ENGR 210 / CSCI B441

Truth Tables

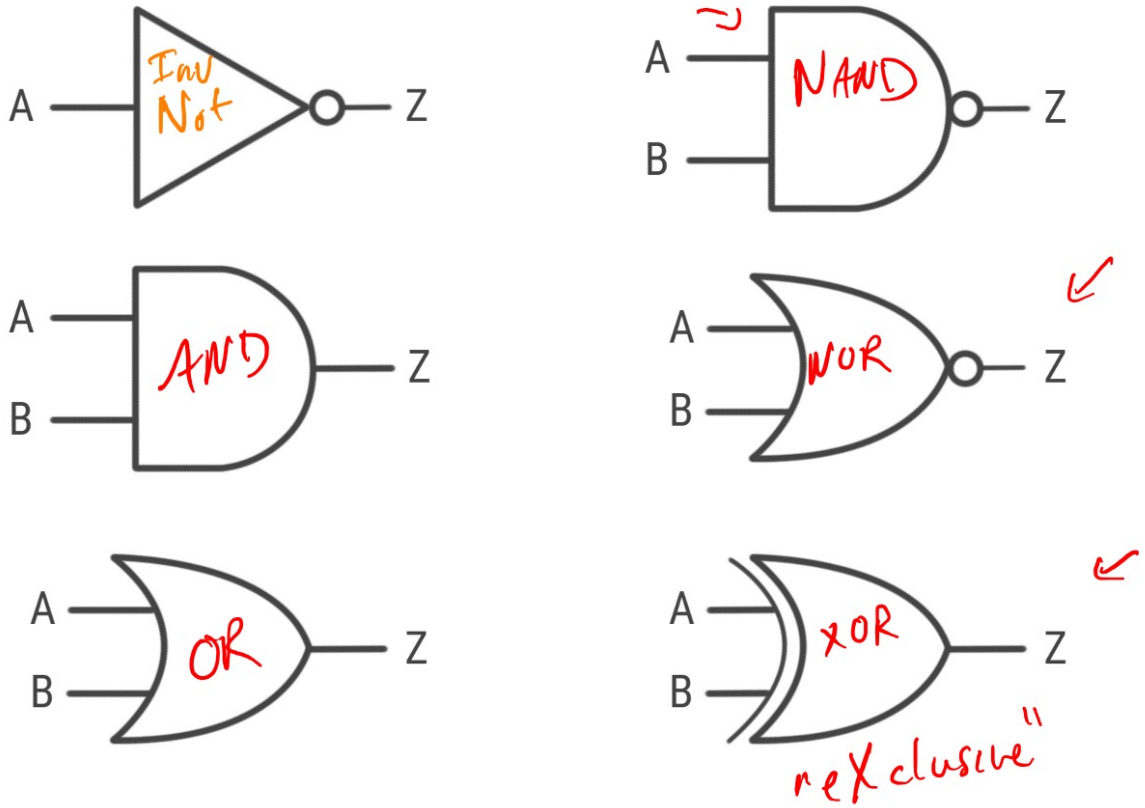
Andrew Lukefahr

Announcements

- P5 is due Friday
- P6 is out

Last Time

- Logic Gates

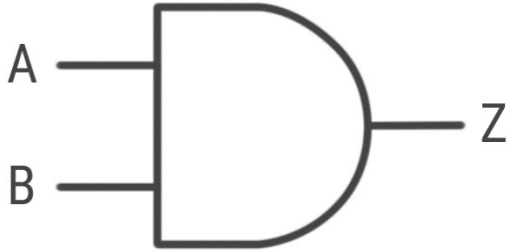


Truth Table

- “A **truth table** is a mathematical table used in logic which sets out the functional values of logical expressions on each of their functional arguments, that is, for each combination of values taken by their logical variables” [wiki]

