RV College of Engineering®

Verilog Fundamentals

Dr. Uma B.V

HDL

Hardware Description Languages (HDLs) are absolutely foundational to modern digital design.

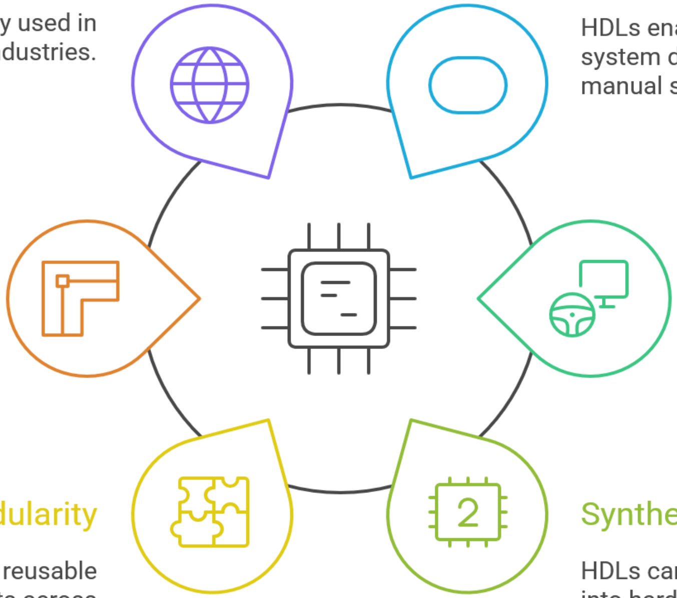
HDLs are specialized programming languages used to describe the structure, behavior, and timing of electronic circuits.

The two most widely used HDLs are

1. VHDL (VHSIC Hardware Description Language)

2. Verilog

Advantages of Hardware Description Languages



Industry Standard

HDLs are widely used in various industries.

Digital Circuit Design

HDLs enable complex system design without manual schematics.

Parallelism

HDLs inherently support parallel execution.

Simulation

HDLs allow error detection before physical fabrication.

Modularity

HDLs support reusable components across projects.

Synthesis

HDLs can be converted into hardware representations.

Significance of HDL

- **Digital Circuit Design**
  HDLs allow engineers to design complex digital systems without manually drawing schematics.
- **Simulation Before Fabrication**
  Before a chip is physically built, HDLs let designers simulate its behavior to catch errors early. This saves time, money, and headaches.
- **Synthesis to Hardware**
  HDL code can be synthesized into gate-level representations, which are then used to fabricate actual hardware (like FPGAs or ASICs).
- **Modularity and Reusability**
  Just like software, HDL designs can be modular. You can reuse components like adders, multiplexers, or memory blocks across multiple projects.
- **Parallelism**
  Unlike traditional programming languages, HDLs inherently support parallel execution, which mirrors how hardware operates in real time.
- **Industry Standard**
  HDLs are the backbone of chip design in industries ranging from consumer electronics to aerospace and defense.

RV College of Engineering®

Verilog HDL

Go, change the world®

- Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.
- Syntax is pre-defined, lowercase, identifiers that define the language constructs.
- Example : *module, endmodule, input, output wire, and, or, not* , etc.,
- Any text between two slashes (//) and the end of line is interpreted as a comment.
- Blank spaces are ignored and names are <u>case sensitive.</u>

- Comments
- `//   The rest of the line is a comment`
- `/*   Multiple line comment  */`
- `/*   Nesting /* comments */ do NOT work  */`

module: example1

A *module* is the building block in Verilog.

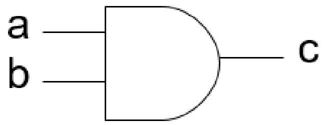
- declared by the keyword ***module***
- always terminated by the keyword ***endmodule***

Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.

&	→ bitwise AND
|	→ bitwise OR
~	→ bitwise NOT
^	→ bitwise XOR
~^ or ^~	→ bitwise XNOR

Operation on bit by bit basis

❑ Circuit



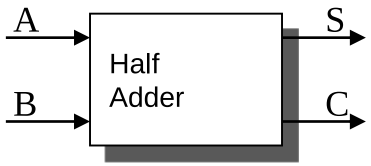
```
// A simple example              ← comment line

module and2 (c, a ,b);           ← module name
                                   port list
input a, b;
                                 ← port declarations
output c;

assign c = a & b;                ← body

end module                       ← end module
```

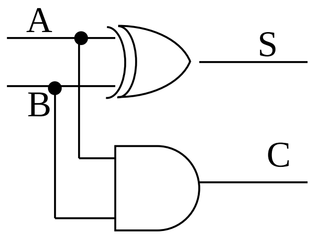
RV College of Engineering®

Go, change the world®

module: example 2

Example: Half Adder





module name

port types

```
module half_adder(S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

module contents

&	→ bitwise AND
|	→ bitwise OR
~	→ bitwise NOT
!	→ logical NOT
^	→ bitwise XOR
~^ or ^~	→ bitwise XNOR

Operation on bit by bit basis

Identifiers

Formed from {[A-Z], [a-z], [0-9], _, $},

but ..

.. can't begin with $ or [0-9]

`myidentifier`	allowed
`m_y_identifier`	allowed
`3my_identifier`	Not allowed
`$my_identifier`	Not allowed
`_myidentifier$`	allowed

Case sensitivity

`myid` $\neq$ **`Myid`**

module Port List

type of port
input
output
inout (bidirectional)

Multiple ways to declare the ports of a module

Scalar (single bit) - don't specify a size

```
input        cin;
```

```
module Add_half(c_out, sum, a, b);
     output sum, c_out;
     input a, b;
     …
endmodule
```

```
module Add_half(output c_out, sum,
input a, b);
     …
endmodule
```

Vector (multiple bits) - specify size using range
Range is MSB to LSB (left to right)

```
output [7:0] OUT;
input  [0:4] IN;
```

```
module xor_8bit(xout, a, b);

     output [7:0] xout;

     input [7:0] a, b;

     …

endmodule
```

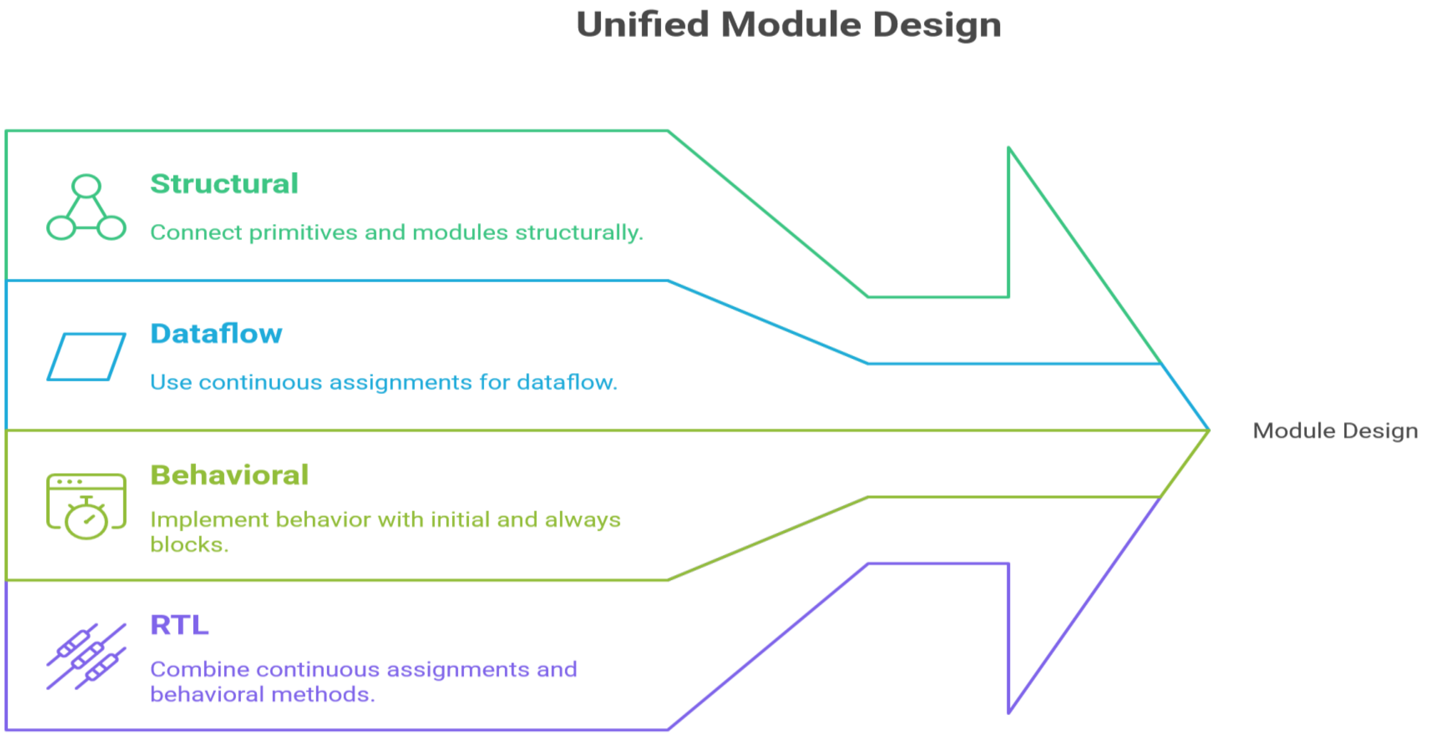
```
module xor_8bit(output [7:0] xout,  input [7:0] a, b);

