

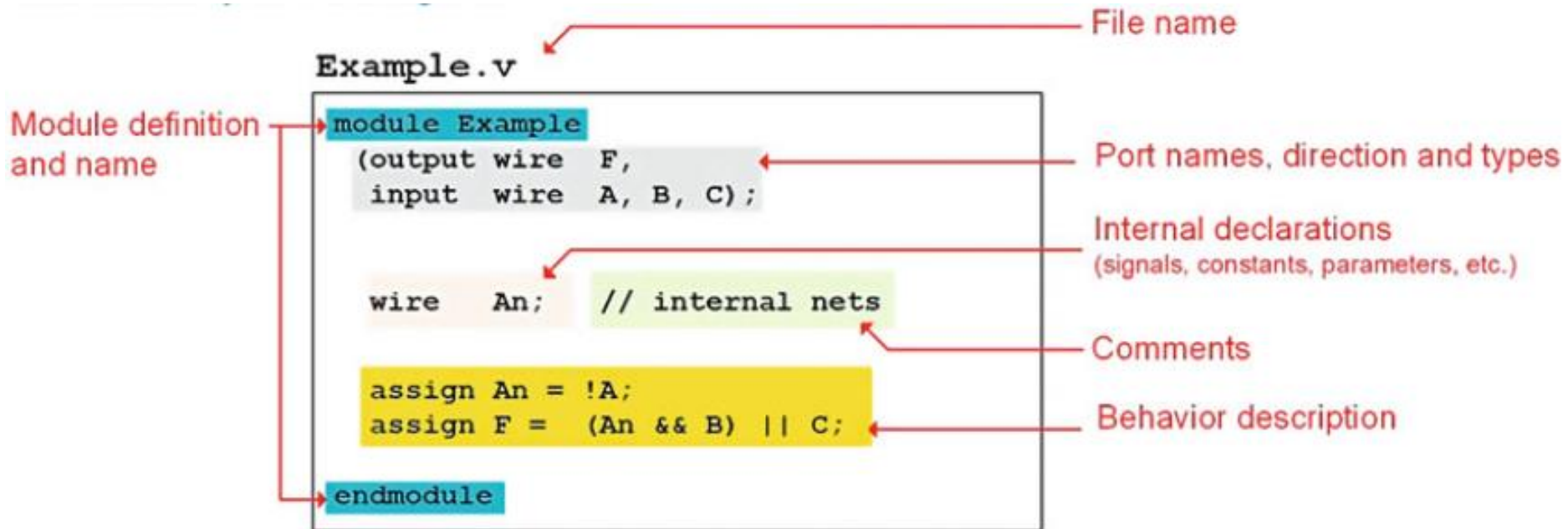
Verilog HDL: A solution for Everybody

Pravin Zode

Outline

- Structure of Verilog Program
- Ports, Signals, Operators
- Gate Level Primitives
- Continuous Signal Assignment
- Assignments with delay

Structure of Verilog program



Structure of Verilog program

- All systems in Verilog are encapsulated inside a module

```
module module_name (port_list);                                // Pre Verilog-2001
    // port_definitions
    // module_items
endmodule
```

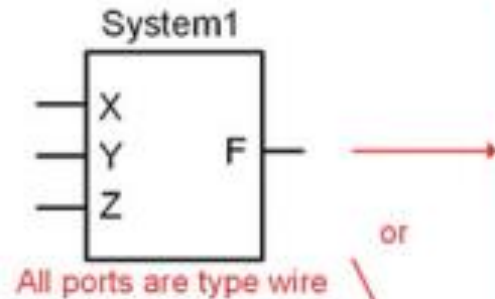
or

```
module module_name (port_list and port_definitions);          // Verilog-2001 and after
    // module_items
endmodule
```

Port Definitions

- The first item in a module is the definition of inputs and outputs (ports).
- Each port must have:
 - User-defined name (case-sensitive, must start with an alphabetic character).
 - Direction: input, output, or inout.
 - Type: Wires, Registers, or Integers (only these are synthesizable).
 - Multiple ports of the same type and direction can be listed on the same line, separated by commas.

Port Definitions

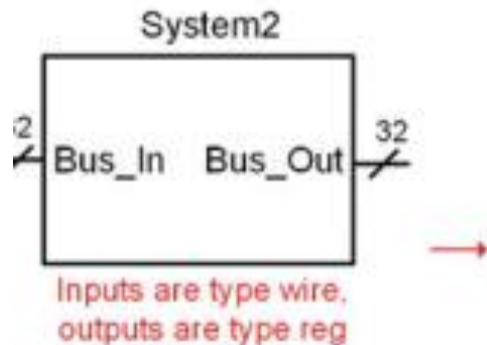


Pre Verilog-2001 Approach: Port names are listed after the module name with directions and types listed separately within the module.

```
module System1 (F, X, Y, Z);  
  
    output F;           // Port directions  
    input  X, Y, Z;  
  
    wire F;             // Port types  
    wire X, Y, Z;  
  
    //-- module items go here...  
  
endmodule
```

Post Verilog-2001 Approach: Port names, directions, and types are listed after the module name.

```
module System1 (output wire F,  
               input  wire X, Y, Z);  
  
    //-- module items go here...  
  
endmodule
```



Post Verilog-2001 Approach:

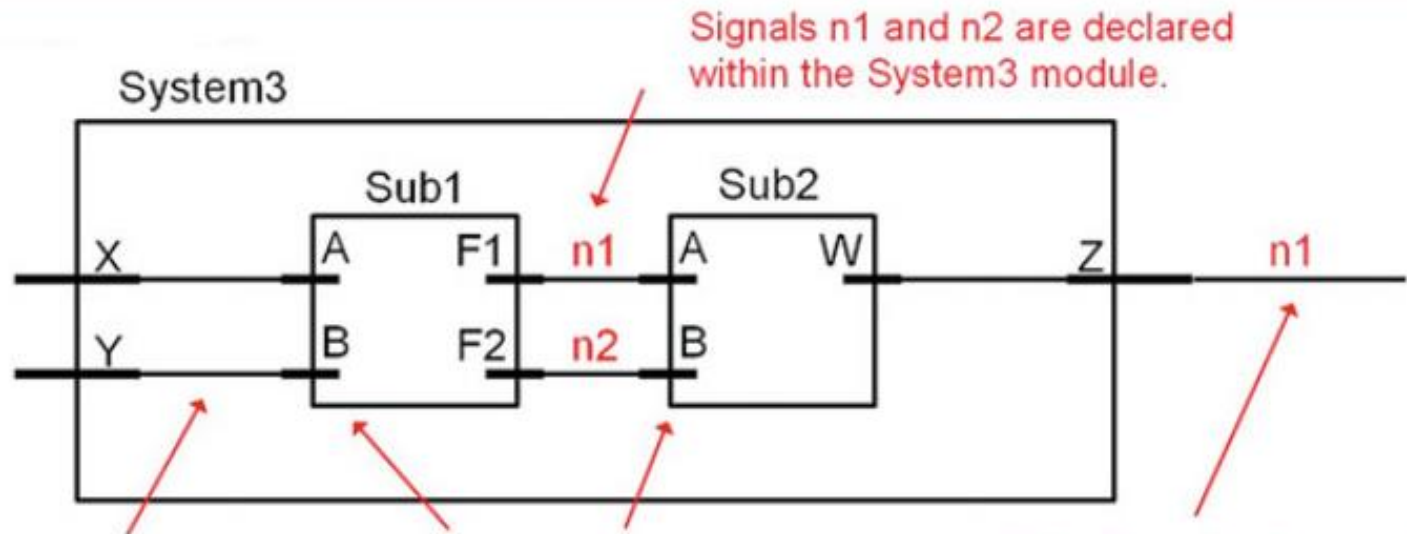
```
module System2 (output reg Bus_Out[31:0],  
               input  wire Bus_In[31:0]);  
  
    //-- module items go here...  
  
endmodule
```

Signal Declarations

- Internal signals are used for connections within a module.
- Signals must be declared before their first use in the module.
- Declaration format: Type (e.g., wire, reg, integer)
- User-defined name (case-sensitive, must start with an alphabetic character)
- Multiple signals of the same type can be declared on the same line, separated by commas.
- Synthesizable signal types:
 - net (e.g., wire, tri)
 - Reg
 - integer

`<type> name;`

Signals and Systems



A new signal is not needed for these connections. The port names can be used to signify the connections instead.

The port names A and B are used in two sub-systems. This is legal since they are named within the lower-level sub-systems. They are not connected to each other implicitly and there is no conflict.

Using the signal name n1 is legal here. The signal does not "see" the duplicate signal name "n1" within the System3 module because they are at different levels of hierarchy.

Operators

- Verilog provides a variety of predefined operators for different operations.
- Operators are designed to work on specific data types (e.g., reg, wire, integer).
- Not all operators are synthesizable (some are for simulation purposes only).
- Understanding which operators are synthesizable is crucial for hardware design.

Operators

Common categories of Verilog operators

- Arithmetic operators (+, -, *, /, %)
- Bitwise operators (&, |, ^, ~)
- Logical operators (&&, ||, !)
- Relational operators (==, !=, >, <, >=, <=)
- Shift operators (<<, >>)
- Reduction operators (&, |, ^, ~&, ~|, ~^)
- Concatenation and replication ({}, {{{}}})

Assignment Operator

- Assignment Operator (=)
- Used to assign values to signals.
- Left-hand side (LHS) → Target signal.
- Right-hand side (RHS) → Input (can be signals, constants, or expressions).

Example :

```
F1 = A;    // Assign signal A to F1
```

```
F2 = 8'hAA; // Assign 8-bit value 10101010 to F2
```

Continuous Assignment

- Uses the assign keyword for continuous signal assignment
- LHS must be a net type (e.g., wire)
- RHS can contain nets, registers, constants, and operators.
- Models combinational logic: any change in RHS updates LHS immediately.
- Multiple assignments to the same net → Higher drive strength takes priority.

```
assign F1 = A;    // F1 updates whenever A changes
```

```
assign F2 = 1'b0; // F2 is assigned constant 0
```

```
assign F3 = 4'hA; // F3 gets 4-bit hexadecimal value A (1010)
```

Concurrent Execution

- Each assign statement is executed simultaneously
- Synthesized as separate logic circuits
- Unlike traditional programming, Verilog does not execute assignments sequentially

```
assign X = A;  
assign Y = B;  
assign Z = C;
```

- These assignments occur at the same time, like three separate wires.

Bitwise Logical Operator

Syntax	Operation
<code>~</code>	Negation
<code>&</code>	AND
<code> </code>	OR
<code>^</code>	XOR
<code>~^</code> or <code>^~</code>	XNOR
<code><<</code>	Logical shift left (fill empty LSB location with zero)
<code>>></code>	Logical shift right (fill empty MSB location with zero)

```
wire [3:0] A = 4'b1101;  
wire [3:0] B = 4'b1010;  
wire [3:0] X;
```

```
assign X = A & B; // X = 1000 (Bitwise AND)  
assign X = A | B; // X = 1111 (Bitwise OR)  
assign X = A ^ B; // X = 0111 (Bitwise XOR)  
assign X = ~A;    // X = 0010 (Bitwise NOT)
```

Reduction Operator

- Reduction operators perform a logical operation on all bits of a vector.
- The entire vector is treated as multiple inputs to a single logic operation.
- The result is always a single-bit output.

Syntax	Operation
<code>&</code>	AND all bits in the vector together (1-bit result)
<code>~&</code>	NAND all bits in the vector together (1-bit result)
<code> </code>	OR all bits in the vector together (1-bit result)
<code>~ </code>	NOR all bits in the vector together (1-bit result)
<code>^</code>	XOR all bits in the vector together (1-bit result)
<code>~^</code> or <code>^^</code>	XNOR all bits in the vector together (1-bit result)

Boolean Logic Operators

- Boolean logic operators return TRUE (1) or FALSE (0) based on a logical operation.
- Used in decision-making statements such as if, while, and case.
- Operate on single-bit inputs or evaluate entire expressions as Boolean conditions

Syntax	Operation
!	Negation
&&	AND
	OR

```
!X           // TRUE if all values in X are 0, FALSE otherwise
X && Y       // TRUE if the bitwise AND of X and Y results in all ones, FALSE otherwise
X || Y       // TRUE if the bitwise OR of X and Y results in all ones, FALSE otherwise
```


Relational Operators

- Relational operators compare two values and return TRUE (1) or FALSE (0)
- Used in conditional statements like if, while, and case
- Operate on scalars and vectors, performing bitwise or numerical comparisons.

