Verilog Lecture 4

- Using Procedures
- Simple Arithmetic Circuits
- Modular Design
- Vectors
- Constants
- Verification

```
module Mux2to1D( A, B, S, W )
  input A, B, S;
  output reg W;
  // using a procedure
  always @( A, B, S ) begin
    if ( S ) W = B;
    else W = A;
  end
endmodule
```

Mux2to1D.v

Procedural Programming Statements

if-statement

```
if ( expression) begin
  // statements
end
```

if/else-statement

```
if ( expression) begin
  // statements
else begin
 // statements
end
```

case-statement

```
case ( expression )
  case_item: begin … end
  case_item, case_item: begin … end
  default: begin … end
endcase
```

also: `casez` and `casex` which we'll talk about later

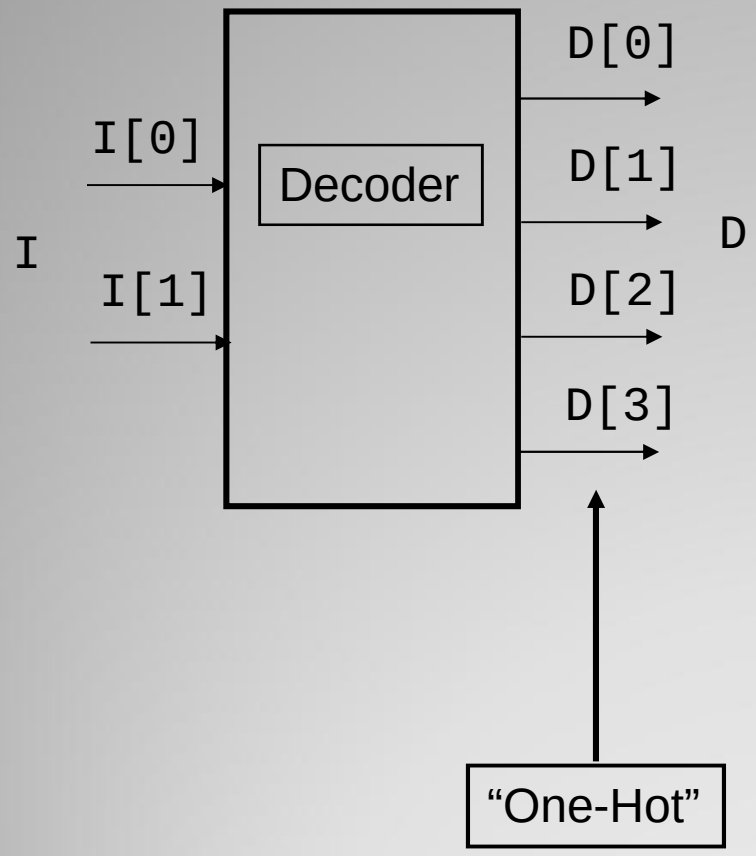
for-loop : be very careful

```
for (initial_assignment; expression; step_assignment) begin
  // statements
end
```

BCD7seg.v

BCD to 7-Seg Decoder

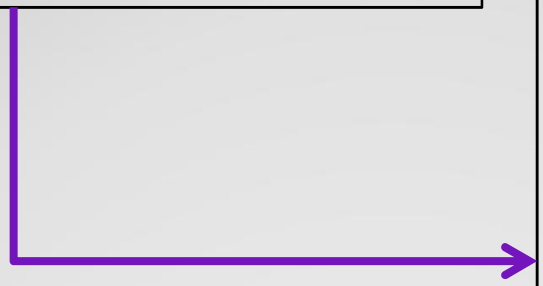
Decoder Example




```
module TwoDecoder( I, D );
  input [1:0] I;
  output reg [3:0]D;

  always @( I ) begin
    if ( I == 2'd0 ) begin
      D = 4'b0001;
    end  // I == 0
    else if ( I == 2'd1 ) begin
      D = 4'b0010;
    end  // I == 1
    else if ( I == 2'd2 ) begin
      D = 4'b0100;
    end  // I == 2
    else begin
      D = 4'b1000;
    end  // I == 3 (default)
  end

endmodule
```

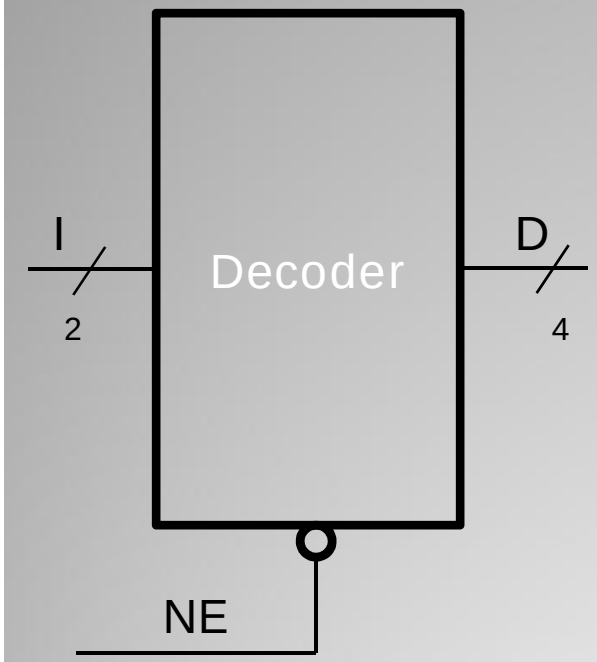
(To do: add Enable)

```
always @ ( C )
    case ( C )
      4'h0: Display = 7'b0000001;
      4'h1: Display = 7'b1001111;
      4'h2: Display = 7'b...;
      4'h3: Display = 7'b...;
      4'h4: Display = 7'b...;
      4'h5: Display = 7'b...;
      4'h6: Display = 7'b..;
      4'h7: Display = 7'b...;
      4'h8: Display = 7'b...;
      4'h9: Display = 7'b0000100;
      default: Display = 7'b1111111;  // blank
  endcase
```



```
module TwoDecoder( I, D );
  input [1:0] I;
  output reg [3:0]D;

  always @( I ) begin
    if ( I == 2'd0 ) begin
      D = 4'b0001;
    end  // I == 0
    else if ( I == 2'd1 ) begin
      D = 4'b0010;
    end  // I == 1
    else if ( I == 2'd2 ) begin
      D = 4'b0100;
    end  // I == 2
    else begin
      D = 4'b1000;
    end  // I == 3 (default)
  end

endmodule
```