                    …

endmodule
```

Basic gates in Verilog

```
module basic_gate(C, A, B);
output [3:0]C;
input [3:0]A,B;
wire C, A, B;

assign C[0] = A[0] & B[0];    // AND GATE
assign C[1] = A[1] | B[1];    // OR GATE
assign C[2] = A[2] ^ B[2];    // XOR GATE
assign C[3] = !(A[3] & B[3]);// NAND GATE
endmodule
```

//Multiple inputs and output using vector

Module style


Unified Module Design
Structural
Connect primitives and modules structurally.
Dataflow
Use continuous assignments for dataflow.
Behavioral
Implement behavior with initial and always blocks.
RTL
Combine continuous assignments and behavioral methods.
Module Design

Dataflow model

- Dataflow models describe **combinational circuits** by their *function* rather than by their gate structure.
- A common form of dataflow modeling of combinational logic uses **concurrent signal assignment statements** and built-in language operators to express how signals are assigned values.

Example

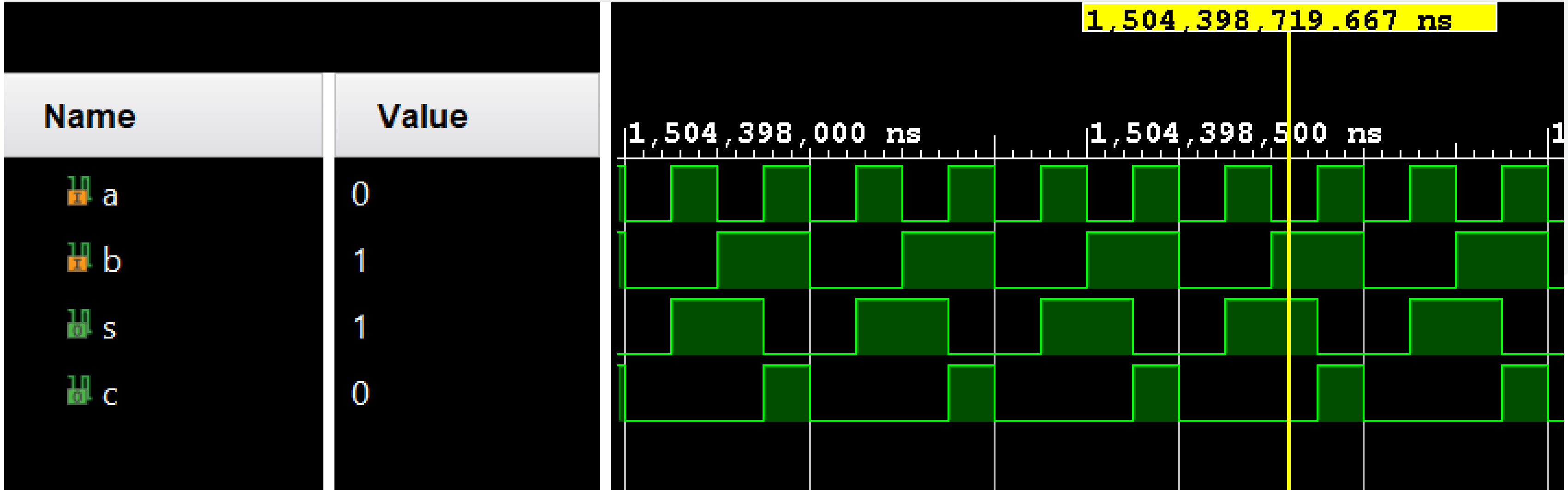
```
module half_adder(S, C, A, B);
output S, C;
input A, B;
wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

RV College of Engineering®

Go, change the world®

Half adder simulation

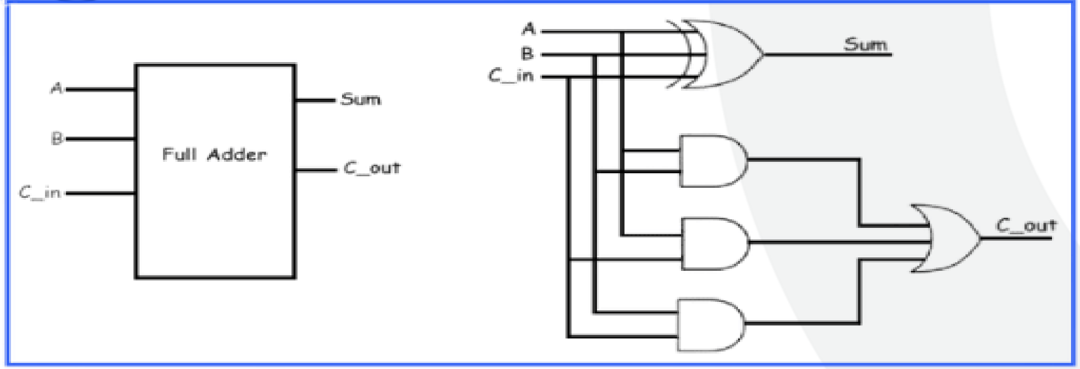
```
module HA1(
     input a,b,
     output s,c
     );
     assign s = a^b;
     assign c = a&b;
endmodule
```



