

Digital Design with Verilog

Verilog

Lecture 14: Behavioral Modeling of Combinational Circuits





Learning Objectives

- Design of combinational circuits using Verilog in particular:
 - Decoders, and Encoders
 - Multiplexers, and De-multiplexers
 - Half-adder, Full adder, and Binary adder
 - Ripple Carry Adder, Carry Look ahead Adder
 - Comparators
 - Parity Generators & Checkers
 - Simple Calculator

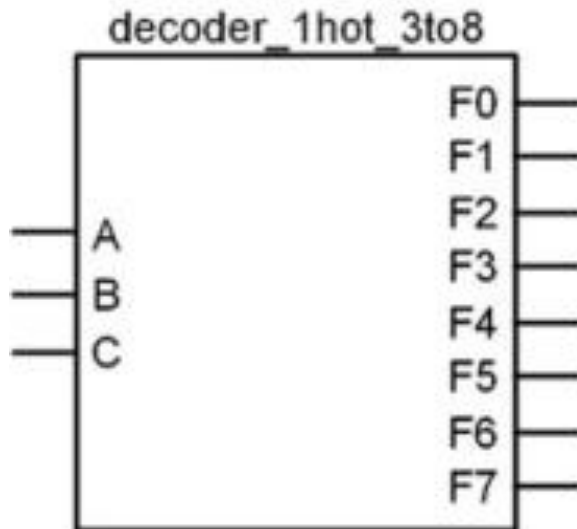


Decoders

- A decoder is a circuit that takes in a binary code and has outputs that are asserted for specific values of that code.
- The code can be of any type or size (e.g., unsigned, two's complement, etc.).
- Each output will assert for only specific input codes.
- Since combinational logic circuits only produce a single output, this means that within a decoder, there will be a separate combinational logic circuit for each output

One-Hot Decoder

- A one-hot decoder is a circuit that has n inputs and 2^n outputs. Each output will assert for one and only one input code.



A	B	C	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



One-Hot Decoder

```
module decoder_1hot_3to8
  (output wire F0, F1, F2, F3, F4, F5, F6, F7,
   input wire A, B, C);

  assign F0 = ~A & ~B & ~C;
  assign F1 = ~A & ~B & C;
  assign F2 = ~A & B & ~C;
  assign F3 = ~A & B & C;
  assign F4 = A & ~B & ~C;
  assign F5 = A & ~B & C;
  assign F6 = A & B & ~C;
  assign F7 = A & B & C;

endmodule
```



One-Hot Decoder

- We can use vector notation for the ports and describe the functionality using conditional operators.

```
module decoder_1hot_3to8
  (output wire [7:0] F,
   input  wire [2:0] ABC);

  assign F = (ABC == 3'b000) ? 8'b0000_0001 :
             (ABC == 3'b001) ? 8'b0000_0010 :
             (ABC == 3'b010) ? 8'b0000_0100 :
             (ABC == 3'b011) ? 8'b0000_1000 :
             (ABC == 3'b100) ? 8'b0001_0000 :
             (ABC == 3'b101) ? 8'b0010_0000 :
             (ABC == 3'b110) ? 8'b0100_0000 :
             (ABC == 3'b111) ? 8'b1000_0000 :
             8'bXXXX_XXXX;

endmodule
```



3x8 Decoder

```
module three_to_eight_decoder(y,x);  
  
    input [2:0] x;  
    output reg [7:0] y;  
  
    initial  
        y = 8'b0;  
  
    always @ (x)  
        case(x)  
            3'b000 : y = 8'b00000001;  
            3'b001 : y = 8'b00000010;  
            3'b010 : y = 8'b00000100;  
            3'b011 : y = 8'b00001000;  
            3'b100 : y = 8'b00010000;  
            3'b101 : y = 8'b00100000;  
            3'b110 : y = 8'b01000000;  
            3'b111 : y = 8'b10000000;  
        endcase  
  
endmodule
```



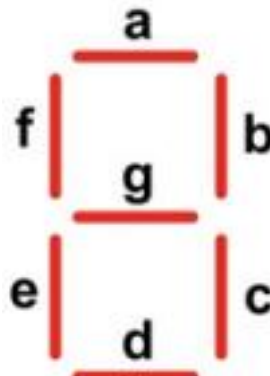
7-Segment Display Decoder









- A 7-segment display decoder is a circuit used to drive character displays that are commonly found in digital clocks and household appliances.
- A character display is made up of 7 individual LEDs, typically labeled a-g.
- The input to the decoder is the binary equivalent of the decimal or Hex character that is to be displayed.
- The output of the decoder is the arrangement of LEDs that will form the character.

