Module – 3

Gate-Level Modeling

- Most digital design is done at gate level or at higher levels of abstraction. At gate level, the circuit is described in terms of logic gates (e.g., and, nand).
- Hardware design at this level is easy to understand for a user with a basic knowledge of digital logic design because there is a one-to-one correspondence between the logic circuit diagram and the Verilog description.

Gate Types

- A logic circuit can be designed by using of logic gates. Verilog supports basic logic gates as predefined primitives.
- These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.
- All logic circuits can be designed by using basic gates. There are two classes of basic gates
  1. and/or gates
  2. buf/not gates

And/Or Gates

- And/or gates have one scalar output and multiple scalar inputs.
- The first terminal in the port list is an output terminal and the other terminals are inputs.
- The output of a gate is evaluated as soon as one of the inputs changes.
- The and/or gates supported by Verilog are

and	**or**	**xor**
nand	**nor**	**xnor**

- The logic symbols for the corresponding gates are shown in fig.3.1. The output terminal is denoted by out and the input terminals are denoted by i1 and i2 for a two input gate.

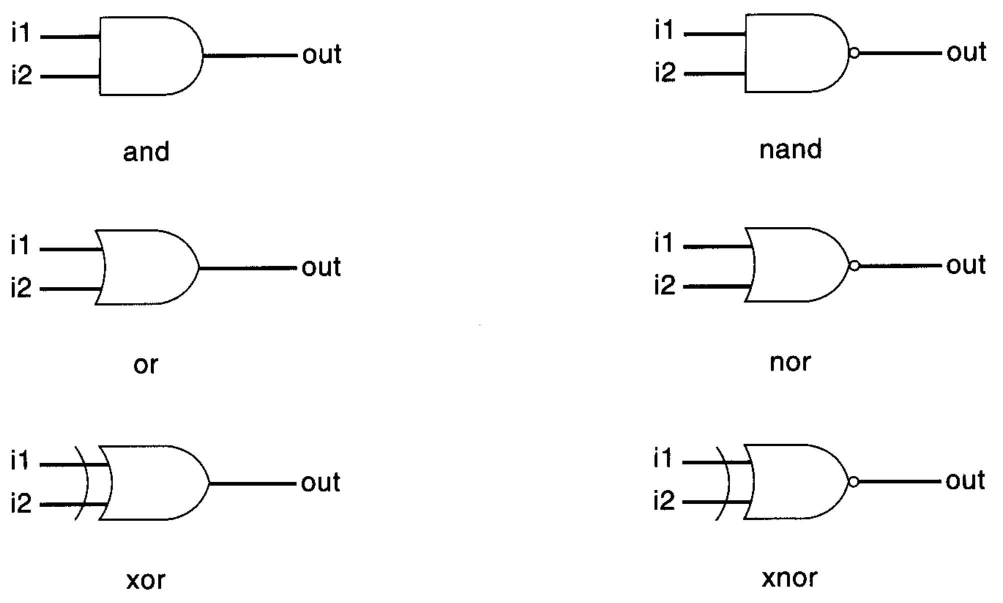


Fig.3.1: Logic Gates

- These gates are instantiated to build logic circuits in Verilog.
- Gate instantiation statement of And/Or gates shown in fig.3.1 can be written as

```
and a1(out, i1, i2);
nand na1(out, i1, i2);
or or1(out, i1, i2);
nor nor1(out, i1, i2);
xor x1(out, i1, i2);
xnor nx1(out, i1, i2);
```

- More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation. For example a 3-input nand gate instantiation can be written as

```
nand na1_3inp(out, i1, i2, i3);
```

- The instance name is optional (need not to be specified) for primitives. Gate instantiation without instance name can be written as

```
and (out, i1, i2);          // legal gate instantiation
```

- The truth tables for And/Or gates with two inputs and one output are shown in table 3.1.

Table 3.1: Truth Tables for And/Or Gates

and		i1: 0	1	X	Z
i2	0	0	0	0	0
	1	0	1	X	X
	X	0	X	X	X
	Z	0	X	X	X

nand		i1: 0	1	X	Z
i2	0	1	1	1	1
	1	1	0	X	X
	X	1	X	X	X
	Z	1	X	X	X

or		i1: 0	1	X	Z
i2	0	0	1	X	X
	1	1	1	1	1
	X	X	1	X	X
	Z	X	1	X	X

nor		i1: 0	1	X	Z
i2	0	1	0	X	X
	1	0	0	0	0
	X	X	0	X	X
	Z	X	0	X	X

xor		i1: 0	1	X	Z
i2	0	0	1	X	X
	1	1	0	X	X
	X	X	X	X	X
	Z	X	X	X	X

xnor		i1: 0	1	X	Z
i2	0	1	0	X	X
	1	0	1	X	X
	X	X	X	X	X
	Z	X	X	X	X

Buf/Not Gates

- Buf/not gates have one scalar input and one or more scalar outputs.
- The last terminal in the port list is connected to the input. Other terminals are connected to the outputs.
- Two basic buf/not gate primitives are provided in Verilog.

  buf **not**
- The logic symbols for these gates are shown in fig.3.2, have one input and one output.

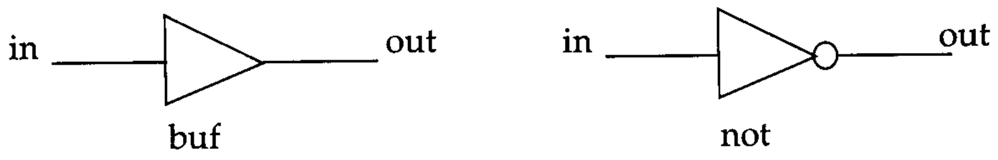


Fig.3.2: Buf and Not Gates

- These gates can have multiple outputs but exactly one input, which is the last terminal in the port list.
- Gate instantiation statement of Buf/Not gates shown in fig.3.2 can be written as

```
buf b1(out, in);
not n1(out, in);
```

- More than two outputs gate instantiation statement of Buf/Not gates can be written as

```
buf b1_2out(out1, out2, in);
```

- The instance name is optional (need not to be specified) for primitives. Gate instantiation without instance name can be written as

```
not (out, in);          // legal gate instantiation
```

- Truth tables for Buf/Not gates with one input and one output are shown in table 3.2.

Table 3.2: Truth Tables for Buf/Not Gates

buf	in	out
	0	0
	1	1
	X	X
	Z	X

not	in	out
	0	1
	1	0
	X	X
	Z	X

bufif/notif

- Gates with an additional control signal on buf and not gates are also supported by Verilog.

bufif1	**notif1**
bufif0	**notif0**

- These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted.
- The logic symbols for bufif/notif are shown in fig.3.3.