Rewrite (Streamline) TwoDecoder



If NE is LOW the chip is enabled (normal operation). If NE is HIGH the chip is disabled and all outputs are 0

Using an ‘if-else’ statement add NE to your decoder.

```
if ( expression) begin
  // statements
else begin
 // statements
end
```

(Not) Enable

```
module Decoder( I, NE, D );
  input [1:0] I;   // inputs
  input NE;        // notEnable
  output reg [3:0] D;  // outputs

  always @( I, NE ) begin
    if ( NE == 1'b1 )
      D = 4'b0;
    else begin
       D = 4'b0001 << I; // shift operator
    end // else

  end  // always

endmodule
```

Yet Another Decoder Method

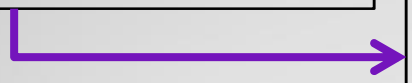
```
module Decoder( I, NE, D );
  parameter NAddr = 2;  // default number of address lines

  input [NAddr-1:0] I;  // address lines
  input NE;             // notEnable
  output reg [2**NAddr-1:0] D;  // 2^Naddr decoder outputs

  always @( I, NE ) begin

    if ( NE == 1'b1 )
      D = 0;          // truncation, but no warning
    else
      D = 1'b1 << I;  // Verilog fills in with 0s, but no warning

  end  // always

endmodule
```

Parameterized Version

```
module Decoder( I, NE, D );

  input [1:0] I;   // address lines
  input NE;        // notEnable

  output reg [3:0] D;  // decoder outputs

  always @( I, NE ) begin
    D = 0;                  // default output
    if ( NE == 0 ) begin  // circuit is enabled
       D[I] = 1'b1;
    end // if

  end  // always

endmodule
```

Last Version of Decoder

```
module Decoder( I, NE, D );
  parameter NAddr = 2;
  input [NAddr-1:0] I;
  input NE;
  output reg [2**NAddr-1:0] D;
  always @( I, NE ) begin
    if ( NE == 1'b1 )
      D = 0;
    else
      D = 1'b1 << I;
  end  // always

endmodule
```



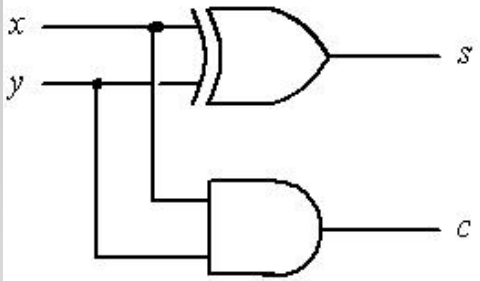
```
module Decoder( I, NE, D );

  input [1:0] I;   // address lines
  input NE;        // notEnable

  output reg [3:0] D;  // decoder outputs

  always @( I, NE ) begin
    D = 0;                // default output
    if ( NE == 0 ) begin // circuit enabled
      D[I] = 1'b1;
    end // if

  end  // always

endmodule
```

