Verilog HDL

Introduction

- Verilog HDL is a general purpose HDL that is easy to learn and use.
- Syntax is similar to the C programming language.
- It allows different levels of abstraction to be mixed in the same model.(switch, gates, RTL or behavioral code)
- Most popular logic synthesis tools support verilog HDL.
- This is a case sensitive language.
- It has built in gate level and switch level primitives.

VHDL & VERILOG

	VHDL	Verilog HDL
Origin (start)	US Department of Defense (1987)	Cadence Design Systems (1985)
Domain	Public (IEEE Standard 1076)	Public (IEEE Standard 1364)
Levels of modeling (mixing allowed?)	system, functional, structural, RTL, gate (yes)	functional, structural, RTL, gate, switching (yes)
Looks like	Ada	c
Level of complexity	difficult	average
Easy to read	no	average
Predefined features	poor	good
Designer friendly	no	yes
Designers Acceptance	widely used	widely used
Tool independent	yes	more or less
Straightforward hardware meaning	not really	partially
Clock model	asynchronous, multi- phase, multiple clocks	asynchronous, multi- phase, multiple clocks
Timing properties	sequential and concurrent	sequential and concurrent
Finite state machine	implicit, verbose	implicit, verbose
Semantics for simulation (speed)	yes (average)	yes (fast)
Semantics for synthesis	no	no

Module – Basic Building Block

Module <module-name> (<module_terminal_list);

…..

<module internals>

…..

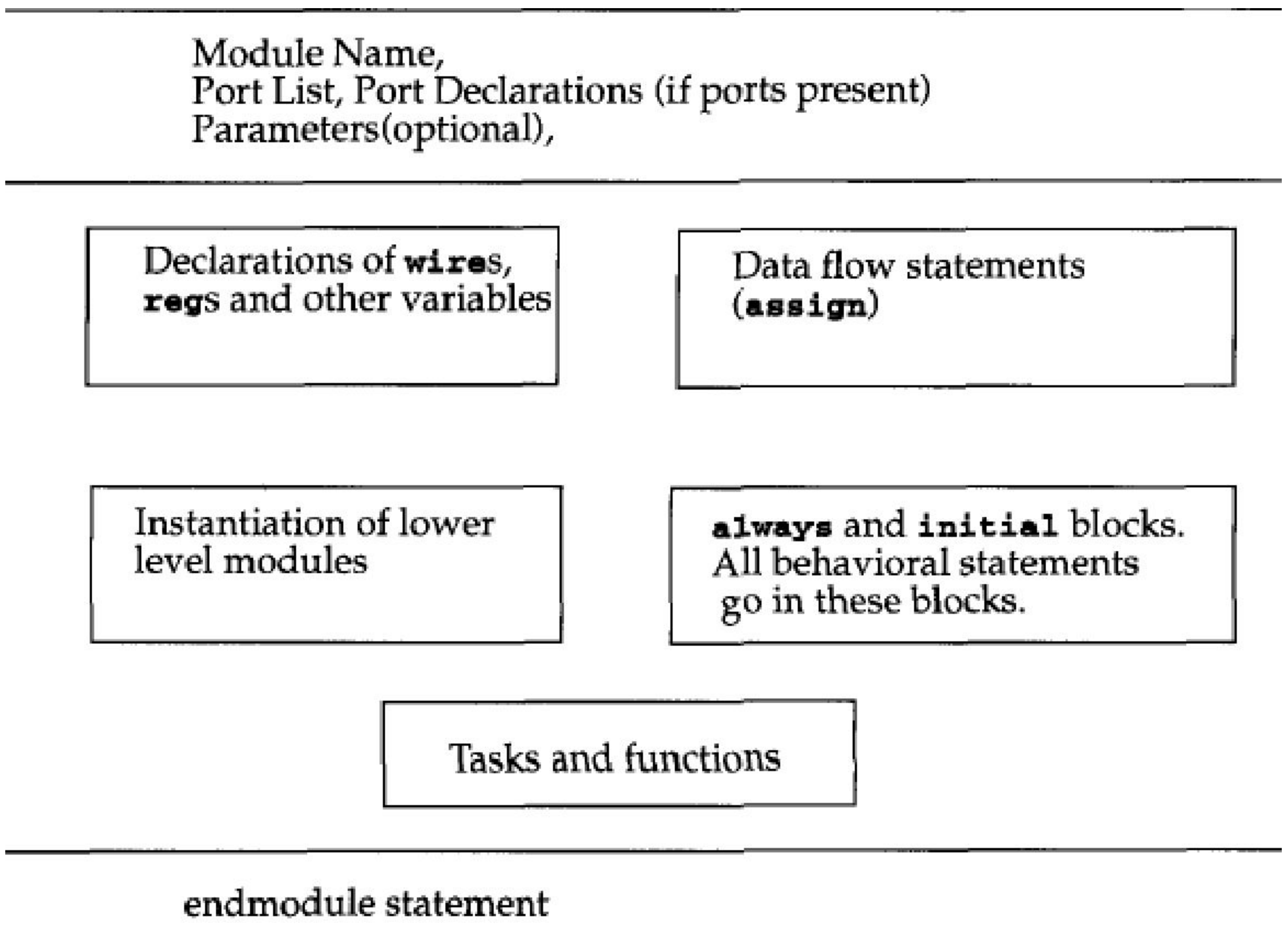
…..

