

Verilog (II)

- Signal Resolution
- Inertial & Transport Delay
- Modelling Combinational Behaviour

Four-Value Logic System

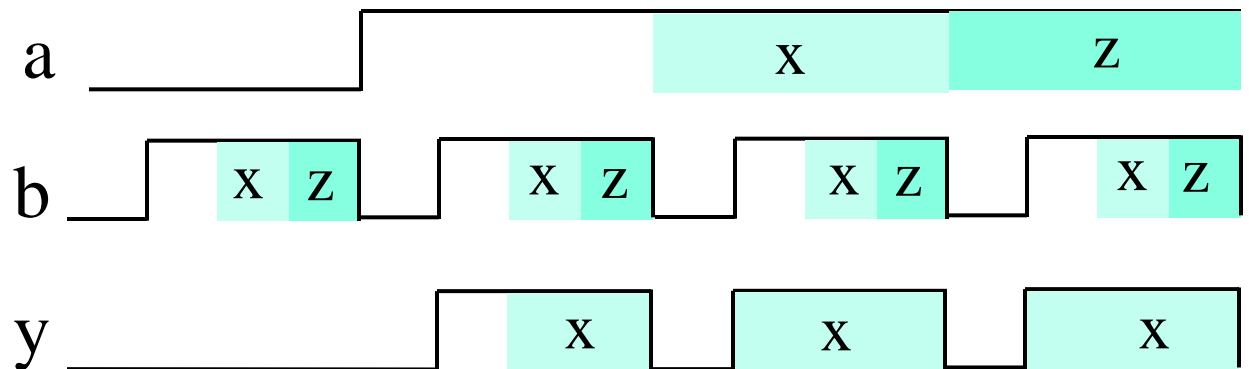
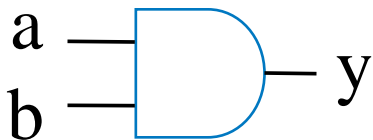
1 – assertion (True)

0 – de-assertion (False)

x – unknown (ambiguous)

z – high impedance (disconnected)

AND primitive signal resolution



Propagation Delay

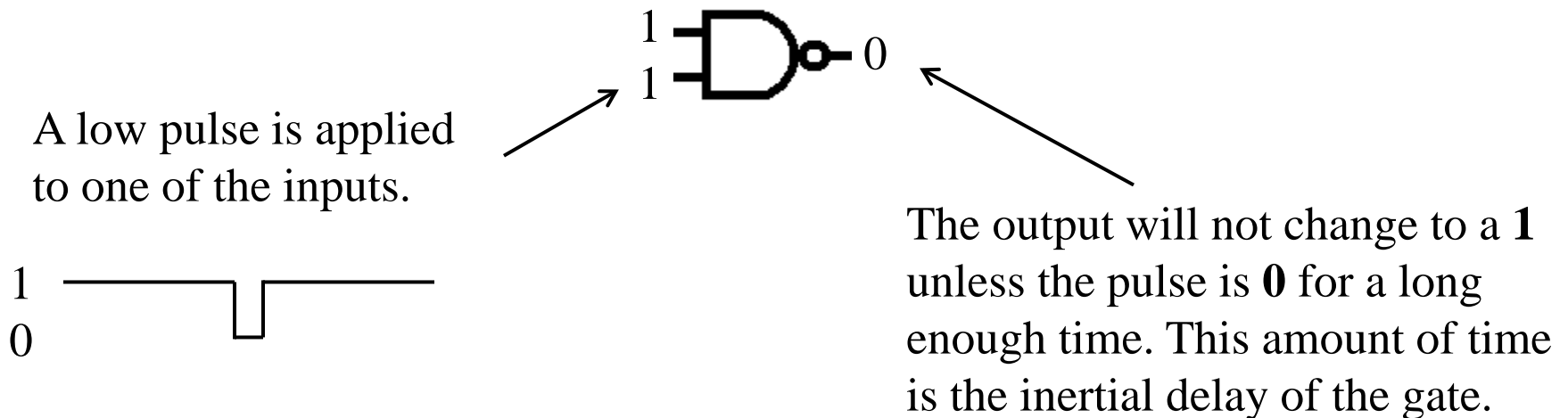
- **Propagation Delay:** time from the input changing to the output responding.
- Default delay is 0 for Verilog primitives, used to quickly verify functionality.
- Unit delay can be used to show the time ordering of signal changes.
- Delays can be added to a Verilog primitive.
- ASIC libraries provide simulation models for gates which contain accurate timing information.

```
module Add_half_unit_delay (output c_out, sum, input a, b);  
    xor    #1      M1(sum, a, b);  
    and    #1      M2(c_out, a, b);  
endmodule
```