Parameterize This Version

Half Adder

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

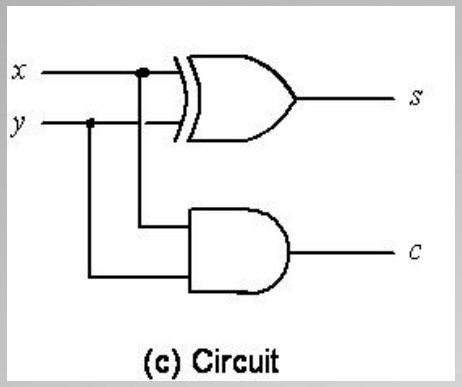
(b) Truth table



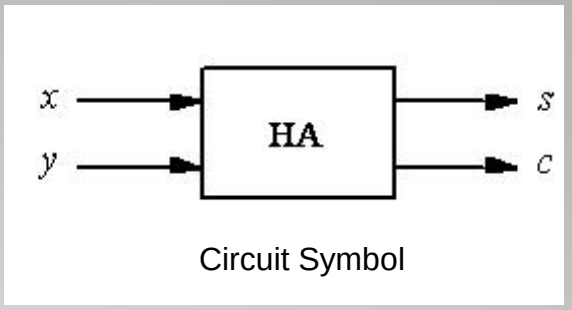
(c) Circuit

(d) Graphical symbol

Half Adder in Verilog

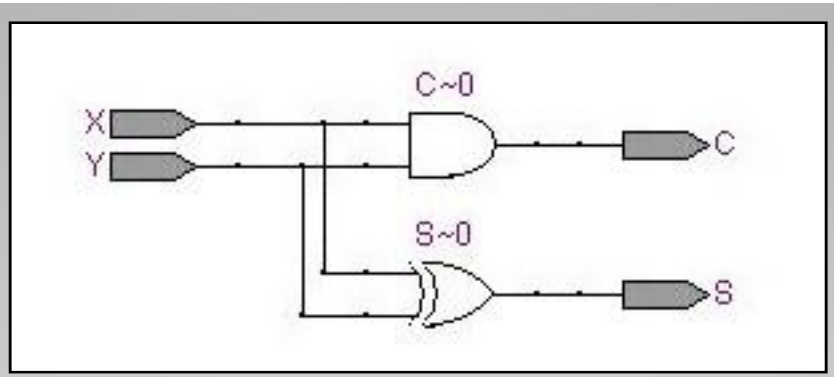


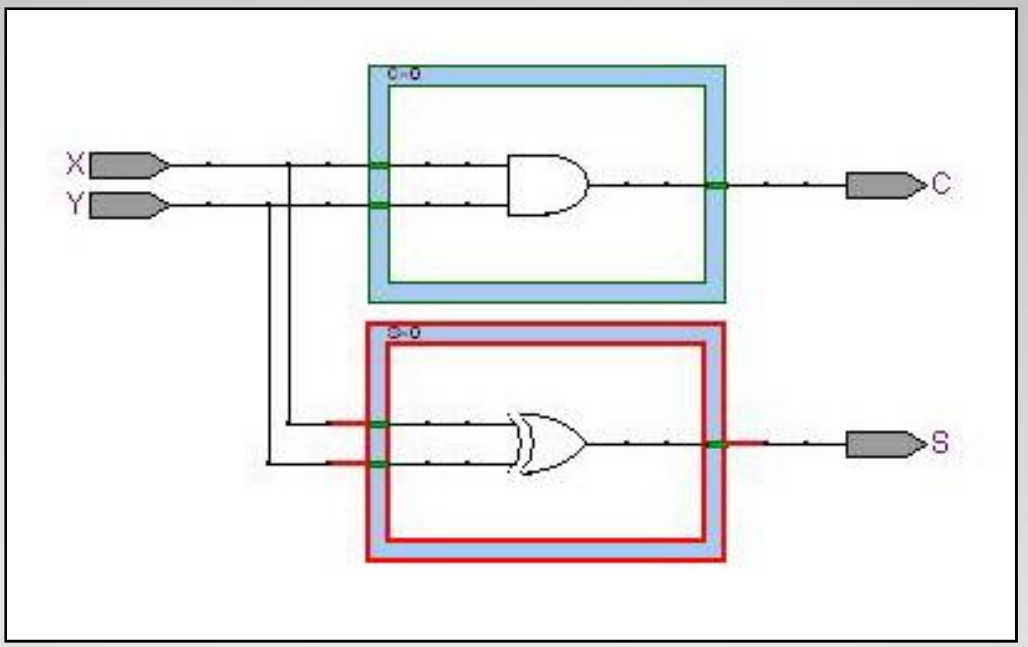
(c) Circuit



Circuit Symbol

```
module HalfAdd( input X, Y, output S, C );

  assign S = X ^ Y;
  assign C = X & Y;

endmodule
```