endmodule

--module-name□identifier

--module_terminal_list□input & output terminals

Module



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

Declarations of **wires**, **regs** and other variables

Data flow statements (**assign**)

Instantiation of lower level modules

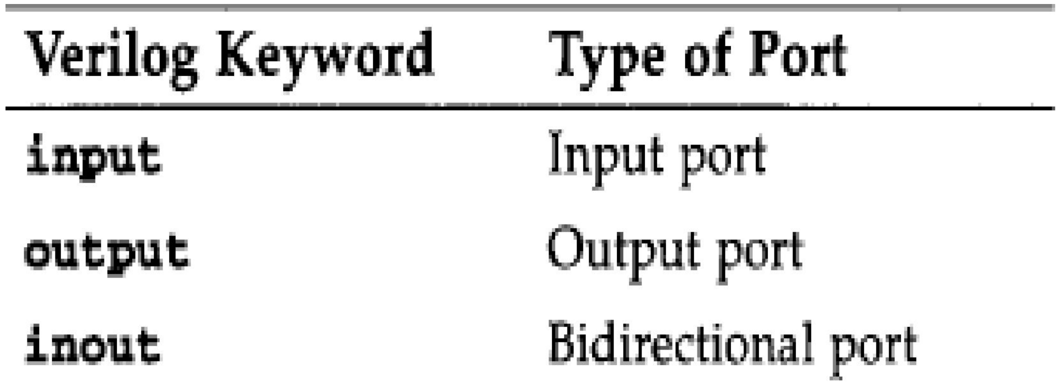
always and **initial** blocks. All behavioral statements go in these blocks.

Tasks and functions

endmodule statement

Ports

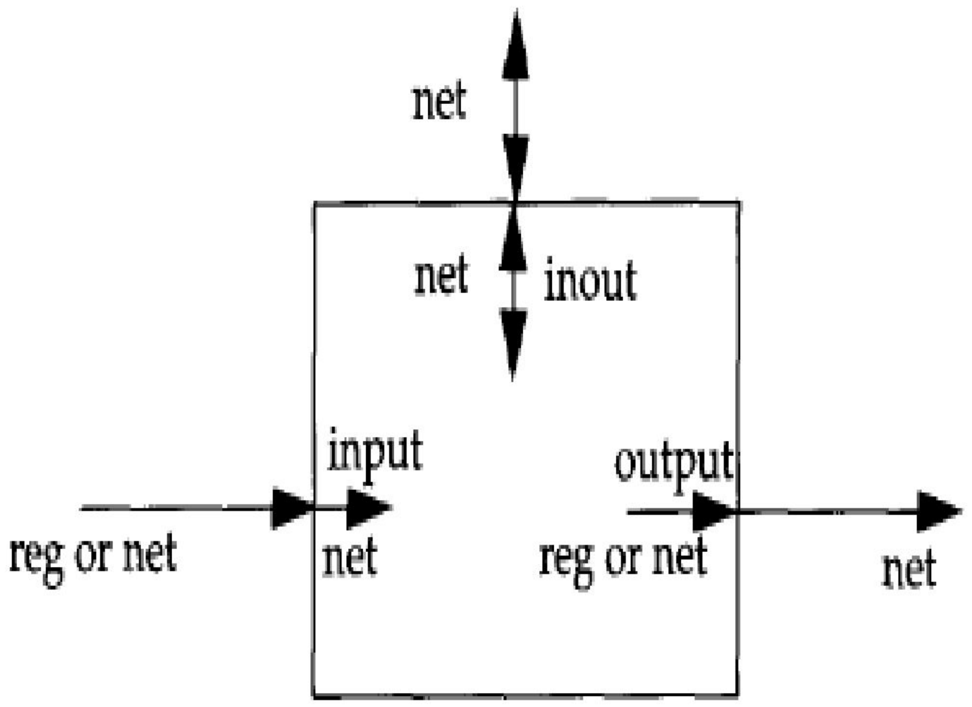
- Ports provide the interface by which a module can communicate with its environment.
- List of Ports



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

- All port declarations are implicitly declared as wire.
- Input or inout ports are normally declared as wires.
- However if output ports hold their value, they must be declared as reg.
- Input and inout cannot be declared as reg.