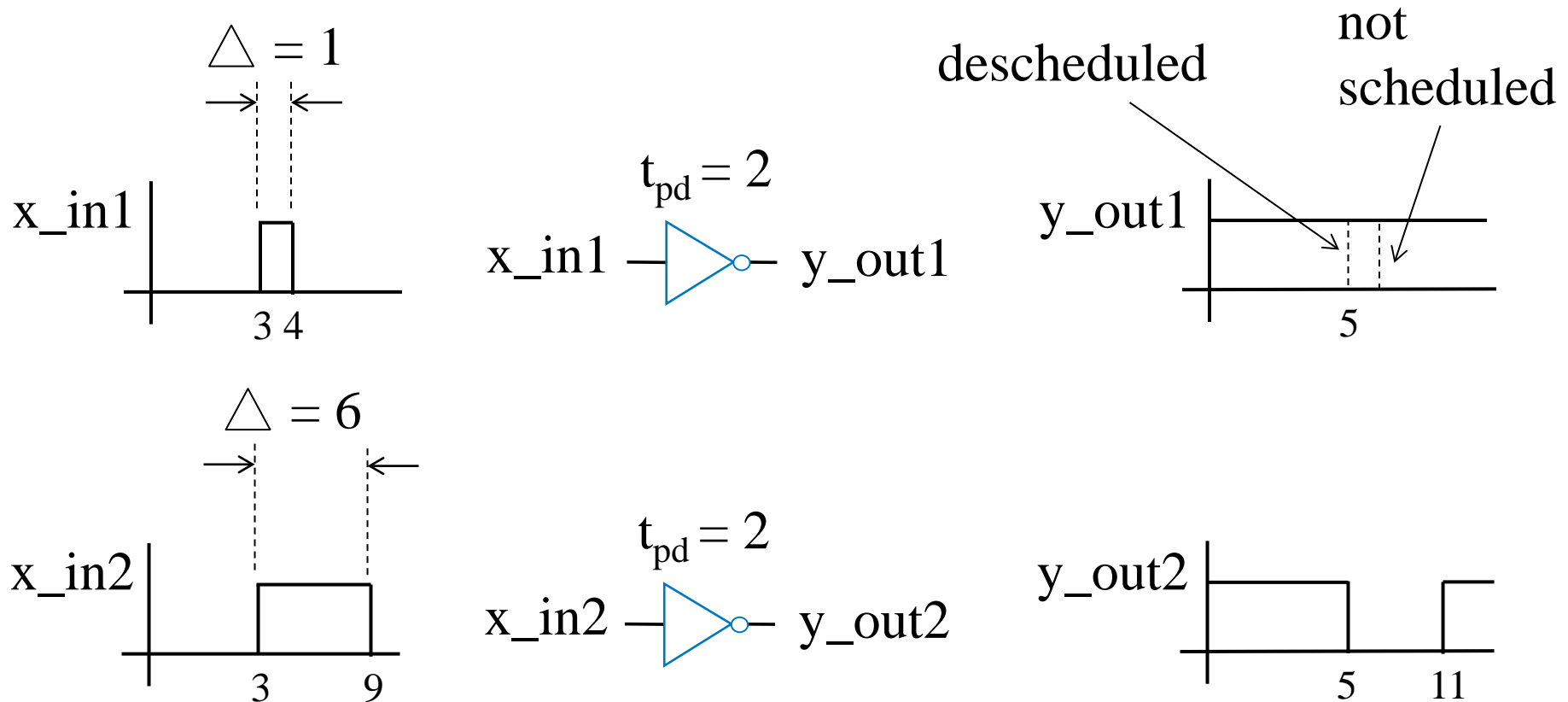
Inertial Delay Model

- Changes to the output of logic gates can not happen instantly due to the charging and discharging of capacitances.
- The transition of a signal is said to have **inertia**.
- Primitive Verilog gates follow an inertial delay model.