7-Segment Display Decoder

Example: 7-Segment Display Decoder - Truth Table

LED Labels



A	B	C		F _a	F _b	F _c	F _d	F _e	F _f	F _g
0	0	0		1	1	1	1	1	1	0
0	0	1		0	1	1	0	0	0	0
0	1	0		1	1	0	1	1	0	1
0	1	1		1	1	1	1	0	0	1
1	0	0		0	1	1	0	0	1	1
1	0	1		1	0	1	1	0	1	1
1	1	0		1	0	1	1	1	1	1
1	1	1		1	1	1	0	0	0	0



7-Segment Display Decoder

```
module decoder_7seg
  (output wire Fa, Fb, Fc, Fd, Fe, Ff, Fg,
   input wire A, B, C);

  assign Fa = (~A & ~C) | (B) | (A & C);
  assign Fb = (~B & ~C) | (~A) | (B & C);
  assign Fc = (A) | (~B) | (C);
  assign Fd = (~A & ~C) | (~A & B) | (B & ~C) | (A & ~B & C);
  assign Fe = (~A & ~C) | (B & ~C);
  assign Ff = (~B & ~C) | (A & ~C) | (A & ~B);
  assign Fg = (~A & B) | (A & ~C) | (A & ~B);

endmodule
```



7-Segment Display Decoder

```
module decoder_7seg
    (output wire [6:0] F,
     input  wire [2:0] ABC);

    assign F = (ABC == 3'b000) ? 7'b111_1110 :
               (ABC == 3'b001) ? 7'b011_0000 :
               (ABC == 3'b010) ? 7'b110_1101 :
               (ABC == 3'b011) ? 7'b111_1001 :
               (ABC == 3'b100) ? 7'b011_0011 :
               (ABC == 3'b101) ? 7'b101_1011 :
               (ABC == 3'b110) ? 7'b101_1111 :
               (ABC == 3'b111) ? 7'b111_0000 :
               8'bXXXX_XXXX;

endmodule
```



7-Segment Display Decoder

```
module decoder_7seg(in1,out1);

input [3:0] in1;
output reg [6:0] out1;

always @ (in1)
case (in1)
    4'b0000 : out1=7'b1000000; //0
    4'b0001 : out1=7'b1111001; //1
    4'b0010 : out1=7'b0100100; //2
    4'b0011 : out1=7'b0110000; //3
    4'b0100 : out1=7'b0011001; //4
    4'b0101 : out1=7'b0010010; //5
    4'b0110 : out1=7'b0000010; //6
    4'b0111 : out1=7'b1111000; //7
    4'b1000 : out1=7'b0000000; //8
    4'b1001 : out1=7'b0010000; //9
    4'b1010 : out1=7'b0001000; //A
    4'b1011 : out1=7'b0000011; //B
    4'b1100 : out1=7'b1000110; //C
    4'b1101 : out1=7'b0100001; //D
    4'b1110 : out1=7'b0000110; //E
    4'b1111 : out1=7'b0001110; //F
endcase

endmodule
```



Encoders

- An encoder works in the opposite manner as a decoder.
- An assertion on a specific input port corresponds to a unique code on the output port.

