Department of Electronics & Communication Engg.

Course : Verilog HDL-15EC53. **Sem.: 5th**

Course Coordinator:

Prof. Sachin S Patil

Verilog HDL -Introduction

Ref: Verilog – HDL by samir palnitkar 2nd Edition

Module- Basic building block

```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

A module can be an element or collection of low level design blocks

Levels of Abstraction-1

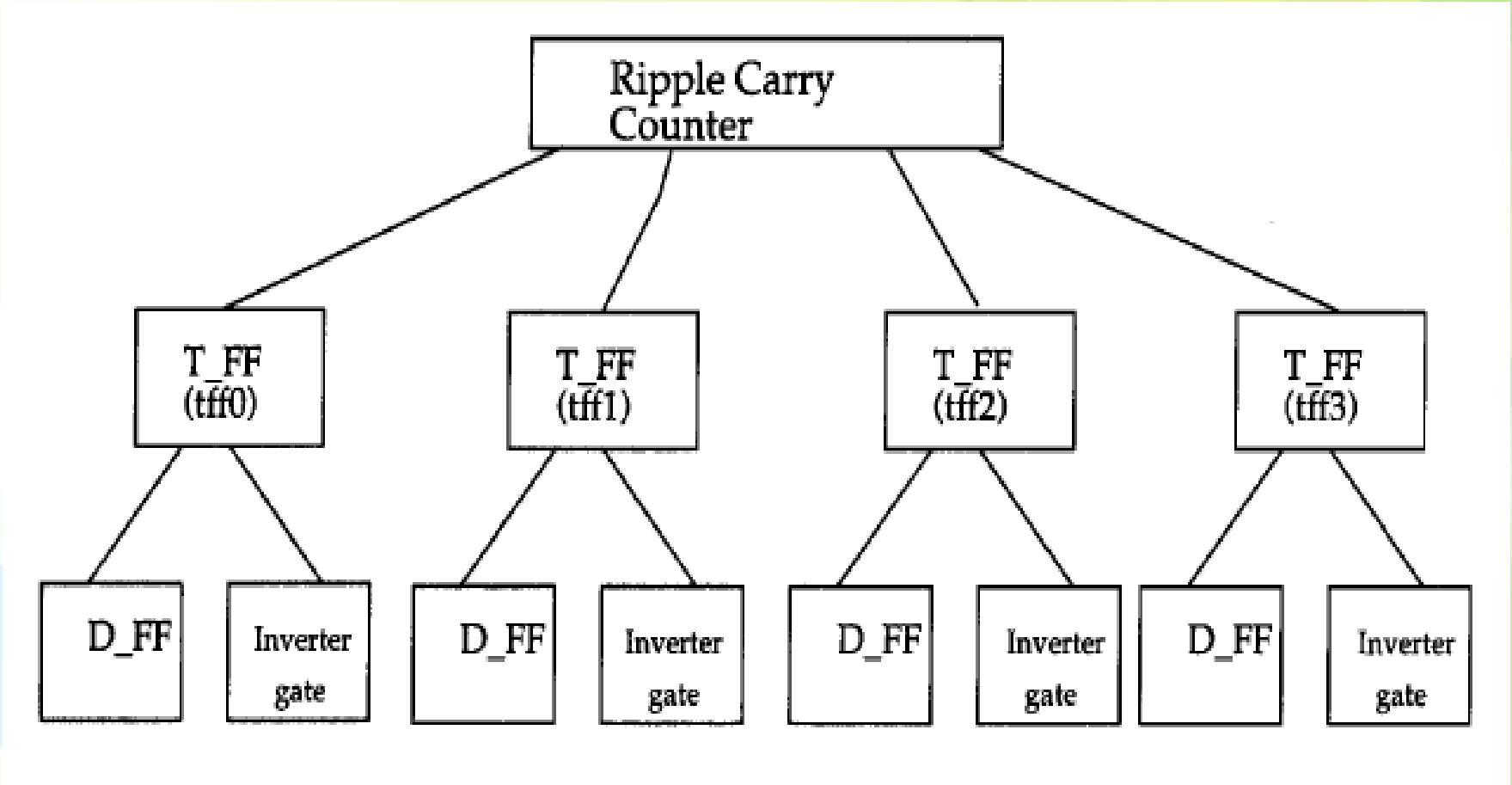
- Switch Level: Module implemented with switches and interconnects. Lowest level of Abstraction

- Gate Level: Module implemented in terms of logic gates like (and ,or) and interconnection between gates

Levels of Abstraction-2

- **Dataflow Level**: Module designed by specifying dataflow. The designer is aware of how data flows between hardware registers and how the data is processed in the design
- **Behavioral Level :**Module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Very similar to C programming

Hierrarchy


Ripple Carry Counter
T_FF (tff0)
T_FF (tff1)
T_FF (tff2)
T_FF (tff3)
D_FF
Inverter gate
D_FF
Inverter gate
D_FF
Inverter gate
D_FF
Inverter gate

Basic Concepts

- Number is specified as

<size>'<baseformat><number>

```
4'b1111 // This is a 4-bit   binary number
12'habc // This is a 12-bit  hexadecimal number
16'd255 // This is a 16-bit  decimal number.
```

```
23456 // This is a 32-bit   decimal number by default
'hc3 // This is a 32-bit   hexadecimal number
'o21 // This is a 32-bit   octal number
```

Contd.

