Truth Tables

Andrew Lukefahr

Project 0: Logic Gates

- Implement AND,OR, and NOT on a Basys3 FPGA

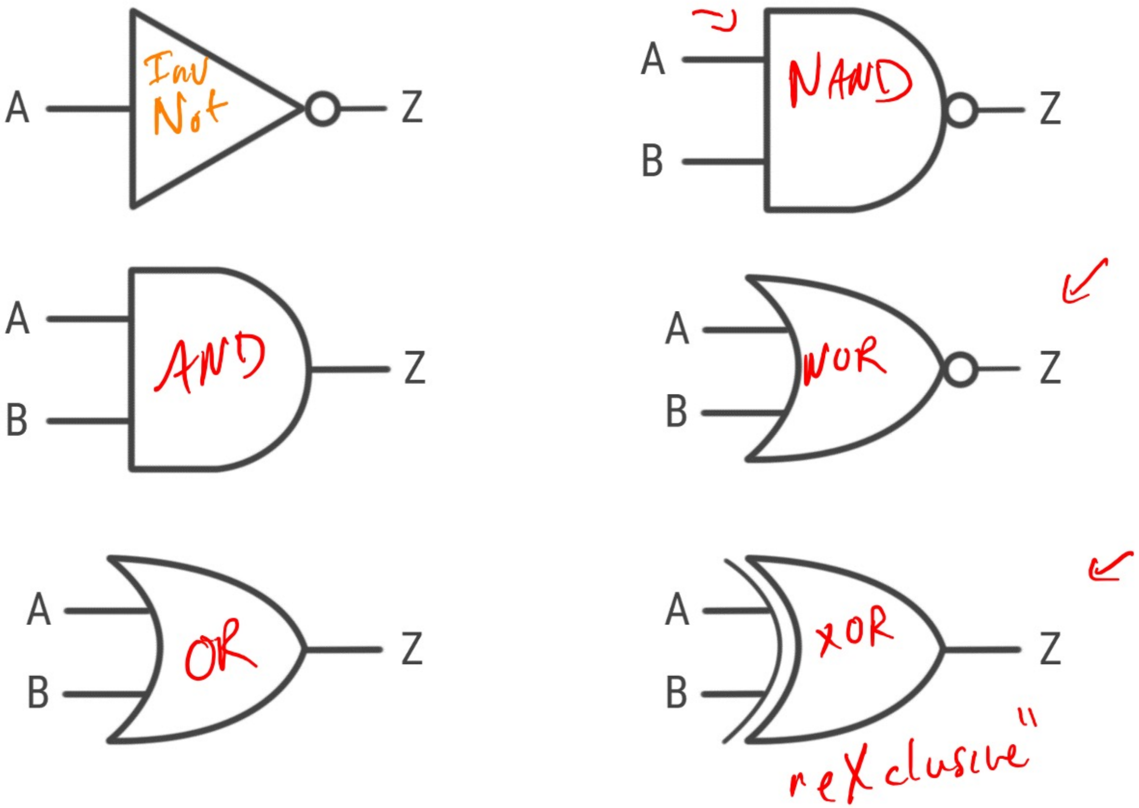
- This is a “completion” lab, no new code

- Expect you to complete this *by next week*

- For questions, email Chris Sozio

Last Time

- Logic Gates

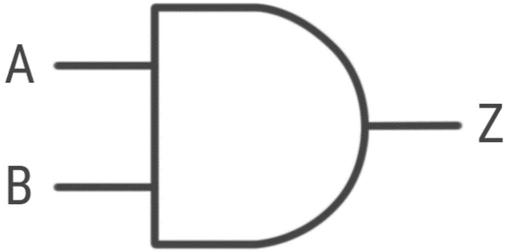


Truth Table

- “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]

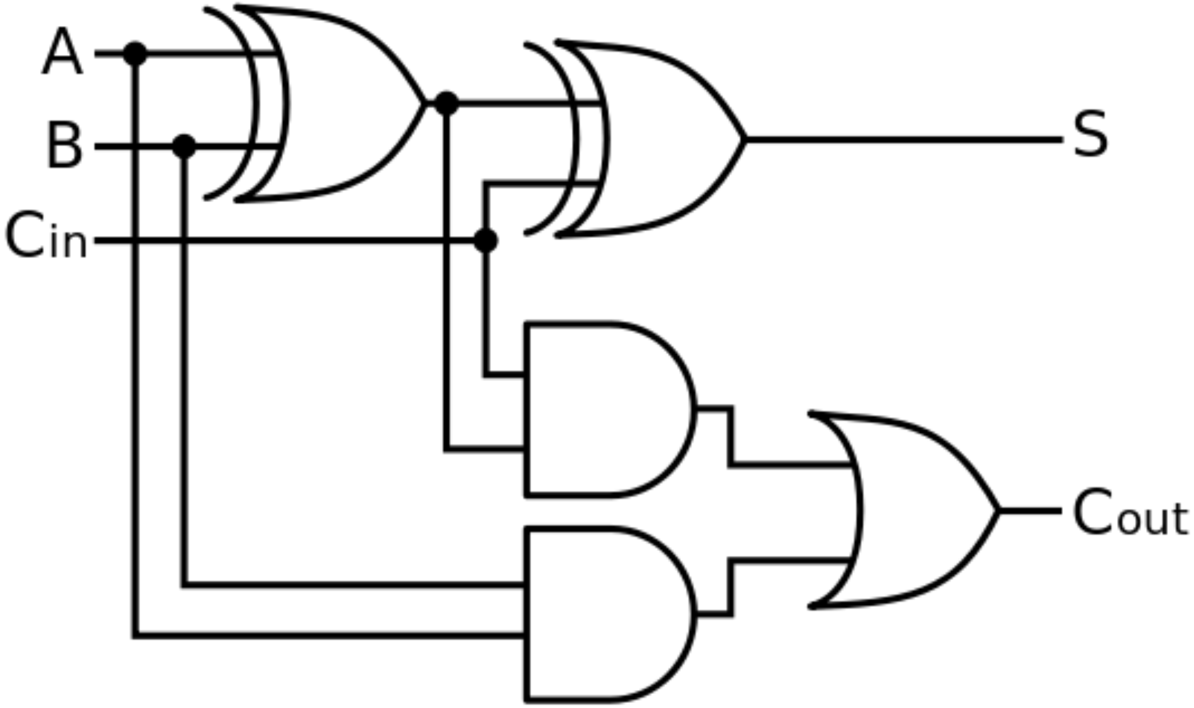
- A mapping of **all possible input values** to output values