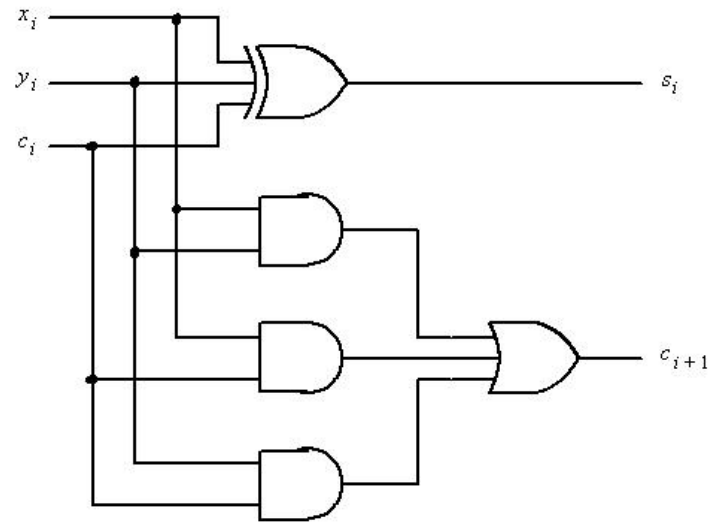


Half Adder RTL and Tech Map

Full Adder

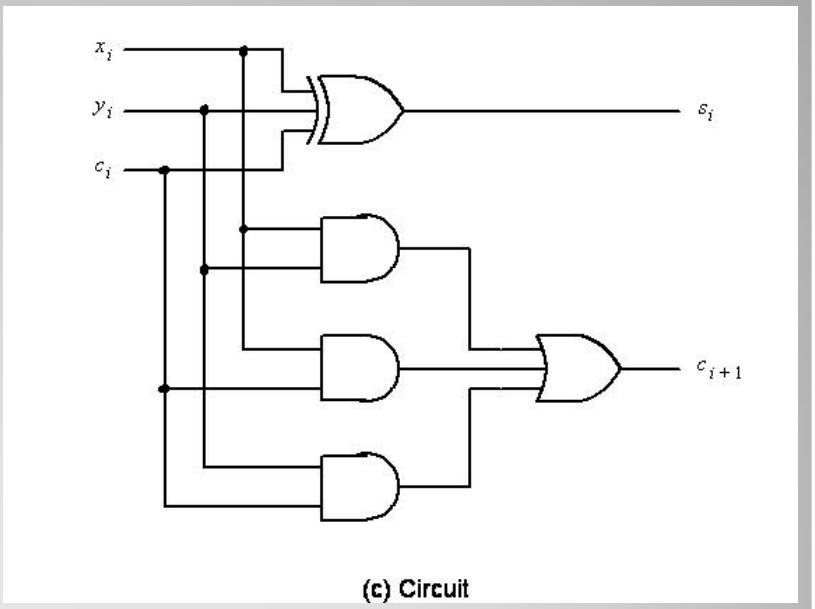
c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table



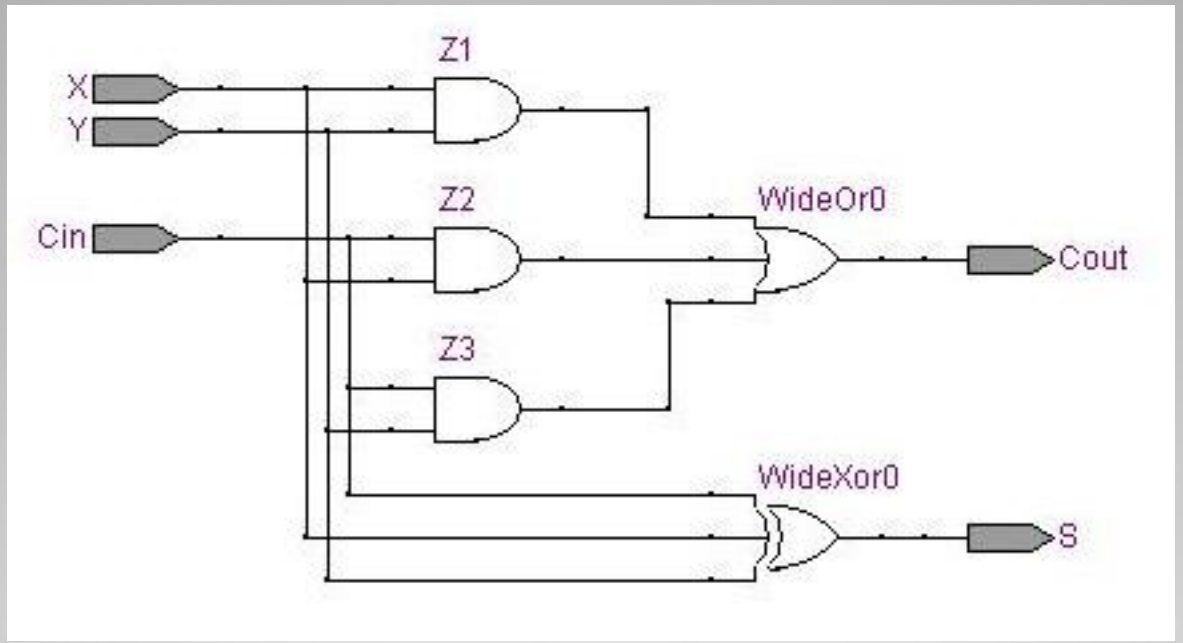
(c) Circuit

```
module FullAdd (Cin, X, Y, S, Cout);
  input Cin, X, Y;
  output S, Cout;

  wire Z1, Z2, Z3;

  xor ( S, X, Y, Cin );
  and ( Z1, X, Y );
  and ( Z2, X, Cin );
  and ( Z3, Y, Cin );
  or  ( Cout, Z1, Z2, Z3 );

endmodule
```