Syntax	Description
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Conditional Operator

- The conditional operator (`? :`) is a shorthand for if-else statements.
- Used for concise and readable assignments in combinational logic.

`<target_net> = <Boolean_condition> ? <true_assignment> : <false_assignment>;`

```
F = (A == 1'b0) ? 1'b1 : 1'b0;           // If A is a zero, F=1, otherwise F=0.  
                                           This models an inverter.
```

```
F = (sel == 1'b0) ? A : B;               // If sel is a zero, F=A, otherwise F=B.  
                                           This models a selectable switch.
```

```
F = ((A == 1'b0) && (B == 1'b0)) ? 1'b'0 : // Nested conditional statements.  
    ((A == 1'b0) && (B == 1'b1)) ? 1'b'1 : // This models an XOR gate.  
    ((A == 1'b1) && (B == 1'b0)) ? 1'b'1 :  
    ((A == 1'b1) && (B == 1'b1)) ? 1'b'0;
```

```
F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;    // This models the logic expression  
                                           // F = C' · (A' + B) .
```

Concatenation Operator

- Curly brackets {} are used to combine multiple signals into a single vector
- The target signal must have a bit width equal to the sum of the input signal sizes..

```
Bus1[7:0] = {Bus2[7:4], Bus3[3:0]}; // Assuming Bus1, Bus2, and Bus3 are all 8-bit
// vectors, this operation takes the upper
// 4-bits of
// Bus2, concatenates them with the lower
// 4-bits of
// Bus3, and assigns the 8-bit combination
// to Bus1.

BusC = {BusA, BusB}; // If BusA and BusB are 4-bits, then BusC
// must be 8-bits.

BusC[7:0] = {4'b0000, BusA}; // This pads the 4-bit vector BusA with
// 4x leading
// zeros and assigns to the 8-bit vector BusC.
```

Replication Operator

- The replication operator (`{{}}`) allows a vector to be repeated multiple times.
- Useful for extending bit-widths and pattern generation
- Replicates a vector multiple times, reducing redundant code
- Commonly used for zero-padding, sign extension, and test pattern generation.
- Can be combined with concatenation for flexible signal assignment

```
{<number_of_replications>{<vector_name_to_be_replicated>}}
```

```
BusX = {4{Bus1}};           // This is equivalent to: BusX = {Bus1, Bus1, Bus1, Bus1};  
BusY = {2{A,B}};           // This is equivalent to: BusY = {A, B, A, B};  
BusZ = {Bus1, {2{Bus2}}};  // This is equivalent to: BusZ = {Bus1, Bus2, Bus2};
```

Numerical Operator

- Verilog supports several numerical operators for arithmetic operations.
- These operators work on integers, real numbers, and bit-vector data types.

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2's complement negation (when placed in front of an argument)
*	Multiplication
/	Division
%	Modulus
**	Raise to the power
<<<	Shift to the left, fill with zeros
>>>	Shift to the right, fill with sign bit

Operator Precedence

- Operators with higher precedence execute first unless overridden by parentheses
- Always use parentheses () for clarity in complex expressions.
- Assignment (=) has the lowest precedence, ensuring other operations execute before assignment

Operators	Precedence	Notes
! ~ + -	Highest	Bitwise/Unary
{ } { }		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ -	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
~	↓	OR/NOR Operators
&&		Boolean AND
		Boolean OR
?:	Lowest	Conditional Operator

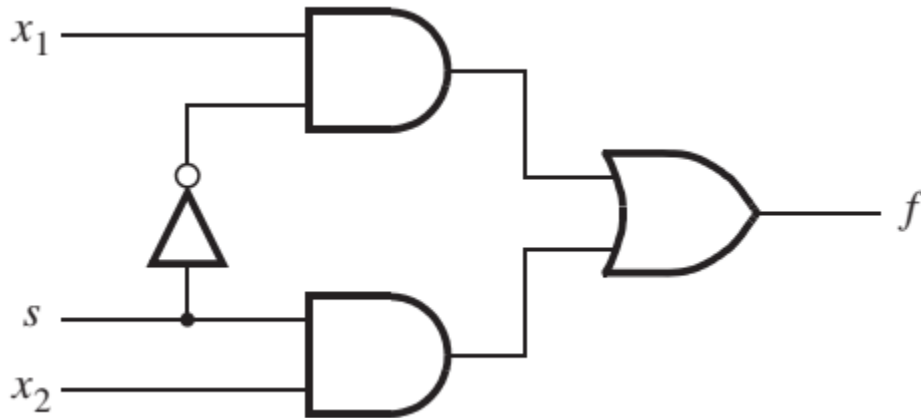
Gate Level Primitive

- Works well for small circuits with a limited number of gates
- Intuitive for designers familiar with digital logic design
- Requires manual instantiation and connection of individual gates

