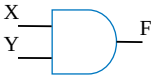
Verilog for Digital Design

Chapter 2:
Combinational Logic Design

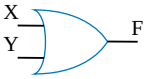
AND/OR/NOT Gates

AND/OR/NOT Gates

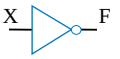
Verilog Modules and Ports




```
module And2(X, Y, F);

   input X, Y;
   output F;
   ...
```




```
module Or2(X, Y, F);

   input X, Y;
   output F;
   ...
```

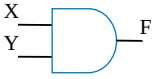



```
module Inv(X, F);

   input X;
   output F;
   ...
```

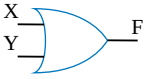
- module – Declares a new type of component
  - Named “And2" in first example above
  - Includes list of ports (module's inputs and outputs)
- input – List indicating which ports are inputs
- output – List indicating which ports are outputs
- Each port is a bit – can have value of *0*, *1*, or *x* (unknown value)
- *Note*: Verilog already has built-in primitives for logic gates, but instructive to build them

AND/OR/NOT Gates

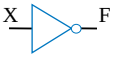
Modules and Ports




```
module And2(X, Y, F);

   input X, Y;
   output F;
   ...
```




```
module Or2(X, Y, F);

   input X, Y;
   output F;
   ...
```



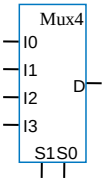

```
module Inv(X, F);

   input X;
   output F;
   ...
```

- Verilog has several dozen keywords
  - User cannot use keywords when naming items like modules or ports
  - *module*, *input*, and *output* are keywords above
  - Keywords must be **lower case**, not UPPER CASE or a MixTure thereof
- User-defined names – Identifiers
  - Begin with letter or underscore (_), optionally followed by any sequence of letters, digits, underscores, and dollar signs ($)
  - Valid identifiers: *A*, *X*, *Hello*, *JXYZ*, *B14*, *Sig432*, *Wire_23*, *_F1*, *F$2*, *_Go_$_$*, *_*, *Input*
    - *Note: "_" and "Input" are valid, but unwise*
  - Invalid identifiers: *input* (keyword), *$ab* (doesn't start with letter or underscore), *2A* (doesn't start with letter or underscore)
- *Note*: Verilog is ***case sensitive***. Sig432 differs from SIG432 and sig432
  - We'll initially capitalize identifiers (e.g., Sig432) to distinguish from keywords

AND/OR/NOT Gates

Modules and Ports

- Q: Begin a module definition for a 4x1 multiplexor
  - Inputs: I3, I2, I1, I0, S1, S0. Outputs: D



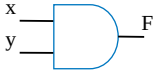
4x1 mux

```
module Mux4(I3, I2, I1, I0, S1, S0, D);

   input I3, I2, I1, I0;
   input S1, S0;
   output D;
   ...
```

Note that input ports above are separated into two declarations for clarity

AND/OR/NOT Gates

Module Procedures—always




```
module And2(X, Y, F);

   input X, Y;
   output F;
   reg F;

   always @(X, Y) begin
       F <= X & Y;
   end

endmodule
```

wait until X or Y changes

F <= x AND y

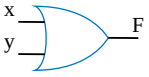
vldd_ch2_And2.v

- One way to describe a module's behavior uses an "always" procedure
  - always – Procedure that executes repetitively (infinite loop) from simulation start
  - @ – event control indicating that statements should only execute when values change
    - "(X,Y)" – execute if X changes or Y changes (change known as an event)
    - Sometimes called "*sensitivity list*"
    - We'll say that procedure is "*sensitive to X and Y*"
  - "F <= X & Y;" – Procedural statement that sets F to AND of X, Y
    - & is built-in bit AND operator
    - <= assigns value to variable
  - reg – Declares a variable data type, which holds its value between assignments
    - Needed for F to hold value between assignments
    - *Note*: "reg", short for "register", is an unfortunate name. A reg variable may or may not correspond to an actual physical register. There obviously is no register inside an AND gate.