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

$\cdot \rightarrow$ AND

$+ \rightarrow$ OR

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

1. Find each '1' output
2. Write the equation for that output
3. 'OR' the above equations together

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

"Demorgan"

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. Write the equation for that output
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

1. Find each '1' output
2. 'AND' the inputs ~~Write the equation~~ for that output's row
3. 'OR' the above equations together

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

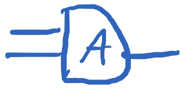
$\cdot$ - AND

$+$ - OR

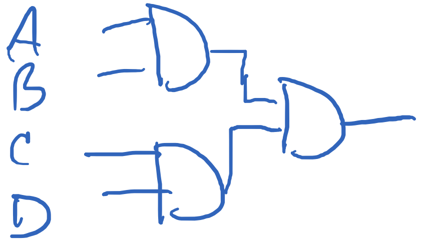
Truth Table to Boolean Equations

A	B	C	Z
0	0	0	$\boxed{\mathbf{1}}$
0	0	1	0
0	1	0	$\boxed{\mathbf{1}}$
0	1	1	0
1	0	0	0
1	0	1	$\boxed{\mathbf{1}}$
1	1	0	0
1	1	1	0

$$
\begin{aligned}
Z = {} & \overline{A}\cdot\overline{B}\cdot\overline{C} \; + \\
& \overline{A}\cdot B\cdot\overline{C} \; + \\
& A\cdot\overline{B}\cdot C
\end{aligned}
$$



⇒ $A \cdot B$ ⇒ A & B

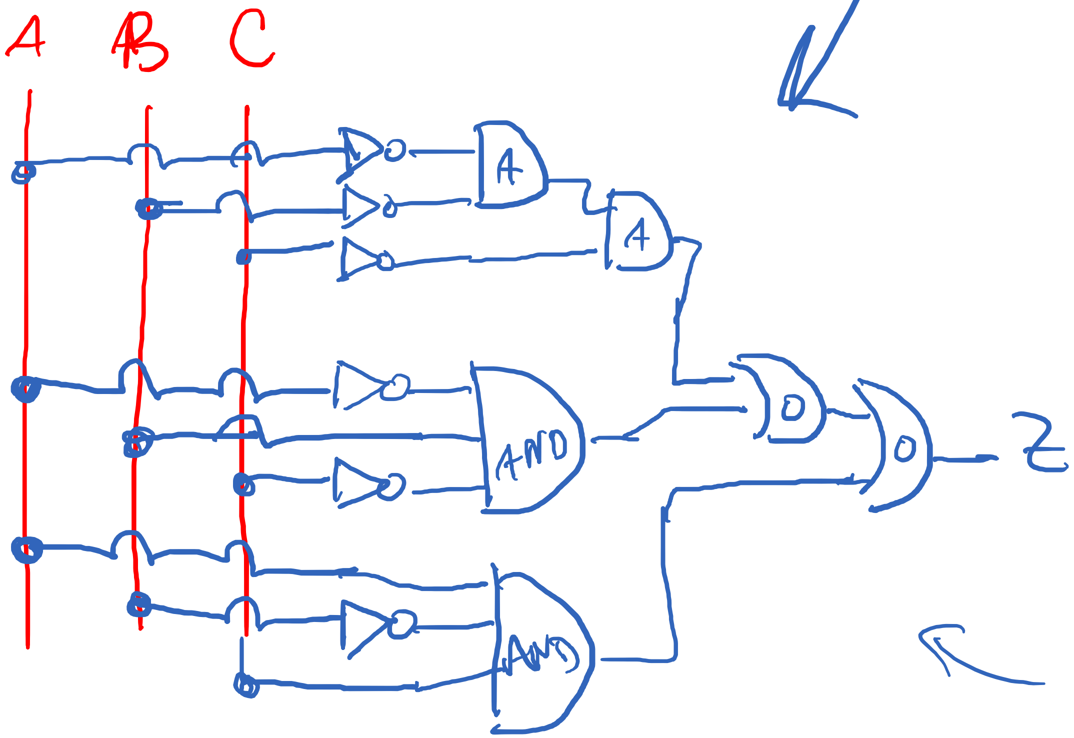


Truth Table to Boolean Equations

$Z = (A \& B) \& C \& D$

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

$$\begin{aligned} Z = {} & \overline{A} \cdot \overline{B} \cdot \overline{C} \; + \\ & \overline{A} \cdot B \cdot \overline{C} \; + \\ & A \cdot \overline{B} \cdot C \end{aligned}$$



More Truth Tables!

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!

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

decoder

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

demultip..