Dataflow Modeling

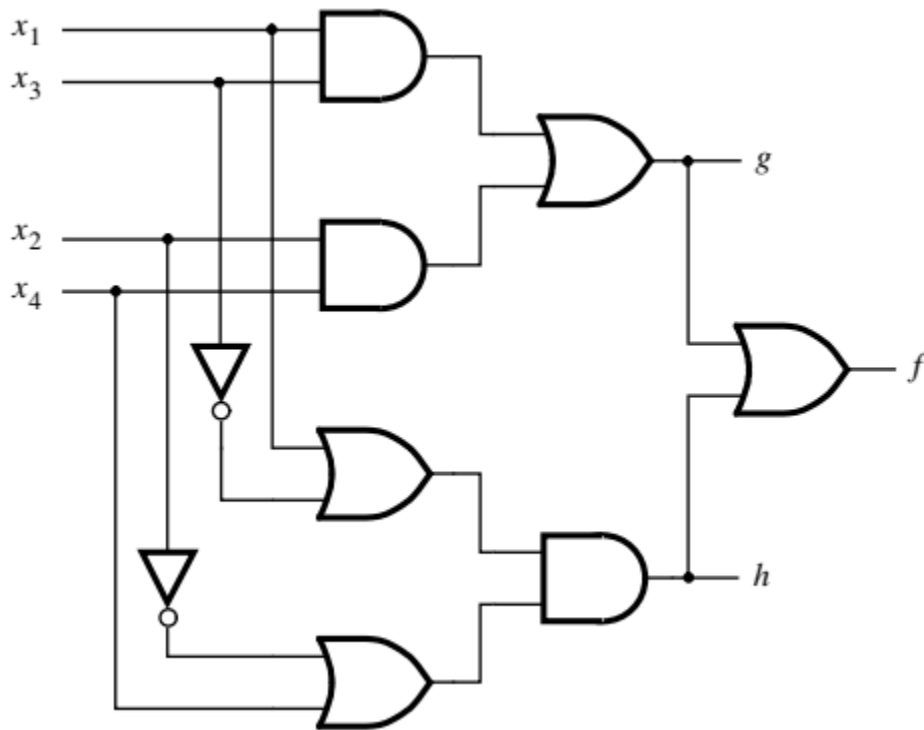
- Focuses on data movement and processing rather than individual gates.
- Describes circuits in terms of data flow between registers.
- Helps designers optimize the circuit more effectively.

Gate Level Primitive



```
module example1 (x1, x2, s, f);  
  input x1, x2, s;  
  output f;  
  
  not (k, s);  
  and (g, k, x1);  
  and (h, s, x2);  
  or (f, g, h);  
  
endmodule
```


Gate Level Primitive

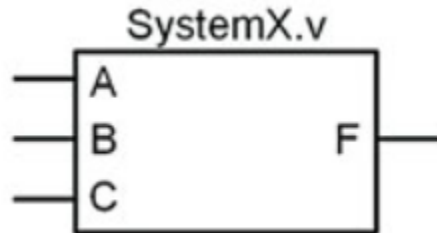


$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \bar{x}_3)(\bar{x}_2 + x_4)$$
$$f = g + h$$

```
module example2 (x1, x2, x3, x4, f, g, h);  
  input x1, x2, x3, x4;  
  output f, g, h;  
  
  and (z1, x1, x3);  
  and (z2, x2, x4);  
  or (g, z1, z2);  
  or (z3, x1, ~x3);  
  or (z4, ~x2, x4);  
  and (h, z3, z4);  
  or (f, g, h);
```

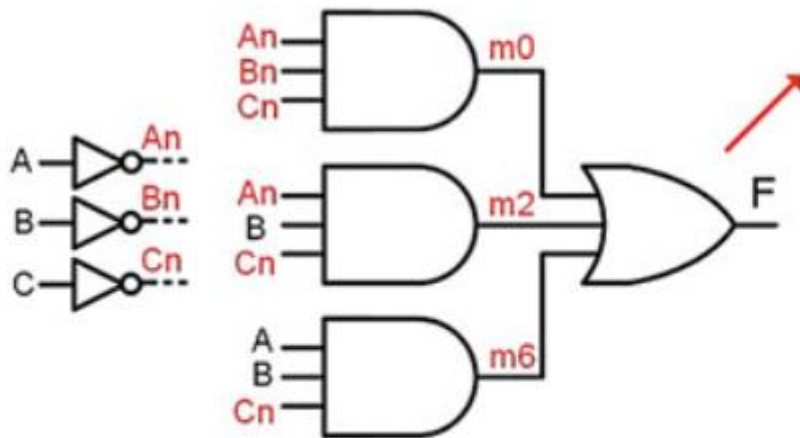
```
endmodule
```

Continuous Assignment with Logical Operators



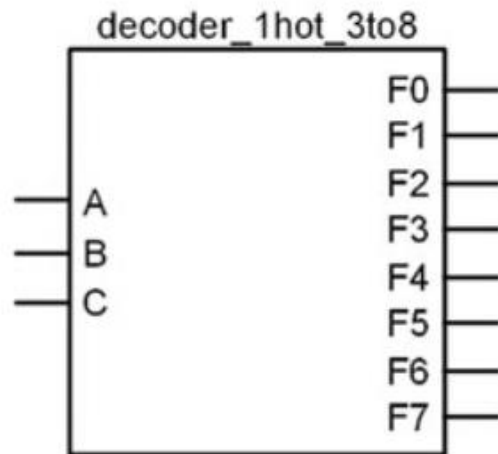
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

$$F = \sum_{A,B,C}(0,2,6) = A' \cdot B' \cdot C' + A' \cdot B \cdot C' + A \cdot B \cdot C'$$



```
module SystemX (output wire F,  
                input  wire A, B, C);  
  
    wire  An, Bn, Cn;    // internal nets  
    wire  m0, m2, m6;  
  
    assign An = ~A;      // Not's  
    assign Bn = ~B;  
    assign Cn = ~C;  
  
    assign m0 = An & Bn & Cn; // AND's  
    assign m2 = An & B  & Cn;  
    assign m6 = A  & B  & Cn;  
  
    assign F  = m0 | m2 | m6; // OR  
  
endmodule
```

One-Hot Decoder



A	B	C	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

$$F0 = \sum_{A,B,C}(0) = A'B'C'$$

$$F1 = \sum_{A,B,C}(1) = A'B'C$$

$$F2 = \sum_{A,B,C}(2) = A'B \cdot C'$$

$$F3 = \sum_{A,B,C}(3) = A'B \cdot C$$

$$F4 = \sum_{A,B,C}(4) = A \cdot B' \cdot C'$$

$$F5 = \sum_{A,B,C}(5) = A \cdot B' \cdot C$$

$$F6 = \sum_{A,B,C}(6) = A \cdot B \cdot C'$$

$$F7 = \sum_{A,B,C}(7) = A \cdot B \cdot C$$

```

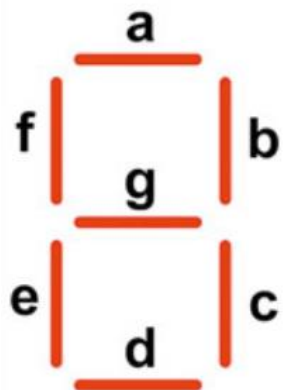
module decoder_1hot_3to8
  (output wire F0, F1, F2, F3, F4, F5, F6, F7,
   input wire A, B, C);

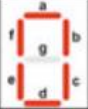



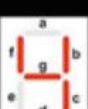
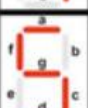
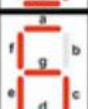

  assign F0 = ~A & ~B & ~C;
  assign F1 = ~A & ~B & C;
  assign F2 = ~A & B & ~C;
  assign F3 = ~A & B & C;
  assign F4 = A & ~B & ~C;
  assign F5 = A & ~B & C;
  assign F6 = A & B & ~C;
  assign F7 = A & B & C;

endmodule
    
```