Port Connection Rules



Basic Concepts

- Lexical Conventions – stream of tokens
- Tokens can be comments, delimiters, strings, numbers, identifiers and keywords.
- Whitespace

  \b – blank space

  \t – tabs

  \n – newlines
- Comments

  // - single line

  /*…*/ - Multiple line
- Operators

  Unary operators - ~

  Binary operators - &&

  Ternary operators - ?

Basic Concepts

- Number Specification

  sized numbers:

represented as <size>’<base format><number>

<size> - is written only in decimal and specifies the no of bits in the number.

‘d or ‘D – Decimal

‘h or ‘H – Hexadecimal

‘b or ‘B – Binary

‘o or ‘O – Octal

Egs. 4’b1111 //4-bit binary number

12’habc //12-bit hexadecimal number

16’d255 //16-bit decimal number

Basic Concepts

- Unsized numbers

  --Numbers that are specified without a base
  format are decimal numbers by default

  -- Default bit size – 32 bit
- X or Z Values

  -- Unknown value – x

  -- High Impedance – z
- Negative numbers

  -6’d3 // 8-bit negative no stores as 2’s complement of 3
- Underscore & Question Marks

  “_” is allowed anywhere in a no except the first character.

  eg: 12’b1111_0000_1010 //for readability

  “?” is alternative for z in the context of numbers.

  eg: 4’b10?? // equivalent of a 4’b10zz

Basic Concepts

- Strings

  “Hello World” //is a string
- Identifiers and Keywords

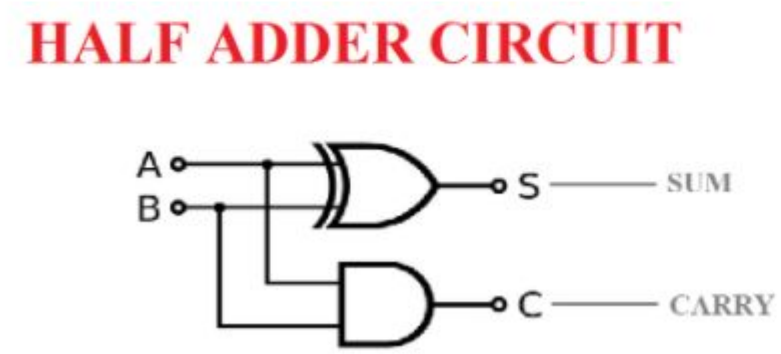
  -- Identifiers are made up of alphanumeric characters, an underscore, $ sign.

  -- Identifiers are case sensitive

  -- Must start with alphabet or an underscore

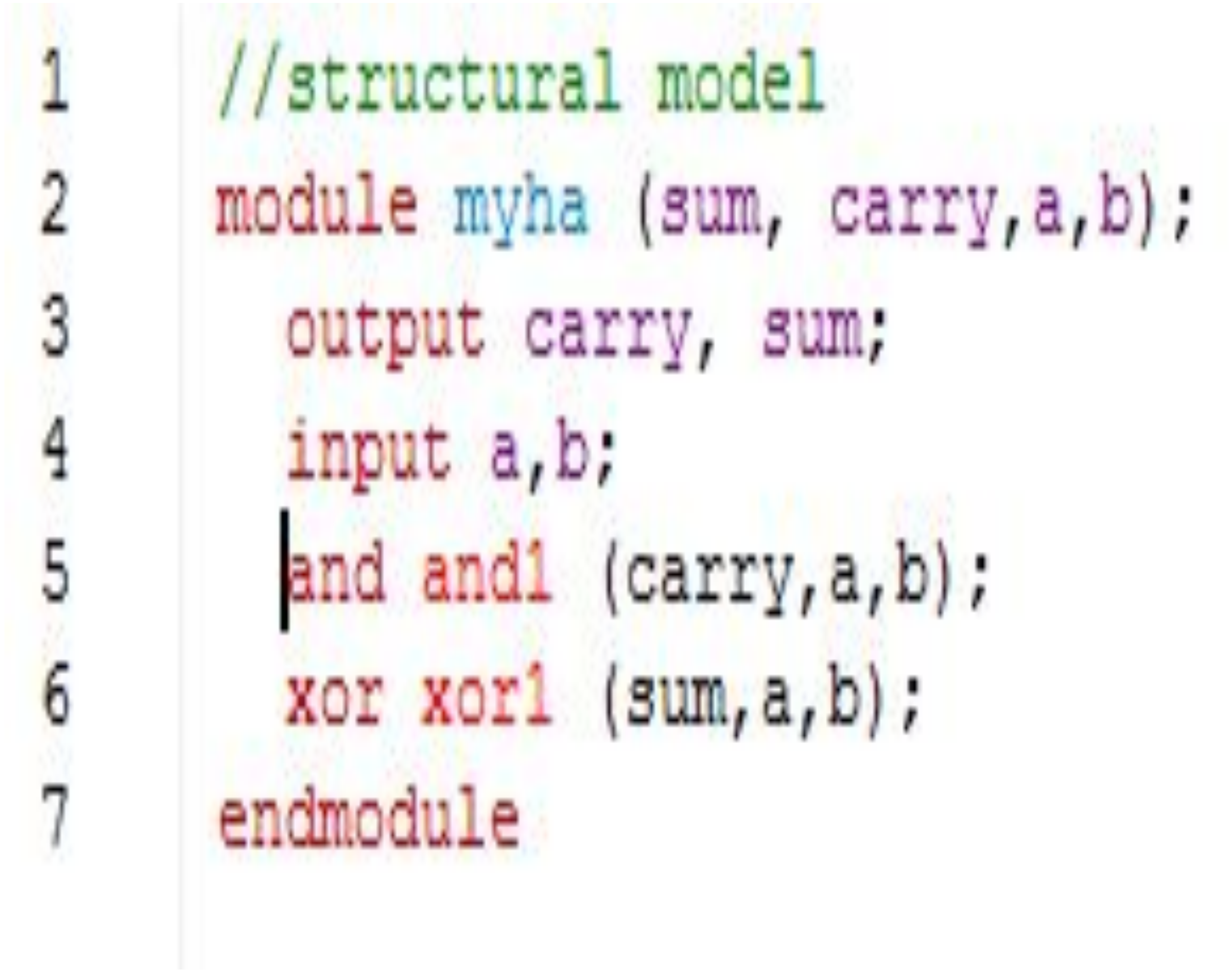
  -- Must not start with a digit or $ sign

