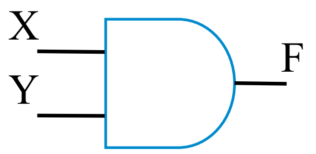
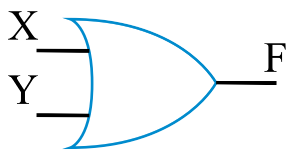
2 – Combinational Logic and Verilog Review

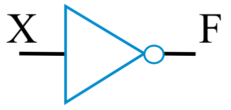
ECE 474A/574A
COMPUTER-AIDED LOGIC DESIGN

AND/OR/NOT Gates

Verilog Modules and Ports







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

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

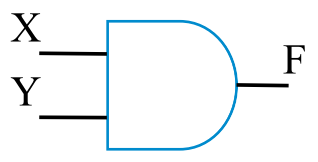
```
module Or2(X, Y, F);

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

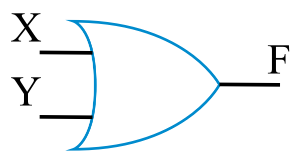
```
module Inv(X, F);

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

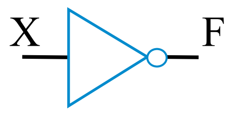
- module – Declares a new type of component
  - Named “And2" in first example above
  - Includes list of ports (module's inputs and outputs)
- input – List indicating which ports are inputs
- output – List indicating which ports are outputs
- Each port is a bit – can have value of *0*, *1*, or *x* (unknown 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;
   ...
```

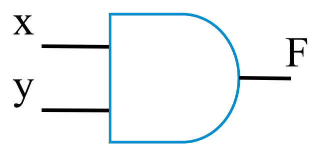
```
module Or2(X, Y, F);

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

```
module Inv(X, F);

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

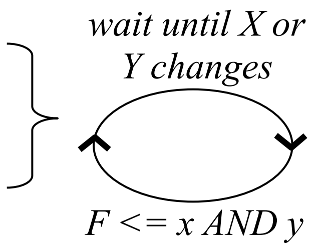
- Verilog has several dozen keywords
  - User cannot use keywords when naming items like modules or ports
  - *module*, *input*, and *output* are keywords above
  - Keywords must be **lower case**, not UPPER CASE or a MixTure thereof
- User-defined names – Identifiers
  - Begin with letter or underscore (_), 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

Module Procedures—always



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

   input X, Y;
   output F;
   reg F;

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

endmodule
```

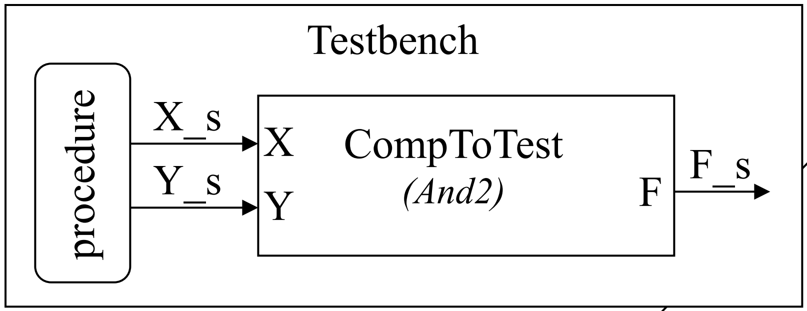


- One way to describe a module's behavior uses an "always" procedure
  - always – Procedure that executes repetitively (infinite loop) from simulation start
  - @ – event control indicating that statements should only execute when values change
    - "(X,Y)" – execute if X changes or Y changes (change 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

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

vldd_ch2_And2TB.v

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.

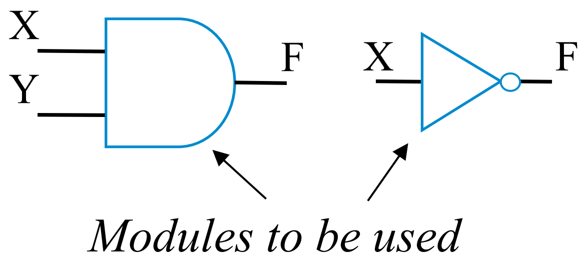
```
`timescale 1 ns/1 ns

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
```

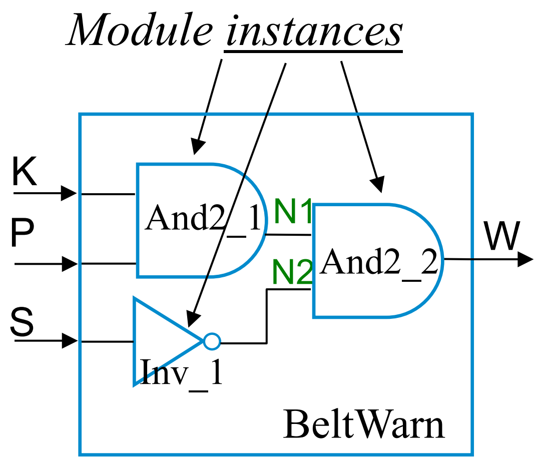
Note: We appended "_s" to reg/wire identifiers to distinguish them from ports, though not strictly necessary

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.