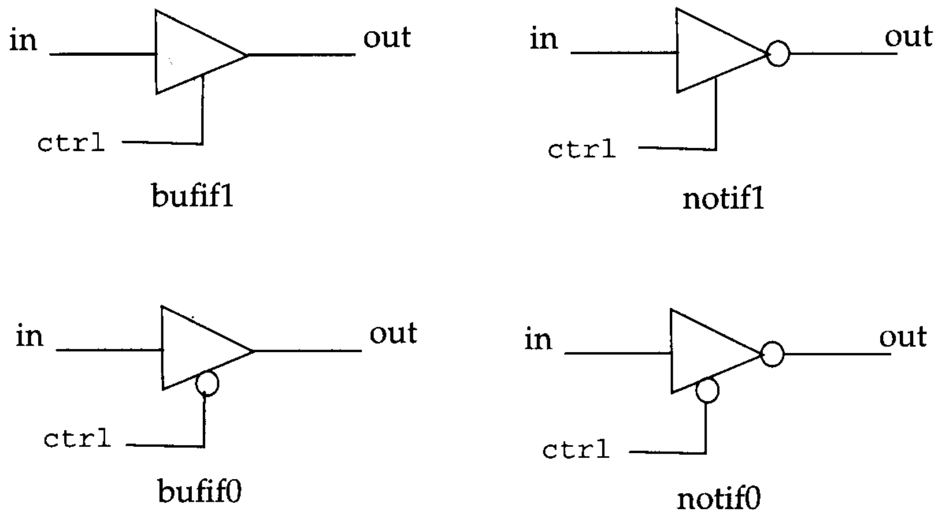


Fig.3.3: Gates Bufif and Notif

- ➢ The truth tables for these gates are shown in table 3.3.

Table 3.3: Truth Tables for Bufif/Notif Gates

		ctrl			
	bufif1	**0**	**1**	**X**	**Z**
	0	Z	0	L	L
	1	Z	1	H	H
in	X	Z	X	X	X
	Z	Z	X	X	X

		ctrl			
	bufif0	**0**	**1**	**X**	**Z**
	0	0	Z	L	L
	1	1	Z	H	H
in	X	X	Z	X	X
	Z	X	Z	X	X

		ctrl			
	notif1	**0**	**1**	**X**	**Z**
	0	Z	1	H	H
	1	Z	0	L	L
in	X	Z	X	X	X
	Z	Z	X	X	X

		ctrl			
	notif0	**0**	**1**	**X**	**Z**
	Z	1	Z	H	H
	Z	0	Z	L	L
in	Z	X	Z	X	X
	Z	X	Z	X	X

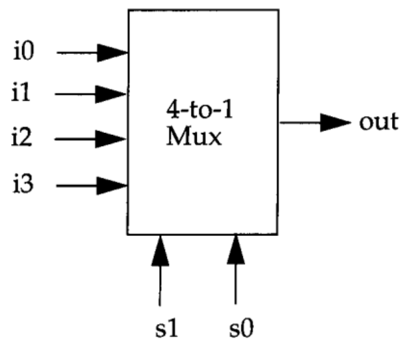
- ➢ These gates are used when a signal is to be driven only when the control signal is asserted. Such a situation is applicable when multiple drivers drive the signal. These drivers are designed to drive the signal on mutually exclusive control signals.

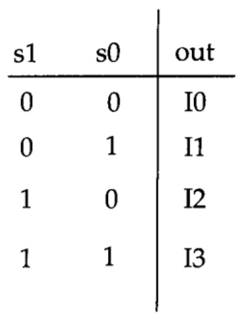
- Symbols L and H have a special meaning. The symbol L means that the output has 0 or z value. The symbol H means that the output has 1 or z value. Any transition to H or L is treated as a transition to x.
- Gate instantiation statement of Bufif/Notif gates shown in fig.3.3 can be written as

```
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

Gate level 4:1 Multiplexer

- 4:1 multiplexer has 4 input signals, 2 select signals and 1 output signal as shown in fig.3.4.
- i0, i1, i2, i3 are input signals, s1 and s0 are select signals and out is the output signal.





s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Fig.3.4: 4:1 Multiplexer

- The logic diagram for the 4:1 multiplexer is as shown in fig.3.5.

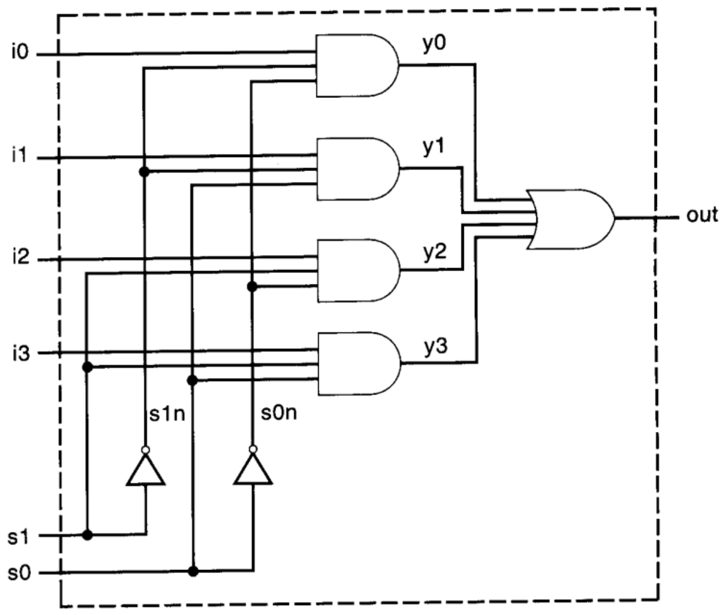


Fig.3.5: Logic Diagram for 4:1 Multiplexer

- Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0. Internal nets y0, y1, y2, y3 are the connection between and gate and or gate.
- Verilog description for 4:1 Multiplexer using gate level is written as

```
module mux4_to_1 (i0, i1, i2, i3, s1, s0, out);
        // Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
        // Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
        // not gate instantiations
not (s1n, s1);
not (s0n, s0);
        // 3-input and gates instantiations
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
```

```
          // 4-input or gate instantiation
or (out, y0, y1, y2, y3);
endmodule
```

- The 4:1 multiplexer can be tested with the stimulus written in the stimulus block / testbench
- The stimulus checks each combination of select signals and then connects the appropriate input to the output.