Consider the nand gate inputs to have been high for a long time.



Verilog uses the value of the propagation delay of a gate as the inertial delay. If the width of an input pulse is shorter than the inertial delay, the input pulse is suppressed and the output will not change in response to it.



Transport Delay Model

- Time taken for a signal to propagate along a wire
- Usually zero in an ASIC
- May be a factor in multi chip modules or on a PCB
- Narrow pulses are not suppressed

If a transport delay is required, it is stated in the wire declaration.

wire #3 long_wire_name

Timescale compiler directive

`timescale <time_unit>/<time_precision>

time_unit is a multiplier of time values, the amount of time #1 represents

`timescale 1ns/1ps instructs the simulator to use time units of 1ns

#2 ; // 2 x 1ns delay = 2ns

`timescale 10ns/1ps instructs the simulator to use time units of 10ns

#3 ; // 3 x 10ns delay = 30ns

time_precision is the smallest time step used in the simulator and determines rounding

`timescale 1ns/1ps instructs the simulator to use a precision of 1ps i.e. round to 1ps

Example:

```
`timescale 10ps / 1ps
```

```
module timedelay (f, a ,b);
```

```
input a, b;
```

```
output f;
```

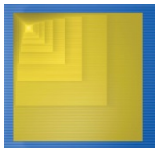
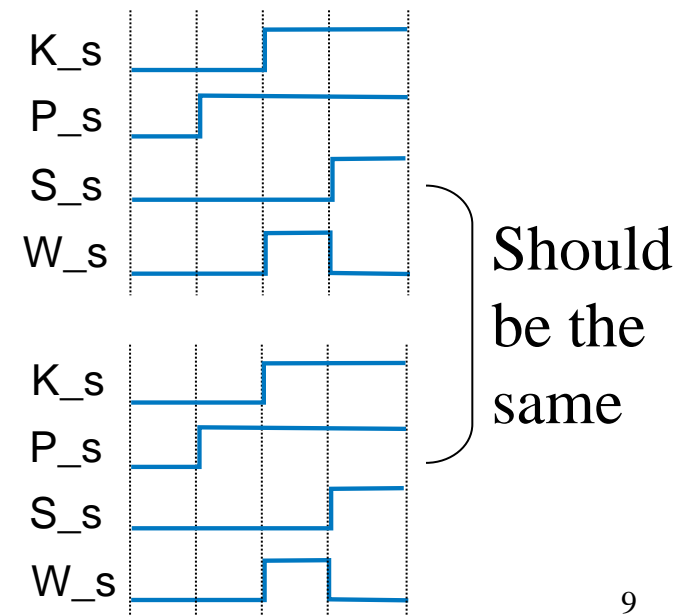
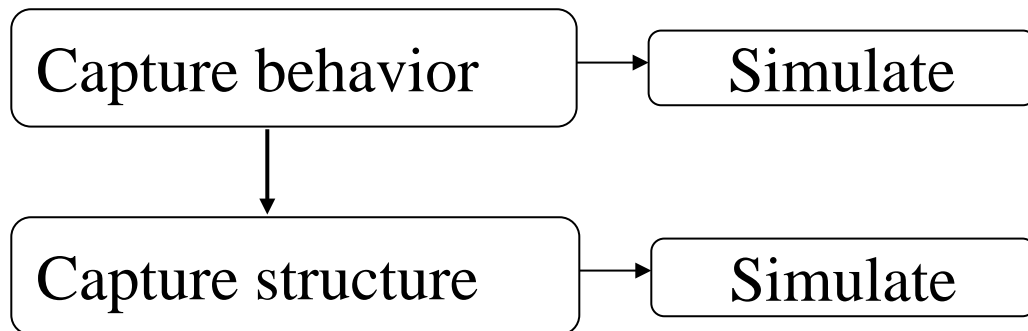
```
nand #2.58 (f, a, b);
```

```
Endmodule
```