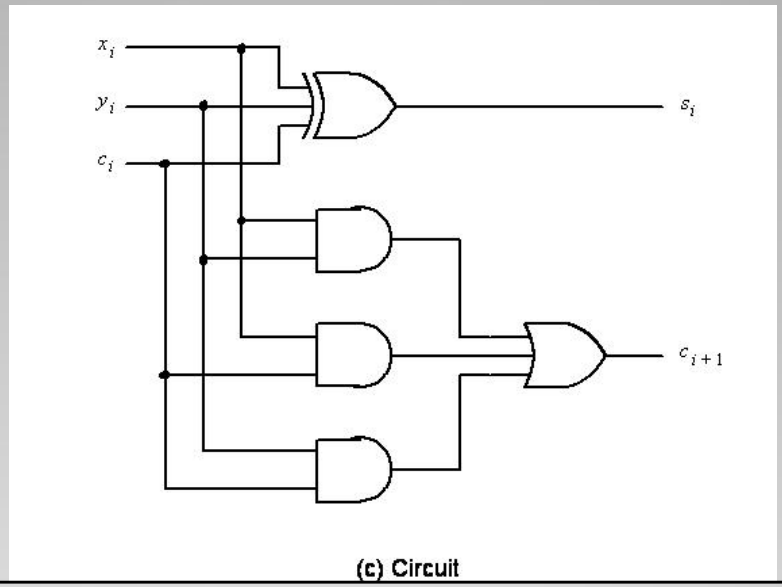


(c) Circuit

Full Adder in Structural Verilog



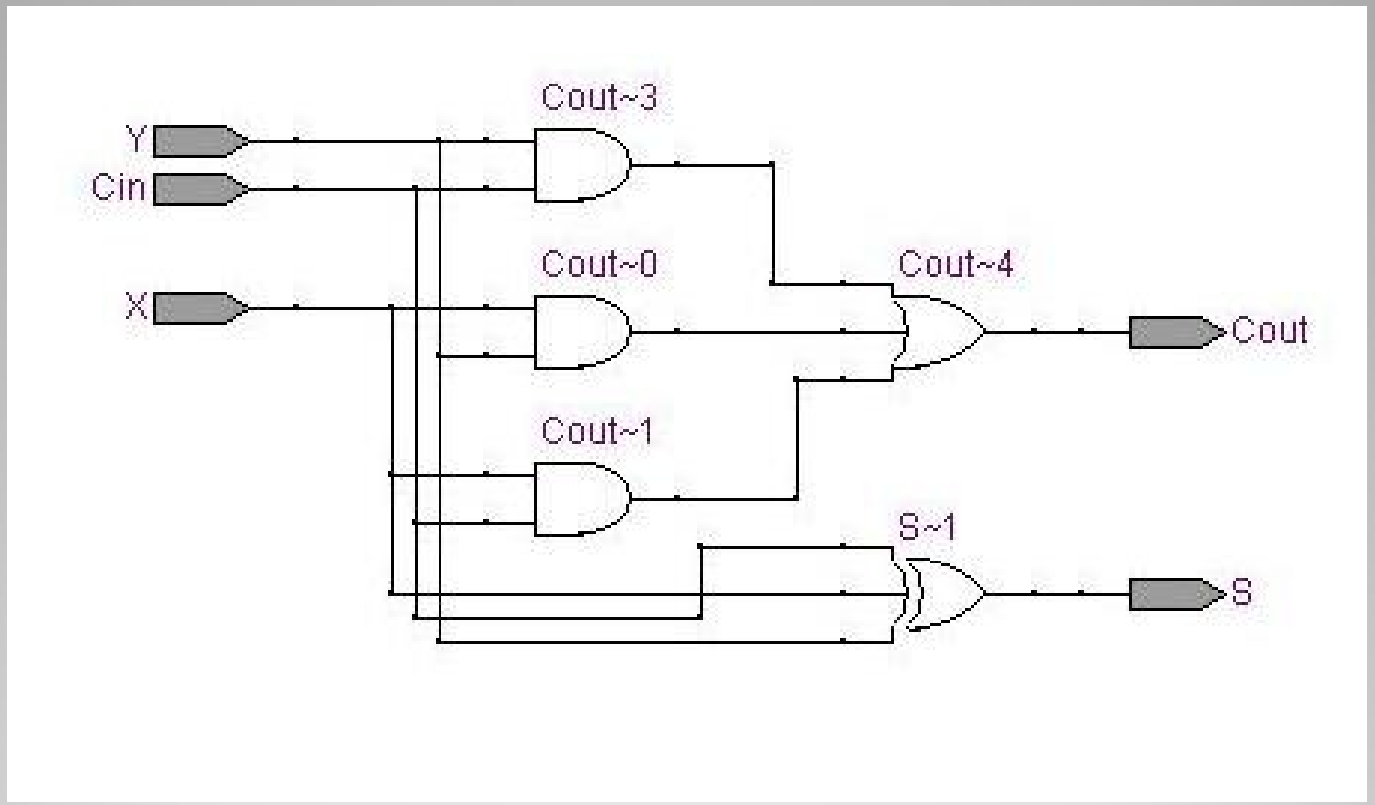
Full Adder From Netlist Viewer



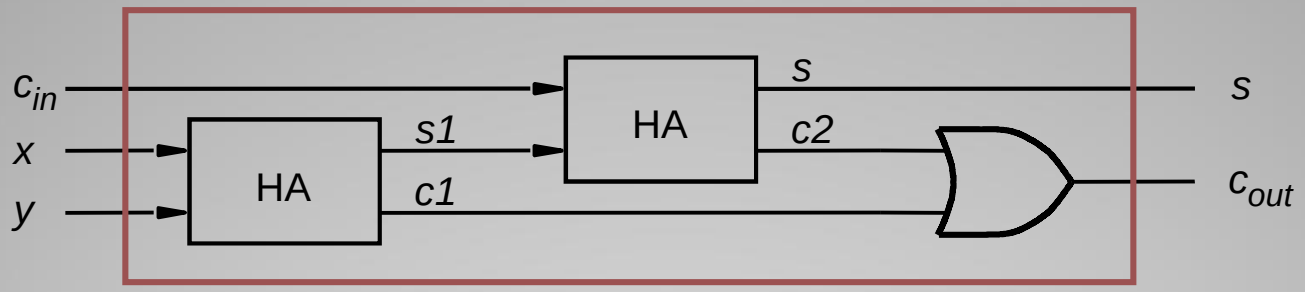
(c) Circuit

```
module FullAdd( Cin, X, Y, S, Cout );
  input Cin, X, Y;
  output S, Cout;

  assign S = X ^ Y ^ Cin;
  assign Cout = (X & Y) | (X & Cin) | (Y & Cin);

endmodule
```