(Skip)

More Truth Tables!

A	**B**	**S**	**Z**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	**1**
1	0	1	0
1	1	0	**1**
1	1	1	**1**

Verilog

Verilog → programmins language for describins digital logic circuits.

"SystemVerilog" = "Verilog"

$x += 2$ $x = x + 2$

Verilog	Python
happen in parallel	happen in sequence

$x = a \& B$

$y = x \& c$

① $x = A \& B$

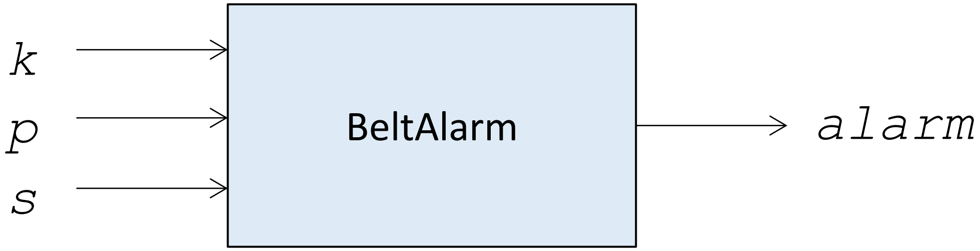
$x = new, y = old$

②

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

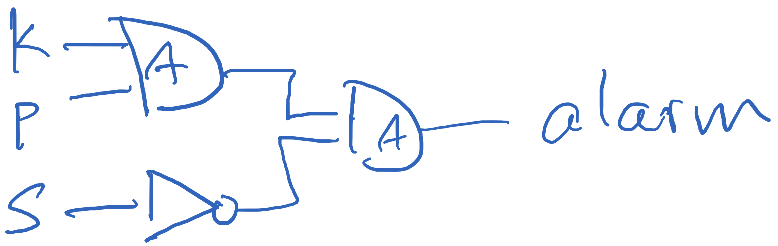
3 inputs = 2^3 rows

- 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)

	k	p	s	alarm
0	0	0	0	0
1	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
6	1	1	0	1
7	1	1	1	0

$$alarm = (k \cdot p) \cdot (\overline{s})$$

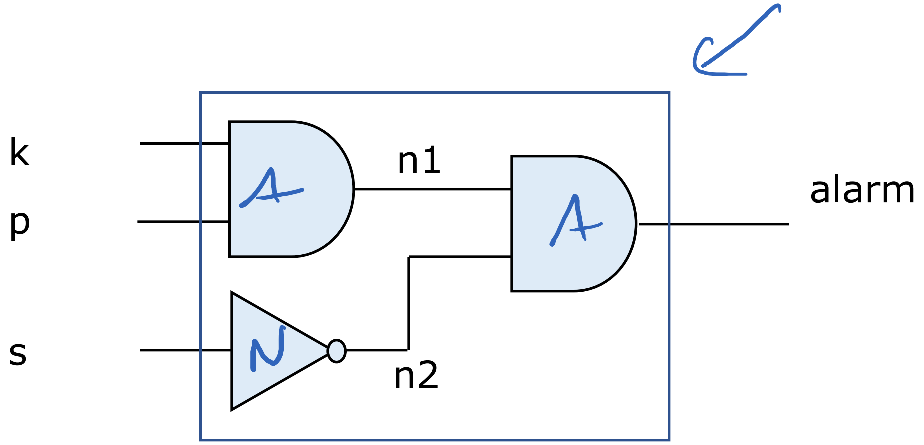


Example: Seat Belt Alarm

- 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`)

Example: Seat Belt Alarm

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



Hardware Description Languages

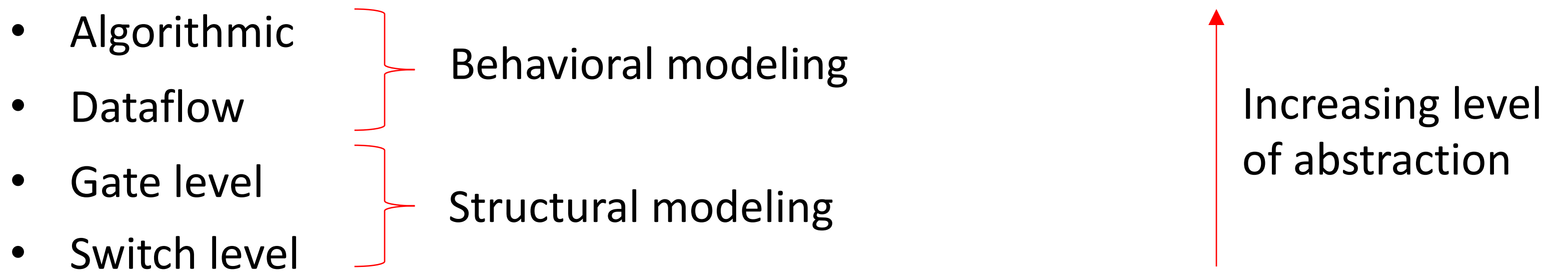
- Different ways represent same same digital circuit:
  - Circuit Schematic
  - Boolean function alarm = $K \cdot S \cdot \bar{P}$
  - Truth Table
- All 3 fail with "big" digital circuits
- 4th Option: Hardware description language (HDL)
- HDL: a programming language, specialized to model digital circuits
- Two main HDLs today:
  - Verilog (this course)
  - VHDL

"System Verilog"

Versatile HDL / Very High Speed HDL

Structural and behavioral modeling

Verilog supports modeling at four levels of abstraction:



- Algorithmic
- Dataflow
- Gate level
- Switch level

Behavioral modeling

Structural modeling

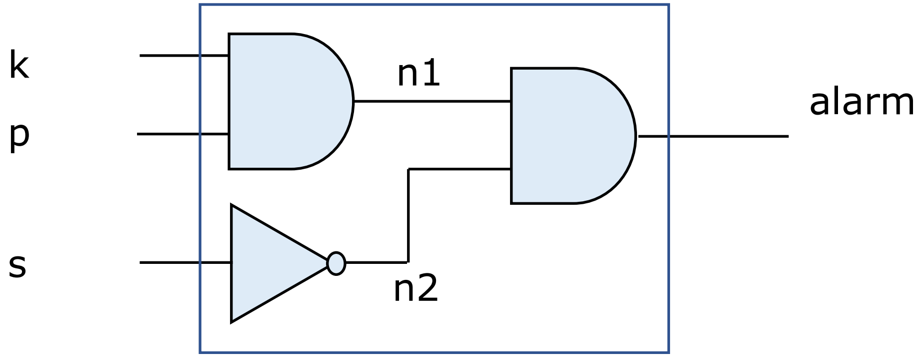
Increasing level of abstraction

In some books, algorithmic modeling is considered to be the only behavioral modeling.

In addition to this, Verilog support hierarchical models, in which the models at a higher level use the instantiations of the lower level models.

We will use algorithmic, dataflow and gate level modeling.

Boolean Logic in Verilog

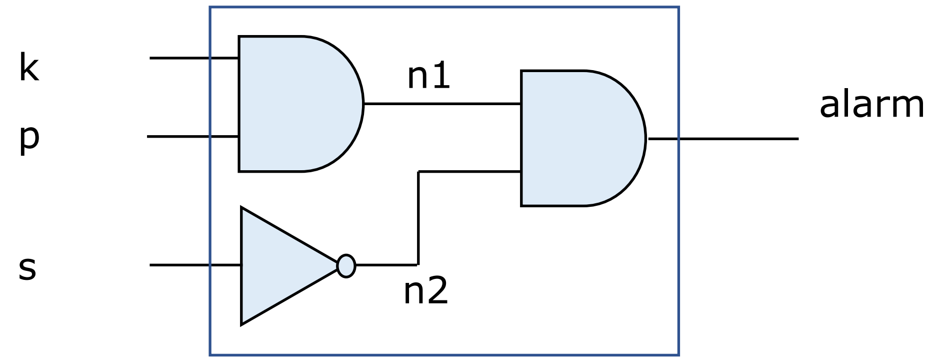


- We can use Boolean logic models in Verilog:

```
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



$\text{alarm} = k \cdot p \cdot \overline{s}$

- We can use Boolean logic models in Verilog:

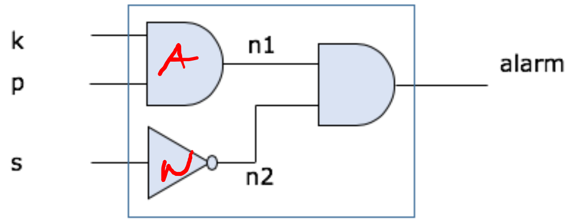
NOT

```
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

Verilog Example

$\frac{1}{10^9}$ seconds



```
'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
```

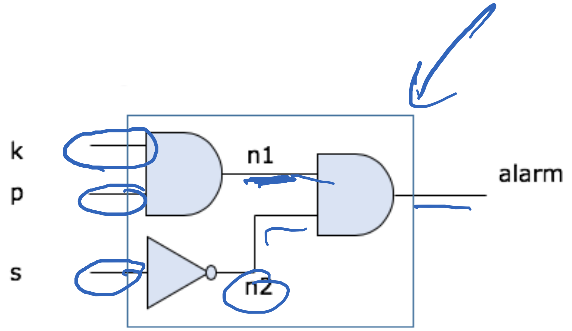
Verilog Gate-level modeling

- Verilog can ***also*** use logic-gate level models
- Verilog supports predefined logic gates:
  - `and, or, xor, nand, nor, xnor`
  - `buf, not, bufif, notif`
- Example: `and and_1(n1, k, p);`
- Instantiated like submodules, but they do not need a module definition.
    - Cover submodules soon…

HDL's in general

Stopped here!