One-Hot Binary Encoder

- A one-hot binary encoder has n outputs and 2^n inputs.
- The output will be an n -bit, binary code which corresponds to an assertion on one and only one of the inputs



One-Hot Binary Encoder

```
module encoder_1hot_4to2 (output wire [1:0] YZ,  
                          input  wire [3:0] ABCD);  
  
    assign YZ[1] = ABCD[3] | ABCD[2];  
    assign YZ[0] = ABCD[3] | ABCD[1];  
  
endmodule
```

```
module encoder_1hot_4to2 (output wire [1:0] YZ,  
                          input  wire [3:0] ABCD);  
  
    assign YZ = (ABCD == 4'b0001) ? 2'b00 :  
                (ABCD == 4'b0010) ? 2'b01 :  
                (ABCD == 4'b0100) ? 2'b10 :  
                (ABCD == 4'b1000) ? 2'b11 :  
                2'bXX;  
  
endmodule
```



8x3 Encoder

```
module eight_to_three_encoder(y,x);

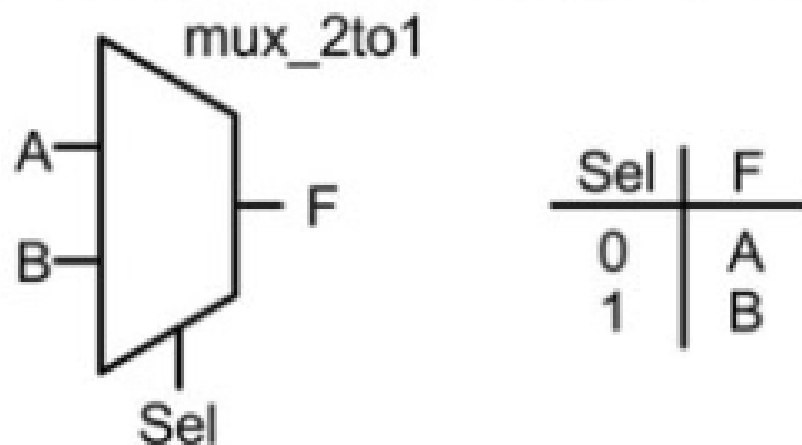
input [7:0] x;
output reg [2:0] y;

initial
y = 3'bzzz;

always @ (x)
case(x)
    8'b1xxxxxxx : y = 3'b111;
    8'b01xxxxxx : y = 3'b110;
    8'b001xxxxx : y = 3'b101;
    8'b0001xxxx : y = 3'b100;
    8'b00001xxx : y = 3'b011;
    8'b000001xx : y = 3'b010;
    8'b0000001x : y = 3'b001;
    8'b00000001 : y = 3'b000;
    default :      y = 3'bzzz;
endcase
endmodule
```

Multiplexers

- A multiplexer is a circuit that passes one of its multiple inputs to a single output based on a select input.
- This can be thought of as a digital switch.
- The multiplexer has n select lines, 2^n inputs, and one output.





Multiplexers

- A multiplexer can be implemented using continuous assignment with either logical or conditional operators.

```
module mux_4to1 (output wire F,  
                input  wire A, B, C, D,  
                input  wire [1:0] Sel);
```

```
    assign F = (A & ~Sel[1] & ~Sel[0]) |  
              (B & ~Sel[1] &  Sel[0]) |  
              (C &  Sel[1] & ~Sel[0]) |  
              (D &  Sel[1] &  Sel[0]);
```

```
endmodule
```

```
module mux_4to1 (output wire F,  
                input  wire A, B, C, D,  
                input  wire [1:0] Sel);
```

```
    assign F = (Sel == 2'b00) ? A :  
              (Sel == 2'b01) ? B :  
              (Sel == 2'b10) ? C :  
              (Sel == 2'b11) ? D : 1'bX;
```

```
endmodule
```



2x1 Multiplexer

// Behavioral description of two-to-one-line multiplexer

```
module mux_2x1_beh (m_out, A, B, select);  
  output      m_out;  
  input       A, B, select;  
  reg         m_out;  
  
  always      @(A or B or select)  
    if (select == 1) m_out = A;  
    else m_out = B;  
endmodule
```



4x1 Multiplexer

// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

```
module mux_4x1_beh
( output reg m_out,
  input      in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select) // Verilog 2001, 2005 syntax
  case (select)
    2'b00:      m_out = in_0;
    2'b01:      m_out = in_1;
    2'b10:      m_out = in_2;
    2'b11:      m_out = in_3;
  endcase
endmodule
```