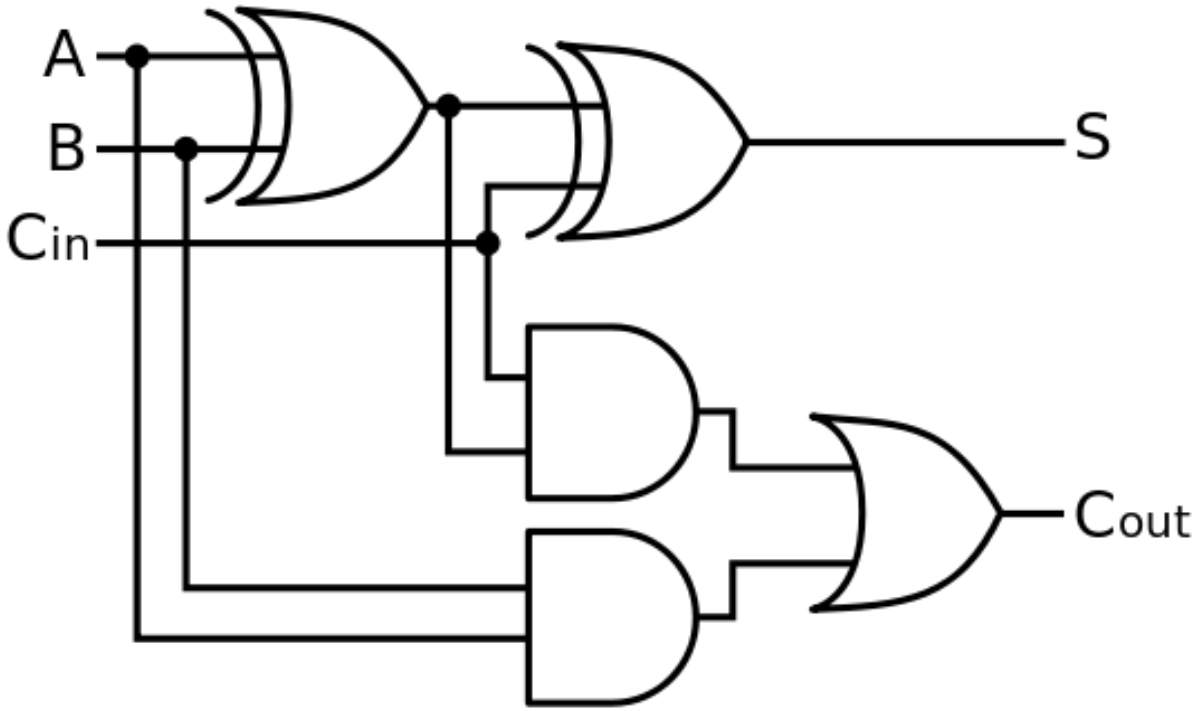
- A mapping of **all possible input values** to output values

Logic Gate Truth Table



A	B	Z

Truth Table Practice



A	B	C	Cout	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Truth Table to Boolean Equations

A	B	C	Z
0	0	0	**1**
0	0	1	0
0	1	0	**1**
0	1	1	0
1	0	0	0
1	0	1	**1**
1	1	0	0
1	1	1	0

1. Find each ’1’ output
2. AND the inputs for that output’s row
3. ‘OR’ the above equations together

Truth Table to Boolean Equations

A	B	C	Z
0	0	0	**1**
0	0	1	0
0	1	0	**1**
0	1	1	0
1	0	0	0
1	0	1	**1**
1	1	0	0
1	1	1	0

$$Z = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot \overline{C} + A \cdot \overline{B} \cdot C$$

Your Turn:

Inputs		Outputs			
a	b	d0	d1	d2	d3
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**

More Truth Tables!