```
-6'd3 // 8-bit   negative number stored as 2's complement of 3
```

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown value
z	High impedance, floating state

Nets

- Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.

```
wire a; // Declare net a for the above circuit
wire b,c; // Declare two wires b,c for the above circuit
```

Registers

- *Registers* represent data storage elements. Registers retain value until another value is placed onto them.
- In Verilog, the term *register* merely means a variable that can hold a value.
- Unlike a net, a register does not need a driver.

```
reg reset; // declare a variable reset that can hold its value
```

```
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
```

Vectors

- Arrays of Regs and Nets

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit   bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; //Vector register,virtual address 41 bits wide
```

Integers and Parameters

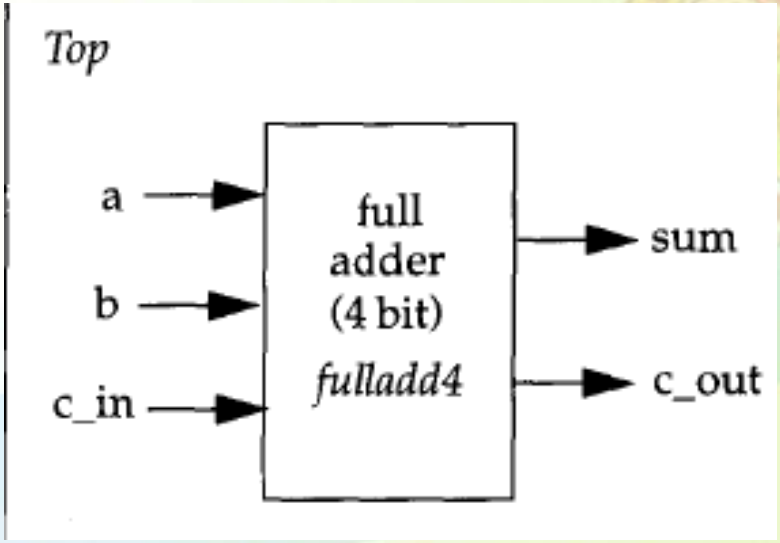
```
integer counter; // general purpose variable used as a counter.
initial
    counter = -1; // A negative one is stored in the counter
```

```
parameter port_id = 5; //Defines a constant port_id
parameter cache_line_width=256; //Constant defines width of cache line
```

Ports

- Ports provide interface for by which a module can communicate with its environment

```
module fulladd4(sum, c_out, a, b, c_in);
```



Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

Module

Module Name,
Port List, Port Declarations (if ports present)
Parameters(optional),

Declarations of **wire**s, **reg**s and other variables

Data flow statements (**assign**)