HALF ADDER using Dataflow Modeling



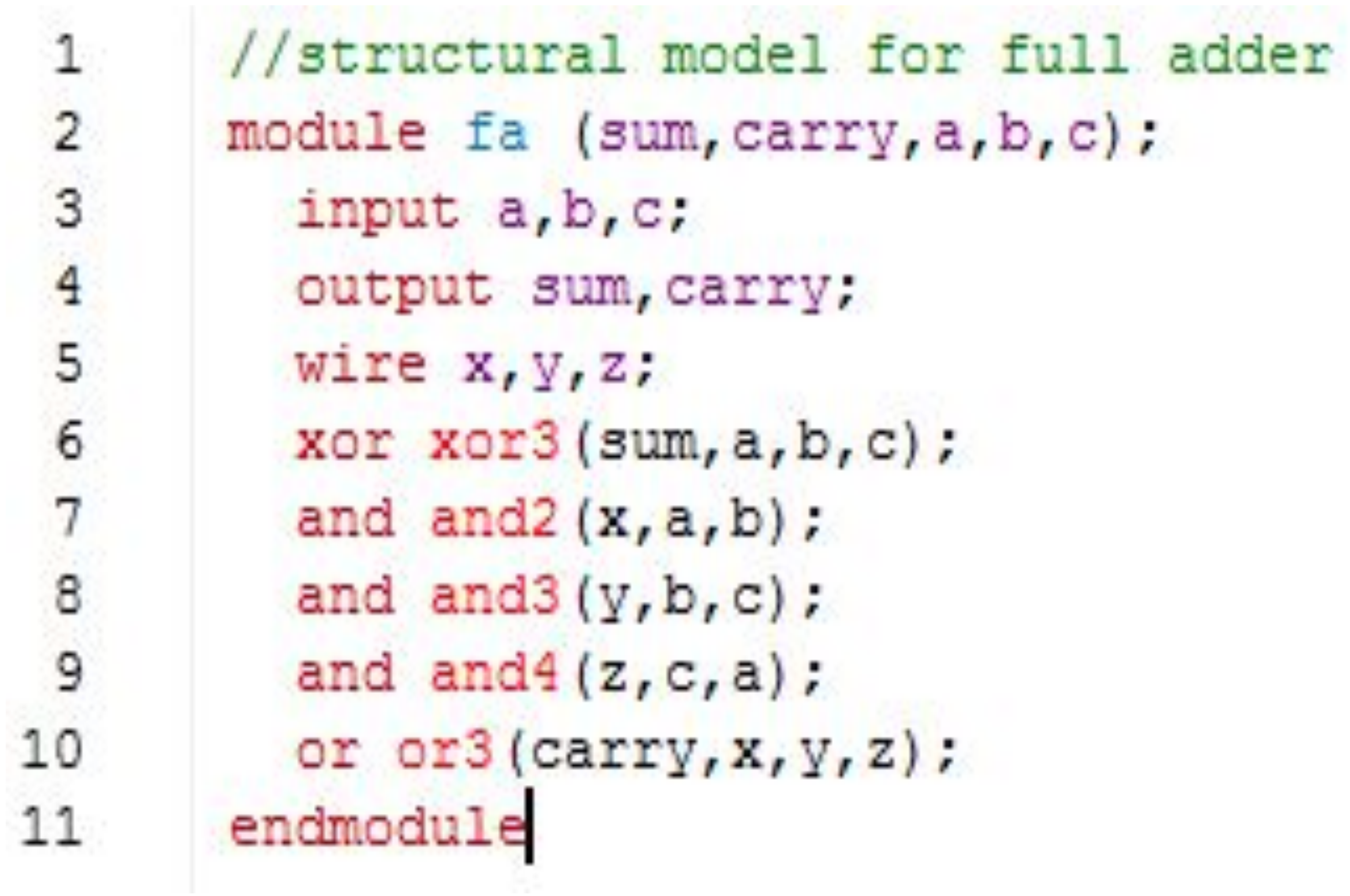
```
module ha_df(sum,carry,a,b);
        input a,b;
        output sum,carry;
        assign sum=a^b;
         assign carry=a&b;
endmodule
```