$d_0 = \bar{a} \cdot \bar{b}$ decoder

$d_1 = \bar{a} \cdot b$

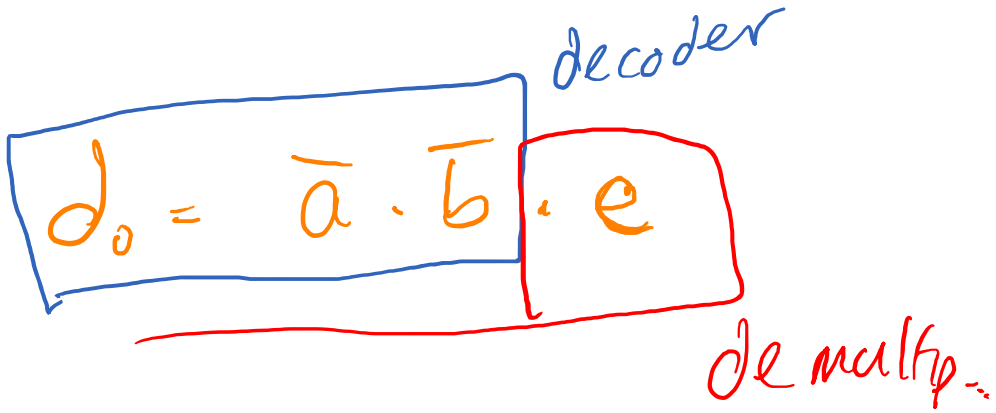
Inputs		Outputs			
a	b	d0	d1	d2	d3
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**

More Truth Tables!

Inputs			Outputs			
a	b	e	d0	d1	d2	d3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

More Truth Tables!

Inputs			Outputs			
a	b	e	d0	d1	d2	d3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1



Verilog

Example: Seat Belt Alarm

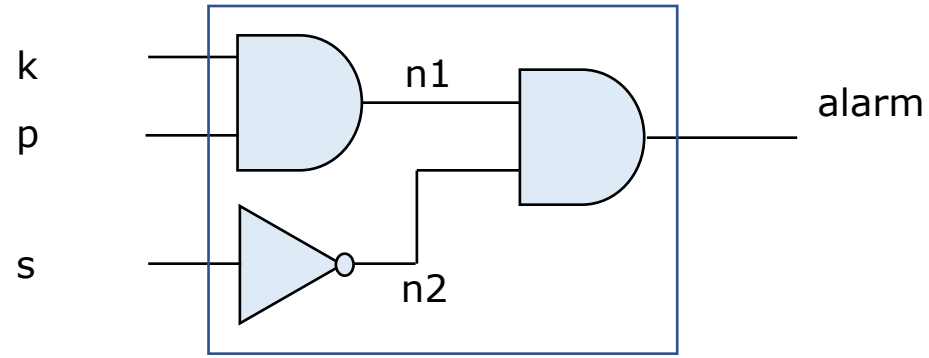
- Inputs:
  - `k`: a car's key in the ignition slot (logic 1)
  - `p`: a passenger is seated (logic 1)
  - `s`: the passenger's seat belt is buckled (logic 1)

- Goal: Set an output `alarm` to logic 1 if:
  - The key is in the car's ignition slot (`k==1`), and
  - A passenger is seated (`p==1`), and
  - The seat belt is not bucked (`s==0`)

Example: Seat Belt Alarm

- Goal: Set an output alarm to logic 1 if:
  - The key is in the car's ignition slot (*k==1*), and
  - A passenger is seated (*p==1*), and
  - The seat belt is not bucked (*s==0*)

Boolean Logic in Verilog

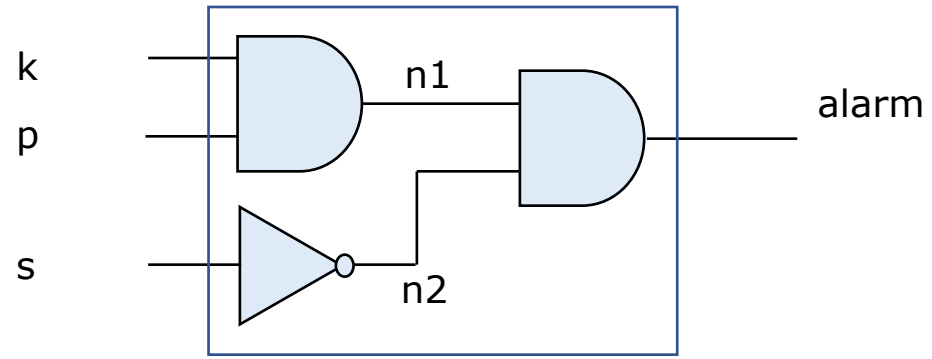


- We can use Boolean logic models in Verilog:

```
assign alarm = (k & p) & ~s;
```