RV College of Engineering®

Dataflow model-Full adder

Go, change the world®



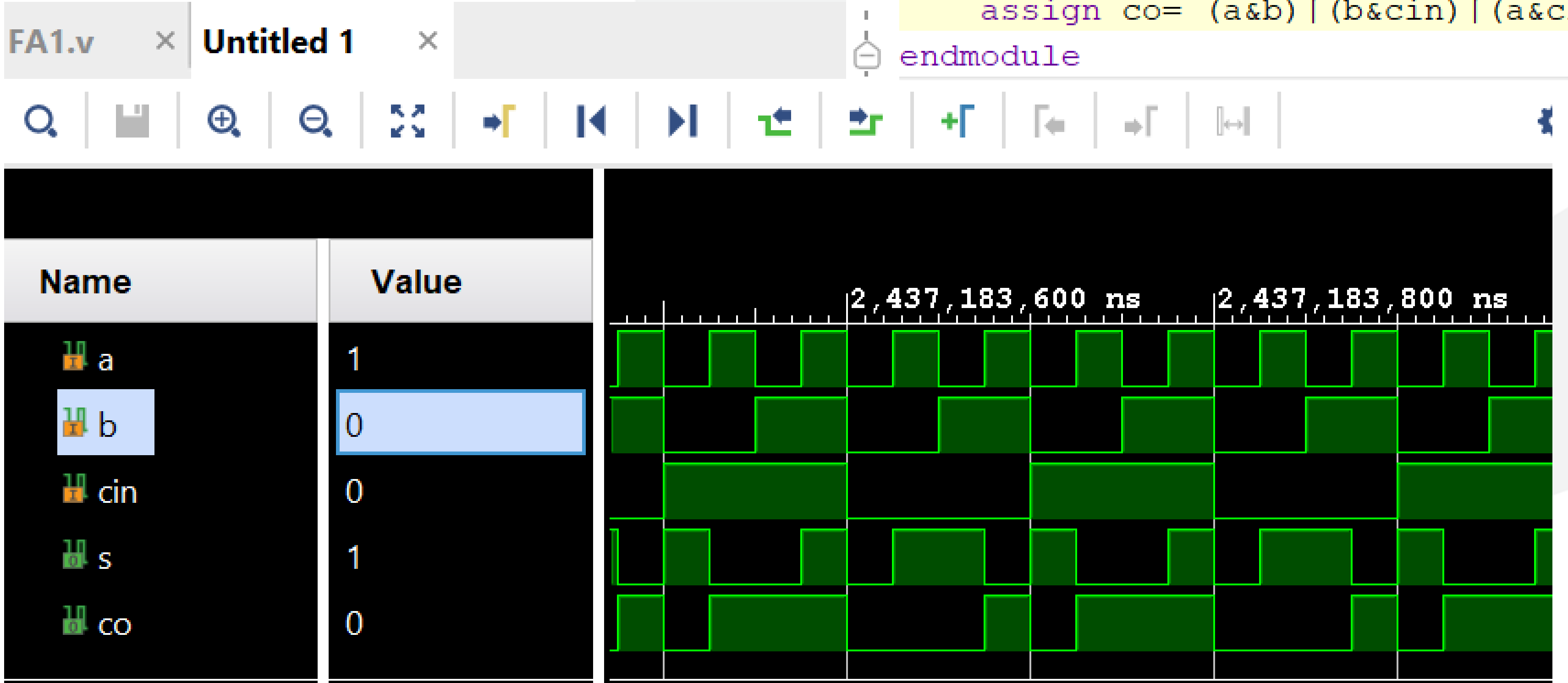
Example 3

```
module full_adder(Sum, C_out, A, B,Cin);
output Sum, C_out;
input A, B, Cin;
//wire Sum, C_out, A, B, Cin;
assign Sum = A ^ B ^ Cin;
assign C_out = (A & B)|(B & Cin)|(A &
Cin);
endmodule
```

RV College of Engineering®

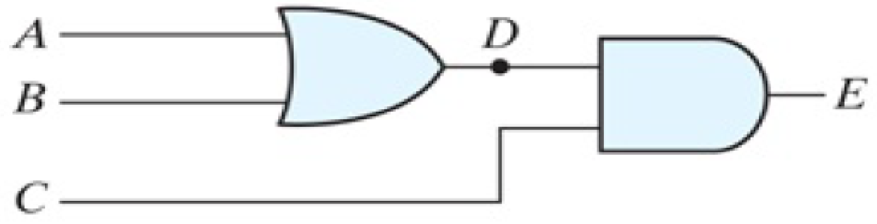
Full adder simulation

```
module FA1(
      input a,b,cin,
      output s,co
      );
      assign s = a^b^cin;
      assign co= (a&b)|(b&cin)|(a&cin);
endmodule
```



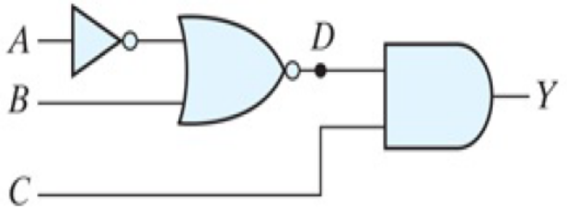
Dataflow model- Logic circuit

```
module  or_and (
 output E,
 input  A, B, C
);
 wire  D;
 assign  D = A || B;     // | is logical “OR” operator
 assign  E = C && D;     // & is the logical “AND” operator
// This is a single-line comment
/* The text here and below
   form a multi-line comment
*/
endmodule
```



Dataflow model-Boolean function

1. Write a continuous assignment statement that describes *Y* in the logic diagram in Fig. PE3.12, where *A*, *B*, *C*, and *D* are Boolean variables.



Answer: assign Y=(!((!A)| |B))&&C

Boolean Function

E=A+BC+B′D F=B′C+BC′D′

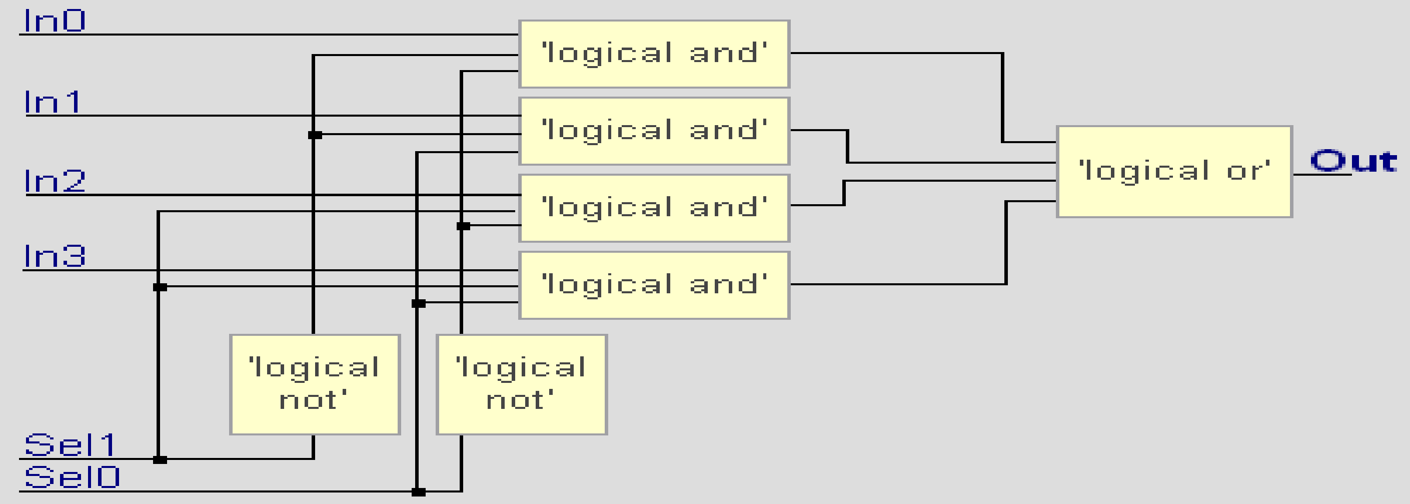
Verilog Code

```
module Circuit_Boolean_CA (E, F, A, B, C, D);
 output E, F;
 input  A, B, C, D;
 assign E = A || (B && C) || ((!B) && D);
 assign F = ((!B) && C) || (B && (!C) && (!D));
 endmodule
```

RV College of Engineering®

Data flow model- MUX 4:1

Go, change the world®

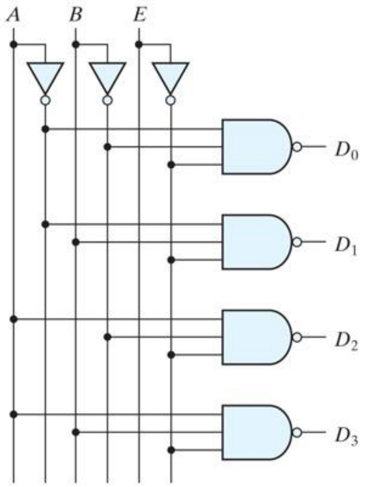


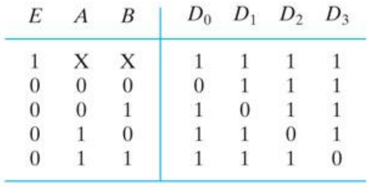
```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

RV College of Engineering®

Dataflow Model of 2:4 decoder

Go, change the world®





E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