Half Adder – Structural Modeling



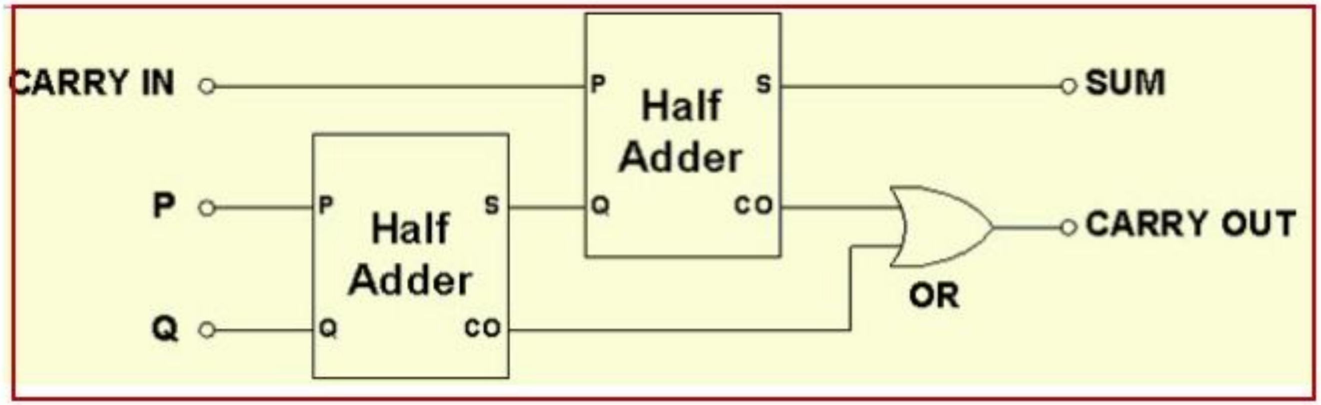
```
//structural model
module myha (sum, carry,a,b);
  output carry, sum;
  input a,b;
  and and1 (carry,a,b);
  xor xor1 (sum,a,b);
endmodule
```