The nand gate's delay is 26ps ($2.58 \times 10\text{ps} = 25.8\text{ps}$ rounded to 26)

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



- How to 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

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

```
input K, P, S;
```

```
output W;
```

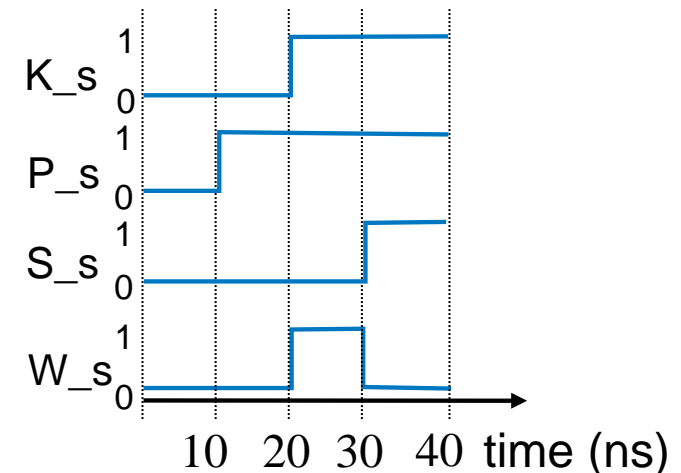
```
reg W;
```

```
always @(K, P, S) begin
```

```
W <= K & P & ~S;
```

```
end
```

```
endmodule
```



- **Procedural assignment statement**

- Assigns value to variable
- Right side may be expression of operators

- Built-in bit operators include

$\& \rightarrow \text{AND}$ $| \rightarrow \text{OR}$
 $\sim \rightarrow \text{NOT}$
 $\wedge \rightarrow \text{XOR}$ $\sim\wedge \rightarrow \text{XNOR}$

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

```
input K, P, S;
```

```
output W;
```

```
reg W;
```

```
always @(K, P, S) begin
```

```
W <= K & P & ~S;
```

```
end
```

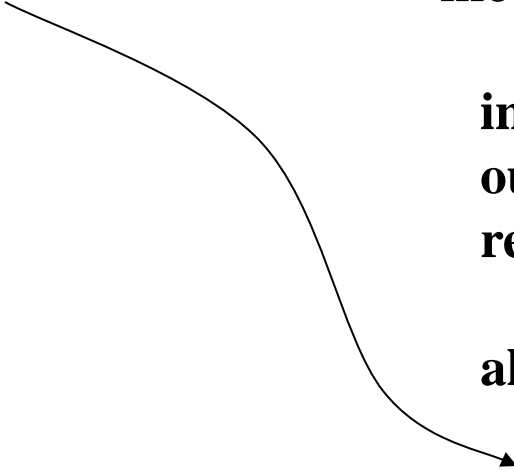
```
endmodule
```

Q: Create an always procedure to compute: $F = C'H + CH'$

1: **always** @(C,H) **begin**
 $F <= (\sim C \& H) | (C \& \sim H);$
end

2: **always** @(C,H) **begin**
 $F <= C \wedge H;$
end


- Procedure may have multiple assignment statements



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

- 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

```
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
```



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

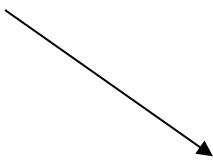
```

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

```

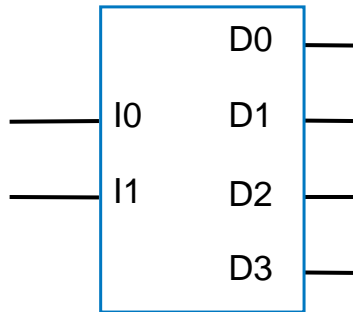


&& → logical AND

& : bit AND (operands are bits, returns bit)

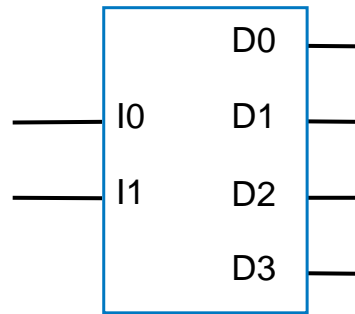
&& : logical AND (operands are true/false values, returns true/false)

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



2x4 decoder

```
module Dcd2x4(I1, I0, D3, D2, D1, D0);  
  input I1, I0;  
  output D3, D2, D1, D0;  
  reg D3, D2, D1, D0;
```



2x4 decoder

Order of assignment statements
does not matter.