```
module decoder_2to4_df (output [0: 3] D, input
A, B, enable );
assign D[0] = !((!A) && (!B) && (!enable)),
D[1] = !((!A) && B && (!enable)),
D[2] = ((A) && (! B) && (!enable)),
D[3] = !(A && B && (!enable));
endmodule
```

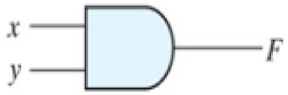
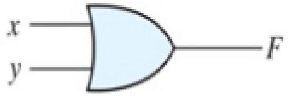
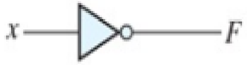
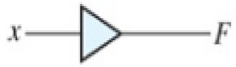
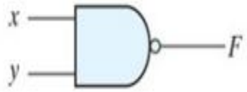
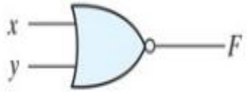
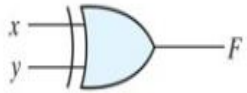
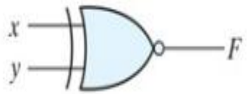
Symbol	Operation	Symbol	Operation
+	binary addition		
−	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT

= =	equality
>	greater than
<	less than
{ }	concatenation

RV College of Engineering®

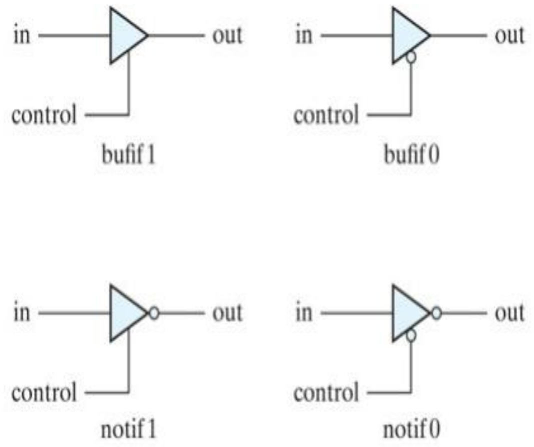
Verilog - predefined primitives

- Verilog includes 12 basic logic gates as predefined primitives.
- Four of these primitive gates are of the three-state type.
- The other eight are the basic gates.
- They are declared with the lowercase keywords **and**, **nand, or, nor, xor, xnor**, **not, and buf**.
- Primitives such as **and** are n-input primitives, because they can have any number of scalar inputs (e.g., a three-input and primitive).
- The **buf** and **not** primitives are n-output primitives because a single input to a buf or not gate can drive multiple outputs.

Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

Verilog: four types of predefined three-state gates

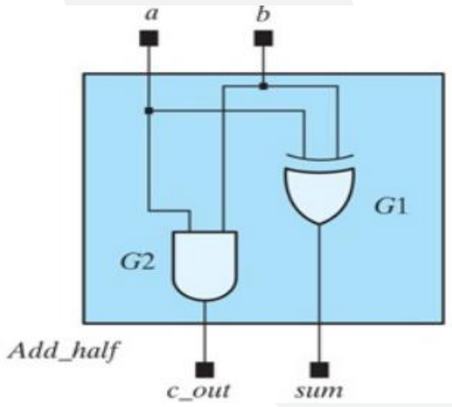
`gate name (output, input, control);`



- Verilog has four types of predefined three-state gates, as shown in Fig.
- The bufif1 gate behaves like a normal buffer if c o n t r o l = 1. The output goes to a high-impedance state z when c o n t r o l = 0.
- The bufif0 gate behaves in a similar fashion, except that the high-impedance state occurs when c o n t r o l = 1.
- The two notif gates operate in a similar manner, but the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement
- g a t e n a m e (o u t p u t , i n p u t , c o n t r o l) ;

RV College of Engineering®

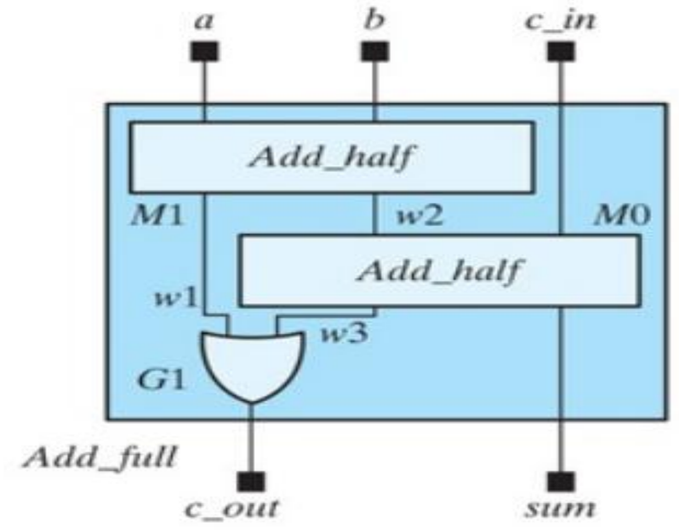
Structural design: Half adder using Gates



```
module Add_half (input a, b, output c_out, sum),
  xor G1(sum, a, b);          // Gate instance names are option
  and G2(c_out, a, b);
endmodule
```

RV College of Engineering®

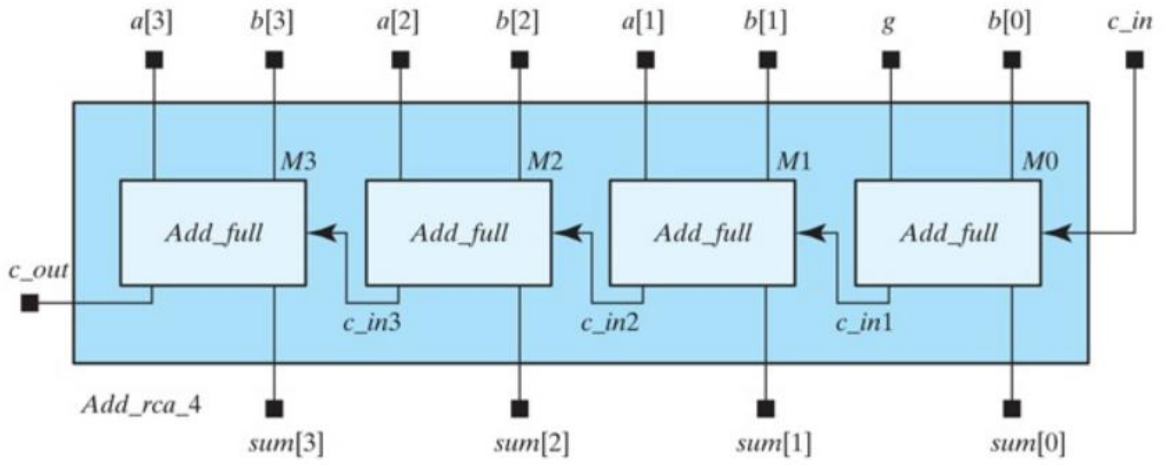
Structural design: Full Adder using two half adder



```
module Add_full (input a, b, c_in, output c_out, sum);   // see
Fig.
4.8

  wire w1, w2, w3;           // w1 is c_out; w2 is sum
  Add_half M1 (a, b, w1, w2);
  Add_half M0 (w2, c_in, w3, sum);
  or (c_out, w1, w3);
endmodule
```

RV College of Engineering®

Structural design: 4 bit Adder using 4 full adders