Verilog Gate-Level Example



pl wire constraints

office hours

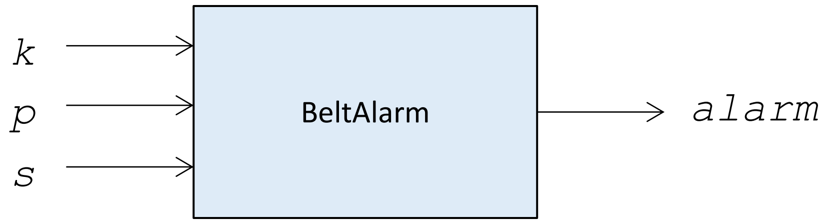
```
'timescale 1 ns/1 ns

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

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

    wire n1, n2;                // definition of wires

    and and_n1(n1, k, p);         // instantiations of
    not not_n2(n2, s);        // primitive gates
    and and_al(alarm, n1, n2);

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)

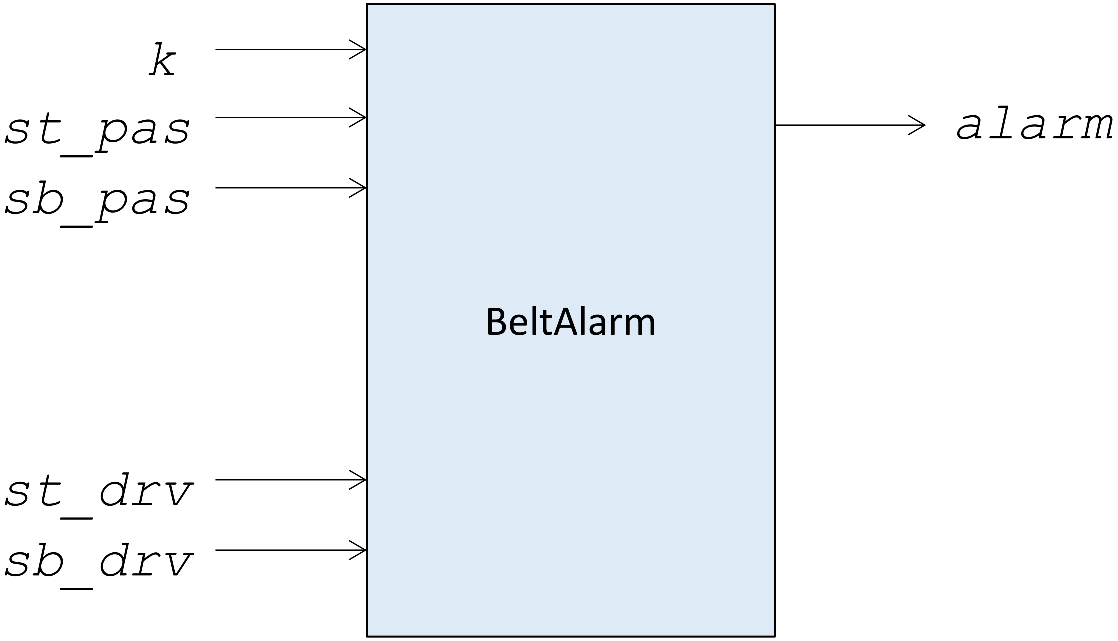
Goal: Set an output `alarm` to logic 1 if:

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

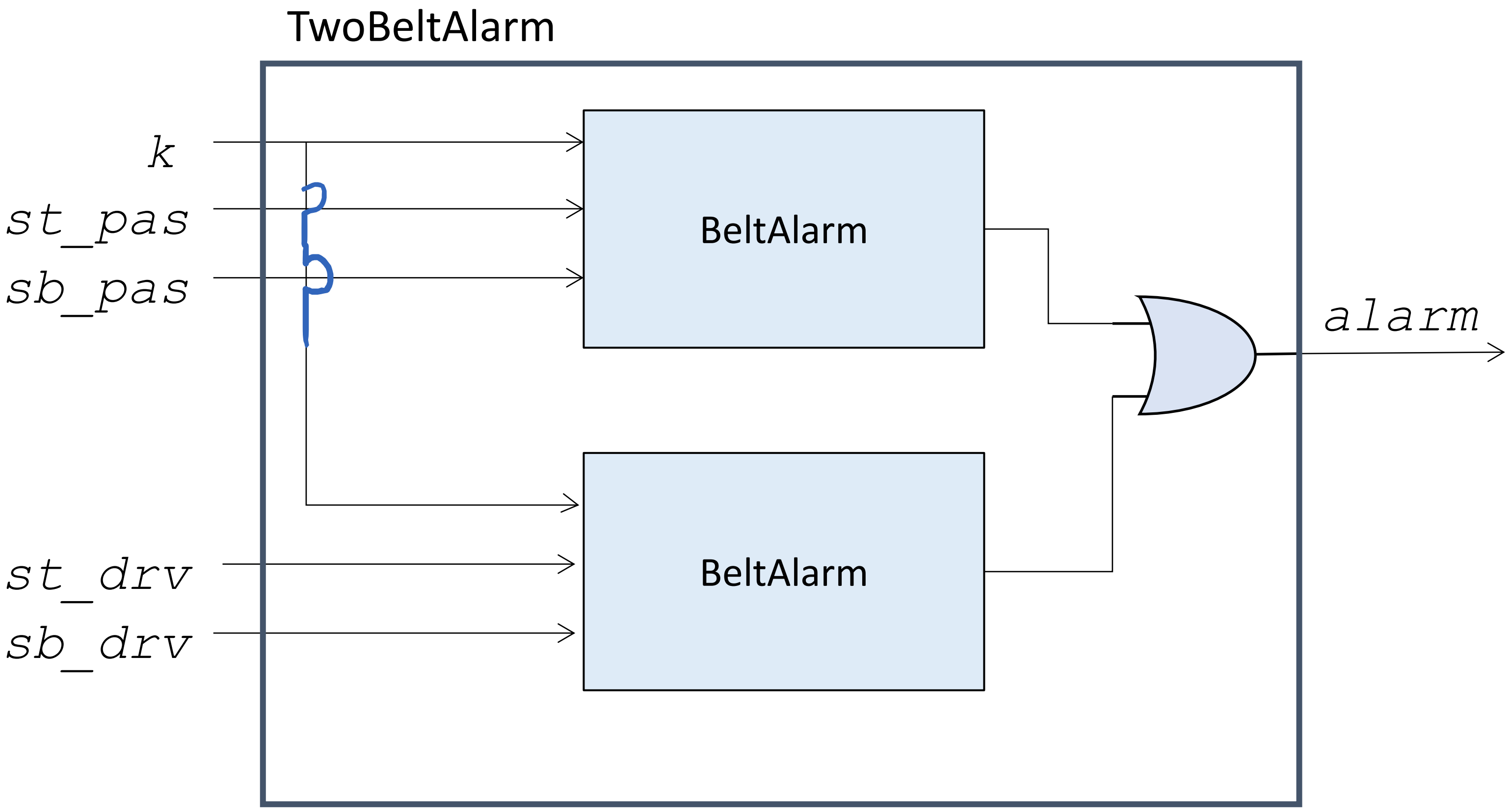
((st_drv==1 and sb_drv == 0) or

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

2 seats: Solution 1



Solution 2: Use Submodules



Submodule Example