```
          // Define the stimulus module (no ports)
module mux4_to_1_tb();
          // Declare variables to be connected to inputs
reg i0, i1, i2, i3;
reg s1, s0;
          // Declare output wire
wire out;
          // Instantiate the multiplexer
mux4_to_1 m1(i0, i1, i2, i3, s1, s0, out);
          // Define the stimulus
initial
begin
          // set input lines
i0 = 1; i1 = 0; i2 = 1; i3 = 0; s1 = 0; s0 = 0;
#20 s1 = 0; s0 = 1;
#20 s1 = 1; s0 = 0;
#20 s1 = 1; s0 = 1;
end
endmodule
```

4-Bit Ripple Carry Full Adder

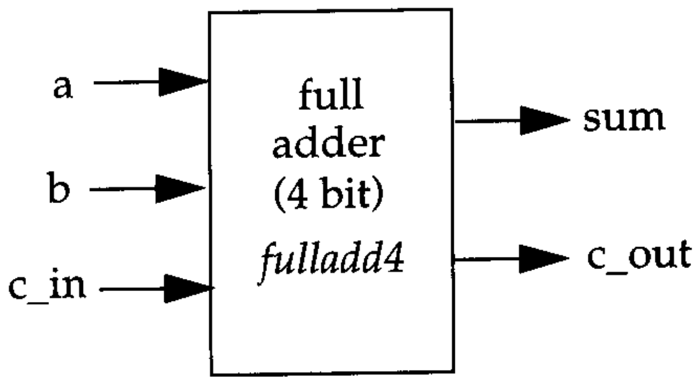


Fig.3.6: 4-bit Ripple Carry Full Adder

- A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in the fig.3.7 where fa0, fa1, fa2, and fa3 are instances of the module fulladd (1-bit full adder).

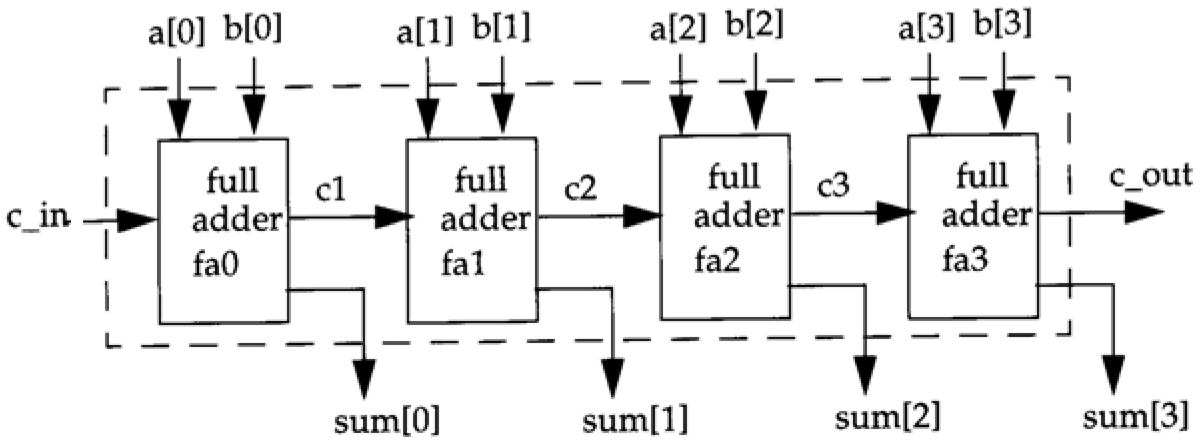


Fig.3.7: 4-bit Ripple Carry Full Adder using full adder as instance

```
        // Define a 4-bit full adder
module fulladd4(a, b, c_in, sum, c_out);
        // I/O port declarations
input[3:0] a, b;
input c_in;
output [3:0] sum;
output c_out;
```

```
        // Internal nets
wire c1, c2, c3;
        // Instantiate four 1-bit full adders.
fulladd fa0(a[0], b[0], c_in, sum[0], c1,);
fulladd fa1(a[1], b[1], c1, sum[1], c2);
fulladd fa2(a[2], b[2], c2, sum[2], c3);
fulladd fa3(a[3], b[3], c3, sum[3], c_out);
endmodule
```

- The logic diagram for a 1-bit full adder is shown in fig.3.8

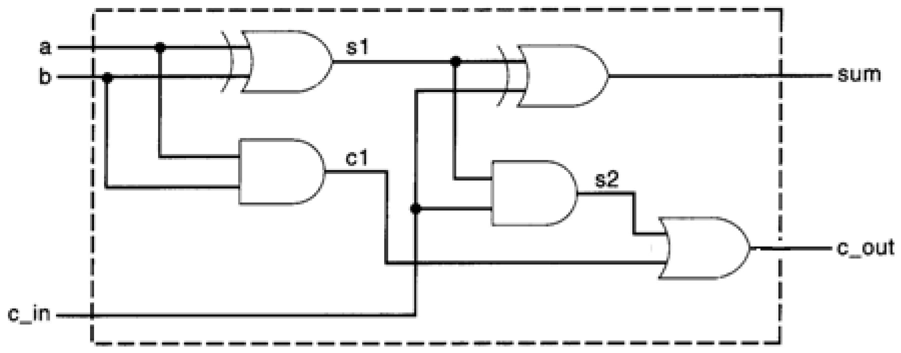


Fig.3.8: 1-bit Full Adder

```
        // Define a 1-bit full adder
module fulladd(a, b, c_in, sum, c_out);
        // I/O port declarations
input a, b, c_in;
output sum, c_out;
        // Internal nets
wire s1, c1, s2;
        // Instantiate logic gate primitives
xor (s1, a, b);
```

```
and (c1, a, b);
xor (sum, s1, c_in);
and (s2, s1, c_in);
or (c_out, s2, c1);
endmodule
```

```
// Define the stimulus / Testbench (for top level module)
module stimulus ();
    // Declare variables to be connected to inputs
reg [3:0] a, b;
reg c_in;
wire [3:0] sum;
wire c_out;
    // Instantiate the 4-bit full adder.
fulladd4 FA1_4(a, b, c_in, sum, c_out);
    // Define the stimulus
initial
begin
a = 4'd0; b = 4'd0; c_in = 1'b0;
#5 a = 4'd3; b = 4'd4;
#5 a = 4'd2; b = 4'd5;
#5 a = 4'd9; b = 4'd9;
#5 a = 4'd10; b = 4'd15;
#5 a = 4'd10; b = 4'd5; c_in = 1'b1;
end
endmodule
```

Gate Delays

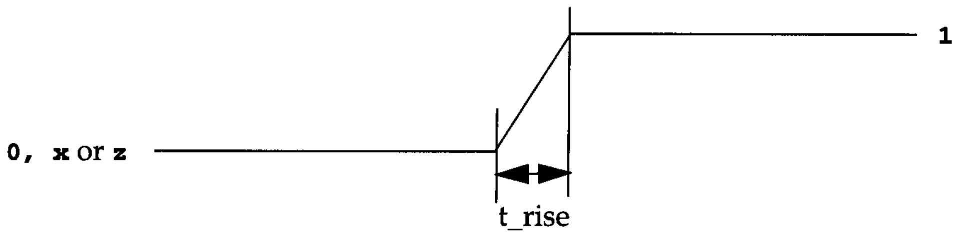
- Gate delays allow the user to specify delays through the logic circuits.
- There are three types of delays from the inputs to the output of a primitive gate.
    i. Rise

ii. Fall

iii. Turn-off Delays

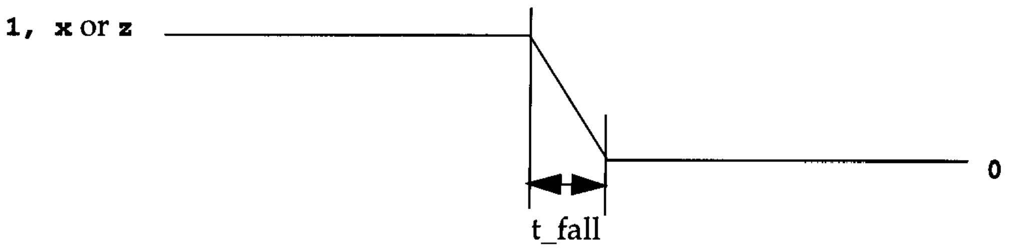
Rise delay

- The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

- The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

- The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

➢ If the value changes to x, the minimum of the three delays is considered.

➢ Three types of delay specifications are allowed in Verilog.

i. If only one delay is specified, this value is used for all transitions.

```
        // Delay of delay_time for all transitions
and #(delay_time) a1 (out, i1, i2);
and #(5) a1(out, i1, i2);              // Delay of 5 for all transitions
```

ii. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.