```
module Add_rca_4 (input [3:0] a, b, input c_in output c_out, ou
  wire c_in1, c_in3, c_in4;          // Intermediate carries
  Add_full M0 (a[0], b[0], c_in, c_in1, sum[0]);
  Add_full M1 (a[1], b[1], c_in1, c_in2, sum[1]);
  Add_full M2 (a[2], b[2], c_in2, c_in3, sum[2]);
  Add_full M3 (a[3], b[3], c_in3, c_out, sum[3]);
endmodule
```

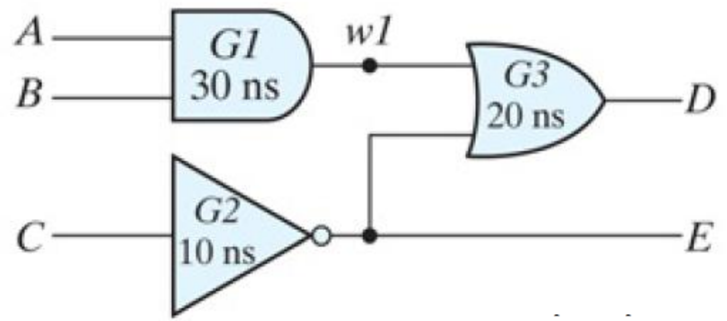
RV College of Engineering®

Data flow model of 4 bit binary Adder

```
module binary_adder ( output C_out, output [3: 0] Sum, input [3: 0] A, B, input C_in
);
assign {C_out, Sum} = A + B + C_in // Continuous assignment statement
endmodule
```

RV College of Engineering

Structural design: Logic Circuit with propagation delay



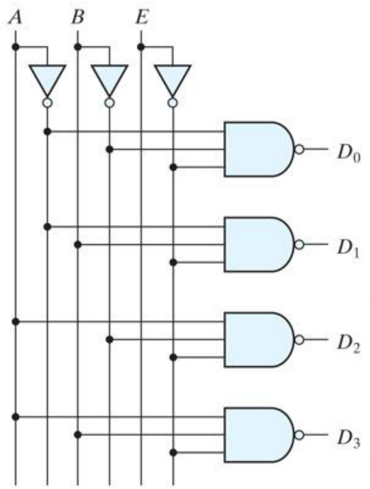
	Time Units (ns)	A	B	C	E	w1	D
Initial	—	0	0	0	1	0	1
Change	—	1	1	1	1	0	1
	10	1	1	1	0	0	1
	20	1	1	1	0	0	1
	30	1	1	1	0	1	0
	40	1	1	1	0	1	0

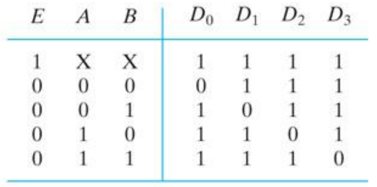
```
module and_or_prop_delay (
   input A, B, C;
   output D, E);
);
   wire w1;

   and G1 #30 (w1, A, B);
   not G2 #10 (E, C);
   or G3 #20 (D, w1, E);
endmodule
```

RV College of Engineering®

Structural Model of 2:4 decoder

Go, change the world®





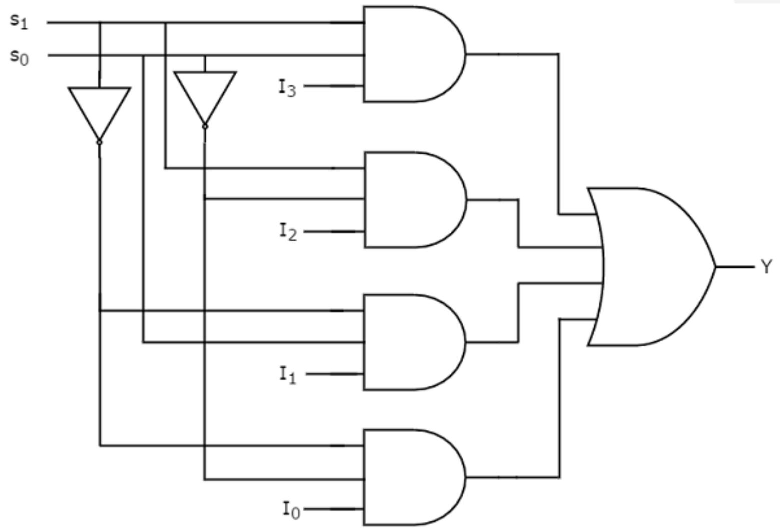
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

```
module decoder_2to4_gates (D, A, B, enable);
output [0: 3] D;
input A, B;
input enable;
wire A_not, B_not, enable_not;
not
G1 (A_not, A), // Comma-separated list of primitives
G2 (B_not, B),
G3 (enable_not, enable);
nand
G4 (D[0], A_not, B_not, enable_not),
G5 (D[1], A_not, B, enable_not),
G6 (D[2], A, B_not, enable_not),
G7 (D[3], A, B, enable_not);
endmodule
```

RV College of Engineering®

Structural Model of 4:1 MUX

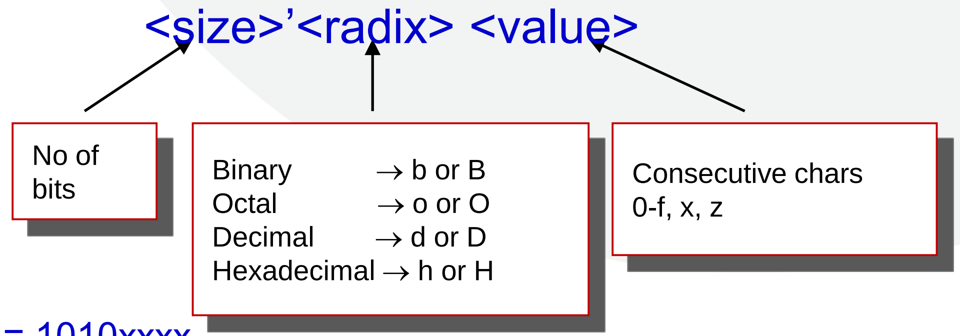
Go, change the world®



```
module m41(y, I0, I1, I2, I3, s0, s1);
output y;
input I0, I1, I2, I3, s0, s1;
wire sobar, s1bar, T1, T2, T3, T4;

not
G1(s0bar, s0),
G2(s1bar, s1);
and
G3(T1,I0, s0bar, s1bar),
G4(T2,I1, s0bar, s1),
G5(T3,I2, s0, s1bar),
G6 T4,I3, s0, s1);
Or G7(y, T1, T2, T3, T4);
endmodule
```

Value and numbers in Verilog

0	represents low logic level or false condition
1	represents high logic level or true condition
x	represents unknown logic level
z	represents high impedance logic level



<size>’<radix> <value>

No of bits

Binary	$\rightarrow$ b or B
Octal	$\rightarrow$ o or O
Decimal	$\rightarrow$ d or D
Hexadecimal	$\rightarrow$ h or H

Consecutive chars 0-f, x, z

- 8’h ax = 1010xxxx
- 12’o 3zx7 = 011zzzxxx111

Number example

Number	Decimal Equivalent	Actual Binary
4'd3	3	0011
3'hA	10	010
8'o26	22	00010110
5'b111	7	00111
8'bx1101	-	xxxx1101
'o7	7	00000………111(32 bits)
10	10	????

Numbers with MSB of **x** *or z extended with that value*

RV College of Engineering®

Numbers in verilog

Go, change the world®

- can insert "_" for readability
  - 12'b 000_111_010_100
  - 12'b 000111010100
  - 12'o 07_24

Represent the same number

- Bit extension
  - MS bit = 0, x or z $\Rightarrow$ extend this
    - 4'b x1 = 4'b xx_x1
  - MS bit = 1 $\Rightarrow$ zero extension
    - 4'b 1x = 4'b 1xxx

- If *size* is ommitted it is inferred from the *value* or takes the simulation specific number of bits or takes the machine specific number of bits

- If *radix* is ommitted then decimal is assumed

15 = <size>'d 15

Logical Operators

- && → logical AND
- || → logical OR
- ! → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 6;            A && B → 1 && 0 → 0
B = 0;     →      A || !B → 1 || 1 → 1
C = x;            C || B → x || 0 → x
```

Bitwise Operators (i) & Reduction operator

&	$\rightarrow$ bitwise AND
|	$\rightarrow$ bitwise OR
~	$\rightarrow$ bitwise NOT
^	$\rightarrow$ bitwise XOR
~^ or ^~	$\rightarrow$ bitwise XNOR