```
‘timescale 1 ns/1 ns

module 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
```

```
‘timescale 1 ns/1 ns

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

    assign alarm = k & p & ~s;

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

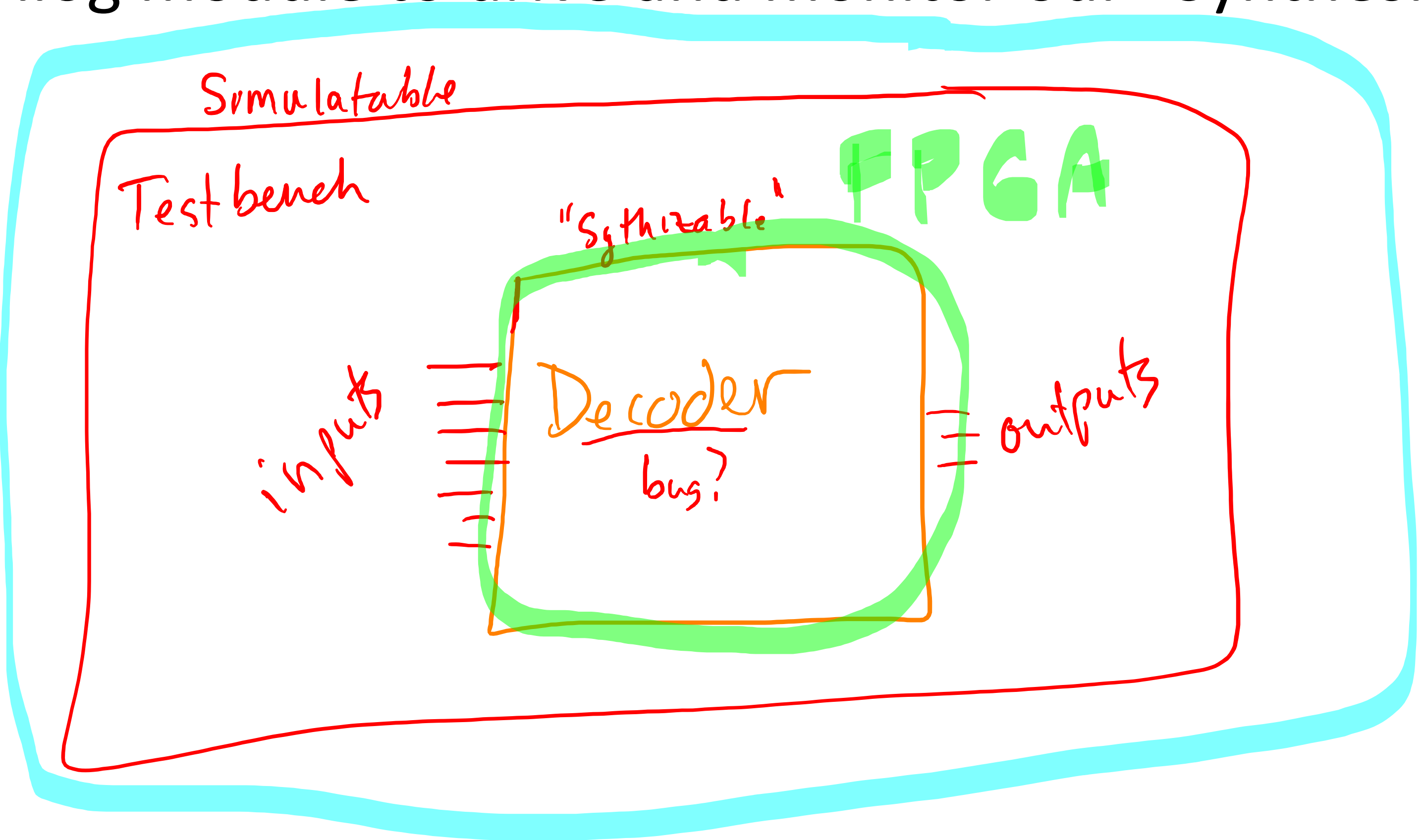
- Simulation: Test our design with software
  - ”Non-synthesizable” Verilog allowed
  - $initial
  - $display

TestBenches

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

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

```
`timescale 1ns/1ps
```

Delayed execution

#10.0005 ← appear the

#10.0001 ← same

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

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

10 ns

- We can use this to test different inputs over time on 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
```

$monitor

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

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

%b = binary

%h = hex

%d = decimal

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
```

Tasks in Verilog

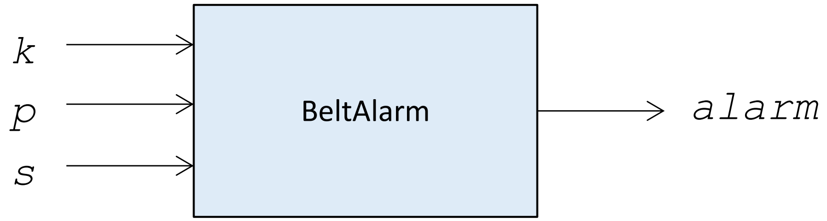
- A task in a Verilog simulation behaves similarly to a C function call.

```
task taskName
    input localVariable1;
    input localVariable2;

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

endtask
```