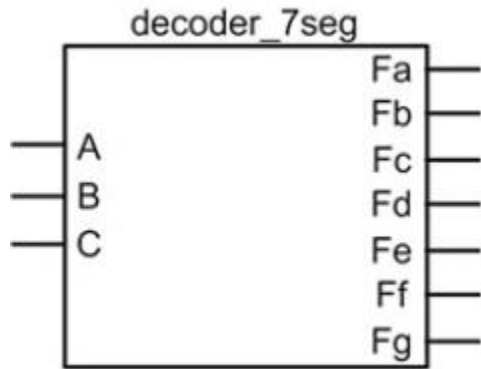
Exercise : 7 Segment Decoder

LED Labels

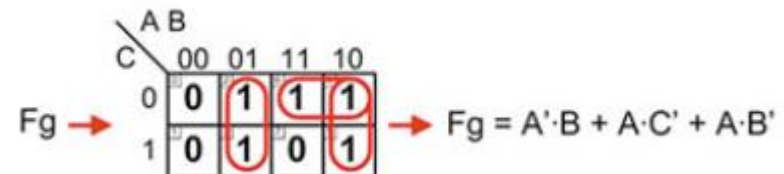
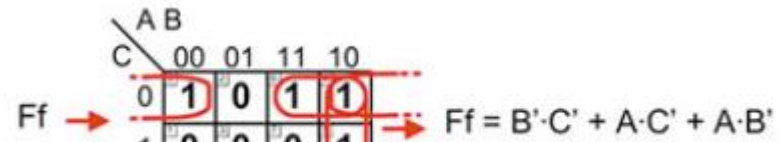
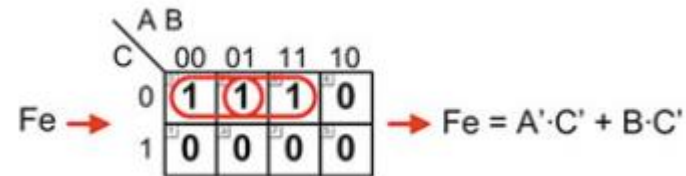
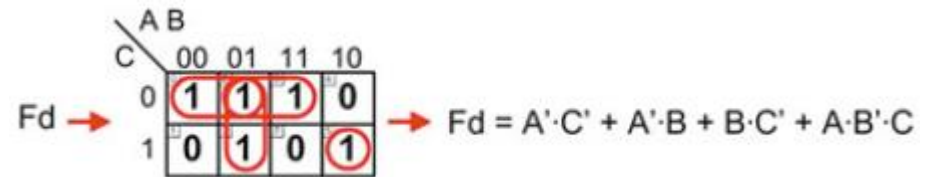
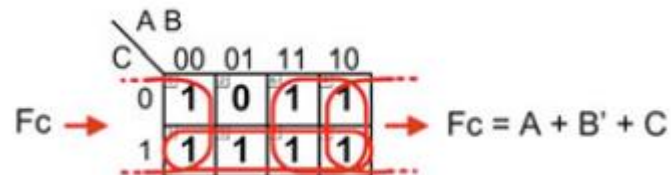
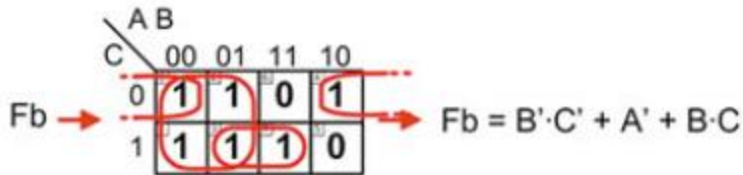
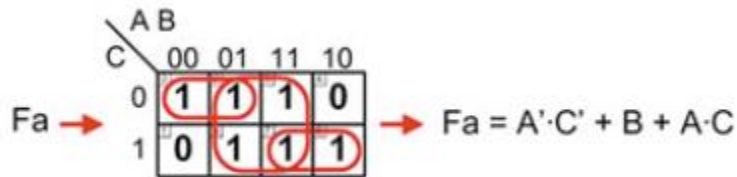


A	B	C		F _a	F _b	F _c	F _d	F _e	F _f	F _g
0	0	0		1	1	1	1	1	1	0
0	0	1		0	1	1	0	0	0	0
0	1	0		1	1	0	1	1	0	1
0	1	1		1	1	1	1	0	0	1
1	0	0		0	1	1	0	0	1	1
1	0	1		1	0	1	1	0	1	1
1	1	0		1	0	1	1	1	1	1
1	1	1		1	1	1	0	0	0	0

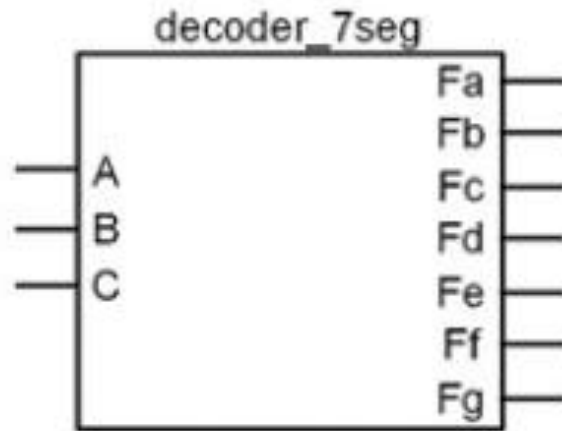
Exercise : 7 Segment Decoder



A	B	C	Fa	Fb	Fc	Fd	Fe	Ff	Fg
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0