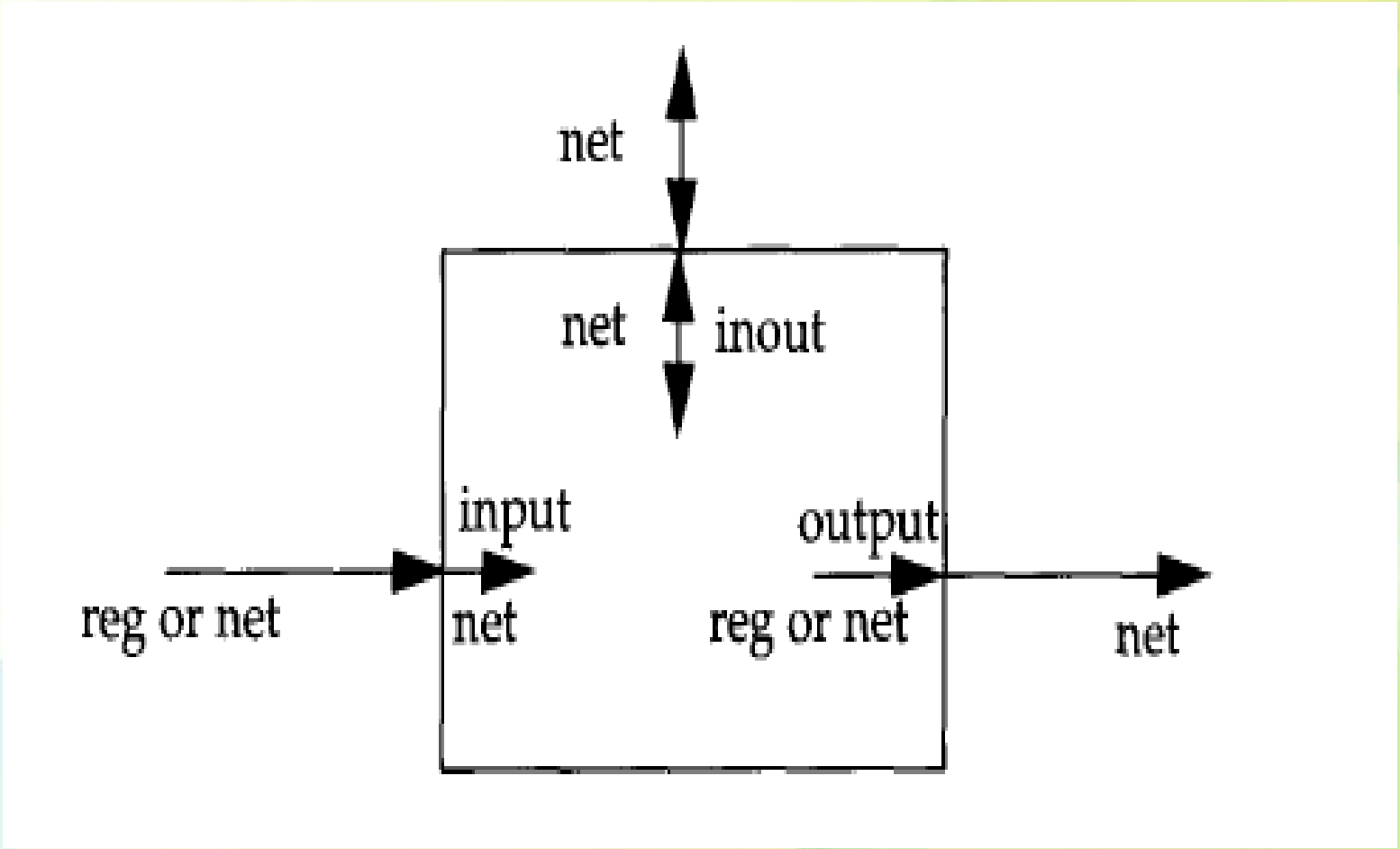
Instantiation of lower level modules

always and **initial** blocks. All behavioral statements go in these blocks.

Tasks and functions

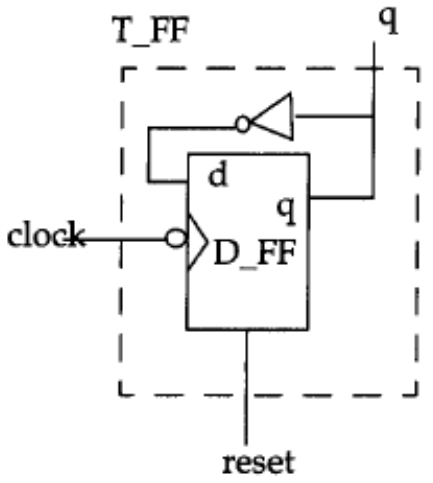
endmodule statement

Port connection rules



Example

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value;
input d, clk, reset;
...
...
endmodule
```



Connecting Ports

- Suppose we have a module

```
module fulladd4(sum, c_out, a, b, c_in);
```

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa_ordered.
    //Signals are connected to ports in order (by position)
    fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
```

```
fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected
```

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

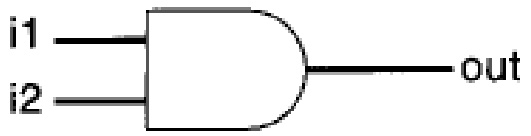
```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),
.a(A),);
```

Gate Level Modeling

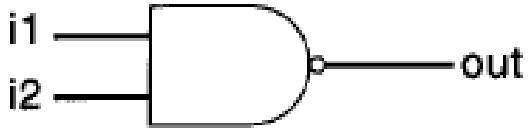
- A logic circuit can be designed by use of logic gates.
- Verilog supports basic logic gates as predefined *primitives.* These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.

and	or	xor
nand	nor	xnor

Gate gate_name(out,in1,in2…)



and



nand

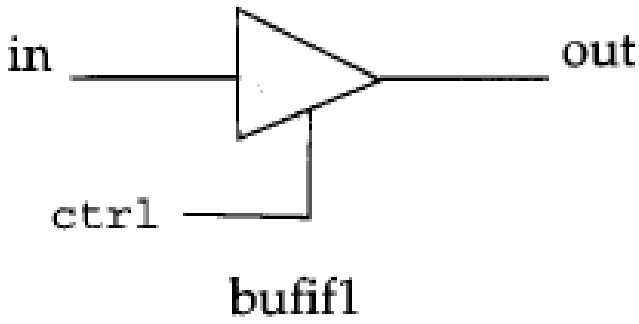
```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
```

Buf/not gates

- *Buf/not* gates have one scalar input and one or more scalar outputs.