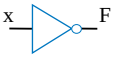
AND/OR/NOT Gates

Module Procedures—always

- Q: Given that "|" and "~" are built-in operators for OR and NOT, complete the modules for a 2-input OR gate and a NOT gate




```
module Or2(X, Y, F);

   input X, Y;
   output F;
   reg F;

   always @(X, Y) begin
       F <= X | Y;
   end

endmodule
```




```
module Inv(X, F);

   input X;
   output F;
   reg F;

   always @(X) begin
       F <= ~X;
   end

endmodule
```

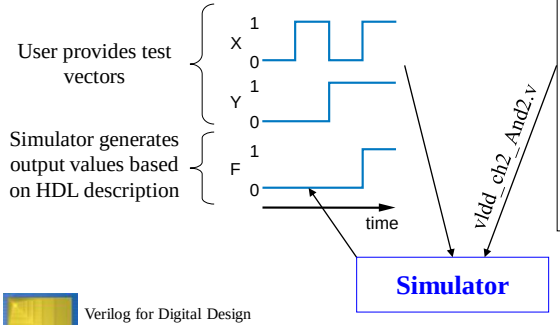
AND/OR/NOT Gates

Simulation and Testbenches — A First Look

- How does our new module behave?
- Simulation
  - User provides input values, simulator generates output values
    - Test vectors – sequence of input values
    - Waveform – graphical depiction of sequence

Timescale directive is for simulation. More later.

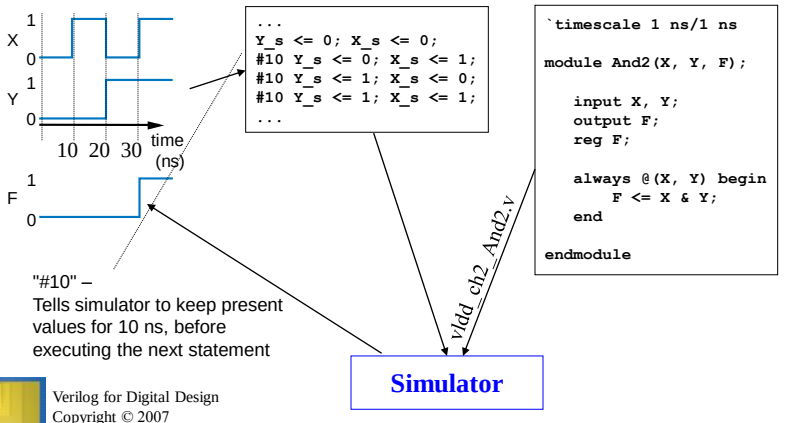
```
`timescale 1 ns/1 ns

module And2(X, Y, F);

   input X, Y;
   output F;
   reg F;

   always @(X, Y) begin
       F <= X & Y;
   end

endmodule
```



AND/OR/NOT Gates

Simulation and Testbenches — A First Look

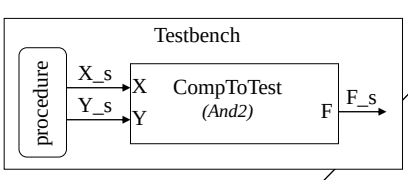
- Instead of drawing test vectors, user can describe them with HDL



AND/OR/NOT Gates

Simulation and Testbenches

Idea: Create new "Testbench" module that provides test vectors to component's inputs




```
`timescale 1 ns/1 ns
module Testbench();

  reg X_s, Y_s;
  wire F_s;

  And2 CompToTest(X_s, Y_s, F_s);

  initial begin
     // Test all possible input combinations
     Y_s <= 0; X_s <= 0;
     #10 Y_s <= 0; X_s <= 1;
     #10 Y_s <= 1; X_s <= 0;
     #10 Y_s <= 1; X_s <= 1;
  end

endmodule
```

- HDL testbench
  - Module with no ports
  - Declare reg variable for each input port, wire for each output port
  - Instantiate module, map variables to ports (more in next section)
  - Set variable values at desired times