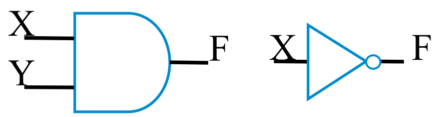


Combinational Circuits

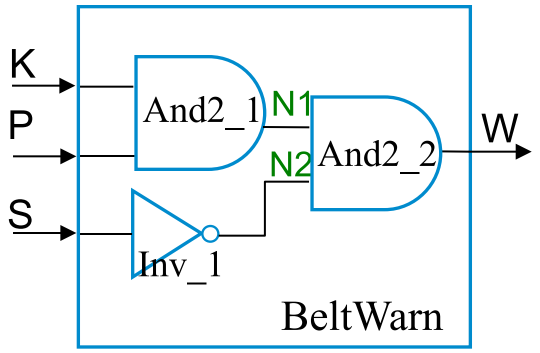
Module Instantiations

- 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)



Combinational Circuits

Module Instantiations

- Module instantiation statement

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

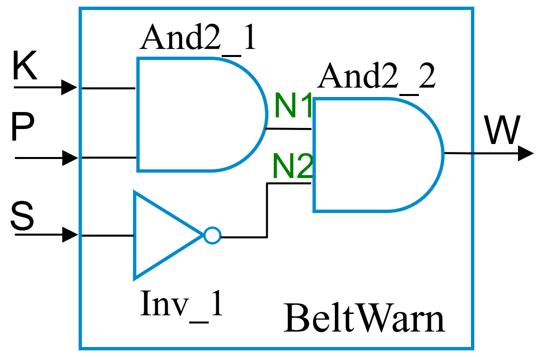
Note: Ports ordered as in original And2 module definition

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

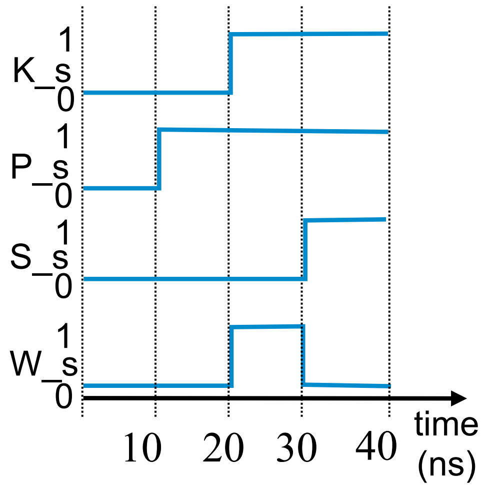
```
`timescale 1 ns/1 ns

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
```



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
```



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

```
`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
```

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

Procedures with If-Else Statements
Multiplexer

- 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

Suppose S1S0 change to 01

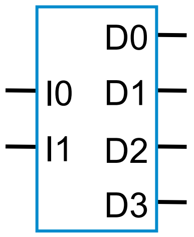
```
`timescale 1 ns/1 ns

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
```

&& → logical AND

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

Procedures with If-Else Statements
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
      begin
         D3 <= 1; D2 <= 0;
         D1 <= 0; D0 <= 0;
      end
   end
endmodule
```



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"

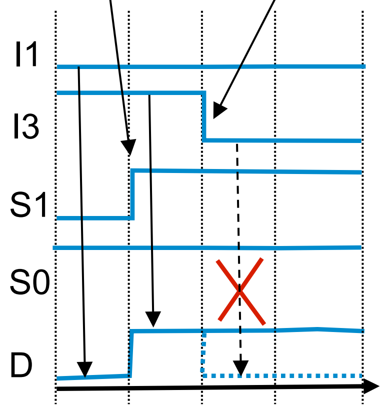
Pitfall: Missing Inputs from Event Control Expression

- 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

Missing I3-I0 from sensitivity list

Recomputes D if S1 or S0 changes

Fails to recompute D if I3 (or I2, I1, I0) changes



```
`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
```

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: 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
```

Missing assignments to outputs D2, D1, D0

I1I0=10 → D2=1, others=0

I1I0=11 → D3=1, but D2 stays same