```
        // Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2 (out, i1, i2);
```

and #(4, 6) a2 (out, i1, i2); // Rise = 4, Fall = 6

iii. If all three delays are specified, they refer to rise, fall, and turn-off delay values.

// Rise, Fall, and Turn-off Delay Specification

and #(rise_val, fall_val, turnoff_val) a3 (out, i1, i2);

and #(3, 4, 5) a3 (out, i1, i2); // Rise = 3, Fall = 4, Turn-off = 5

iv. If no delays are specified, the default value is zero.

and a4 (out, i1, i2);

and a4 (out, i1, i2); // Rise = 0, Fall = 0, Turn-off = 0

Min/Typ/Max Values

- Verilog provides an additional level of control for each type of delay. For each type of delay (rise, fall, and turn-off) three values (min, typ, and max) can be specified. Any one value can be chosen at the start of the simulation.
- Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.
- **Min value:** The min value is the minimum delay value that the designer expects the gate to have.
- **Typ value:** The typ value is the typical delay value that the designer expects the gate to have.
- **Max value:** The max value is the maximum delay value that the designer expects the gate to have.
- The values are chosen by specifying options +maxdelays, +typdelays, and +mindelays at run time. If no option is specified, the typical delay value is the default.
- For example,

```
// One delay is specified
// if +mindelays, delay = 4
// if +typdelays, delay = 5
// if +maxdelays, delay = 6
        and #(4:5:6) a1(out, i1, i2);
// Two delays are specified
// if +mindelays, rise = 3, fall = 5, turn-off = min(3,5)
// if +typdelays, rise = 4, fall = 6, turn-off = min(4,6)
// if +maxdelays, rise = 5, fall = 7, turn-off = min(5,7)
        and #(3:4:5, 5:6:7) a2(out, i1, i2);
// Three delays are specified
// if +mindelays, rise = 2 fall = 3 turn-off = 4
// if +typdelays, rise = 3 fall = 4 turn-off = 5
// if +maxdelays, rise = 4 fall = 5 turn-off = 6
        and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

Delay Example

- Consider a simple example to illustrate the use of gate delays to model timing in the logic circuits.
- A simple module called gates which implements the following logic equation

  out = (a·b) + c
- The gate-level implementation of the logic equation is shown in fig. 3.9. The module contains two gates with delays of 5 and 4 time units.

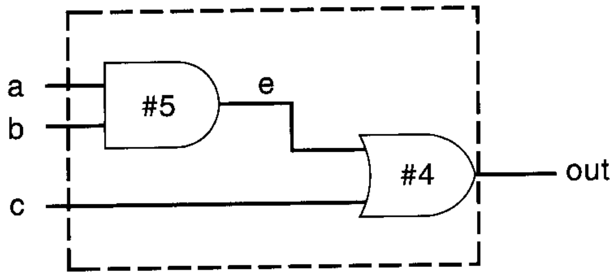


Fig.3.9: Module gates

- Verilog definition for module gates with delay is as given below

```
        // Define a simple combination module called gates
module gates (out, a, b, c);
        // I/O port declarations
output out;
input a, b, c;
        // Internal nets
wire e;
        // Instantiate primitive gates to build the circuit
```

```
and #(5) a1(e, a, b);          // Delay of 5 on gate a1
or #(4) o1(out, e, c);         // Delay of 4 on gate o1
endmodule
```

- This module is tested by the stimulus code defined below

```
        // Stimulus (top-level module)
module stimulus;
        // Declare variables
reg a, b, c;
wire out;
        // Instantiate the module gates
gates d1( out, a, b, c);
        // Stimulate the inputs. Finish the simulation at 40 time units.
initial
begin
        a = 1'b0; b = 1'b0; c = 1'b0;
        #10 a = 1'b1; b = 1'b1; c = 1'b1;
        #10 a = 1'b1; b = 1'b0; c = 1'b0;
        #20 $finish;
end
endmodule
```

- Simulation waveform for the above design is shown in fig.3.10. The outputs E and OUT are initially unknown.
- At time 10, A, B, and C all have transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after a delay of 5 time units.
- At time 20, B and C have transition to 0. E changes value to 0 after 5 time units and OUT changes value to 0, 4 time units after E changes.

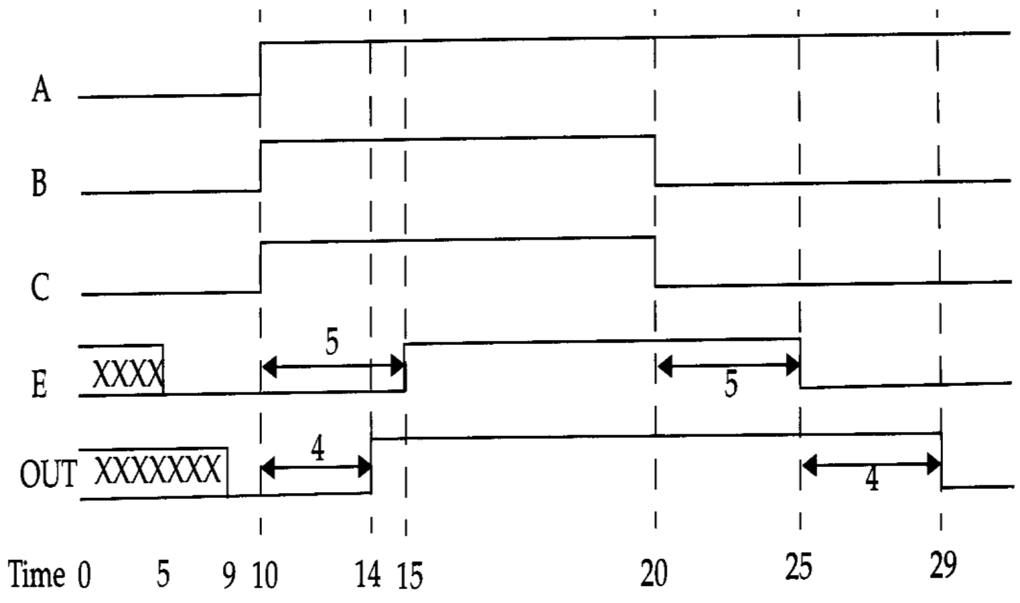


Fig.3.10: Simulation waveforms with delay

Dataflow Modeling

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connects every gate individually.
- Gate-level modeling is very easy to use and understand for a designer with a basic knowledge of digital logic design.
- However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level.
- Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.
- With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance.
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis.

- Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow.
- For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design.
- In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous Assignments

- A continuous assignment is a basic statement in dataflow modelling which is used to drive a value onto a net.
- This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
- The assignment statement starts with the keyword assign.
- Syntax of an continuous assignment statement is

  assign [#(delay)] net_name = expression;

  where parameters inside [ ] is optional.
- Continuous assignments have the following characteristics:
  1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
  2. The operands on the right hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
  3. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right hand side operands changes and the value is assigned to the left hand side net.
  4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.
- For example

```
// Continuous assign using nets. out is a net. i1 and i2 are nets.
        assign out = i1 & i2;
// Continuous assign using vector nets. num is a 16-bit vector net
// num1 and num2 are 16-bit vector registers.
        assign num[15:0] = num1[15:0] ^ num2[15:0];
// Continuous assign using Concatenation.
        assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit Continuous Assignment

- The implicit continuous assignment combines the net declaration and continuous assignment into one statement. The explicit assignment requires two statements: one to declare the net and one to continuously assign a value to it.