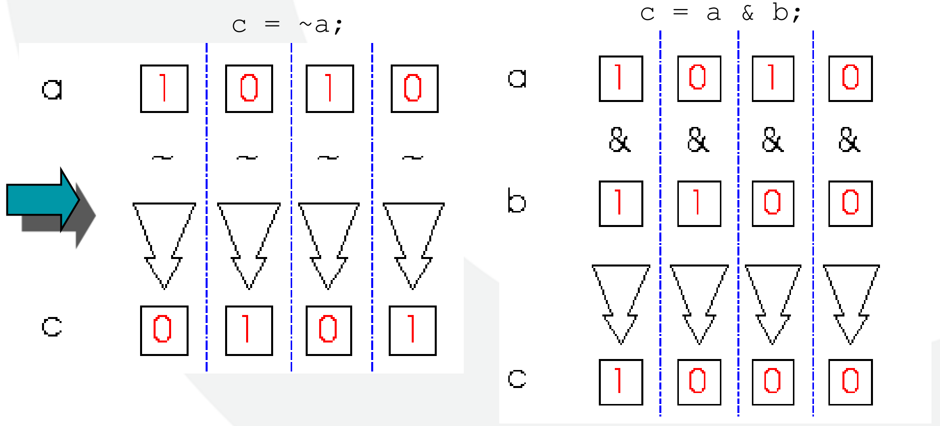
Operation on bit by bit basis

- & $\rightarrow$ AND
- | $\rightarrow$ OR
- ^ $\rightarrow$ XOR
- ~& $\rightarrow$ NAND
- ~| $\rightarrow$ NOR
- ~^ or ^~ $\rightarrow$ XNOR
- One multi-bit operand $\rightarrow$ One single-bit result

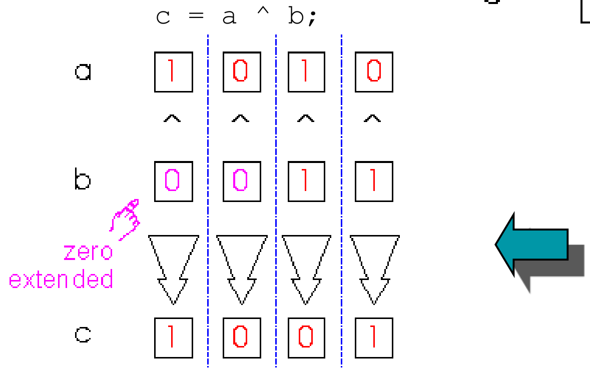
```
a = 4'b1001;
c = |a; // c = 1|0|0|1 = 1
 Y=  &A;//  Y=  1&0&0&1 = 0
```

Bitwise Operators (ii)

-
  ```
  a = 4'b1010;
  b = 4'b1100;
  ```



```
a = 4'b1010;
b = 2'b11;
```



RV College of Engineering®

Concatenation Operator

Go, change the world®

- {op1, op2, ..} → concatenates op1, op2, .. to single number
- Operands must be sized !!

```
reg a;
reg [2:0] b, c;
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};                   // catx = 1_010_101
caty = {b, 2'b11, a};    // caty = 010_11_1
catz = {b, 1};           // WRONG !!
```

- Replication ..

```
catr = {4{a}, b, 2{c}}; // catr = 1111_010_101101
```

Shift Operators

- >> → shift right
- << → shift left

- Result is same size as first operand, **always zero filled**

```
a = 4'b1010;
...
d = a >> 2; // d = 0010
c = a << 1; // c = 0100
```

RV College of Engineering®

Net /Wire

Go, change the world®

Nets

•**Purpose**: Represent physical connections between components.

•**Behavior**: Do **not store** values; instead, they reflect the value driven by connected sources.

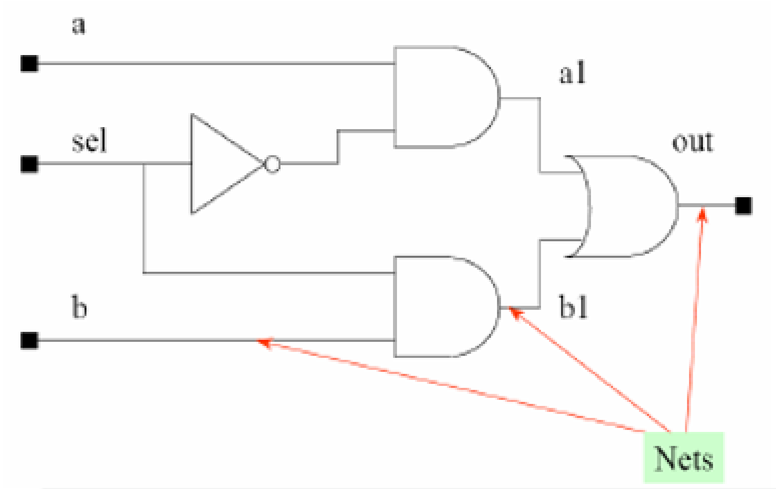
•**Common Types:** wire, tri, wand, wor, trireg,

•**Usage Scenarios**:

•Connecting module ports.

•Left-hand side of continuous assignments (assign).

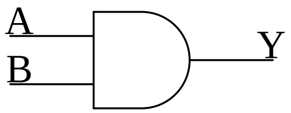
•Signals driven by gates or other modules.



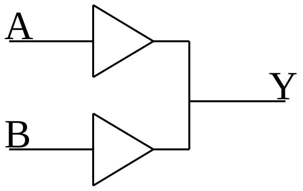
```
wire a, b, c;
assign c = a & b;
```

Nets (ii)

Y is evaluated, ***automatically***, every time A or B changes



```
wire Y;   // declaration
assign Y = A & B;
```

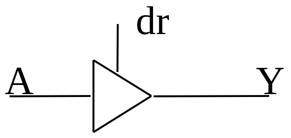


```
wand Y;   // declaration
assign Y = A;
assign Y = B;
```

Y		A = 0	A = 1
B	0	0	0
B	1	0	1

```
wor Y;   // declaration
assign Y = A;
assign Y = B;
```

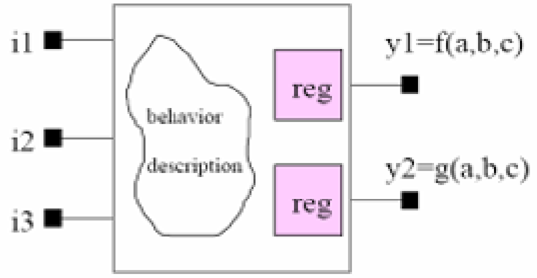
Y		A = 0	A = 1
B	0	0	1
B	1	1	1



```
tri Y;   // declaration
assign Y = (dr) ? A : z;
```

Registers (reg)

- **Purpose**: Represent **storage elements** inside procedural blocks.
- **Behavior**: Retain their value until explicitly changed.
- **Common Types**: reg, integer, real, time.
- **Usage Scenarios**
- Inside always block
- Used in the design of latches, flipflps etc
- Example



```
module D_ff(q,d,clk,res);
output q;
input d,clk,res;
reg q;
```

```
always @(posedge res or negedge clk);
If (res)
q=1'b0;
else
q=d;
endmodule
```

Register

- Variables that stores values
- Do not represent real hardware
- Assigned a value only in procedural statements, user defined sequential statements, tasks or function
- Reg: can not be output of primitive gates, input, inout, output port of module or target of continuous assignment statement
- Example: `reg`

```
reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0
```

- Register values are updated explicitly!!

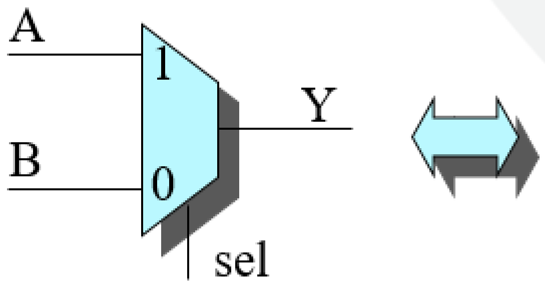
RV College of Engineering®

Go, change the world®

Conditional Operator

- `cond_expr ? true_expr : false_expr`

- 2-to-1 mux ..