More information on next slides

Note: CompToTest short for Component To Test

AND/OR/NOT Gates
Simulation and Testbenches

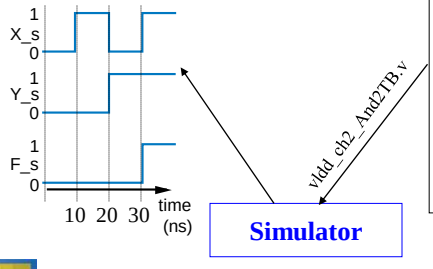
- wire – Declares a net data type, which does not store its value
  - Vs. reg data type that stores value
  - Nets used for connections
  - Net's value determined by what it is connected to
- initial –procedure that executes at simulation start, but *executes only once*
  - Vs. "always" procedure that also executes at simulation start, but that *repeats*
- # – Delay control – number of time units to delay this statement's execution relative to previous statement
  - `timescale – compiler directive telling compiler that from this point forward, 1 time unit means 1 ns
    - Valid time units – s (seconds), ms (milliseconds), us (microseconds), ns (nanoseconds), ps (picoseconds), and fs (femtoseconds)
    - 1 ns/1 ns – time unit / time precision. Precision is for internal rounding. For our purposes, precision will be set same as time unit.

```
`timescale 1 ns/1 ns

module Testbench();

  reg X_s, Y_s;
  wire F_s;

  And2 CompToTest(X_s, Y_s, F_s);

  initial begin
     // Test all possible input combinations
     Y_s <= 0; X_s <= 0;
     #10 Y_s <= 0; X_s <= 1;
     #10 Y_s <= 1; X_s <= 0;
     #10 Y_s <= 1; X_s <= 1;
  end

endmodule
```

Note: We appended "_s" to reg/wire identifiers to distinguish them from ports, though not strictly necessary

AND/OR/NOT Gates
Simulation and Testbenches

- Provide testbench file to simulator
  - Simulator generates waveforms
  - We can then check if behavior looks correct



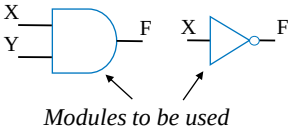

```
`timescale 1 ns/1 ns

module Testbench();

   reg X_s, Y_s;
   wire F_s;

   And2 CompToTest(X_s, Y_s, F_s);

   initial begin
      // Test all possible input combinations
      Y_s <= 0; X_s <= 0;
      #10 Y_s <= 0; X_s <= 1;
      #10 Y_s <= 1; X_s <= 0;
      #10 Y_s <= 1; X_s <= 1;
   end

endmodule
```

Combinational Circuits

Combinational Circuits

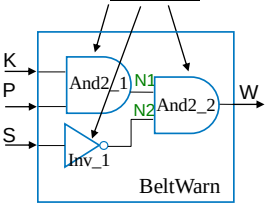
Component Instantiations

- Circuit – A connection of modules
  - Also known as structure
  - A circuit is a second way to describe a module
    - vs. using an always procedure, as earlier
- Instance – An occurrence of a module in a circuit
    - May be multiple instances of a module
    - e.g., Car's modules: tires, engine, windows, etc., with 4 tire instances, 1 engine instance, 6 window instances, etc.



Modules to be used

Module instances

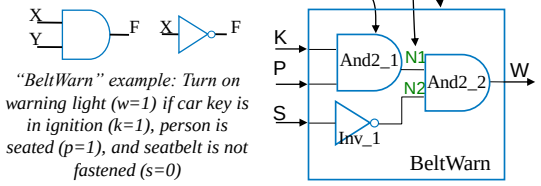


Combinational Circuits

Module Instantiations

- Creating a circuit
  1. Start definition of a new module
  2. Declare nets for connecting module instances
     - N1, N2
     - Note: W is also a declared as a net. By defaults outputs are considered wire nets unless explicitly declared as a reg variable
  3. Create module instances, create connections

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;

   wire N1, N2;