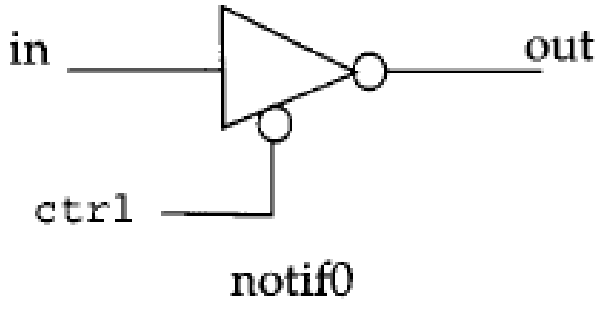
```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```

Bufif/notif



		ctrl			
	bufif1	0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

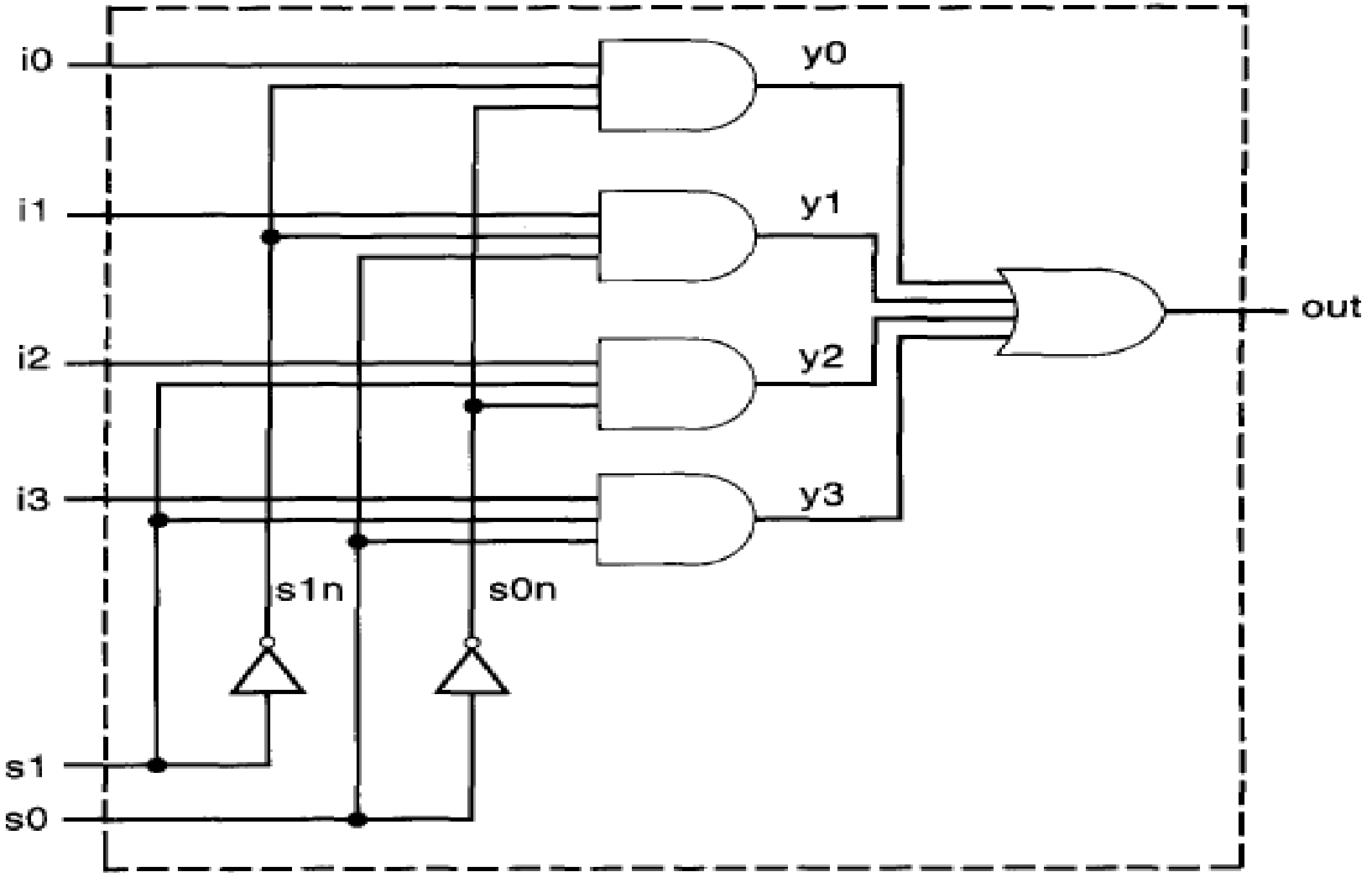
		ctrl			
	notif0	0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x



Instantiation of bufif gates

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);

//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

Design of 4:1 Multiplexer



Contd..

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
```