- Evaluated when any of the right-hand-side operands changes
- Assigns a new value to the left-hand-side operand

Boolean Logic in Verilog



alarm = k · p · $\overline{s}$

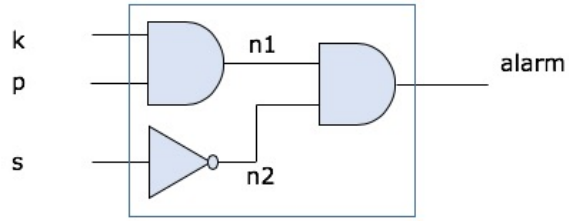
- We can use Boolean logic models in Verilog:

```
assign alarm = (k & p) & ~s;
```

NOT

- Evaluated when any of the right-hand-side operands changes
- Assigns a new value to the left-hand-side operand

Verilog Example




```
`timescale 1 ns/1 ns

//------------------------------------------------
// Example: Belt alarm
// Model:   Boolean level
//------------------------------------------------

module BeltAlarm(
     input k, p, s,     // definition of input ports
     output alarm       // definition of output ports
);

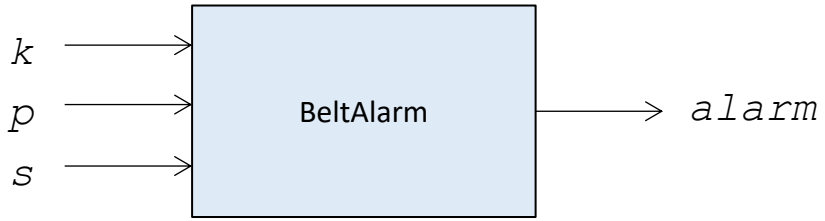
     assign alarm = k & p & ~s; //Boolean equation

endmodule
```

Aside: Synthesis

- In Synthesis, Vivado auto-magically:
  - Translates Boolean models into gate-level models
  - Simplifies and minimizes the gate-level models
- All you have to do is … wait …

2 seats?



- What if I have a car with 2 seats?
  - *k*: a car's key in the ignition slot (logic 1)
  - *st_pas*: the passenger is seated (logic 1)
  - *sb_pas:* the passenger's seat belt is buckled (logic 1)
  - *st_drv*: the driver is seated (logic 1)
  - *sb_drv:* the driver's seat belt is buckled (logic 1)

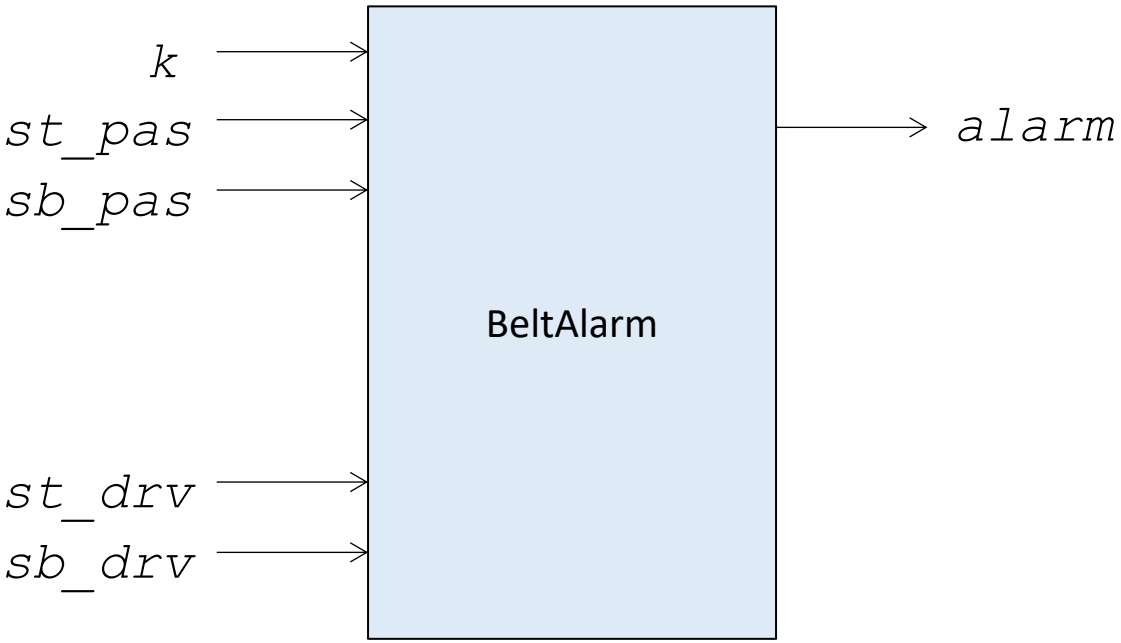
Goal: Set an output `alarm` to logic 1 if:

The key is in the car's ignition slot (*k==1*), and

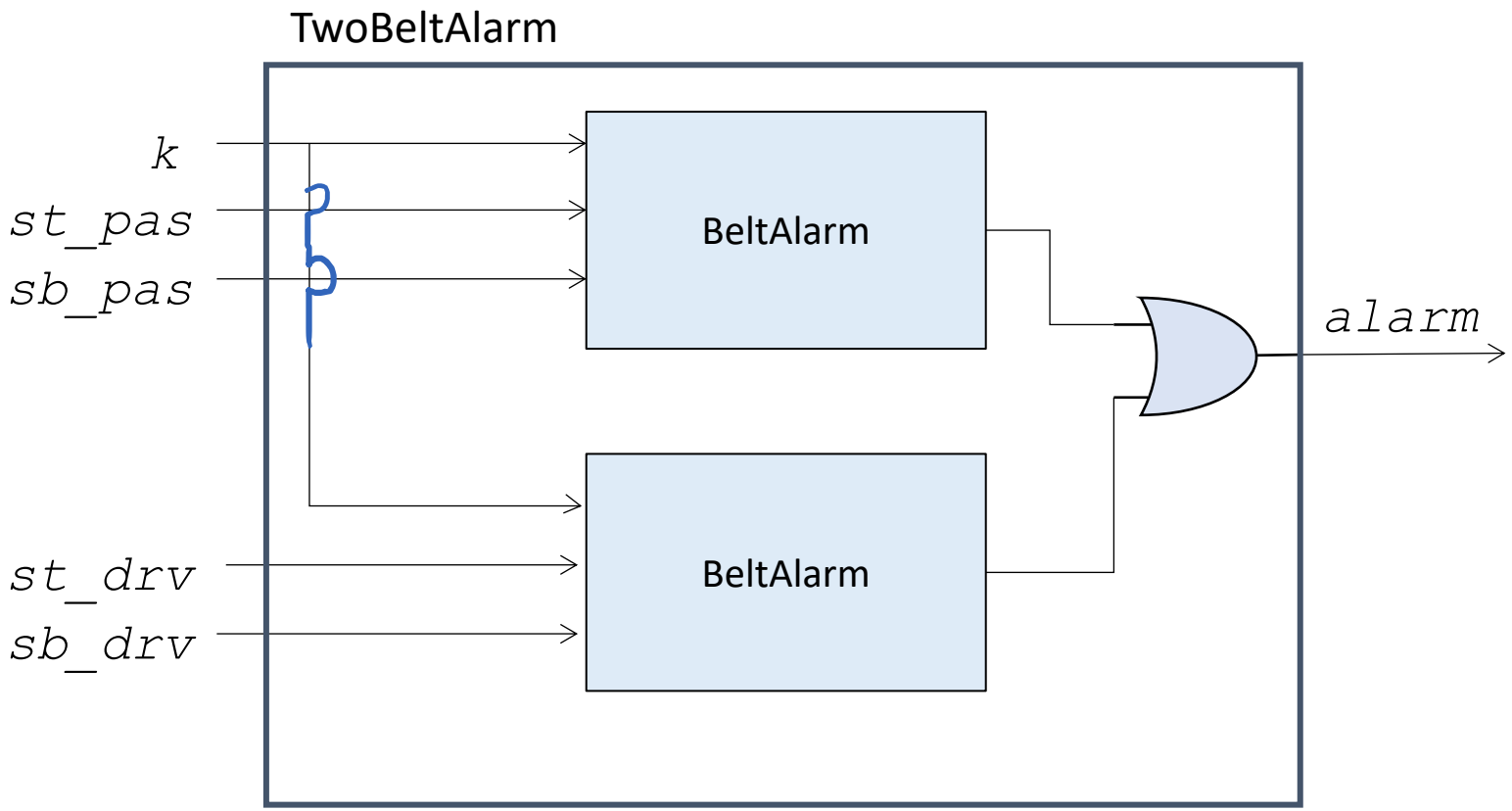
((st_drv==1 and sb_drv == 0) or

(st_pas==1 and sb_pas == 0))

2 seats: Solution 1



Solution 2: Use Submodules



Submodule Example

```
`timescale 1 ns/1 ns

module BeltAlarm(
      input k, p, s,
      output alarm
);

    assign alarm = k & p & ~s;

endmodule
```

```
`timescale 1 ns/1 ns

module TwoBeltAlarm(
      input k, st_pas, sb_pas,
      input st_drv, sb_drv
      output alarm
);
      wire al_pas, al_drv; //intermediate wires

      //submodules, two different examples
      BeltAlarm ba_drv(k, st_drv, sb_drv, al_drv); //no named arguments
      BeltAlarm ba_pas(.k(k), .p(st_pas),
            .s(sb_pas), .alarm(al_pas)); // with named arguments

      assign alarm = al_pas | al_drv;
endmodule
```

Hierarchical Models

- `Modules` are basic building block in Verilog
- Group modules together to form more complex structure

Testing

Unit Testing

- **UNIT TESTING** is a level of software testing where individual components of a software are tested. The purpose is to validate that each unit of the software performs as designed.

- We're going to test (almost) every module!

https://softwaretestingfundamentals.com/unit-testing/

TestBench

- Another Verilog module to drive and monitor our Verilog module
- Goal is to simulate real-world usage to evaluate correctness

Simulation vs Synthesis

- Synthesis: Real gates on real hardware
  - Only “synthesizable” Verilog allowed