   And2 And2_1(K, P, N1);
   Inv  Inv_1(S, N2);
   And2 And2_2(N1, N2, W);

endmodule
```



"BeltWarn" example: Turn on warning light (w=1) if car key is in ignition (k=1), person is seated (p=1), and seatbelt is not fastened (s=0)

vldd_ch2_BeltWarnStruct.v

Combinational Circuits

Module Instantiations

- Module instantiation statement

```
And2 And2_1(K, P, N1);
```

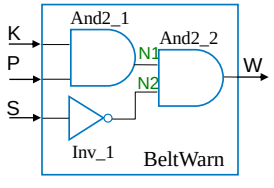
Note: Ports ordered as in original And2 module definition

Connects instantiated module's ports to nets and variables

Name of new module instance
Must be distinct; hence And2_***1*** and And2_***2***

Name of module to instantiate

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;

   wire N1, N2;

   And2 And2_1(K, P, N1);
   Inv  Inv_1(S, N2);
   And2 And2_2(N1, N2, W);

endmodule
```



vldd_ch2_BeltWarnStruct.v

Combinational Circuits
Module Instantiations

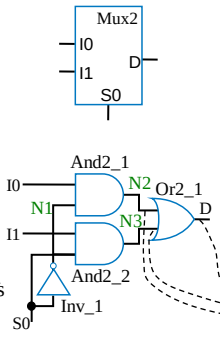
- Q: Complete the 2x1 mux circuit's module instantiations

1. Start definition of a new module (done)

(Draw desired circuit, if not already done)

2. Declare nets for internal wires

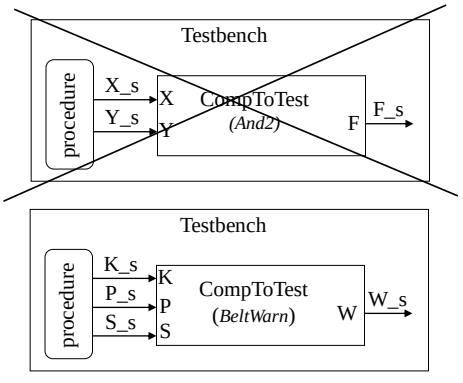
3. Create module instances and connect ports



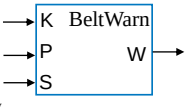

```
`timescale 1 ns/1 ns

module Mux2(I1, I0, S0, D);

   input I1, I0;
   input S0;
   output D;

   wire N1, N2, N3;

   Inv  Inv_1  (S0, N1);
   And2 And2_1 (I0, N1, N2);
   And2 And2_2 (I1, S0, N3);
   Or2  Or2_1  (N2, N3, D);

endmodule
```


Combinational Circuit Structure
Simulating the Circuit

- Same testbench format for BeltWarn module as for earlier And2 module



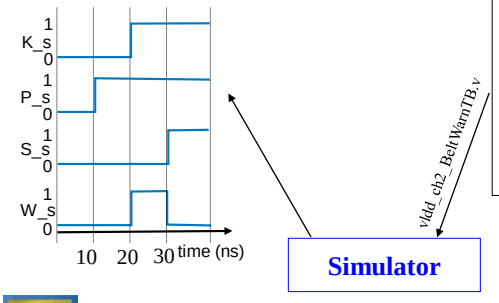

```
`timescale 1 ns/1 ns

module Testbench();

   reg K_s, P_s, S_s;
   wire W_s;

   BeltWarn CompToTest(K_s, P_s, S_s, W_s);

   initial begin
      K_s <= 0; P_s <= 0; S_s <= 0;
      #10 K_s <= 0; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 1;
   end

endmodule
```



Combinational Circuit Structure

Simulating the Circuit

- Simulate testbench file to obtain waveforms




```
`timescale 1 ns/1 ns

module Testbench();

   reg K_s, P_s, S_s;
   wire W_s;

   BeltWarn CompToTest(K_s, P_s, S_s, W_s);

   initial begin
      K_s <= 0; P_s <= 0; S_s <= 0;
      #10 K_s <= 0; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 1;
   end

endmodule
```