Placing two statements on one
line does not matter.

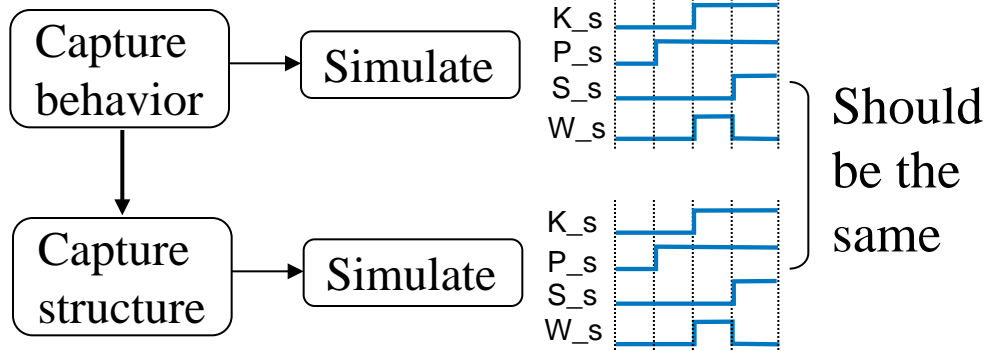
To execute multiple statements
if expression is true, enclose
them between "begin" and "end"

```

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
 - Capture behavior, and simulate
 - Capture structure using a second module, and simulate



```
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
```

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

```
  input K, P, S;
```

```
  output W;
```

```
  wire N1, N2;
```

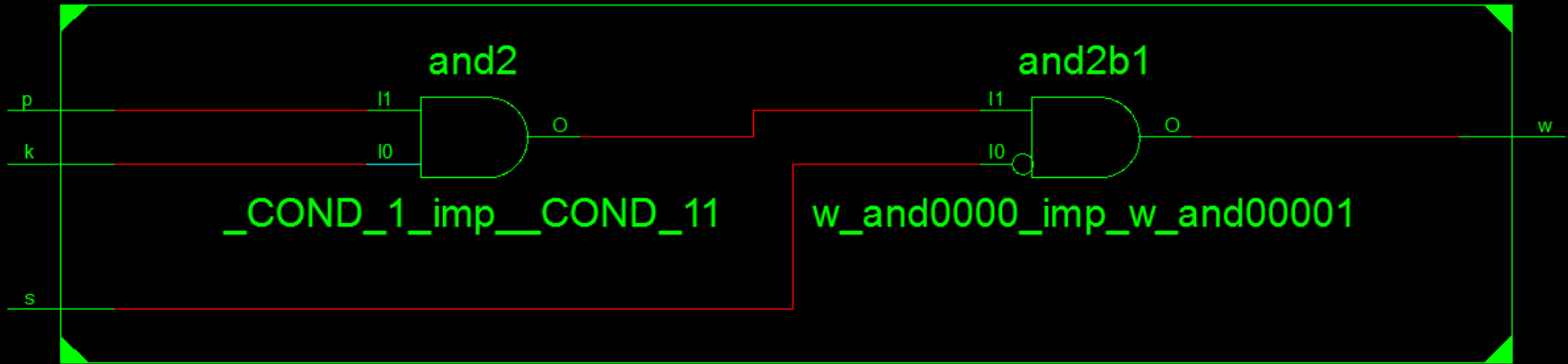
```
  And2 And2_1(K, P, N1);
```