One Behavioral Version

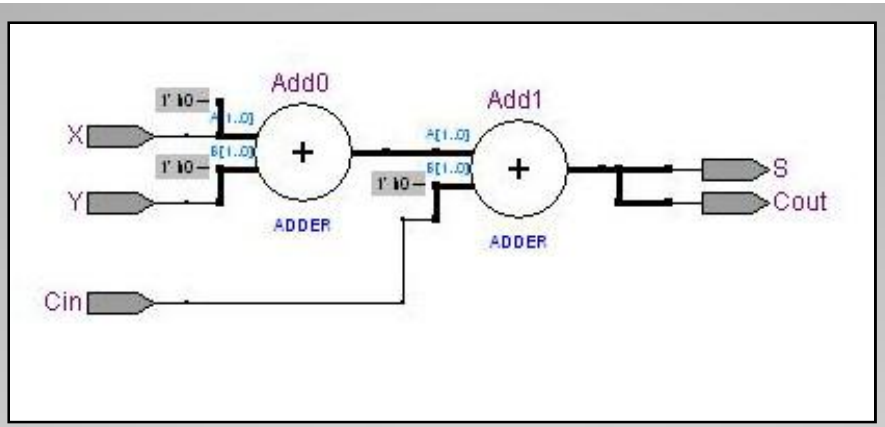


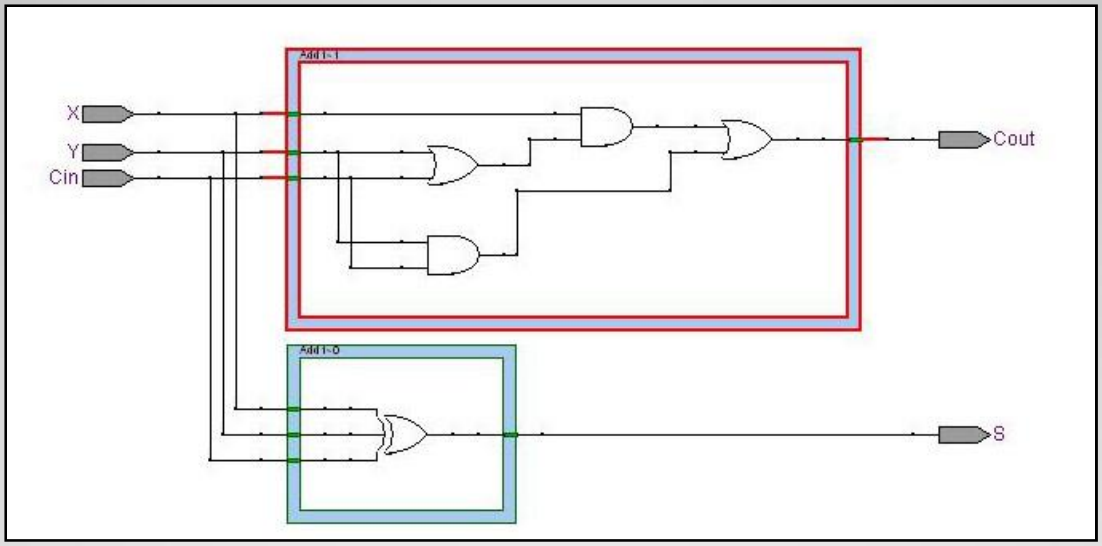
Full Adder From Netlist Viewer




```
// module FullAdd( Cin, X, Y, S, Cout );
module FullAdd( input Cin, X, Y, output S, Cout );
   output S, Cout;
   assign S = X ^ Y;
   assign C1 = X & Y;

endmodule
   HalfAdd H1( X, Y, S1, C1 );
   HalfAdd H2( Cin, S1, S, C2 );
   assign Cout = C1 | C2;

endmodule
```