8x1 Multiplexer

```
module eight_to_one_muxplexer(y,x,s);

    input [2:0] s;
    input [7:0] x;
    output reg y;

    always @ (s or x)
    case(s)
        3'b000 : y = x[0];
        3'b001 : y = x[1];
        3'b010 : y = x[2];
        3'b011 : y = x[3];
        3'b100 : y = x[4];
        3'b101 : y = x[5];
        3'b110 : y = x[6];
        3'b111 : y = x[7];
    endcase

endmodule
```



Demultiplexers

- A demultiplexer works in a complementary fashion to a multiplexer.
- A demultiplexer has one input that is routed to one of its multiple outputs.
- The output that is active is dictated by a select input. A demux has n select lines that chooses to route the input to one of its 2^n outputs.
- When an output is not selected, it outputs a logic 0.



Demultiplexers

```
module demux_1to4 (output wire W, X, Y, Z,  
                  input  wire A,  
                  input  wire [1:0] Sel);  
  
    assign W = (A & ~Sel[1] & ~Sel[0]);  
    assign X = (A & ~Sel[1] & Sel[0]);  
    assign Y = (A & Sel[1] & ~Sel[0]);  
    assign Z = (A & Sel[1] & Sel[0]);  
  
endmodule
```

```
module demux_1to4 (output wire W, X, Y, Z,  
                  input  wire A,  
                  input  wire [1:0] Sel);  
  
    assign W = (Sel == 2'b00) ? A : 1'b0;  
    assign X = (Sel == 2'b01) ? A : 1'b0;  
    assign Y = (Sel == 2'b10) ? A : 1'b0;  
    assign Z = (Sel == 2'b11) ? A : 1'b0;  
  
endmodule
```

Half-Adder

Inputs		Outputs	
x	y	s	co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
module one_bit_half_adder(s,co,x,y);  
  
    // Port definitions  
    input  x,y;  
    output s,co;  
  
    // Structural modeling  
    and g1(co,x,y);  
    xor g2(s,x,y);  
  
    // Dataflow modeling  
    assign co = x & y;  
    assign s = x ^ y;  
  
endmodule
```



Full-Adder

Inputs			Outputs	
x	y	ci	s	co
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

```
module one_bit_full_adder(s,co,x,y,ci);  
  
    // Port definitions  
    input  x,y,ci;  
    // for structural and functional modeling  
    output s,co;  
  
    // for structural modeling  
    wire o1,o2,o3;  
  
    // Structural modeling  
    and g1(o1,x,y);  
    xor g2(o2,x,y);  
    xor g3(s,o1,ci);  
    and g4(o3,o1,ci);  
    or  g5(co,o2,o3);  
  
    // Dataflow modeling  
    assign co = (x & y) | (ci & (x^y));  
    assign s = x^y^ci;  
  
endmodule
```




Full Adder

```
module half_adder (output wire Sum, Cout,  
                  input  wire A, B);
```

```
    xor U1 (Sum, A, B);  
    and U2 (Cout, A, B);
```

```
endmodule
```

```
module full_adder (output wire Sum, Cout,  
                  input  wire A, B, Cin);
```

```
    wire HA1_Sum, HA1_Cout, HA2_Cout;
```

```
    half_adder U1 (.Sum(HA1_Sum), .Cout(HA1_Cout), .A(A), .B(B));  
    half_adder U2 (.Sum(Sum), .Cout(HA2_Cout), .A(HA1_Sum), .B(Cin));
```

```
    or U3 (Cout, HA2_Cout, HA1_Cout);
```

```
endmodule
```



4-bit Ripple Carry Adder

```
module rca_4bit (output wire [3:0] Sum,
                output wire      Cout,
                input  wire [3:0] A, B);

    wire C1, C2, C3;

    full_adder U1 (.Sum(Sum[0]), .Cout(C1), .A(A[0]), .B(B[0]), .Cin(1'b0));
    full_adder U2 (.Sum(Sum[1]), .Cout(C2), .A(A[1]), .B(B[1]), .Cin(C1));
    full_adder U3 (.Sum(Sum[2]), .Cout(C3), .A(A[2]), .B(B[2]), .Cin(C2));
    full_adder U4 (.Sum(Sum[3]), .Cout(Cout), .A(A[3]), .B(B[3]), .Cin(C3));

endmodule
```



Binary Adder

```
module adder_4bit (output wire [3:0] Sum,  
                  output wire      Cout,  
                  input  wire [3:0] A, B);  
  
    assign {Cout, Sum} = A + B;  
  
endmodule
```



Carry Look Ahead Adder

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

$$S_i = P_i \oplus G_i.$$

$$C_{i+1} = C_i.P_i + G_i.$$