```
  Inv Inv_1(S, N2);
```

```
  And2 And2_2(N1, N2, W);
```

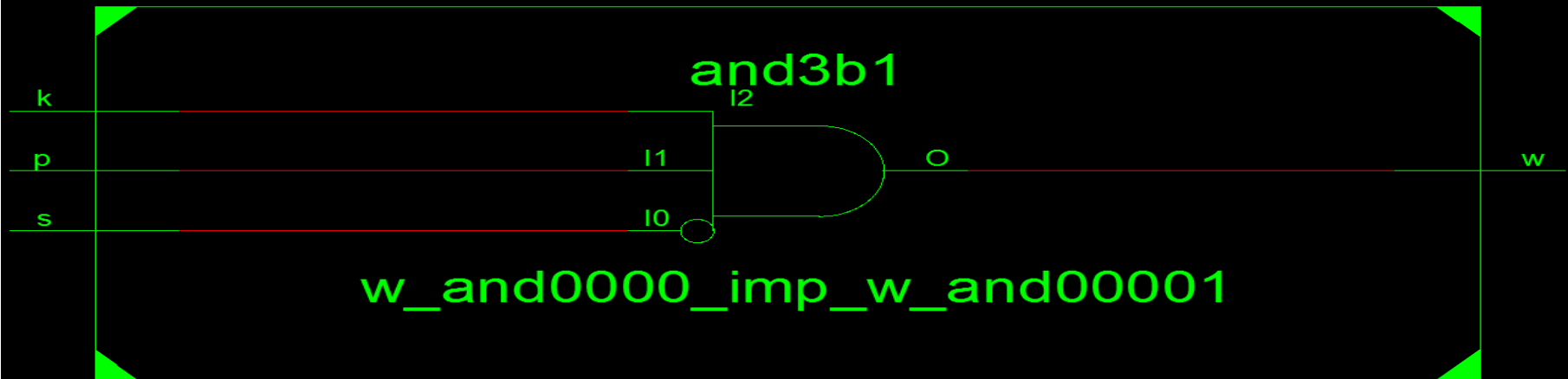
```
endmodule
```

BeltWarn:1



BeltWarn

BeltWarn:1



BeltWarn

Reminder

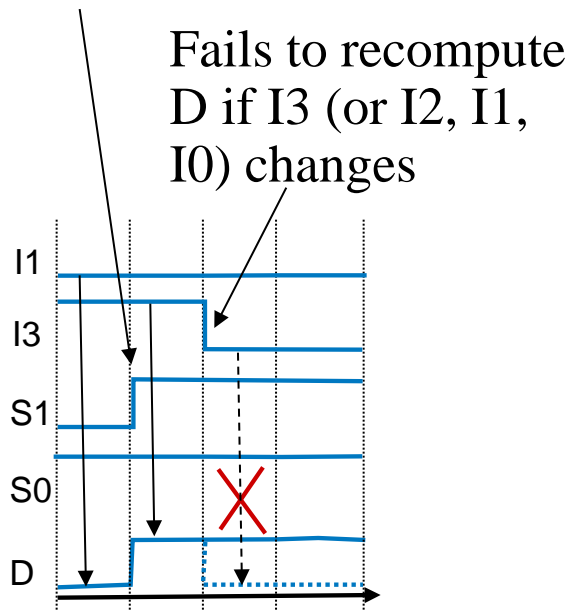
- ***Combinational behavior***: Output value is purely a function of the present input values
- ***Sequential behavior***: Output value is a function of present *and past* input values, i.e., the system has memory

- 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

Recomputes D if
S1 or S0 changes



```

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

```

Missing I3-I0 from
sensitivity list

- 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;

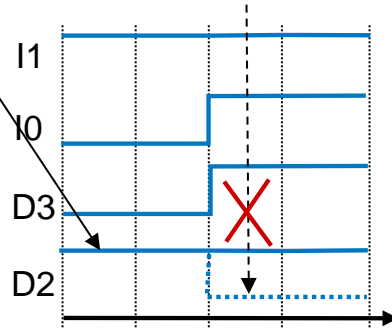
    always @*
    begin
        if ($1==0 && S0==0)
            D <= I0;
        else if (S1==0 && S0==1)
            D <= I1;
        else if (S1==1 && S0==0)
            D <= I2;
        else
            D <= I3;
    end
endmodule
```

- Pitfall – Failing to assign every output on every pass through the procedure for combinational behavior
 - Results in sequential behavior
 - Referred to as inferred latch
 - Not a 2x4 decoder
 - Has memory
 - No compiler error

I1I0=10 → D2=1,
others=0

Missing assignments
to outputs D2, D1, D0

I1I0=11 → D3=1,
but D2 stays same



```

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
end
endmodule

