'b110;
#100
IN=3'b111;
end
endmodule
```



```
/test3x8/IN    111        000       001       010       011       100       101       110       111
/test3x8/OUT   10000000   00000001  00000010  00000100  00001000  00010000  00100000  01000000  10000000
Now            800 ns     200  400  600  800
Cursor 1       0 ns       0 ns
0 ns to 840 ns
```

2.7 Results

Dataflow abstraction makes the code size very small and easy to manage as seen from the code above. Many examples like addition, subtraction, multiplication and comparison ect which was previously implemented by using long codes can be completed using a single statement in dataflow.