Exercise : 7 Segment Decoder



A	B	C	Fa	Fb	Fc	Fd	Fe	Ff	Fg
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

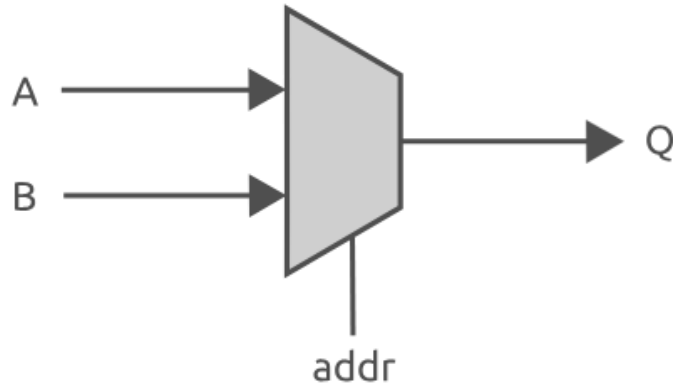
```
module decoder_7seg (output wire Fa, Fb, Fc, Fd, Fe, Ff, Fg,
                    input wire A, B, C);

    assign Fa = (~A & ~C) | (B) | (A & C);
    assign Fb = (~B & ~C) | (~A) | (B & C);
    assign Fc = (A) | (~B) | (C);
    assign Fd = (~A & ~C) | (~A & B) | (B & ~C) | (A & ~B & C);
    assign Fe = (~A & ~C) | (B & ~C);
    assign Ff = (~B & ~C) | (A & ~C) | (A & ~B);
    assign Fg = (~A & B) | (A & ~C) | (A & ~B);

endmodule
```

Continuous Assignment with Conditional Operators

2:1 Multiplexer



`assign q = addr ? b : a;`

```
module mux (  
    input wire addr, // Address or control  
    signal  
    input wire a,    // Input a  
    input wire b,    // Input b  
    output wire q    // Output q  
);  
  
    // Conditional assignment  
    assign q = addr ? b : a;  
  
endmodule
```


Continuous Assignment with Conditional Operators

Implement the following truth table using a continuous assignment with conditional operators.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX (output wire F,
                input wire A, B, C);

    assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
                ((A == 1'b0) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
                ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
                ((A == 1'b0) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
                ((A == 1'b1) && (B == 1'b0) && (C == 1'b0)) ? 1'b0 :
                ((A == 1'b1) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
                ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
                ((A == 1'b1) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
                1'b0;

endmodule
```

```
module SystemX (output wire F,
                input wire A, B, C);

    assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
                ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
                ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
                1'b0;

endmodule
```


Exercise : Conditional Operators

Implement the truth table using continuous assignment with conditional operator

$$F = C' \cdot (A' + B)$$

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX (output wire F,  
                input wire A, B, C);  
  
    assign F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;  
  
endmodule
```

Exercise : Conditional Operators

3-to-8 One-Hot Decoder

ABC	F(7)	F(6)	F(5)	F(4)	F(3)	F(2)	F(1)	F(0)
"000"	0	0	0	0	0	0	0	1
"001"	0	0	0	0	0	0	1	0
"010"	0	0	0	0	0	1	0	0
"011"	0	0	0	0	1	0	0	0
"100"	0	0	0	1	0	0	0	0
"101"	0	0	1	0	0	0	0	0
"110"	0	1	0	0	0	0	0	0
"111"	1	0	0	0	0	0	0	0

```
module decoder_1hot_3to8 (output wire [7:0] F,  
                          input  wire [2:0] ABC);  
  
    assign F = (ABC == 3'b000) ? 8'b0000_0001 :  
                (ABC == 3'b001) ? 8'b0000_0010 :  
                (ABC == 3'b010) ? 8'b0000_0100 :  
                (ABC == 3'b011) ? 8'b0000_1000 :  
                (ABC == 3'b100) ? 8'b0001_0000 :  
                (ABC == 3'b101) ? 8'b0010_0000 :  
                (ABC == 3'b110) ? 8'b0100_0000 :  
                (ABC == 3'b111) ? 8'b1000_0000 :  
                8'bXXXX_XXXX;  
  
endmodule
```

Continuous Assignment with Delay

- Verilog allows modeling of gate delays in continuous assignments.
- The # symbol is used to specify delayed assignments.
- For combinational circuits, delays can be specified for:
 - All transitions (single delay parameter).
 - Rising, falling, and high-impedance (Z) transitions separately (three parameters).
 - Rising ($0 \rightarrow 1$) and falling ($1 \rightarrow 0$) transitions separately (two parameters).

Continuous Assignment with Delay

- If only one delay is given, it applies to all transitions.
- If two delays are given:
 - First parameter → Rise time delay.
 - Second parameter → Fall time delay.
- If three delays are given:
 - First parameter → Rise time delay.
 - Second parameter → Fall time delay.
 - Third parameter → Transition to Z (high-impedance state).
 - Parentheses are optional but recommended when specifying multiple delay parameters.