- Simulation: Test our design with software
  - ”Non-synthesizable” Verilog allowed
  - $initial
  - $display

TestBenches

- Another Verilog module to drive and monitor our “Synthesizable” module

"initial" statement

- **Simulation only!**
- An initial block starts at simulation time 0, executes exactly once, and then does nothing.
- Group multiple statements with `begin` and `end`.
  - `begin/end` are the '**{**' and '**}**' of Verilog.

```
initial
begin
    a = 1;
    b = 0;
end
```

Delayed execution

- If a delay `#<delay>` is seen before a statement, the statement is executed `<delay>` time units after the current simulation time.

```
initial
begin
    #10 a = 1; // executes at 10 time units
    #25 b = 0;// executes at 35 time units
end
```

- We can use this to test different inputs of our circuits

$monitor

- `$monitor` prints a new line every time it's output changes
- C-like format

- ```
  $monitor($time,
      "K= %b, P= %b, S= %b, A= %b\n",
       K,P,S,A);
  ```

Example Output:

```
0  K= 0, P= 0, S= 0, A= 0
5  K= 1, P= 0, S= 0, A= 0
10 K= 1, P= 1, S= 0, A= 1
```
```

A simple testbench

```
`timescale 1ns/1ps
module BeltAlarm_tb();

logic k, p, s;
wire alarm;
BeltAlarm dut0( .k(k), .p(p), .s(s),.alarm(alarm) );

initial
begin
    k = 'h0; p = 'h0; s= 'h0;
    $monitor ("k:%b p:%b s:%b a:%b", k, p, s, alarm);
    #10
    assert(alarm == 'h0) else $fatal(1, "bad alarm");
    $display("@@@Passed");
end
endmodule
```