Combinational Circuit Structure

Simulating the Circuit

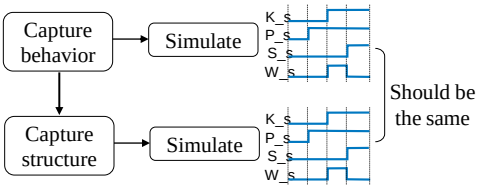
- More on testbenches
  - Note that a single module instantiation statement used
  - reg and wire declarations (K_s, P_s, S_s, W_s) used because procedure cannot access instantiated module's ports directly
    - Inputs declared as regs so can assign values (which are held between assignments)
  - Note module instantiation statement and procedure can both appear in one module

```
`timescale 1 ns/1 ns

module Testbench();

   reg K_s, P_s, S_s;
   wire W_s;

   BeltWarn CompToTest(K_s, P_s, S_s, W_s);

   initial begin
      K_s <= 0; P_s <= 0; S_s <= 0;
      #10 K_s <= 0; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 0;
      #10 K_s <= 1; P_s <= 1; S_s <= 1;
   end

endmodule
```

Top-Down Design – Combinational Behavior to Structure

Top-Down Design – Combinational Behavior to Structure

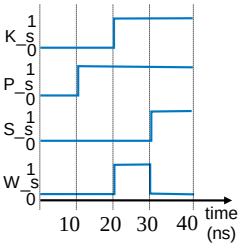
- Designer may initially know system behavior, but not structure
  - BeltWarn: W = KPS'
- Top-down design
  - *Capture behavior*, and simulate
  - *Capture structure (circuit)*, simulate again
  - Gets behavior right first, unfettered by complexity of creating structure



Top-Down Design – Combinational Behavior to Structure
Always Procedures with Assignment Statements

- How describe behavior? One way: Use an always procedure
  - Sensitive to K, P, and S
    - Procedure executes only if change occurs on any of those inputs
  - Simplest procedure uses one assignment statement
- Simulate using testbench (same as shown earlier) to get waveforms
- Top-down design
  - Proceed to capture structure, simulate again using same testbench – result should be the same waveforms

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;
   reg W;

   always @(K, P, S) begin
      W <= K & P & ~S;
   end
endmodule
```



Top-Down Design – Combinational Behavior to Structure
Procedures with Assignment Statements

- Procedural assignment statement
  - Assigns value to variable
  - Right side may be expression of operators
    - Built-in bit operators include
      - & → AND | → OR ~ → NOT
      - ^ → XOR ~^ → XNOR
  - Q: Create an always procedure to compute:
    - F = C'H + CH'

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;
   reg W;

   always @(K, P, S) begin
      W <= K & P & ~S;
   end
endmodule
```

vldd_ch2_BeltWarnBeh.v

Answer 1:

```
always @(C,H) begin
   F <= (~C&H) | (C&~H);
end
```

Answer 2:

```
always @(C,H)
begin
   F <= C ^ H;
end
```

Top-Down Design – Combinational Behavior to Structure
Procedures with Assignment Statements

- Procedure may have multiple assignment statements

```
`timescale 1 ns/1 ns

module TwoOutputEx(A, B, C, F, G);

   input A, B, C;
   output F, G;
   reg F, G;

   always @(A, B, C) begin
      F <= (B & B) | ~C;
      G <= (A & B) | (B & C);
   end
endmodule
```

vldd_ch2_TwoOutputBeh.v

Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- Process may use if-else statements (a.k.a. conditional statements)
  - if (*expression*)
    - If *expression* is true (evaluates to nonzero value), execute corresponding statement(s)
    - If false (evaluates to 0), execute else's statement (else part is optional)
    - Example shows use of operator == → logical equality, returns true/false (actually, returns 1 or 0)
    - True is nonzero value, false is zero