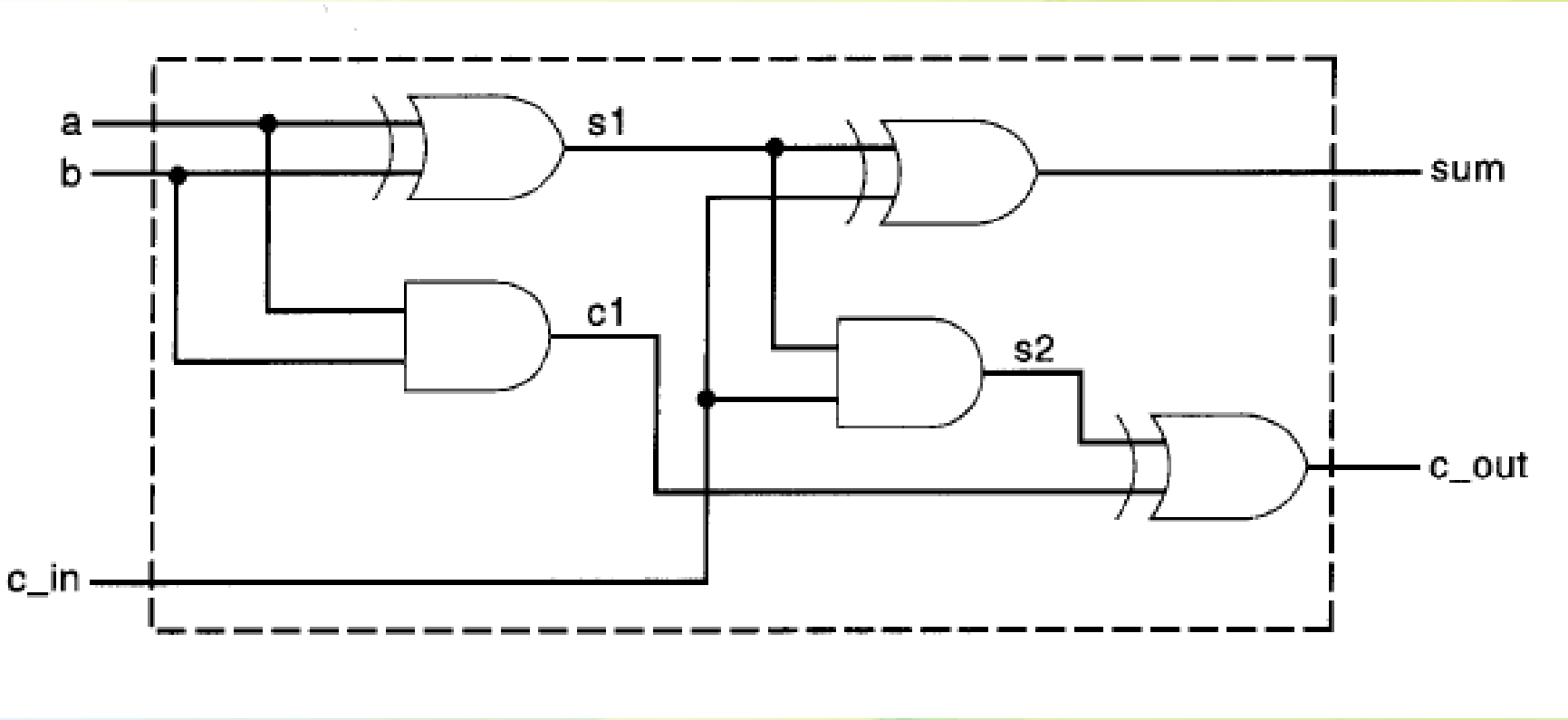
```
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

Stimulus

```
// Define the stimulus module (no ports)
module stimulus;

// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
reg S1, S0;

// Declare output wire
wire OUTPUT;

// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
```

```
// Define the stimulus module (no ports)

// Stimulate the inputs
initial
begin
  // set input lines
  IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
  #1 $display("IN0= %b, IN1= %b, IN2= %b, IN3=
%b\n",IN0,IN1,IN2,IN3);

  // choose IN0
  S1 = 0; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN1
  S1 = 0; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN2
  S1 = 1; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN3
  S1 = 1; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