Full Adder From Half Adders



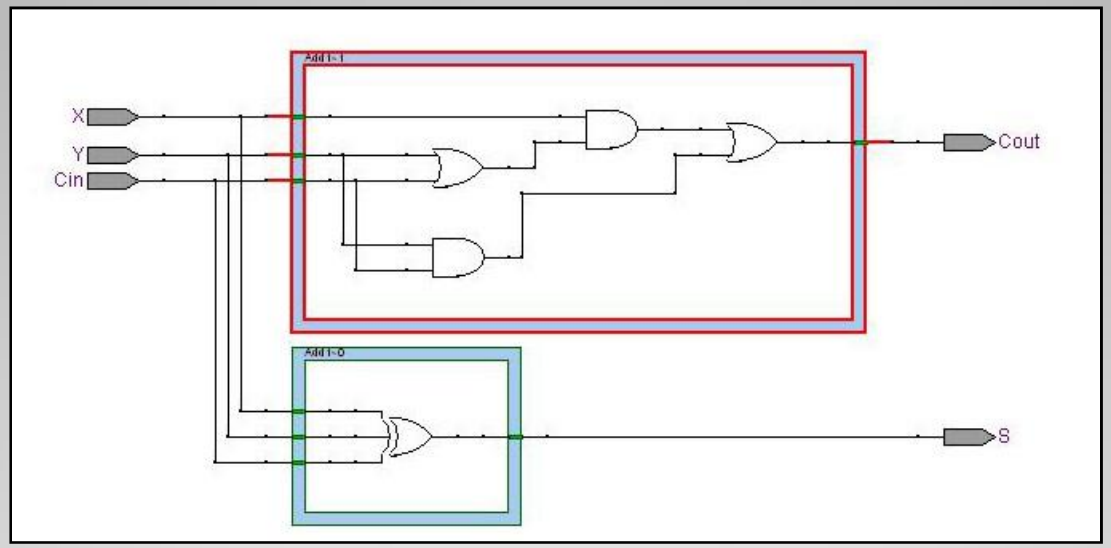


Full Adder From Netlist Viewer

```
// TCES 330
// 4/7/2010
// R. Gutmann
// This circuit implements a full adder
// using  vector concatenation

module FullAdd( Cin, X, Y, S, Cout );
  input Cin, X, Y;
  output S, Cout;

  assign {Cout, S} = X + Y + Cin;

endmodule
```

Full Adder With Vector Notation

- Looks the same as the last Netlist view.



Full Adder From Netlist Viewer

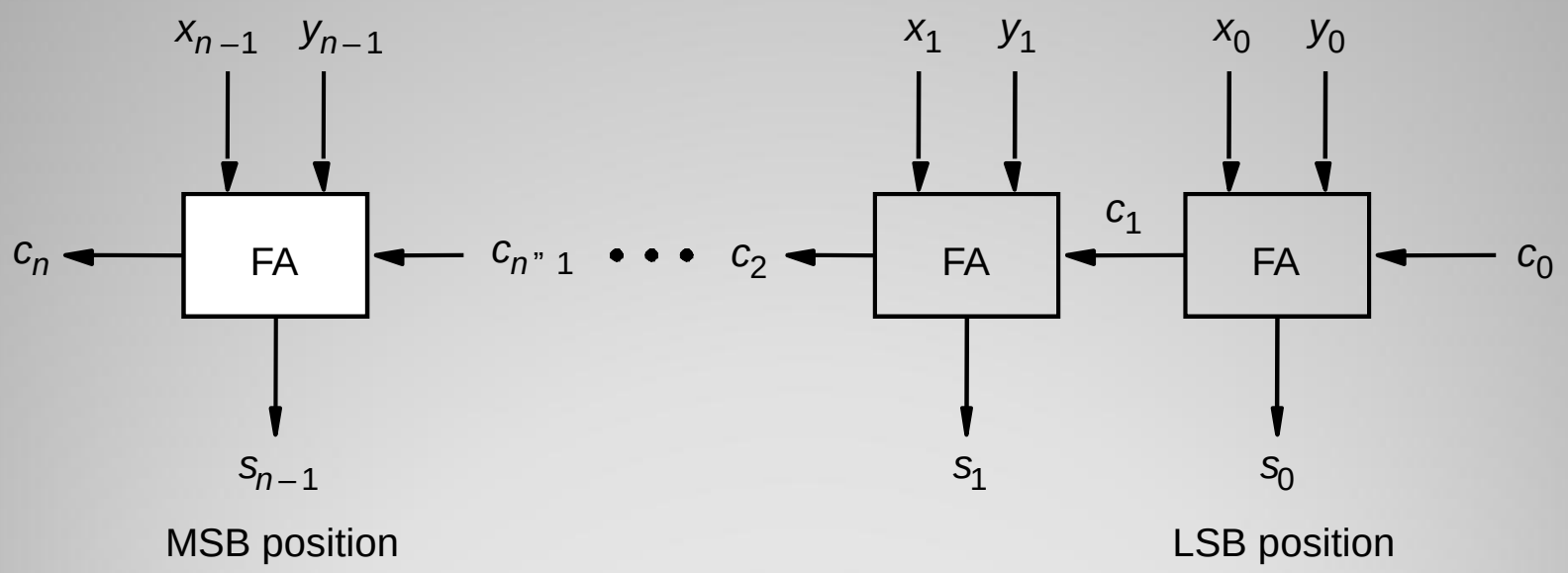


Figure 5.6. An *n*-bit ripple-carry adder.