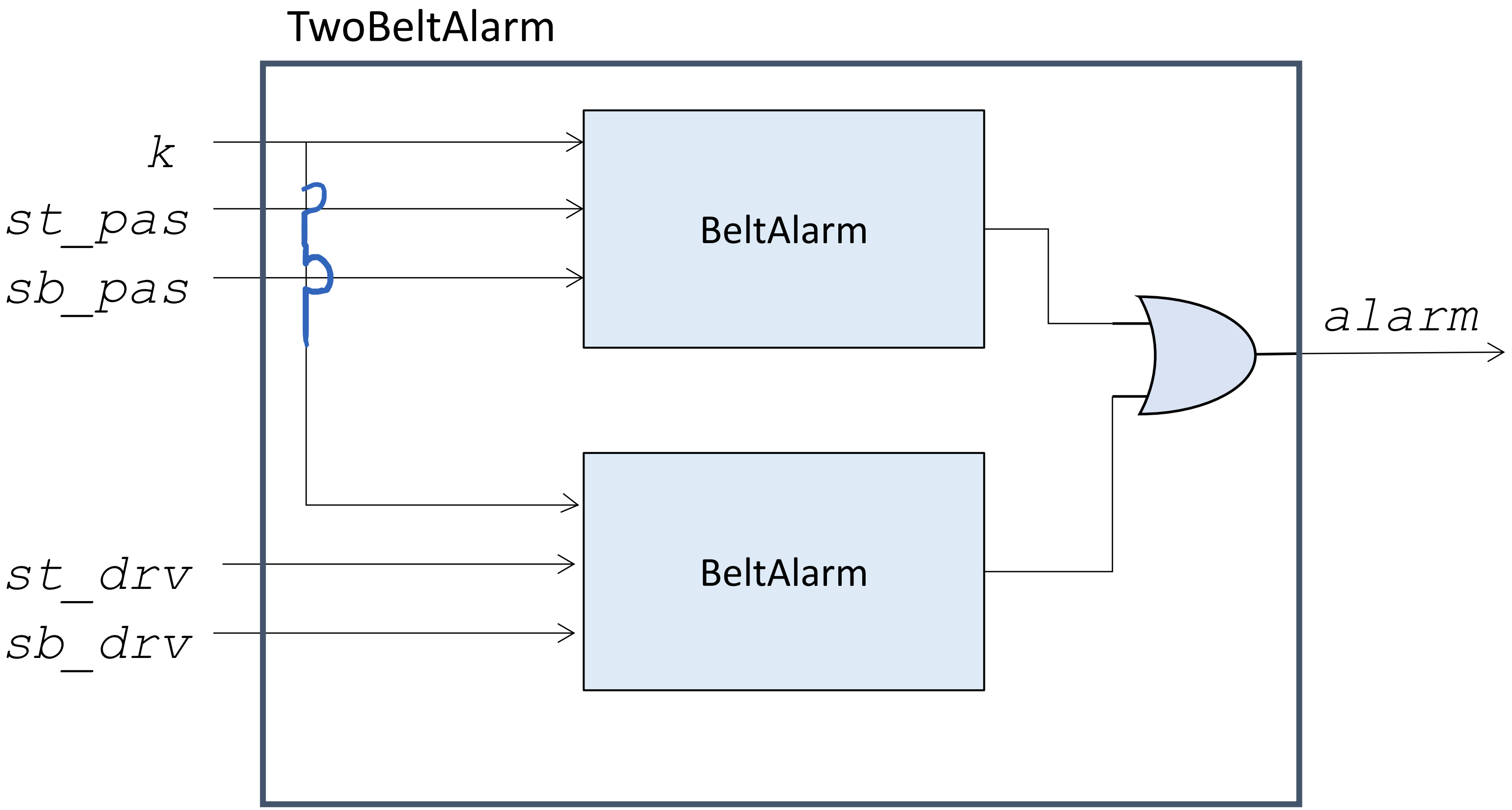
There is a function in Verilog.   We don’t use it.

Seatbelt Testbench

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

    wire n1, n2;

    and and_n1(n1, k, p);
    not not_n2(n2, s);
    and and_al(alarm, n1, n2);

endmodule
```



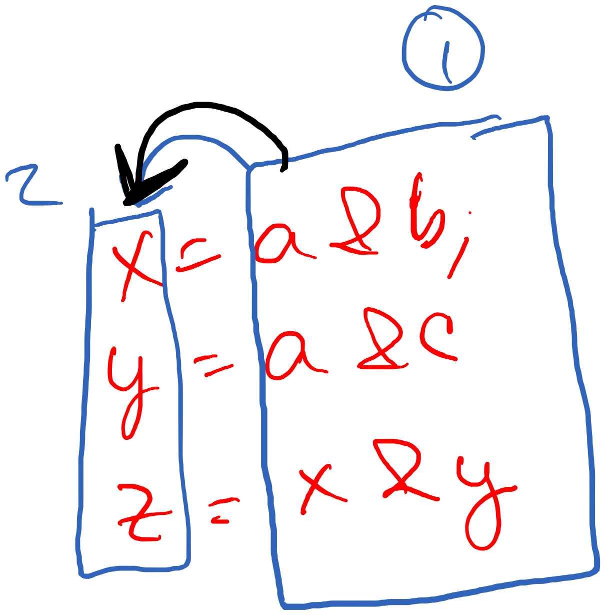
Testbench for 2 SeatBelt!



Next Time

- Continue with Verilog

Verilog



```
x = a & b;
y = a & c
z = x & y
```

Python

```
             1  1  0
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)
```

Testing a Full Adder