4 bit full adder



$$sum = (a \oplus b \oplus cin)$$

$$cout = (a \cdot b) + cin \cdot (a \oplus b)$$

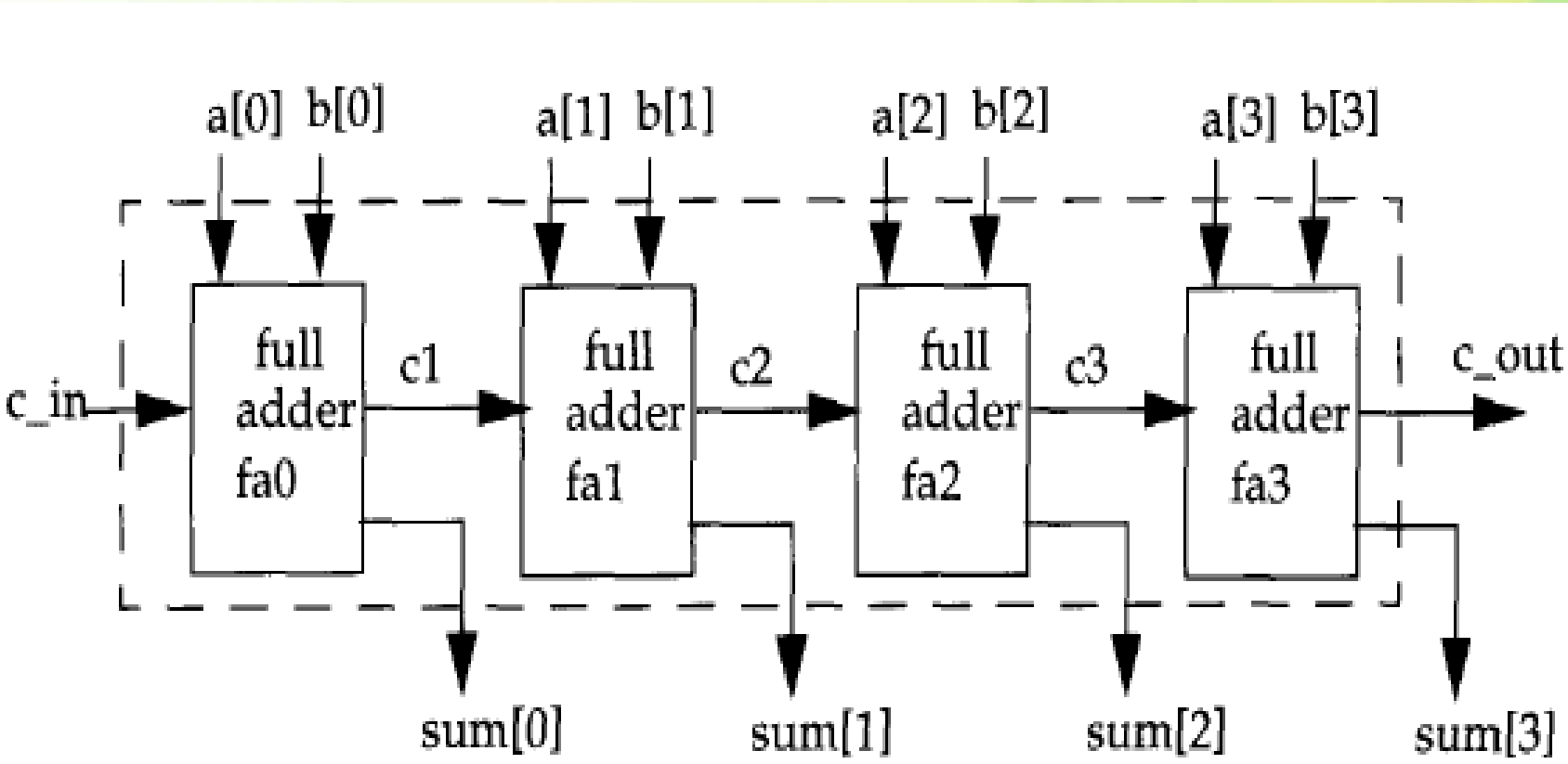
Declaration:

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations
output sum, c_out;
input a, b, c_in;

// Internal nets
wire s1, c1, c2;
```

Code contd..

```
// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);

xor (sum, s1, c_in);
and (c2, s1, c_in);

or  (c_out, c2, c1);

endmodule
```

4 bit adder using 1 bit adder



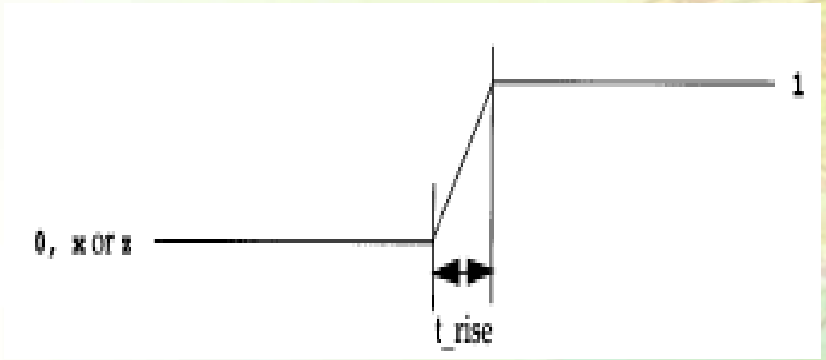
```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Internal nets
wire c1, c2, c3;