```
// Regular continuous assignment
        wire out;
        assign out = in1 & in2;
// Same effect is achieved by an implicit continuous assignment
        wire out = in1 & in2;
```

- Syntax of an implicit continuous assignment statement is

net_data_type [delay] [size] net_name = expression;

Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.
- Delays in a continuous assignment statement can be specified in three ways
    1. regular assignment delay
    2. implicit continuous assignment delay
    3. net declaration delay

Regular Assignment Delay

- In a continuous assignment statement the delay value is specified after the keyword assign.
- Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.

- If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay.
- An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2;          // Delay in a continuous assign
```

Implicit Continuous Assignment Delay

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.
- For example

```
// implicit continuous assignment delay
        wire #10 out = in1 & in2;
// The above statement has the same effect as the following.
        wire out;
        assign #10 out = in1 & in2;
```

Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.
- Net declaration delays can also be used in gate level modeling.
- For example

```
// Net Delays
        wire # 10 out;
        assign out = in1 & in2;
// The above statement has the same effect as the following.
        wire out;
        assign #10 out = in1 & in2;
```

Expressions, Operators, and Operands

- Dataflow modeling describes the design in terms of expressions instead of primitive gates.
- Expressions, operators, and operands form the basis of dataflow modeling.

Expressions

- Expressions are constructs that combine operators and operands to produce a result.
- For example,

```
a ^ b
num1[3:0] + num2[3:0]
in1 | in2
```

Operands

- Operands hold the data and can be any one of the data types.
- Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls
- For example,

```
integer count, final_count;
final_count = count + 1;            // count is an integer operand
real a, b, c;
c = a - b;                          // a and b are real operands
reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];    // reg1[3:0] and reg2[3:0] are part-select
                                    // register operands
reg ret_value;
ret_value = calculate_parity(A, B); // calculate_parity is a function type operand
```

Operators

- Operators act on the operands to produce desired results.
- For example,

```
d1 && d2          // && is an operator on operands d1 and d2
!a[0]             // ! is an operator on operand a[0]
```

B >> 1 // >> is an operator on operands B and 1

Operator Types

- Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional.
- Operator types and symbols supported by Verilog are given in table 3.4

Table 3.4: Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power	two
Logical	!	Logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two

	<<	Left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Arithmetic Operators

- There are two types of arithmetic operators
  1. binary operators
  2. unary operators

Binary Operators

- Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%).
- Binary operators use two operands.
- For example,

```
A = 4'b0011; B = 4'b0100;     // A and B are register vectors
D = 6; E = 4; F = 2           // D and E are integers
A + B                         // Add A and B. Evaluates to 4'b0111
B - A                         // Subtract A from B. Evaluates to 4'b0001
A * B                         // Multiply A and B. Evaluates to 4'b1100
D / E          // Divide D by E. Evaluates to 1. Truncates any fractional part
F = E ** F;                   // E to the power F, yields 16
```

- If any operand bit has a value x, then the result of the entire expression is x.
- For example,

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2;        // sum will be evaluated to the value 4'bx
```

- Modulus operators produce the remainder from the division of two numbers.
- For example,

```
13 % 3          // Evaluates to 1
16 % 4          // Evaluates to 0
-7 % 2          // Evaluates to -1, takes sign of the first operand
```

```
7 % -2          // Evaluates to +1, takes sign of the first operand
```

Unary Operators

- The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand.
- Unary + or – operators have higher precedence than the binary + or – operators.
- For example,

```
-4      // Negative 4
+5      // Positive 5
```

- Negative numbers are represented as 2's complement internally in Verilog.

Logical Operators

- Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators. Operator ! is a unary operator.
- Logical operators always evaluate to a 1bit value, 0 (false), 1 (true), or x (ambiguous).
- If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
- Logical operators take variables or expressions as operands.
- For example,

```
A = 3;              // Non zero value
B = 0;              // Zero value
A && B              // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B              // Evaluates to 1. Equivalent to (logical-1 || logical-0)
! A                 // Evaluates to 0. Equivalent to not (logical-1)
! B                 // Evaluates to 1. Equivalent to not (logical-0)
// For unknown input
A = 2'b0x; B = 2'b10;
A && B              // Evaluates to x. Equivalent to (x && logical 1)
// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.
```

// Evaluates to 0 if either is false.

Relational Operators

- Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).
- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.
- If there are any x or z bits in the operands, then result is x.
- For example,

```
A = 4, B = 3
X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
A <= B              // Evaluates to a logical 0
A > B               // Evaluates to a logical 1
Y >= X              // Evaluates to a logical 1
Y < Z               // Evaluates to an x
```

Equality Operators

- Equality operators are logical equality (= =), logical inequality (!=), case equality (= = =), and case inequality (!= =).
- When used in an expression, equality operators return logical value 1 if true, 0 if false.
- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.

Table 3.5: Equality Operators

Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

- The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits. However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z. The result is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly. Case equality operators never result in an x.
- For example,

```
A = 4, B = 3
X = 4'b1010, Y = 4'b1101
Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx
A == B        // Results in logical 0
X != Y        // Results in logical 1
X == Z        // Results in x
Z === M       // Results in logical 1 (all bits match, including x and z)
Z === N       // Results in logical 0 (least significant bit does not match)
M !== N       // Results in logical 1
```

Bitwise Operators

- Bitwise operators are negation (~), and (&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands.
- They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.
- The unary negation operator (~), takes only one operand and operates on the bits of the single operand.
- For example,

```
X = 4'b1010, Y = 4'b1101, Z = 4'b10x1
~X            // Negation. Result is 4'b0101
X & Y         // Bitwise and. Result is 4'b1000
X | Y         // Bitwise or. Result is 4'b1111
```

```
X ^ Y       // Bitwise xor. Result is 4'b0111
X ^~ Y      // Bitwise xnor. Result is 4'b1000
X & Z       // Result is 4'b10x0
```

- The difference between logical operators and bitwise operators is, logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.
- For example,

```
X = 4'b1010, Y = 4'b0000
X | Y       // bitwise operation. Result is 4'b1010
X || Y      // logical operation. Equivalent to 1 || 0. Result is 1.
```

Reduction Operators

- Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).
- Reduction operators take only one operand and it performs a bitwise operation on a single vector operand and yield a 1-bit result.
- The difference between bitwise operations and reduction operations is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand.
- Reduction operators work bit by bit from right to left. Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

```
X = 4'b1010
&X          // Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X          // Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X          // Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

- A reduction xor or xnor can be used for even or odd parity generation of a vector.

Shift Operators

- Shift operators are right shift (>>), left shift (<<). These operators shift a vector operand to the right or the left by a specified number of bits.
- The operands are the vector and the number of bits to shift.

- When the bits are shifted, the vacant bit positions are filled with zeros.
- For example,

```
X = 4'b1100
Y = X >> 1;        // Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1;        // Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2;        // Y is 4'b0000. Shift left 2 bits.
```

Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands.
- The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
- Concatenations are expressed as operands within braces, with commas separating the operands.
- Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.
- For example,

```
a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = {b , c}                   // result y is 4'b0010
y = {a , b , c , d , 3'b001}  // result y is 11'b10010110001
y = {a , b[0], c[1]}          // result y is 3'b101
```

Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({{ }}).
- For example,

```
reg a;
reg [1:0] b, c;
reg [2:0] d;
```

```
a = 1'b1; b = 2'b00; c = 2'b10; d = 3'b110;
y = { 4{a} }                  // result y is 4'b1111; i.e. replicate a value 4 times
y = { 4{a} , 2{b} }            // result y is 8'b11110000
y = { 4{a} , 2{b} , c }        // result y is 8'b1111000010
```

Conditional Operator

- The conditional operator (?) takes three operands.