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;
   reg W;

   always @(K, P, S) begin
      if ((K & P & ~S) == 1)
         W <= 1;
      else
         W <= 0;
   end
endmodule
```

vldd_ch2_BeltWarnBehIf.v

Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- More than two possibilities
  - Handled by stringing if-else statements together
    - Known as if-else-if construct
- Example: 4x1 mux behavior
  - Suppose S1S0 change to 01
    - if's expression is false
    - else's statement executes, which is an if statement whose expression is true

Note: The following indentation shows if statement nesting, but is unconventional:

```
if (S1==0 && S0==0)
   D <= I0;
else
   if (S1==0 && S0==1)
      D <= I1;
   else
      if (S1==1 && S0==0)
         D <= I2;
      else
         D <= I3;
```

```
`timescale 1 ns/1 ns

module Mux4(I3, I2, I1, I0, S1, S0, D);

   input I3, I2, I1, I0;
   input S1, S0;
   output D;
   reg D;

   always @(I3, I2, I1, I0, S1, S0)
   begin
      if (S1==0 && S0==0)
         D <= I0;
      else if (S1==0 && S0==1)
         D <= I1;
      else if (S1==1 && S0==0)
         D <= I2;
      else
         D <= I3;
   end
endmodule
```

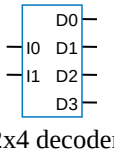
Suppose S1S0 change to 01

&& → logical AND

& : bit AND (operands are bits, returns bit)
&& : logical AND (operands are true/false values, returns true/false)

Top-Down Design – Combinational Behavior to Structure
Procedures with If-Else Statements

- Q: Create procedure describing behavior of a 2x4 decoder using if-else-if construct



2x4 decoder

Order of assignment statements does not matter.

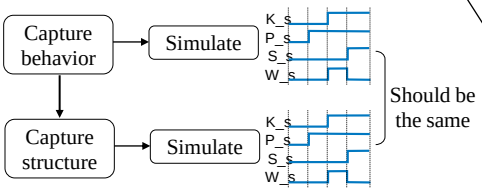
Placing two statements on one line does not matter.

To execute multiple statements if expression is true, enclose them between "begin" and "end"

```
`timescale 1 ns/1 ns

module Dcd2x4(I1, I0, D3, D2, D1, D0);

   input I1, I0;
   output D3, D2, D1, D0;
   reg D3, D2, D1, D0;

   always @(I1, I0)
   begin
      if (I1==0 && I0==0)
      begin
         D3 <= 0; D2 <= 0;
         D1 <= 0; D0 <= 1;
      end
      else if (I1==0 && I0==1)
      begin
         D3 <= 0; D2 <= 0;
         D1 <= 1; D0 <= 0;
      end
      else if (I1==1 && I0==0)
      begin
         D3 <= 0; D2 <= 1;
         D1 <= 0; D0 <= 0;
      end
      else
      begin
         D3 <= 1; D2 <= 0;
         D1 <= 0; D0 <= 0;
      end
   end
endmodule
```

Top-Down Design – Combinational Behavior to Structure

- Top-down design
  - Capture behavior, and simulate
  - Capture structure using a second module, and simulate




```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;
   reg W;

   always @(K, P, S) begin
      W <= K & P & ~S;
   end
endmodule
```

vldd_ch2_BeltWarnBeh.v

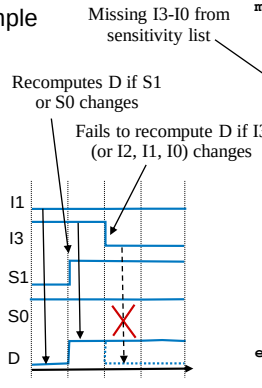
```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;

   wire N1, N2;

   And2 And2_1(K, P, N1);
   Inv  Inv_1(S, N2);
   And2 And2_2(N1, N2, W);

endmodule
```

vldd_ch2_BeltWarnStruct.v

Top-Down Design – Combinational Behavior to Structure
Common Pitfall – *Missing Inputs from Event Control Expression*

- Pitfall – Missing inputs from event control's sensitivity list when describing combinational behavior
  - Results in sequential behavior
  - Wrong 4x1 mux example
    - Has memory
    - No compiler error
      - Just not a mux

Reminder
- *Combinational behavior:* Output value is purely a function of the present input values
- *Sequential behavior:* Output value is a function of present *and past* input values, i.e., the system has memory