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

endmodule
```

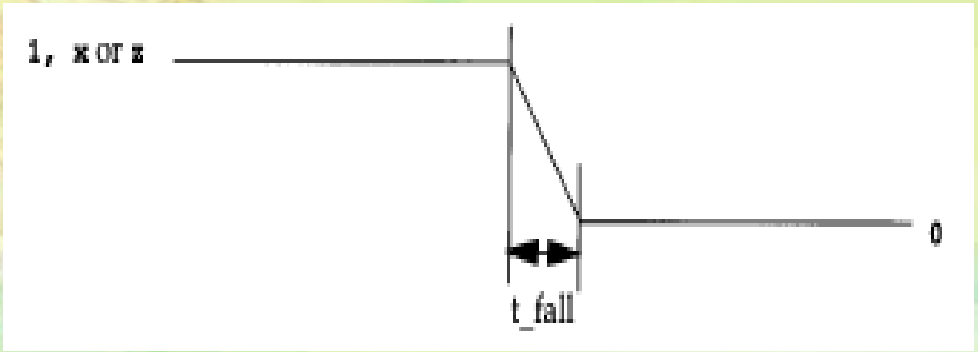
Stimulus

```
// Define the stimulus (top level module)
module stimulus;

// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
```

Gate Delays:

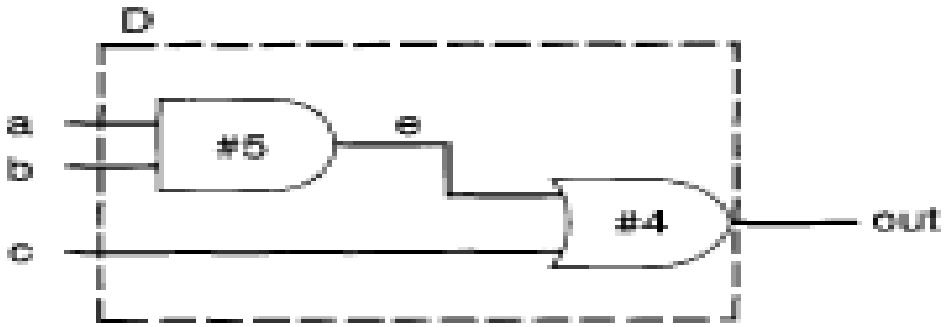
- Rise Delay: Delay associated with a o/p transition to 1 from any value.



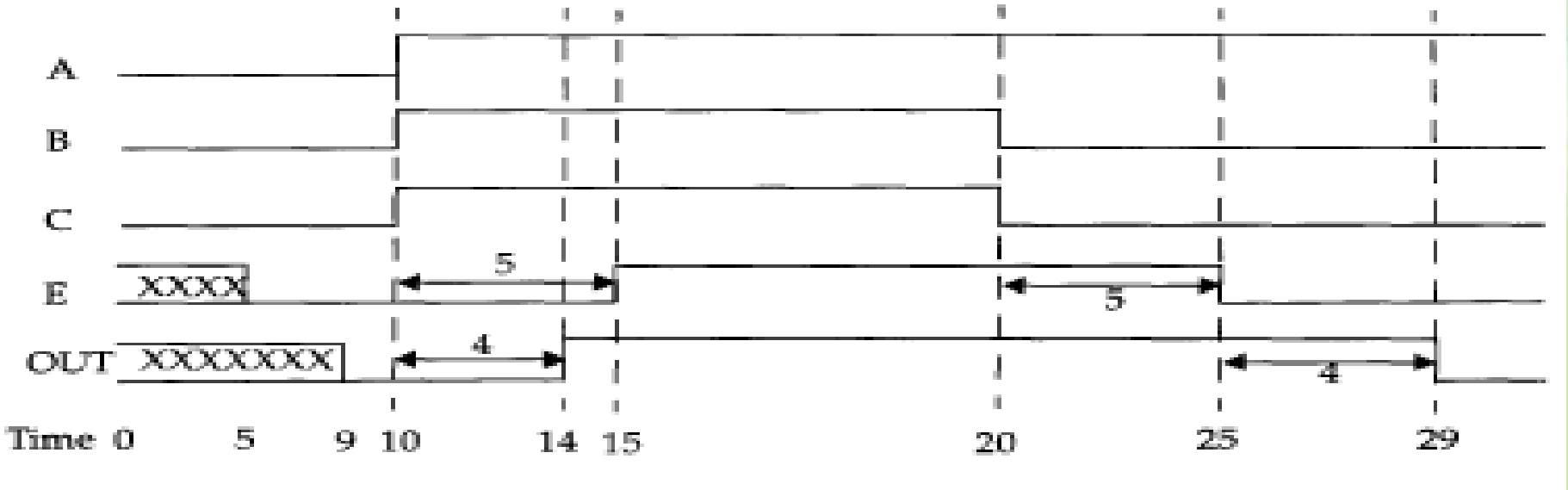


Fall Delay: Delay associated with o/p transition to 0 from any value.

Turn off Delay: Delay associate with o/p transition to Z from another value.