Continuous Assignment with Delay

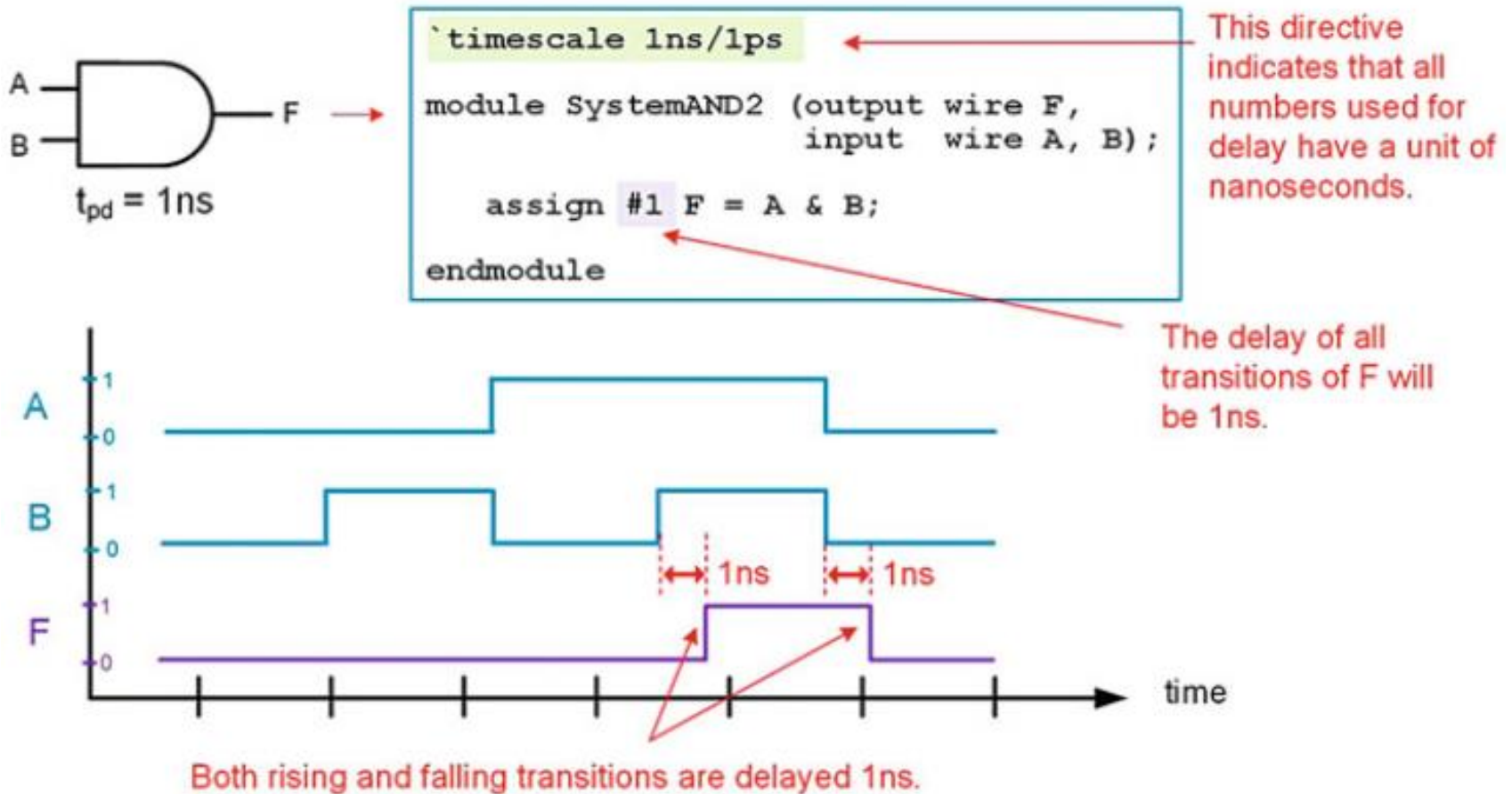
assign #(<del_all>)	<target_net> = <RHS_nets, operators, etc...>;
assign #(<del_rise, del_fall>)	<target_net> = <RHS_nets, operators, etc...>;
assign #(<del_rise, del_fall, del_off>)	<target_net> = <RHS_nets, operators, etc...>;

Example:

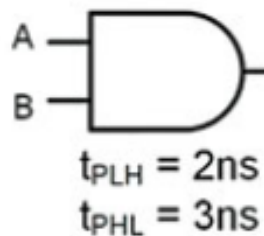
```
assign #1      F = A;  // Delay of 1 on all transitions.
assign #(2,3)  F = A;  // Delay of 2 for rising transitions and 3 for falling.
assign #(2,3,4) F = A;  // Delay of 2 for rising, 3 for falling, and 4 for
                        off transition.
```

Modeling Delay in Continuous Assignment

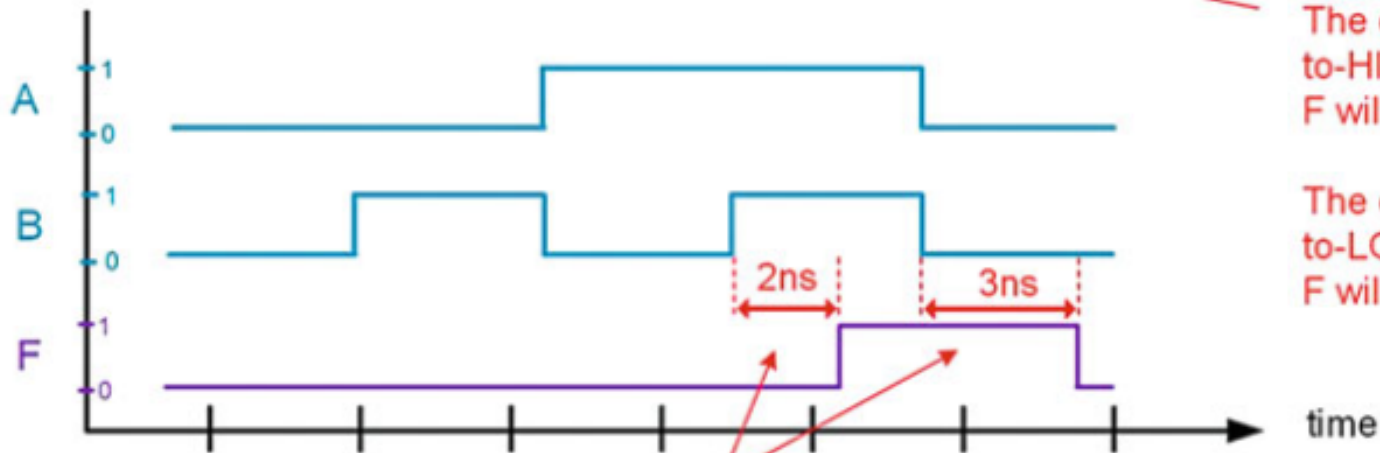
When using delay, it is typical to include the ``timescale` directive to provide the units of the delay being specified.



Modeling Delay in Continuous Assignment



```
`timescale 1ns/1ps  
module SystemAND2 (output wire F,  
                  input wire A, B);  
    assign #(2,3) F = A & B;  
endmodule
```



The delay of all LOW-to-HIGH transitions of F will be 2ns.