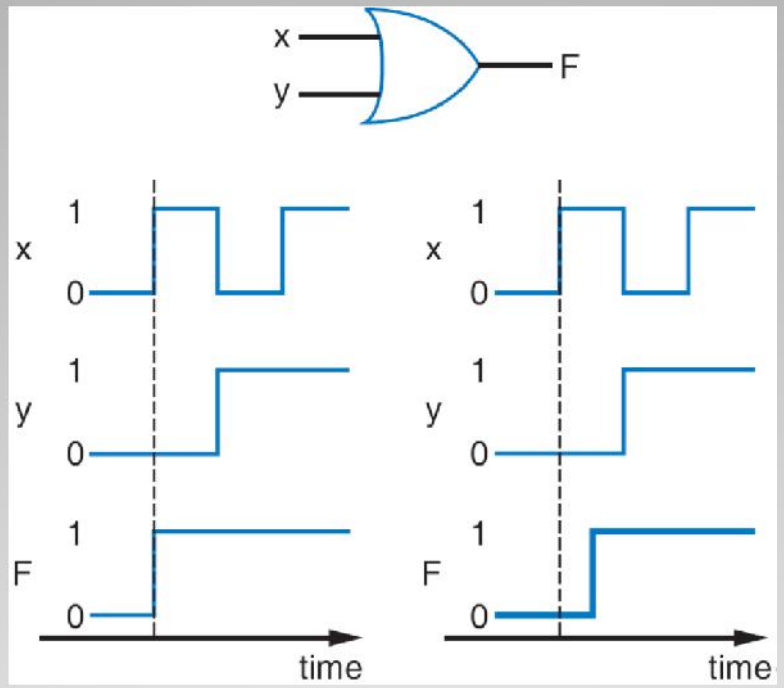
Ripple-Carry Adder

```
module Adder4( Carryin, X3, X2, X1, X0, Y3, Y2, Y1, Y0,
               S3, S2, S1, S0, Carryout );
       input Carryin, X3, X2, X1, X0, Y3, Y2, Y1, Y0;
       output S3, S2, S1, S0, Carryout;
       wire C3,C2,C1;

       FullAdd Stage0 ( Carryin, X0, Y0, S0, C1 );
       FullAdd Stage1 ( C1, X1, Y1, S1, C2 );
       FullAdd Stage2 ( C2, X2, Y2, S2, C3 );
       FullAdd Stage3 ( C3, X3, Y3, S3, Carryout );

endmodule

module FullAdd ( Cin, X, Y, S, Cout );
       input Cin, X, Y;
       output S, Cout;
       assign S = X ^ Y ^ Cin,
       assign Cout = (X & Y) | (X & Cin) | (Y & Cin);

endmodule
```

Four-bit Ripple-Carry Adder

```
module Adder4( Carryin, X, Y, S, Carryout );
  input  Carryin;
  input  [3:0] X, Y;
  output [3:0] S;
  output Carryout;
  wire   [3:1] C;

  FullAdd Stage0 (Carryin, X[0], Y[0], S[0], C[1]);
  FullAdd Stage1 (C[1], X[1], Y[1], S[1], C[2]);
  FullAdd Stage2 (C[2], X[2], Y[2], S[2], C[3]);
  FullAdd Stage3 (C[3], X[3], Y[3], S[3], Carryout);

endmodule
```

Same Adder Using Vectors



- Real gates have some delay
  - Outputs don't change immediately after inputs change

Additional Considerations

Non-Ideal Gate Behavior -- Delay

```
module Adder4( Carryin, X, Y, S, Carryout );

  input  Carryin;
  input  [3:0] X, Y;  // addends

  output [3:0] S;     // sum
  output Carryout;

  assign {Carryout, S} = X + Y + Carryin;

endmodule
```

How We Would Really Do It

```
module AdderN( Carryin, X, Y, S, Carryout );
  parameter N = 8;
  input  Carryin;
  input  [N-1:0] X, Y; // addends

  output [N-1:0] S;    // sum
  output Carryout;

  assign {Carryout, S} = X + Y + Carryin;

endmodule
```

How We Would Really Really Do It

```
// TCES 330
// 04/07/2010
// R. Gutmann
// This circuit adds 5 to X (eight bits)
// and outputs the sum S (eight bits) plus overflow flag

module Add5To( X, S, Overflow );
  input  [7:0] X;  // eight bits of input
  output [7:0] S;  // eight bits of output
  output Overflow;

  AdderN #(.N(8)) U1( 1'b0, X, 8'h5, S, Overflow );

endmodule
```

Example Using Constants

```
module BCD7seg( C, Display );
  input      [3:0] C;        // input code
  output reg [0:6] Display;  // seven-segment display output
  // Note: a '0' turns the segment ON
  // Note: Segment 6 is in the right most bit, below
  always @ ( C )
    if ( C == 4'h0 )
      Display = 7'b0000001;
    else if ( C == 4'h1 )
      Display = 7'b …
    else if ( C == 4'h2 )
      Display = 7'b …
    else if ( C == 4'h3 )
      Display = 7'b …
    else if ( C == 4'h4 )
      Display = 7'b …
    else if ( C == 4'h5 )
      Display = 7'b …
    else if ( C == 4'h6 )
      Display = 7'b …
    else if ( C == 4'h7 )
      Display = 7'b …
    else if ( C == 4'h8 )
      Display = 7'b …
    else if ( C == 4'h9 )
      Display = 7'b …
    else
      Display = 7'b1111111;  // blank
endmodule
```

Constants + Procedure + if

```
module BCD7seg( C, Display );
  input      [3:0] C;        // input code
  output reg [0:6] Display;  // seven-segment display output
  // Note: a '0' turns the segment ON
  // Note: Segment 6 is in the right most bit, below
  always @ ( C )
    case ( C )
      4'h0: Display = 7'b0000001;
      4’h1: Display = 7’b …
      4’h2: Display = 7’b …
      4’h3: Display = 7’b …
      4’h4: Display = 7’b …
      4’h5: Display = 7’b …
      4’h6: Display = 7’b …
      4’h7: Display = 7’b …
      4’h8: Display = 7’b …
      4’h9: Display = 7’b …
      default: Display = 7'b1111111;  // blank
  endcase

endmodule
```

Constants + Procedure + case

Question:

How do we know it works?

Verification

- All of this design in Verilog is well and good, but how do we know if we got it right?
- We need to test or verify the design.
- Traditional Verilog testing techniques are not supported by Quartus II (its purpose is to synthesize circuits).
- However, we can test using ModelSim.