```
`timescale 1 ns/1 ns

module Mux4(I3, I2, I1, I0, S1, S0, D);

   input I3, I2, I1, I0;
   input S1, S0;
   output D;
   reg D;

   always @(S1, S0)
   begin
      if (S1==0 && S0==0)
         D <= I0;
      else if (S1==0 && S0==1)
         D <= I1;
      else if (S1==1 && S0==0)
         D <= I2;
      else
         D <= I3;
   end
endmodule
```

vldd_ch2_Mux4Wrong.v

Top-Down Design – Combinational Behavior to Structure

Common Pitfall – *Missing Inputs from Event Control Expression*

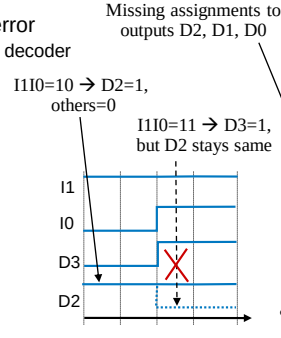
- Verilog provides mechanism to help avoid this pitfall
  - @* – implicit event control expression
    - Automatically adds all nets and variables that are read by the controlled statement or statement group
    - Thus, @* in example is equivalent to @(S1,S0,I0,I1,I2,I3)
    - @(*) also equivalent

```
`timescale 1 ns/1 ns

module Mux4(I3, I2, I1, I0, S1, S0, D);

   input I3, I2, I1, I0;
   input S1, S0;
   output D;
   reg D;

   always @*
   begin
      if (S1==0 && S0==0)
         D <= I0;
      else if (S1==0 && S0==1)
         D <= I1;
      else if (S1==1 && S0==0)
         D <= I2;
      else
         D <= I3;
   end
endmodule
```

Top-Down Design – Combinational Behavior to Structure

Common Pitfall – *Output not Assigned on Every Pass*

- Pitfall – Failing to assign every output on every pass through the procedure for combinational behavior
  - Results in sequential behavior
    - Referred to as inferred latch (more later)
  - Wrong 2x4 decoder example
    - Has memory
    - No compiler error
      - Just not a decoder




```
`timescale 1 ns/1 ns

module Dcd2x4(I1, I0, D3, D2, D1, D0);

   input I1, I0;
   output D3, D2, D1, D0;
   reg D3, D2, D1, D0;