```

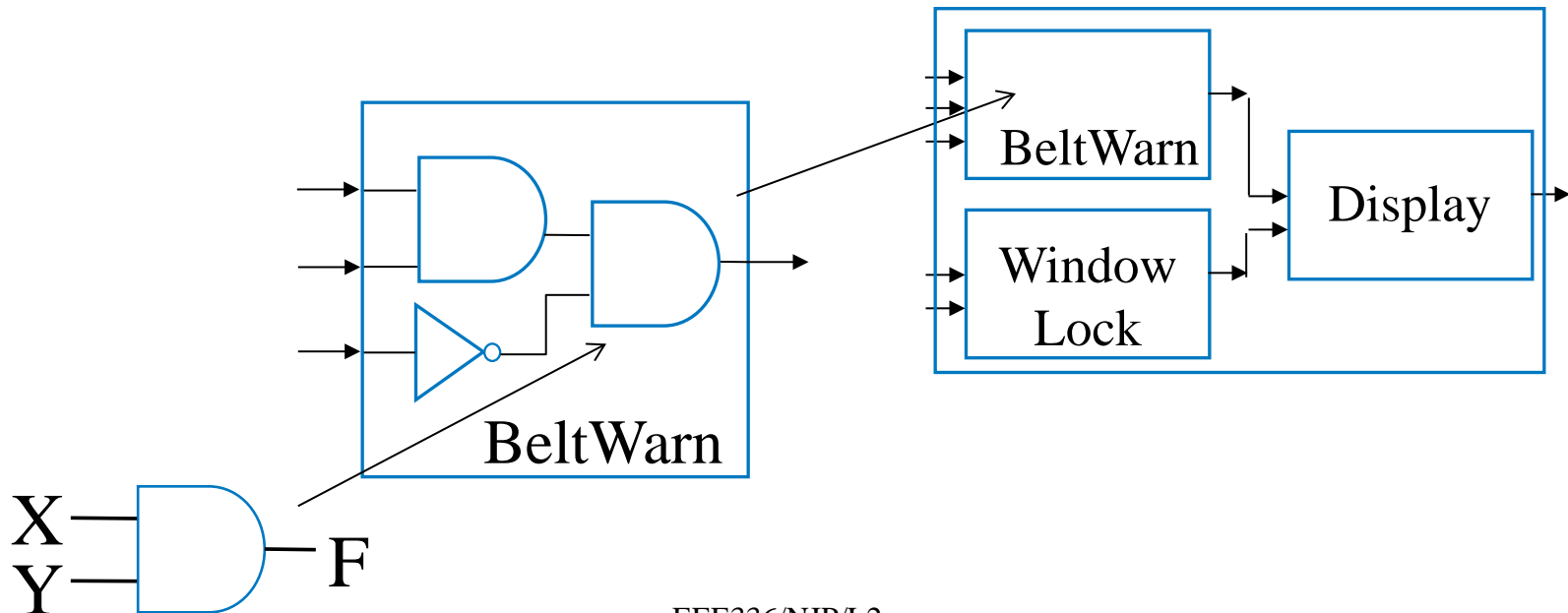
- Same pitfall often occurs due to not considering all possible input combinations

```
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

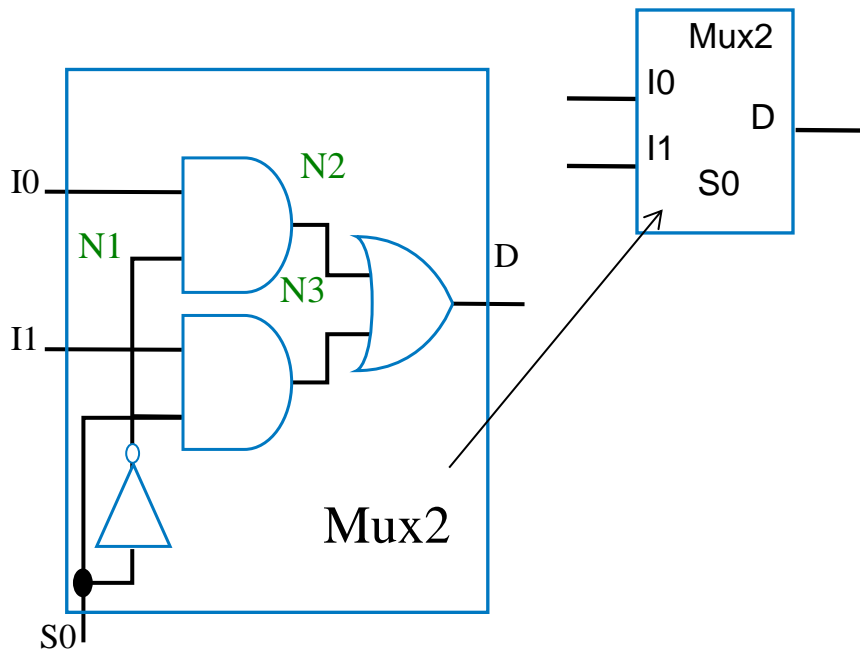
- Module can be used as instance in a new module
 - And2 module used as instance in BeltWarn module
 - BeltWarn module can be used as instance in another module
 - And so on
- Hierarchy powerful mechanism for managing complexity



Using Module Instances in Another Module

- 4-bit 2x1 mux example

2x1 mux circuit from earlier