The delay of all HIGH-to-LOW transitions of F will be 3ns.

The transition from LOW to HIGH has a delay of 2ns while the transition from HIGH to LOW has a delay of 3ns.

Modeling Delay in Continuous Assignment

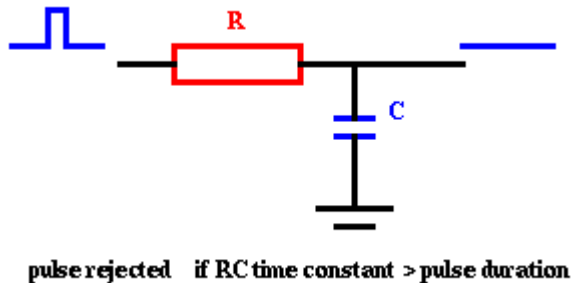
- Verilog allows modeling a range of delays selectable by a switch in the CAD compiler.
- There are three delay categories:
 - Minimum delay
 - Typical delay
 - Maximum delay
- The delays are separated by a colon (:) in the syntax.

```
assign #(<min>:<typ>:<max>) <target_net> = <RHS_nets, operators, etc...>;
```

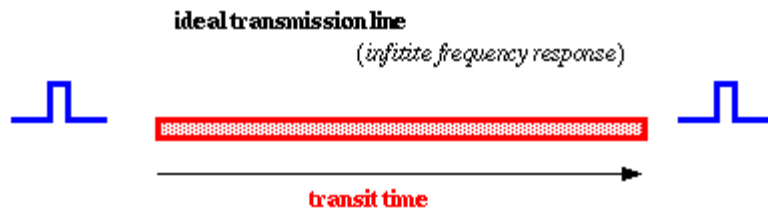
```
assign #(1:2:3)          F = A;  // Specifying a range of delays for all
                             transitions.
assign #(1:1:2, 2:2:3)   F = A;  // Specifying a range of delays for
                             rising/falling.
assign #(1:1:2, 2:2:3, 4:4:5) F = A; // Specifying a range of delays for
                             each transition.
```


Delay modeling for signal propagation

- Verilog provides two types of delay modeling for signal propagation:
 - Inertial Delay (default behavior/ Gate)
 - Transport Delay (Interconnect/Wire)



- Inertial Delay (default behavior/ Gate)

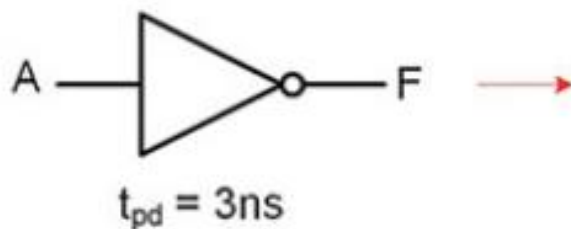


- Transport Delay (Interconnect/Wire)

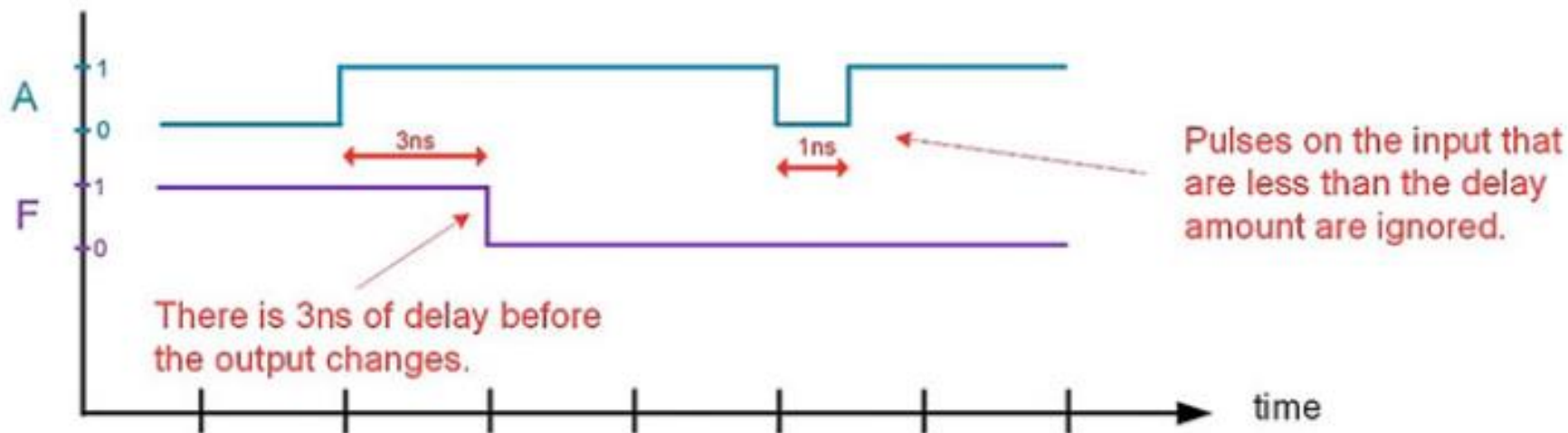
Delay modeling for signal propagation

Feature	Inertial Delay	Transport Delay
Definition	Models real-world gate delays, filtering out short pulses.	Models ideal signal propagation without filtering glitches.
Glitch Filtering	Yes, short pulses are removed.	No, all transitions are propagated.
Stability Check	Required – output changes only if input remains stable for at least the delay duration.	Not required – every input change is propagated after the delay.
Use Case	Modeling logic gates and real-world behavior.	Modeling interconnect delays (wire delays).
Syntax	<code>assign #delay y = a & b;</code>	<code>assign y = #delay a & b;</code>
Behavior	Short pulses are ignored if they are shorter than the delay.	Every change in the input is propagated after the delay.
Default in Verilog?	Yes	No
Example	<code>assign #5 y = a;</code>	<code>assign y = #5 a;</code>

Example : Inertial (Default) Delay



```
`timescale 1ns/1ps  
  
module SystemINV (output wire F,  
                  input  wire A);  
  
    assign #3 F = ~A;  
  
endmodule
```





Thank you !

Happy Learning