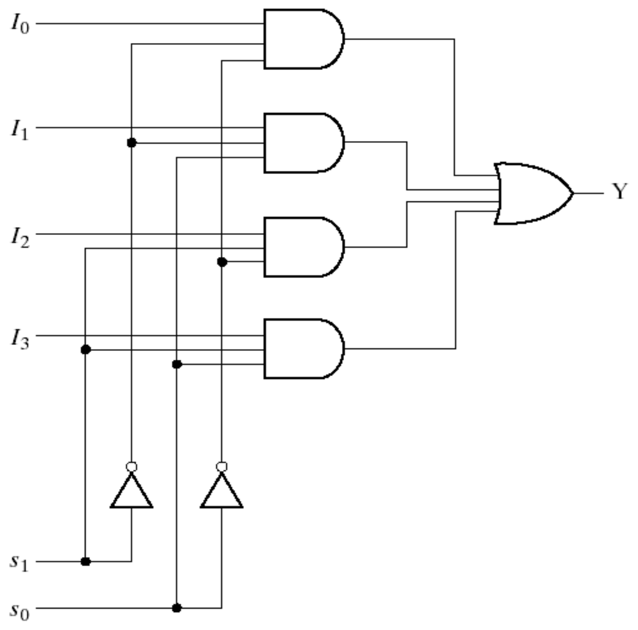


```
Y = (sel)? A : B;
```

RV College of Engineering®

4:1 MUX using conditional operator

Go, change the world®



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

```
module mux4to1(output y, input i0, i1, i2,
i3, s0, s1);
assign y = s1 ? ( s0 ? i3:i2) : (s0 ? i1:i0);
//nested conditional operator
endmodule
```

Assignment statement

- Regular continuous assignment

assign y = in1 & in2;
assign addr[15:0] = addr1[15:0] II addr2[15:0]
assign {cout, sum[3:0]} = a[3:0] + b[3:0] +cin;

- Implicit continuous assignment

wire y = in1 & in2;

- With delay

wire # 10 y = in1 & in2
wire y;
assign #10 y = in1 & in2

Syntax:

```
assign #del <id> = <expr>;
```

Propagation delay(inertial) can be associated with a continuous assignment
Facilitates logic function and timing characteristic of the design to be same as hardware circuit

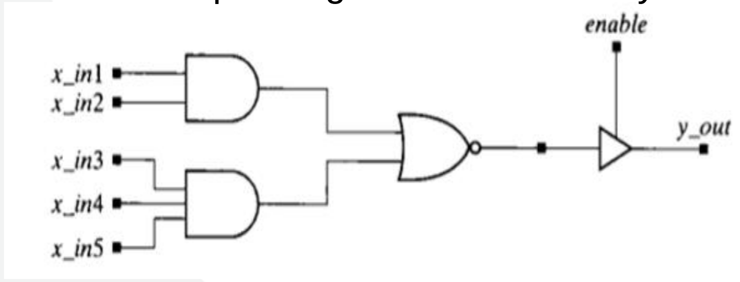
RV College of Engineering®

Continuous Assignements

Go, change the world®

- Where to write them:
  - inside a module
  - outside procedures
- Properties:
  - they all execute in parallel
  - are order independent
  - are continuously active

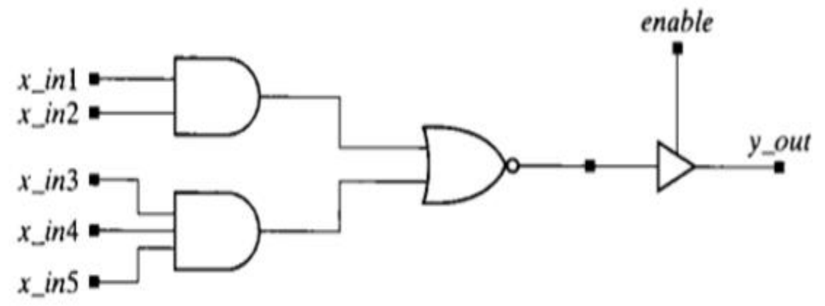
Example: Logic circuit with delay



```
module aoi (xin1,xin2,xin3,xin4,xin5,e,yout);
input xin1,xin2,xin3,xin4,xin5,e;
output yout;
wire #1 y1= (xin1 & xin2);
wire #1 y2= (xin3 & xin4 & xin5):
wire #1 y3= !(y1 I y2);
assign yout = e ? y3: 1b’z;
endmodule
```

With Conditional operator ?

- Same as if then else
- If condition true: expression after ? Is evaluated and assigned to target
- If condition is false: expression after : is evaluated and assigned to target

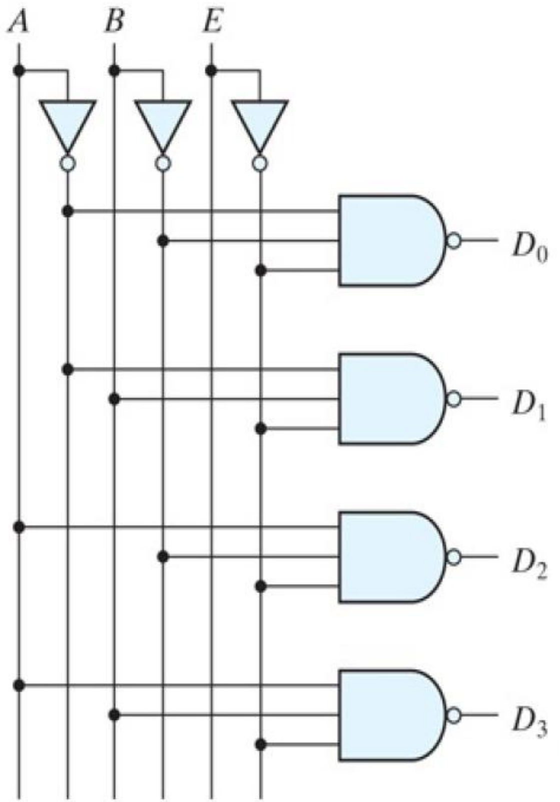


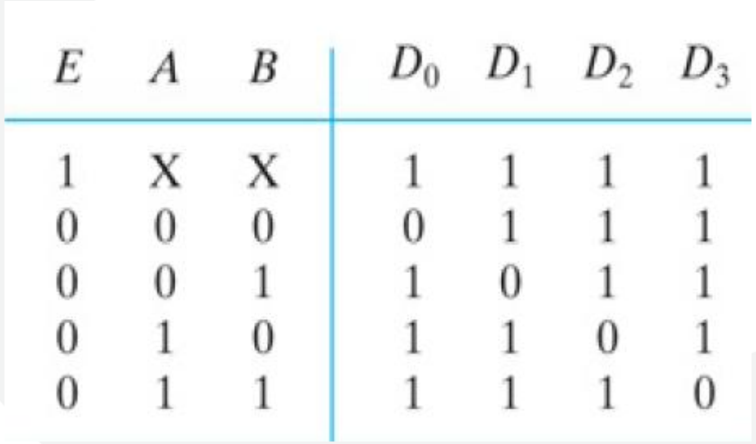
```
module aoi (xin1,xin2,xin3,xin4,xin5,e,yout);
input xin1,xin2,xin3,xin4,xin5,e;
output yout;
assign yout = e ? ! ((xin1 && xin2)ii(xin3 && xin4 &&
xin5)): 1b'z;
endmodule
```