  Syntax: condition_expression ? true_expression : false_expression;
- The condition_expression is first evaluated. If the result is true (logical 1), then the true_expression is evaluated. If the result is false (logical 0), then the false_expression is evaluated. If the result is x (ambiguous), then both true_expression and false_expression are evaluated and their results are compared, bit by bit (returns x if the bits are different and returns the same bit value if they are the same).
- The conditional expression acts as a switching control. The action of a conditional operator is similar to a multiplexer as shown in fig.3.11. Alternately, it can be compared to the if-else expression.

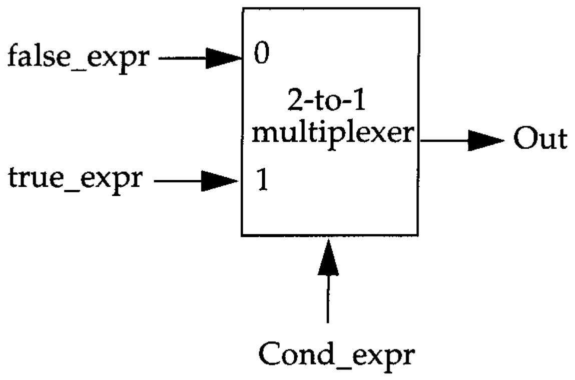


Fig.3.11: 2:1 mux

- For example

  // 2:1 mux function using conditional operator

  ```
  assign out = control ? in1 : in0;
  ```
- Conditional operations can be nested. Each true_expression or false_expression can itself be a conditional operation.
- For example

  ```
  assign out = (A == 3) ? (control ? x : y ) : ( control ? m : n);
  ```

Operator Precedence

- If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence.
- Operators listed in table 3.6 are in order from highest precedence to lowest precedence.

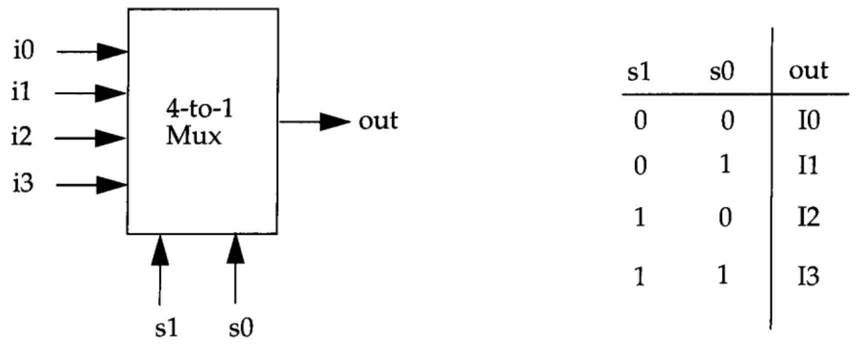
Table 3.6: Operator Precedence

Operators	Operator Symbols	Precedence
Unary Multiply, Divide, Modulus	+ - ! ~ * / %	Highest precedence
Add, Subtract Shift	+ - << >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction Logical	& ~& | ~| ^ ^~ && ||	
Conditional	?:	Lowest precedence

Verilog HDL Code to design 4:1 Multiplexer

- The dataflow description for the 4:1 multiplexer is written using boolean expression of the output signal.
- For the 4:1 multiplexer, the boolean expression of the output signal can be written as

$$\text{out} = (\overline{s1} * \overline{s0} * i0) + (\overline{s1} * s0 * i1) + (s1 * \overline{s0} * i2) + (s1 * s0 * i3)$$



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Fig.3.12: 4:1 Multiplexer

Method 1: Using logic equation

- Assignment statements are used instead of gates to model the logic equations of the multiplexer.

- Dataflow Verilog description is similar to the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations.
- I/O ports remain the same. Only the internals of the module change.

```
// Module 4:1 multiplexer using logic equation
        module mux4_to_1 (i0, i1, i2, i3, s1, s0, out);
// Port declarations from the I/O diagram
        input i0, i1, i2, i3;
        input s1, s0;
        output out;
// Logic equation for out
        assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) | (s1 & ~s0 & i2) | (s1 & s0 & i3);
        endmodule
```

Method 2: Using conditional operator

```
// Module 4:1 multiplexer using conditional operator.
        module mux4_to_1  (i0, i1, i2, i3, s1, s0, out);
// Port declarations from the I/O diagram
        input i0, i1, i2, i3;
        input s1, s0;
        output out;
// Use nested conditional operator
        assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;
        endmodule
```

- The stimulus module will not change. The simulation results will be identical.

```
// Define the stimulus module (no ports)
        module mux4_to_1_tb();
// Declare variables to be connected to inputs
        reg i0, i1, i2, i3;
        reg s1, s0;
// Declare output wire
```

```
		wire out;
	// Instantiate the multiplexer
		mux4_to_1 m1 (i0, i1, i2, i3, s1, s0, out);
	// Define the stimulus
		initial
		begin
			i0 = 1; i1 = 0; i2 = 1; i3 = 0; s1 = 0; s0 = 0;
			#20 s1 = 0; s0 = 1;
			#20 s1 = 1; s0 = 0;
			#20 s1 = 1; s0 = 1;
		end
		endmodule
```

Verilog HDL Code to design 4-bit Full Adder

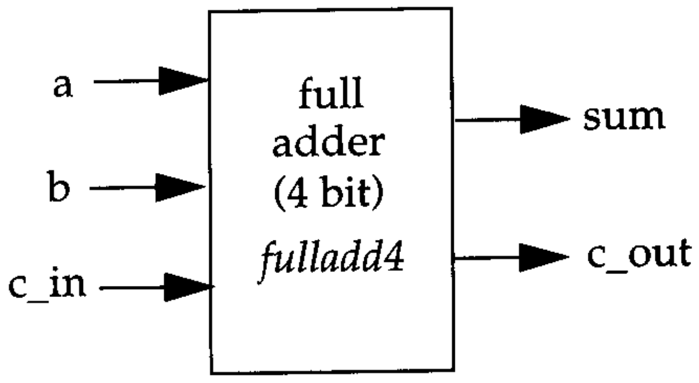


Fig.3.13: 4-bit adder

- The dataflow description for the 4-bit adder shown in fig.3.13 is given below.

Method 1: Using Dataflow Operators

```
	// Define a 4-bit full adder by using dataflow statements.
		module fulladd4 (a, b, c_in, sum, c_out);
	// I/O port declarations
		input[3:0] a, b;
		input c_in;
		output [3:0] sum;
		output c_out;
```

```
// Specify the function of a full adder
        assign {c_out, sum} = a + b + c_in;
        endmodule
```

Method 2: Full Adder with Carry Look ahead logic

- In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals.
- An n-bit ripple carry adder will have 2n gate levels. The propagation time can be a limiting factor on the speed of the circuit.
- One of the most popular methods to reduce delay is to use a carry look ahead mechanism.