Full Adder – Structural Modeling



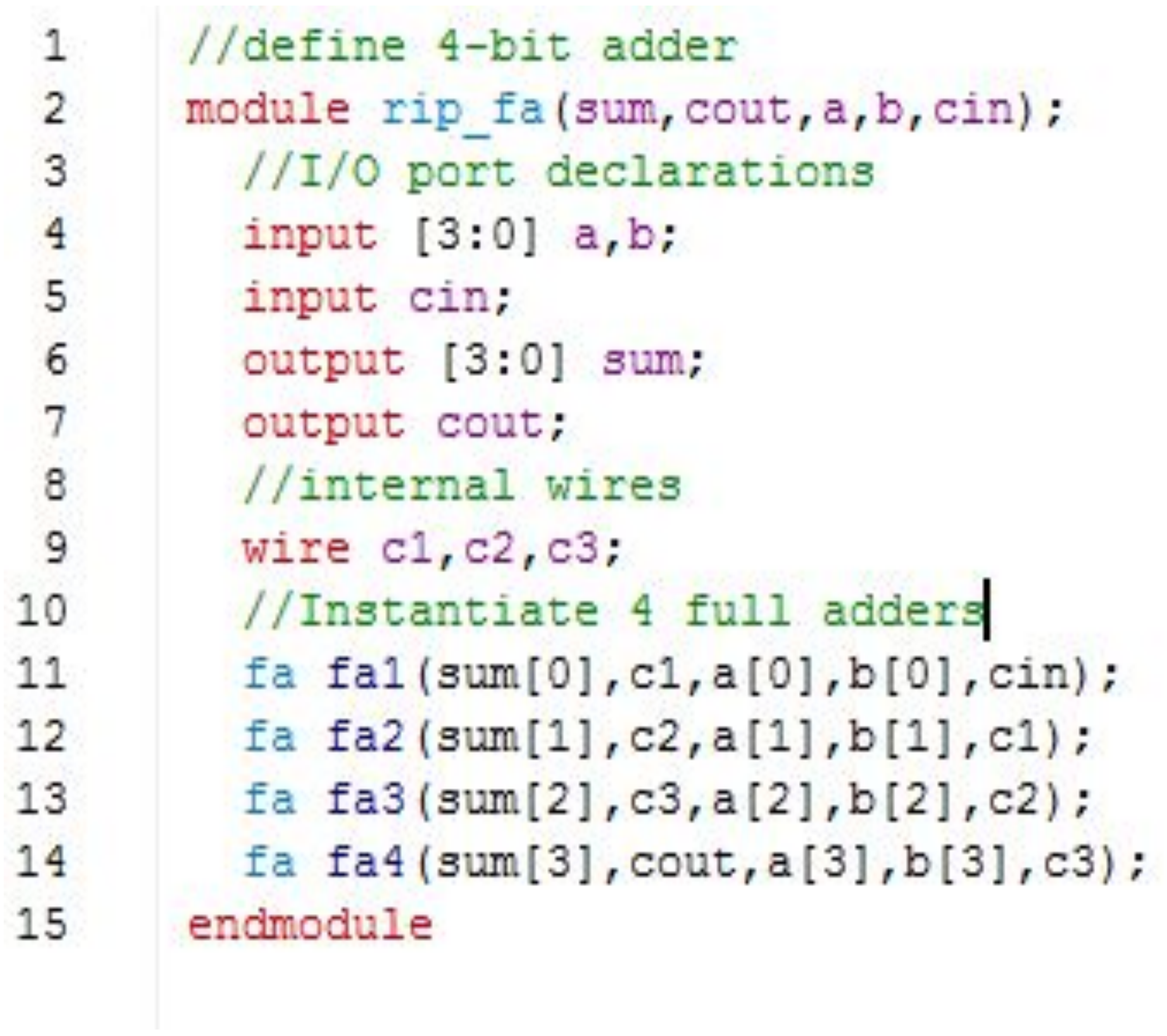
```
//structural model for full adder
module fa (sum,carry,a,b,c);
  input a,b,c;
  output sum,carry;
  wire x,y,z;
  xor xor3(sum,a,b,c);
  and and2(x,a,b);
  and and3(y,b,c);
  and and4(z,c,a);
  or or3(carry,x,y,z);
endmodule
```

Full Adder using Half Adder



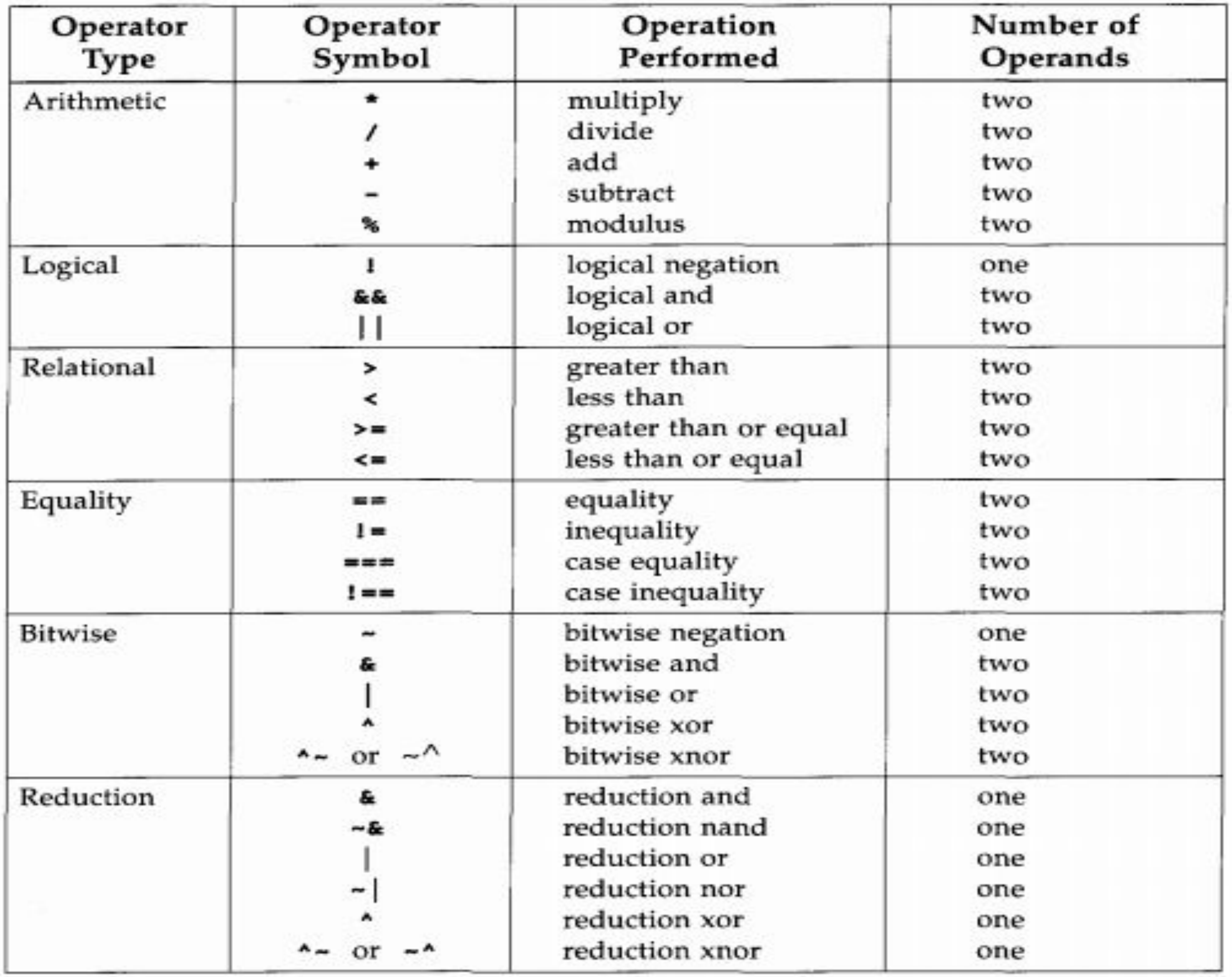
```
module fa_ha(sum,carryout,p,q,carryin);
        input p,q,carryin;
        output sum,carryout;
        wire s,co1,co2;
        ha_df ha1(s,co1,p,q);
        ha_df ha2(sum,co2,s,carryin);
        or o1(carryout,co1,co2);
endmodule
```