```
`timescale 1ns / 1ps

/// initialize FullAddr

initial
begin

     //$monitor optional

     #1 //wait 1ns
     a = 1; b = 0; ci = 0;
     #0.001 // 1ps
     assert( s == 1) else $fatal(1, ”s”);
     assert( co == 0) else $fatal(1, “co”);

     #1 //wait 1ns
     a = 1; b = 1; ci = 0;
     #0.001 // 1ps
     assert( s == 0) else $fatal(1, ”s”);
     assert( co == 1) else $fatal(1, “co”);

     $finish;
end
```

```
module FullAddr (
     input a,b,ci,
     output s, co
     );

   s = a ^ b ^ ci;
   co = a & b | ( a ^ b) & ci;

endmodule
```

return

print

$$\begin{array}{r} 1 \\ +\ 0 \\ +\ 0 \\ \hline 0\ 1 \end{array}$$

$$\begin{array}{r} 1 \\ +\ 1 \\ +\ 0 \\ \hline 1\ 0 \end{array}$$

Testing a Full Adder

```
`timescale 1ns / 1ps

/// initialize FullAddr

initial
begin

     //$monitor optional

     #1 //wait 1ns
     a = 1; b = 0; ci = 0;
     #0.001 // 1ps
     assert( s == 1) else $fatal(1, ”s”);
     assert( co == 0) else $fatal(1, “co”);

     #1 //wait 1ns
     a = 1; b = 1; ci = 0;
     #0.001 // 1ps
     assert( s == 0) else $fatal(1, ”s”);
     assert( co == 1) else $fatal(1, “co”);

     $finish;
end
```

```
module FullAddr (
    input a,b,ci,
    output s, co
    );

   s = a ^ b ^ ci;
   co = a & b | ( a ^ b) & ci;

endmodule
```

Tasks in Testing

```
module FullAddr (
     input a,b,ci,
     output s, co
     );
    s = a ^ b ^ ci;
    co = a & b | ( a ^ b) & ci;
endmodule
```

```
`timescale 1ns / 1ps

// declare a,b,ci, s, & co

FullAddr fa0 (.a(a), .b(b), .ci(ci), .s(s), .co(co));
```

✓

```
task TestOne; //set module signals to T(est) values
     input aT, bT, ciT, sT, coT;

     #1
     a = aT; b= bT; ci = ciT;
     #1
     assert( s  == sT ) else $fatal(1, “s  failed”);
     assert( co == coT) else $fatal(1, “co failed”);
endtask
```

$$\begin{array}{r} 1 \\ +\ 0 \\ +\ 1 \\ \hline 10 \end{array}$$

```
initial
begin
```

a=1   b=0   cin=1   s=1   co=0

```
     TestOne(.aT(1), .bT(0), .ciT(1), .sT(1), .coT(0));
```

.sT(~~1~~ 0), .coT(~~0~~ 1) ✓

```
     TestOne(.aT(1), .bT(1), .ciT(0), .sT(0), .coT(1));
     $finish;
end
```

a=1   b=1   cin=0   s=0   ~~co~~ co=1

Tasks in Testing

```
/// module definition
// declare a0,a1,b0,b1,ci, s0,s1,& co

TwoBitAdder tba0 (a0,a1,b0,b1,ci,s0,s1,co);

task TestTwo;
    input a1T, a0T, b1T, b0T, cinT, CoutT, s1T, s0T;
    #1
    a1=a1T; a0=a0T; b1=b1T; b0=b0T, Cin=CinT;
    #1
    assert( co ==coT) else $fatal(1, "cout failed\n");
    assert((s0==s0T) && (s1=s1T)) else $fatal(1, "sum failed\n");
endtask

               a1 a0 b1 b0 Ci Cout s1 s0
initial begin   ↓  ↓  ↓  ↓  ↓   ↓   ↓  ↓
    TestTwo( 0,0,0,0,0, 0,0,0);   // a=00 + b=00 + ci=0 => s=00 co=0
    TestTwo( 0,0,0,0,1, 0,0,1);   // a=00 + b=00 + ci=1 => s=01 co=0
    //more tests + $finish
end
```

```
module TwoBitAddr(
    input a0, a1, b0, b1, ci,
    output s0, s1, co
    );
    wire r;
    FullAddr fa0 (a0,b0,ci,s0,r);
    FullAddr fa1 (a1,b1,r,s1,co);
endmodule
```

Tasks in Testing

```
module TwoBitAddr(
    input a0, a1, b0, b1, ci,
    output s0, s1, co
    );
    wire r;
    FullAddr fa0 (a0,b0,ci,s0,r);
    FullAddr fa1 (a1,b1,r,s1,co);
endmodule
```

```
/// module definition
// declare a0,a1,b0,b1,ci, s0,s1,& co

TwoBitAdder tba0 (a0,a1,b0,b1,ci,s0,s1,co);

task TestTwo;
    input a1T, a0T, b1T, b0T, ciT;
    input coT, s0T, s1T;

    #1
    a0 = a0T; a1 = a1T; b0 = b0T; b1=b1T, ci = ciT;
    #1
    assert( (s0  == s0T) && (s1 == s1T)) else $fatal(1, “s  failed”);
    assert( co == coT) else $fatal(1, “co failed”);
endtask

initial begin
    TestTwo( 0,0,0,0,0, 0,0,0);  // 00 + 00 + 0 = 000
    TestTwo( 0,0,0,0,1, 0,0,1);  // 00 + 00 + 1 = 001
    //more tests + $finish
end
```

`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`