```
module cla_4bit (output wire [3:0] Sum,  
                output wire      Cout,  
                input  wire [3:0] A, B);
```

```
    wire      c0, c1, c2, c3;  
    wire [3:0] p, g;
```

```
    assign c0  = 1'b0;  
    assign c1  = g[0] | (p[0] & c0);  
    assign c2  = g[1] | (p[1] & c1);  
    assign c3  = g[2] | (p[2] & c2);  
    assign Cout = g[3] | (p[3] & c3);
```

```
    mod_full_adder U0 (.Sum(Sum[0]), .p(p[0]), .g(g[0]), .A(A[0]), .B(B[0]), .Cin(c0));  
    mod_full_adder U1 (.Sum(Sum[1]), .p(p[1]), .g(g[1]), .A(A[1]), .B(B[1]), .Cin(c1));  
    mod_full_adder U2 (.Sum(Sum[2]), .p(p[2]), .g(g[2]), .A(A[2]), .B(B[2]), .Cin(c2));  
    mod_full_adder U3 (.Sum(Sum[3]), .p(p[3]), .g(g[3]), .A(A[3]), .B(B[3]), .Cin(c3));
```

```
endmodule
```



Comparator

```
module one_bit_comparator(g,e,l,x,y);  
  
    // Port definitions  
    input  x,y;  
    // for structural and functional modeling  
    output g,e,l;  
  
    // for structural modeling  
    wire o1,o2,o3;  
  
    // Structural modeling  
    not g1(o1,y);  
    and g2(g,o1,x);  
    xor g3(o2,x,y);  
    not g4(e,o2);  
    not g5(o3,x);  
    and g6(l,o3,y);  
  
    // Dataflow modeling  
    assign g = x & ~y;  
    assign e = ~(x ^ y);  
    assign l = ~x & y;  
  
endmodule
```



Four-bit Comparator

```
module mag_compare
( output          A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]    A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```



3-bit Parity Generator

```
module three_bit_even_parity_generator(pe,b);  
  
    // Port definitions  
    input [2:0] b;  
    output pe;  
  
    // for structural modeling  
    wire w1;  
  
    // Structural modeling  
    xor g1(w1,b[0],b[1]);  
    xor g2(pe,w1,b[2]);  
  
    // Dataflow modeling  
    assign pe = b[0] ^ b[1] ^ b[2];  
  
endmodule
```



3-bit Parity Checker

```
module three_bit_even_parity_checker(c,pe,b);  
  
    // Port definitions  
    input [2:0] b;  
    input pe;  
    output c;  
  
    // for structural modeling  
    wire w1,w2;  
  
    // Structural modeling  
    xor g1(w1,b[0],b[1]);  
    xor g2(w2,pe,b[2]);  
    xor g3(c,w1,w2);  
  
    // Dataflow modeling  
    assign c = b[0] ^ b[1] ^ b[2] ^ pe;  
  
endmodule
```




Simple Calculator

```
module calculator(a,b,op,result);

parameter number_length=4;
input [number_length-1:0] a;
input [number_length-1:0] b;
input [2:0] op;
output reg [15:0] result;

always @ (*)
case(op)
    2'b00 : result <= a + b;
    2'b01 : result <= a - b;
    2'b10 : result <= a * b;
    2'b11 : result <= a / b;
endcase

endmodule
```

References

- Chapter 9, Introduction to Logic Circuits & Logic Design with Verilog by Brock J. LaMerres
- Disclaimer: “I don’t claim the ownership of all the slides, some of the material is picked up from various publicly available sources on the internet”.

Thank you