4-bit Ripple Carry Adder using FA



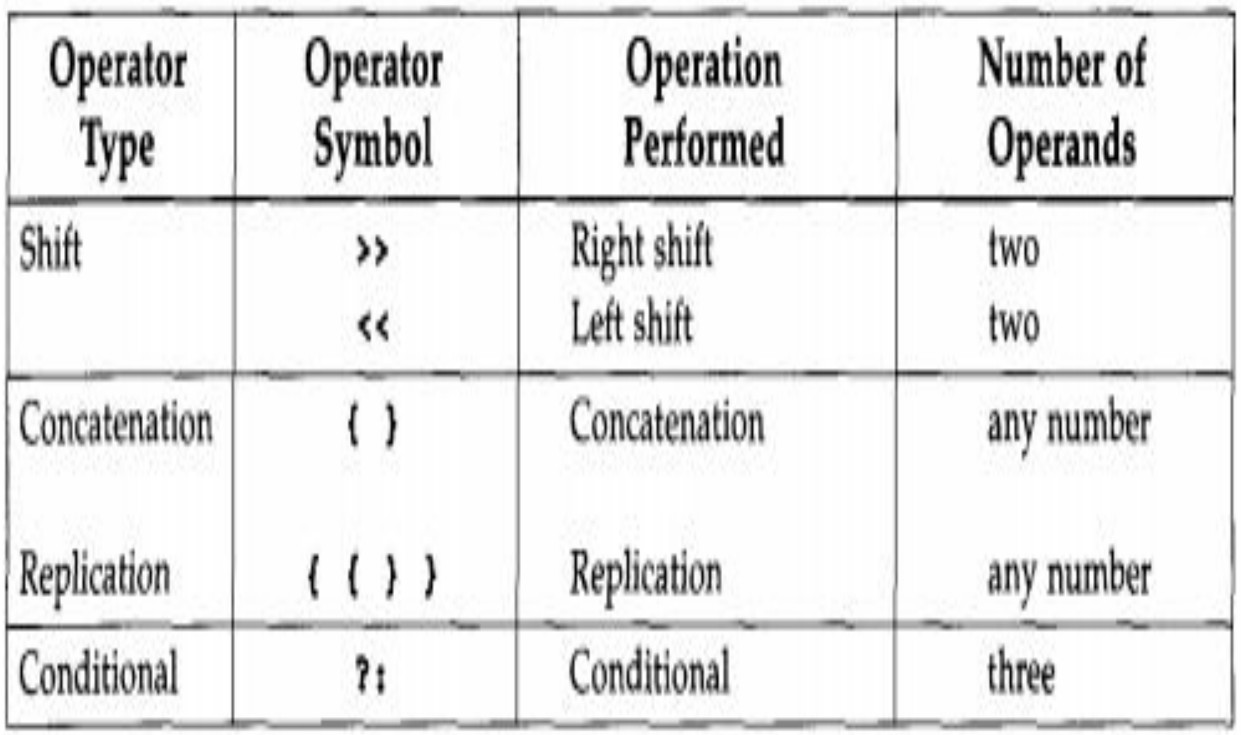
```
//define 4-bit adder
module rip_fa(sum,cout,a,b,cin);
  //I/O port declarations
  input [3:0] a,b;
  input cin;
  output [3:0] sum;
  output cout;
  //internal wires
  wire c1,c2,c3;
  //Instantiate 4 full adders
  fa fa1(sum[0],c1,a[0],b[0],cin);
  fa fa2(sum[1],c2,a[1],b[1],c1);
  fa fa3(sum[2],c3,a[2],b[2],c2);
  fa fa4(sum[3],cout,a[3],b[3],c3);
endmodule
```