```
module BeltAlarm(
     input k, p, s,
     output alarm
);

      assign alarm = k & p & ~s;
endmodule
```

A simple testbench

```
`timescale 1ns/1ps
module BeltAlarm_tb();

logic k, p, s;
wire alarm;
BeltAlarm dut0( .k(k), .p(p), .s(s),.alarm(alarm) );

initial
begin
    k = 'h0; p = 'h0; s= 'h0;
    $monitor ("k:%b p:%b s:%b a:%b", k, p, s, alarm);
    #10
    assert(alarm == 'h0) else $fatal(1, "bad alarm");
    $display("@@@Passed");
end
endmodule
```

```
module BeltAlarm(
    input k, p, s,
    output alarm
);

    assign alarm = k & p & ~s;
endmodule
```

printf("%d", x)

%b ⇒ binary

%h ⇒ hex

%d ⇒ decimal

Tasks in Verilog

- A `task` in a Verilog simulation behaves similarly to a C function call.

```
task taskName(
    input localVariable1,
    input localVariable2,
    );

    #1 //1 ns delay
    globalVariable1 = localVariable1;
    #1 // 1ns delay
    assert( globalVariable2 == localVariable2)
        else $fatal(1, "failed!");

endtask
```

SeatBelt Task

```
task checkAlarm(
    input kV, pV, sV,
    input alarmV
    );

    k = kV; p=pV; s=sV;
    #10
    assert(alarm == alarmV) else
        $fatal (1, "bad alarm, expected:%b got:%b",
                alarmV, alarm);
endtask
```

SeatBelt Testing

```
initial
begin
    k = 'h0; p = 'h0; s= 'h0;
    $monitor ("k:%b p:%b s:%b a:%b",
       k, p, s, alarm);
    checkAlarm('h0,'h0,'h0, 'h0);
    checkAlarm('h0,'h0,'h1, 'h0);
    checkAlarm('h0,'h1,'h0, 'h0);
    checkAlarm('h0,'h1,'h1, 'h0);
    checkAlarm('h1,'h0,'h0, 'h0);
    checkAlarm('h1,'h0,'h1, 'h0);
    checkAlarm('h1,'h1,'h0, 'h1);
    checkAlarm('h1,'h1,'h1, 'h0);
    $display("@@@Passed");
end
```

```
task checkAlarm(
    input kV, pV, sV,
    input alarmV
    );

    k = kV; p=pV; s=sV;
    #10
    assert(alarm == alarmV) else
        $fatal (1, "bad alarm, expected:%b
got:%b",
            alarmV, alarm);
endtask
```

`Tasks` in Testing

- `tasks` are very useful for quickly testing Verilog code

- Call a `task` to quickly change + check things
- A `task` can call another `task`

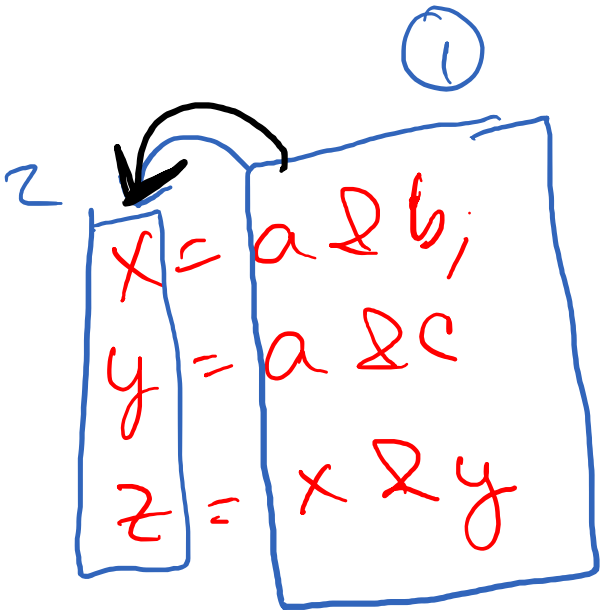
- There is a `function` in Verilog.
- We don't use it.

Next Time

- Continue with Verilog

Next Time

- Continue with Verilog



Verilog

Python

```
def foo(a, b, c)
① ↳ x = a & b ;
② ↳ y = a & c ;
③ ↳ z = x & y ;
return [x, y, z]
```

foo(1, 1, 0)