```
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);
```

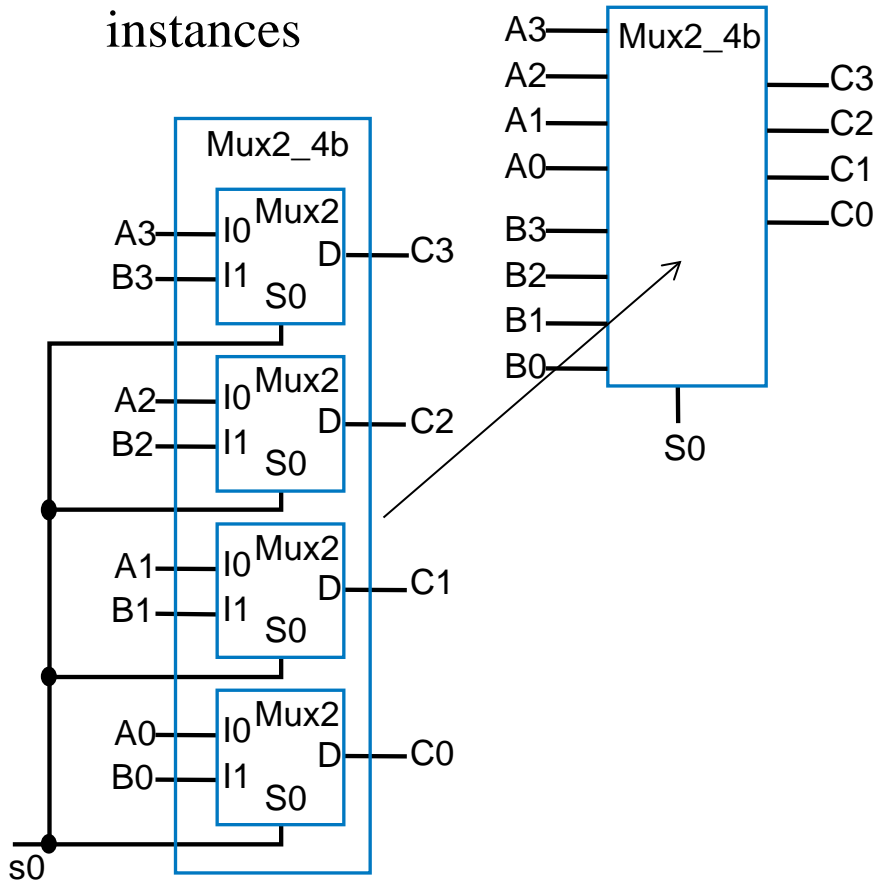
```
And2 And2_2  (I1, S0, N3);
```

```
Or2   Or2_1   (N2, N3, D);
```

```
endmodule
```

- 4-bit 2x1 mux example

Create four Mux2 instances



```
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
```

Can then use Mux2_4b in another module's circuit, and so on ...

- Earlier BeltWarn example using built-in gates

```
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
```