```
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;
```



Dataflow Modeling

- In complex designs the number of gates is very large
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called *logic synthesis*

Continuous Assignment

```
//Syntax of assign statement in the simplest form
<continuous_assign>
        ::= assign <drive_strength>?<delay>? <list_of_assignments>;
```

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

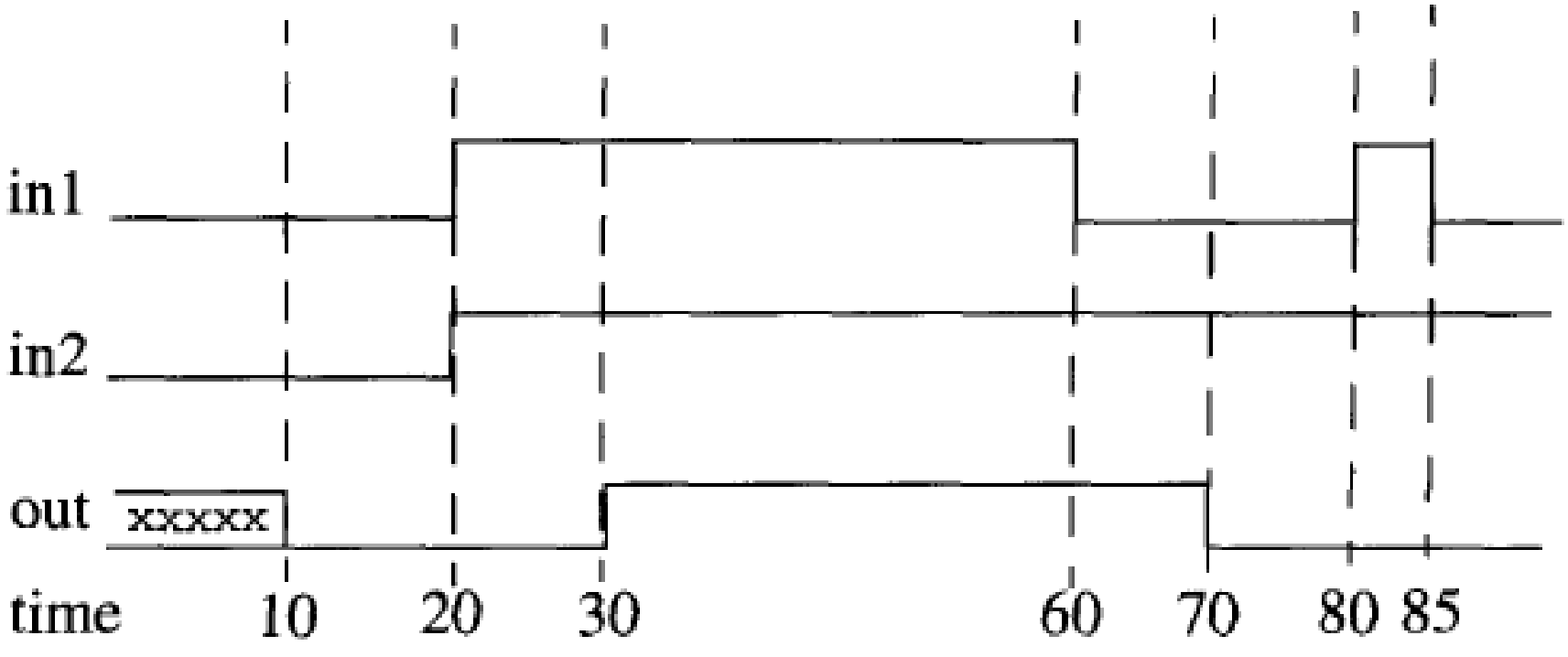
Rules:

- The left hand side of an assignment must always be a scalar or vector net
- It cannot be a scalar or vector register.
- 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.

- The operands on the right-hand side can be registers or nets.

- 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

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



Operator Types

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	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

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Shift	>>	Right shift	two
	<<	Left shift	two
Concatenation	{ }	Concatenation	any number
Replication	{ { } }	Replication	any number
Conditional	?:	Conditional	three
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Conditional Operator

Usage: condition_expr ? true_expr : false_expr ;

```
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

4:1 Multiplexer Example

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =   (~s1 & ~s0 & i0)|
               (~s1 & s0 & i1) |
               (s1 & ~s0 & i2) |
               (s1 & s0 & i3) ;

endmodule
```

Queries …?