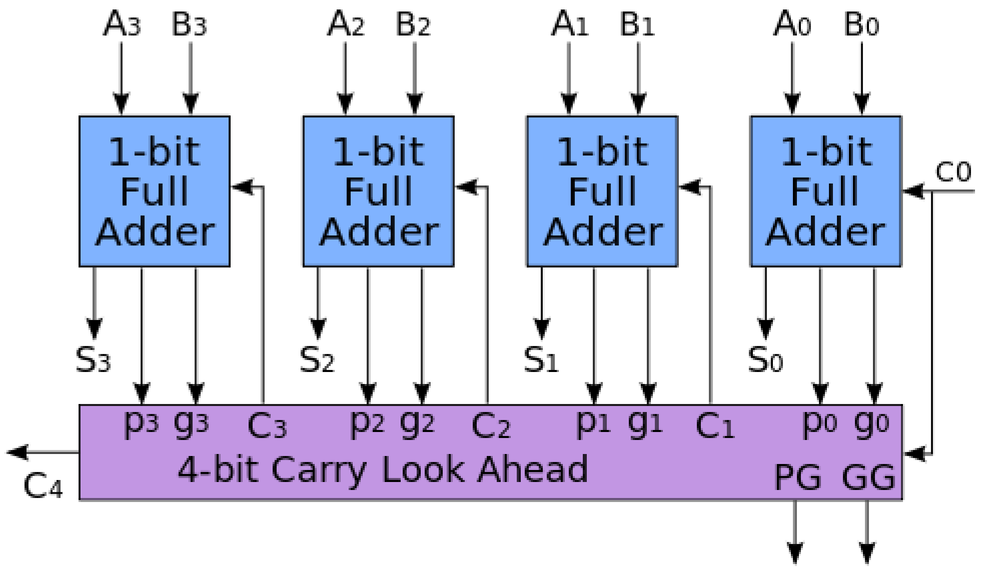


Fig.3.14: Carry look-ahead adder

- The propagation delay is reduced to four gate levels, irrespective of the number of bits in the adder.
- Carry look-ahead adder (CLA adder) show in fig.3.14 is based on the fact that a carry signal will be generated in two cases:
    - i. When both bits A_i and B_i are 1, or
    - ii. When one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.
- To understand the carry propagation problem, let's consider the case of adding two n-bit numbers A and B as shown in fig.3.15.

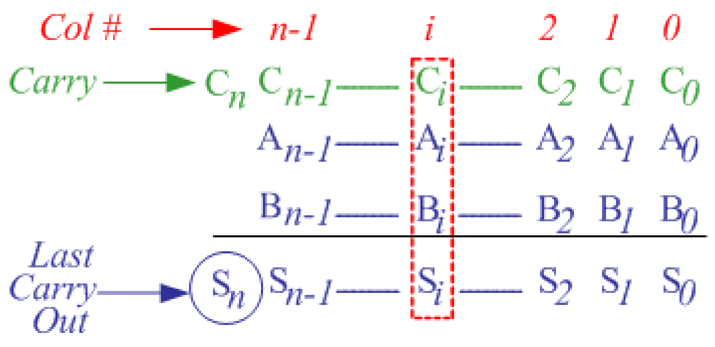


Fig.3.15: Addition of two n-bit numbers A and B

- Fig.3.16 shows the full adder circuit used to add the operand bits in the ith column; namely A_i & B_i and the carry bit coming from the previous column (C_i).

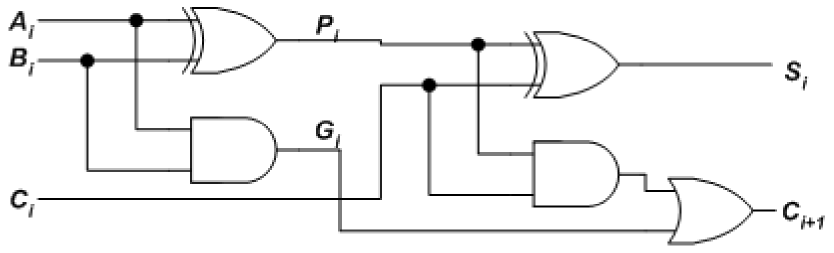


Fig.3.16: Addition of ith bit using full adder

- In this circuit, the 2 internal signals P_i and G_i are given by

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

- The output sum and carry can be defined as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

- G_i is known as the carry generate signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i).
- P_i is known as the carry propagate signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0).
- Therefore computing the values of P_i and G_i only depend on the input operand bits (A_i & B_i).
- The carry expression can be written as

$$\mathbf{C_1 = G_0 + P_0C_0}$$

$$\mathbf{C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)}$$

$$= G_1 + P_1G_0 + P_1P_0C_0$$

$$C_3 = G_2 + P_2C_2$$

$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3C_3$$

$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- The dataflow description of 4-bit adder is as follows

```
module fulladd4(a, b, c_in, sum, c_out);
// Inputs and outputs
        input [3:0] a,b;
        input c_in;
        output [3:0] sum;
        output c_out;
// Internal wires
        wire p0,g0, p1,g1, p2,g2, p3,g3;
        wire c4, c3, c2, c1, c0;
        assign c0 = c_in;
// compute the p for each stage
        assign p0 = a[0] ^ b[0],
                p1 = a[1] ^ b[1],
                p2 = a[2] ^ b[2],
                p3 = a[3] ^ b[3];
// compute the g for each stage
        assign g0 = a[0] & b[0],
                g1 = a[1] & b[1],
                g2 = a[2] & b[2],
                g3 = a[3] & b[3];
// compute the carry for each stage
        assign c1 = g0 | (p0 & c0),
                c2 = g1 | (p1 & g0) | (p1 & p0 & c0),
                c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c0),
                    c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
                    (p3 & p2 & p1 & p0 & c0);
// Compute Sum
          assign sum[0] = p0 ^ c0,
                    sum[1] = p1 ^ c1,
                    sum[2] = p2 ^ c2,
                    sum[3] = p3 ^ c3;
// Assign carry output
          assign c_out = c4;
endmodule
```

Verilog HDL Code to design Ripple Counter

- 4-bit ripple counter shown in fig.3.17 is designed by using four negative edge-triggered T-flipflops.

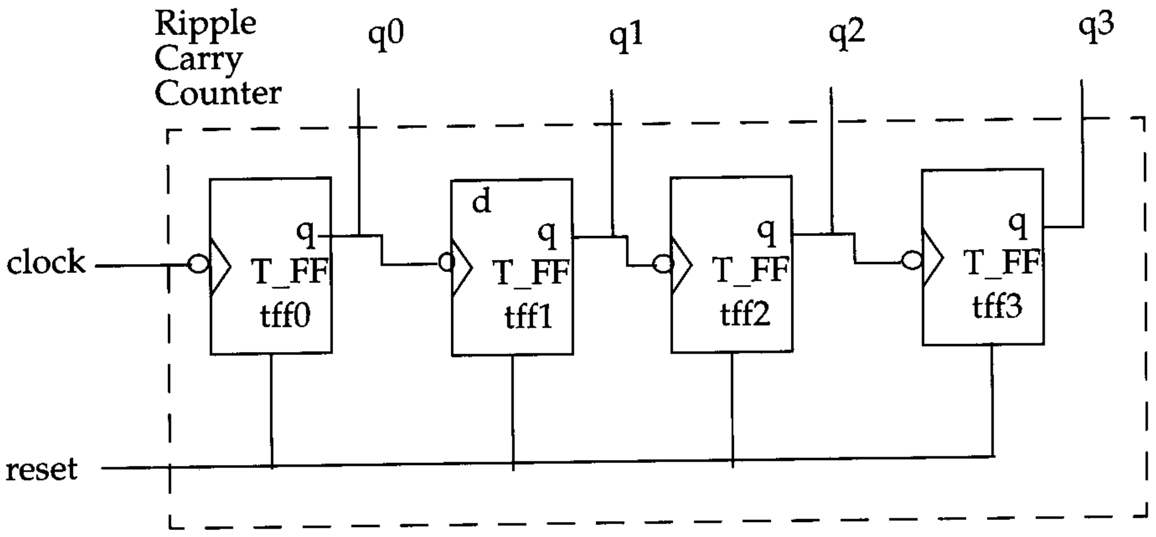


Fig.3.17: 4-bit Ripple Carry Counter

- Fig.3.18 shows that the T-flipflop is built with one D-flipflop and an inverter gate.