   always @(I1, I0)
   begin
      if (I1==0 && I0==0)
      begin
         D3 <= 0; D2 <= 0;
         D1 <= 0; D0 <= 1;
      end
      else if (I1==0 && I0==1)
      begin
         D3 <= 0; D2 <= 0;
         D1 <= 1; D0 <= 0;
      end
      else if (I1==1 && I0==0)
      begin
         D3 <= 0; D2 <= 1;
         D1 <= 0; D0 <= 0;
      end
      else if (I1==1 && I0==1)
      begin
         D3 <= 1;
      end
      // Note: missing assignments
      // to every output in last "else if"
   end
endmodule
```

vldd_ch2_Dcd2x4Wrong.v

Top-Down Design – Combinational Behavior to Structure

Common Pitfall – *Output not Assigned on Every Pass*

- Same pitfall often occurs due to not considering all possible input combinations

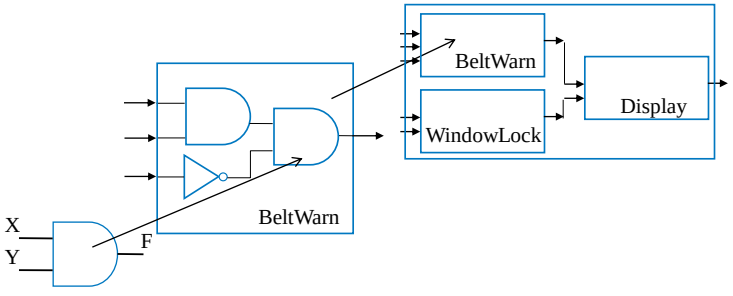
```
if (I1==0 && I0==0)
begin
   D3 <= 0; D2 <= 0;
   D1 <= 0; D0 <= 1;
end
else if (I1==0 && I0==1)
begin
   D3 <= 0; D2 <= 0;
   D1 <= 1; D0 <= 0;
end
else if (I1==1 && I0==0)
begin
   D3 <= 0; D2 <= 1;
   D1 <= 0; D0 <= 0;
end
```

Last "else" missing, so not all input combinations are covered (i.e., I1I0=11 not covered)

Hierarchical Circuits

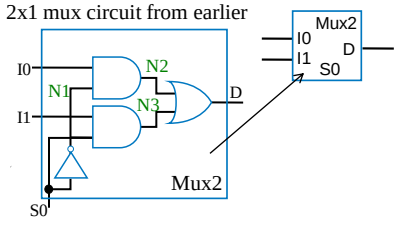
Hierarchical Circuits
Using Modules Instances in Another Module

- Module can be used as instance in a new module
  - As seen earlier: And2 module used as instance in BeltWarn module
  - Can continue: BeltWarn module can be used as instance in another module
    - And so on
- Hierarchy powerful mechanism for managing complexity



Hierarchical Circuits
Using Module Instances in Another Module

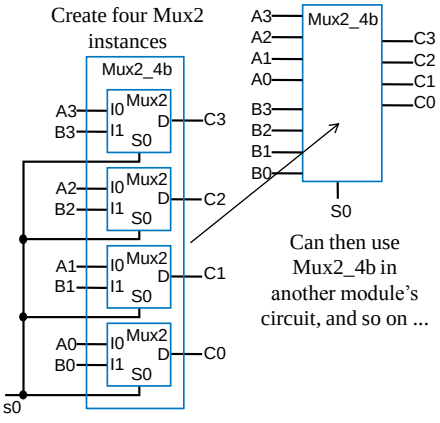
- 4-bit 2x1 mux example




```
`timescale 1 ns/1 ns

module Mux2(I1, I0, S0, D);

   input I1, I0;
   input S0;
   output D;

   wire N1, N2, N3;

   Inv  Inv_1  (S0, N1);
   And2 And2_1 (I0, N1, N2);
   And2 And2_2 (I1, S0, N3);
   Or2  Or2_1  (N2, N3, D);

endmodule
```

Hierarchical Circuits
Using Module Instances in Another Module

- 4-bit 2x1 mux example




```
`timescale 1 ns/1 ns

module Mux2_4b(A3, A2, A1, A0,
               B3, B2, B1, B0,
               S0,
               C3, C2, C1, C0);

   input A3, A2, A1, A0;
   input B3, B2, B1, B0;
   input S0;
   output C3, C2, C1, C0;

   Mux2  Mux2_3  (B3, A3, S0, C3);
   Mux2  Mux2_2  (B2, A2, S0, C2);
   Mux2  Mux2_1  (B1, A1, S0, C1);
   Mux2  Mux2_0  (B0, A0, S0, C0);

endmodule
```


Built-In Gates

Built-In Gates

- We previously defined AND, OR, and NOT gates
- Verilog has several built-in gates that can be instantiated
  - and, or, nand, nor, xor, xor
    - One output, one or more inputs
    - The output is always the first in the list of port connections
  - Example of 4-input AND:

    and a1 (out, in1, in2, in3, in4);
  - not is another built-in gate
- Earlier BeltWarn example using built-in gates
  - Note that gate size is automatically determined by the port connection list

```
`timescale 1 ns/1 ns

module BeltWarn(K, P, S, W);

   input K, P, S;
   output W;

   wire N1, N2;

   and And_1(N1, K, P);
   not Inv_1(N2, S);
   and And_2(W, N1, N2);

endmodule
```

vldd_ch2_BeltWarnGates.v