RV College of Engineering®

Structural design- 2:4 Decoder

Go, change the world®



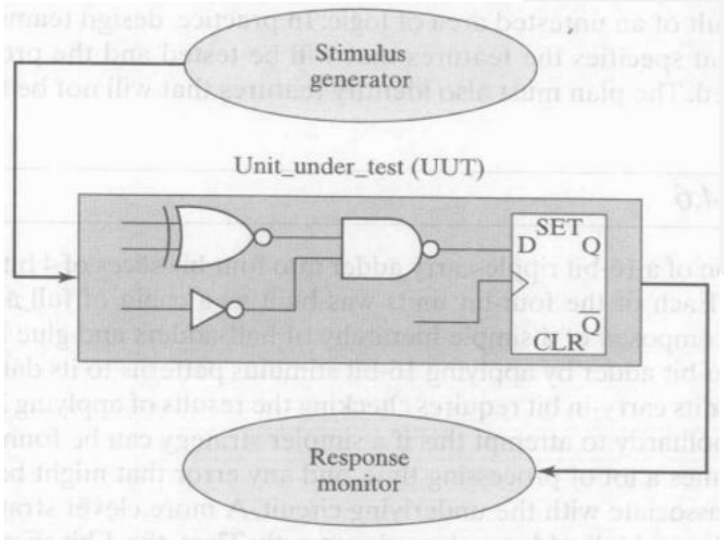


E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

- Write a continuous assignment statement that is equivalent to the logic of *G*4 in *decoder_2x4_gates*.
- **Answer: assign** D[0]=! ((!A) && (!B) && (!enable));

Testing methodology

A logic circuit must be tested systematically to ensure that all of its logic has been exercised and functionally correct



Test bench: applies stimulus patterns to the circuit and displays output waveforms

Separate verilog module which resides at the top of the design

Testing environment consists of stimulus generator, response monitor and an instantiation of the unit under test

Testing methodology - cntd

Stimulus generator: verilog statement to define patterns

Response monitor: selectively gathers data on signals within the design and displays in text or graphical format

Simulator task:

(1)checks source code,

(2)reports any violations of the language's syntax

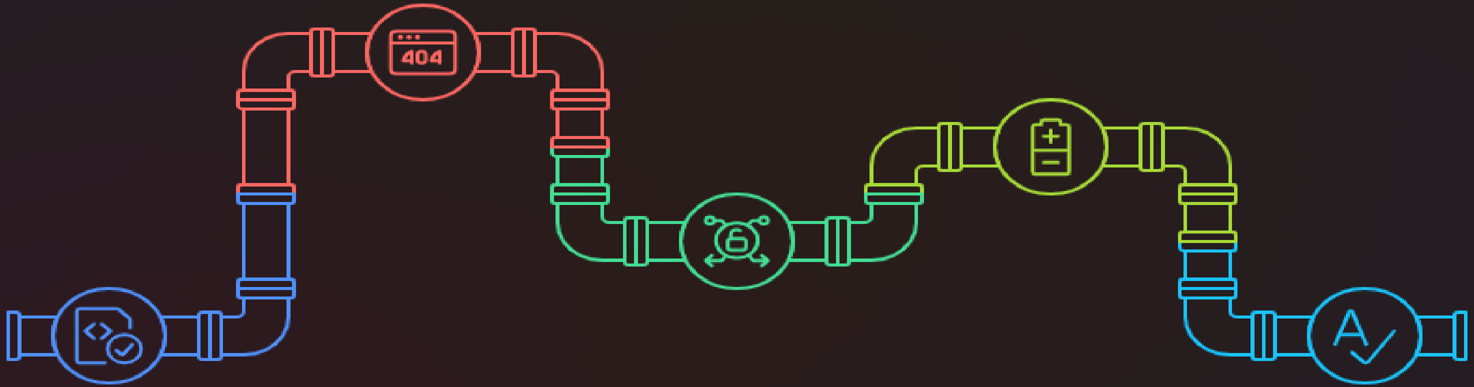
(3) determines whether the number of ports specified by a model are actually connected to it in an instantiation

(4)simulates the behavior of the circuit under the application of input signals that are defined in testbench

Syntax errors must be eliminated before a simulation can run

Testing methodology - cntd

Simulation Process Sequence



01 Check Source Code
Verifies the integrity of the source code

02 Report Syntax Violations
Identifies and reports any syntax errors

03 Verify Port Connections
Ensures all specified ports are correctly connected

04 Simulate Circuit Behavior
Runs the simulation to observe circuit behavior

05 Eliminate Syntax Errors
Corrects any identified syntax errors

Signal generators for testbench

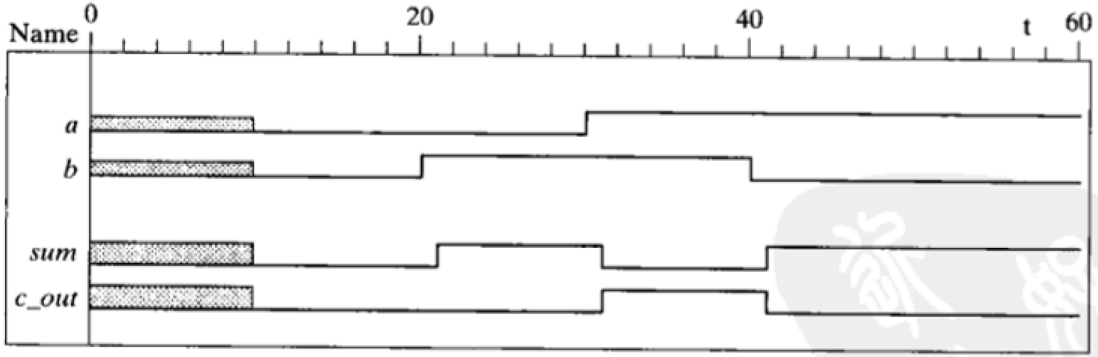
- Initial declares a single pass behavior that begins executing when the simulator is activated at tsim=0
- Statements within begin and end are called procedural statements
- = operator is procedural assignment
- Statements executes sequentially from top to bottom
- # is a delay control operator: suspends the execution of that statement and hence subsequent statement until specified time

```
module  halfadd;
reg a,b;
wire  carry, sum;
halfadd     u1(carry, sum, a ,b);
```

```
Initial
begin
#10 a=0'b0;b=0'b0;
#10 a=0'b0;b=1'b0
#10 a=1'b0;b=0'b0 ;
#10 a=1'b0;b=1'b0;
#100   $finish;
end
endmodule
```

Testbench

- Reg variable takes the value x in the beginning of simulation until it is loaded with different values by explicit statement
- Each statement is delayed by 10 unit
- Delay accumulates (40 u)
- Simulation terminates after 100 u



Writing Testbenches

```
module test;
reg a, b, sel;

mux m(y, a, b, sel);

initial begin
  $monitor($time,, “a = %b b=%b sel=%b y=%b”,  a,  b, sel, y);
    a = 0; b= 0; sel = 0;
  #10 a = 1;
  #10 sel = 1;
  #10 b = 1;
  #10 b = 0;


end
```

Inputs to device under test

Device under test

$monitor is a built-in event driven “printf”

Stimulus generated by sequence of assignments and delays

RV College of Engineering®

Go, change the world®

Testbench-AND gate

```
module test1;
reg a,b;
wire y;
ANDGATE  u1(a,b,y);
initial
begin
a=1'b0;b=1'b0;
#10 a=1'b0;b=1'b1;
#10 a=1'b1;b=1'b0;
#10 a=1'b1;b=1'b1;
#100 $finish;
end
endmodule
```

RV College of Engineering®

Full adder test bench

```
timescale 1ns / 1ps
 module full_adder_tb;
   // Inputs
  reg a;
  reg b;
  reg cin;
   // Outputs
  wire sum;
  wire cout;

  // Instantiate the Unit Under Test
(UUT)
 full_adder uut (.a(a), .b(b),
.cin(cin), .sum(sum),.cout(cout)  );
 initial begin
// Apply all input combinations
  a = 0; b = 0; cin = 0; #10;
  a = 0; b = 0; cin = 1; #10;
  a = 0; b = 1; cin = 0; #10;
  a = 0; b = 1; cin = 1; #10;
  a = 1; b = 0; cin = 0; #10;
  a = 1; b = 0; cin = 1; #10;
  a = 1; b = 1; cin = 0; #10;
  a = 1; b = 1; cin = 1; #10;
  $finish;
end
endmodule
```