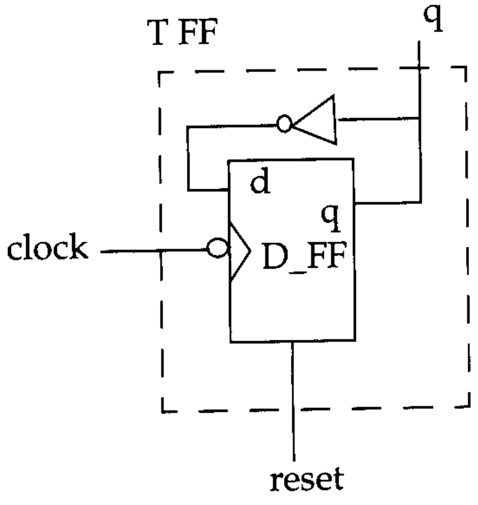


Fig.3.18: T-flipflop

- Fig.3.19 shows the D-flipflop constructed from logic gates.

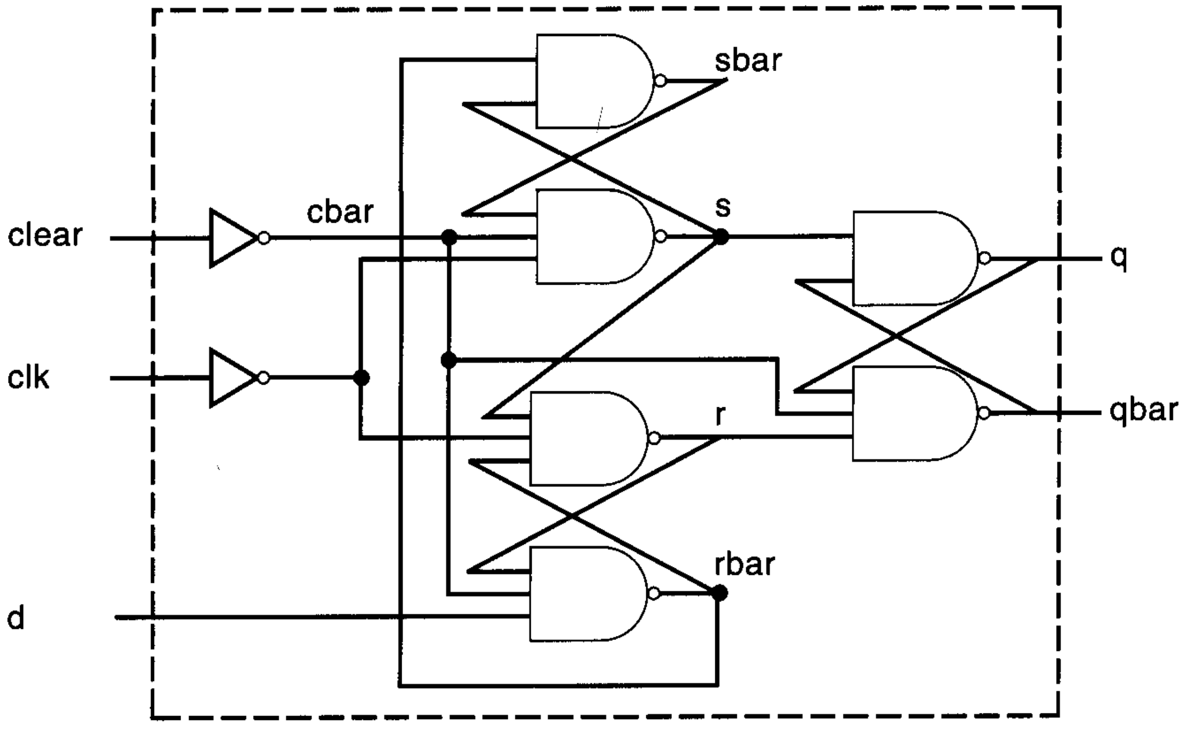


Fig.3.19: Negative Edge-Triggered D-flipflop with Clear

- With reference to the above diagrams, 4-bit counter is designed using dataflow statements in a top-down fashion.
- First we design the module counter which contains instantiation of four T_FF modules.

```
// Ripple counter
        module counter (clock, reset, q);
```

```
// I/O ports
        input clock, reset;
        output [3:0] q;
// Instantiate the T flipflops
        T_FF tff0 (clock, reset, q[0]);
        T_FF tff1 (q[0], reset, q[1]);
        T_FF tff2 (q[1], reset, q[2]);
        T_FF tff3 (q[2], reset, q[3]);
        endmodule
```

- Verilog description for T_FF is written by instantiating d-flipflop. Instead of instantiating the not gate, a dataflow operator ~ negates the signal q, which is fed back.

```
// Edge-triggered T-flipflop. Toggles every clock cycle.
        module T_FF(clk, reset, q);
// I/O ports
        input clk, reset;
        output q;
// Instantiate the edge-triggered DFF and complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
        edge_dff ff1 (clk, clear, q, ,~q,);
        endmodule
```

- The lowest level module D_FF (edge_dff), is designed using dataflow statements. The dataflow statements correspond to the logic diagram shown in fig.3.1.

```
// Edge-triggered D flipflop
        module edge_dff(clk, clear, q, qbar, d);
// Inputs and outputs
        input d, clk, clear;
        output q,qbar;
// Internal variables
        wire s, sbar, r, rbar,cbar;
// Create a complement of signal clear
        assign cbar = ~clear;
```

```
// An edge-sensitive flip-flop is implemented by using 3 SR latches.
        assign sbar = ~(rbar & s),
                s = ~(sbar & cbar & ~clk),
                r = ~(rbar & ~clk & s),
                rbar = ~(r & cbar & d);
// Output latch
        assign q = ~(s & qbar),
        qbar = ~(q & r & cbar);
        endmodule
```

- The design block is tested with the stimulus given below

```
// Top level stimulus module
        module stimulus;
// Declare variables for stimulating input
        reg clock, reset;
        wire [3:0] q;
// Instantiate the design block counter
        counter c1(clock, reset, q);
// Stimulate the reset signal
        initial
        begin
                reset = 1'b1;
                #20 reset = 1'b0;
                #100 reset = 1'b1;
        end
// Set up the clock to toggle every 10 time units
        initial
        begin
                clock = 1'b0;
                forever #10 clock = ~clock;
        end
        endmodule
```