Operator Types & Symbols



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
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Operator Types & Symbols



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

Arithmetic Operators

- Binary Operators

  - multiply(*),divide(/),add(+),subtract(-), power(**)and modulus(%)

- Unary Operators

  - + and – can also work as unary operators.

  eg -4 –negative 4

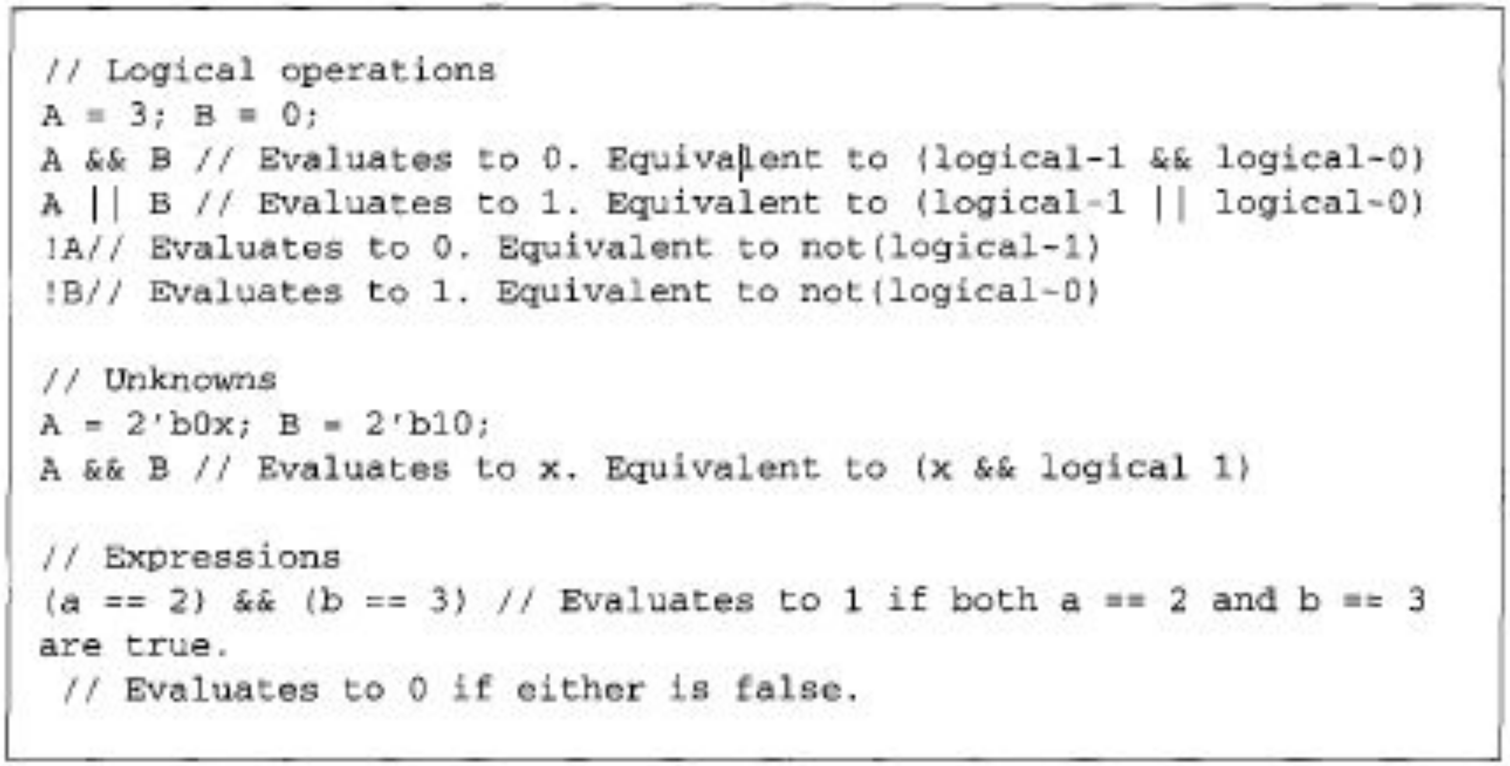
  +5 –positive 5

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; // D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E.Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
```

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

```
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
```

Logical Operators



```
// Logical operations
A = 3; B = 0;
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)

// Unknowns
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.
```

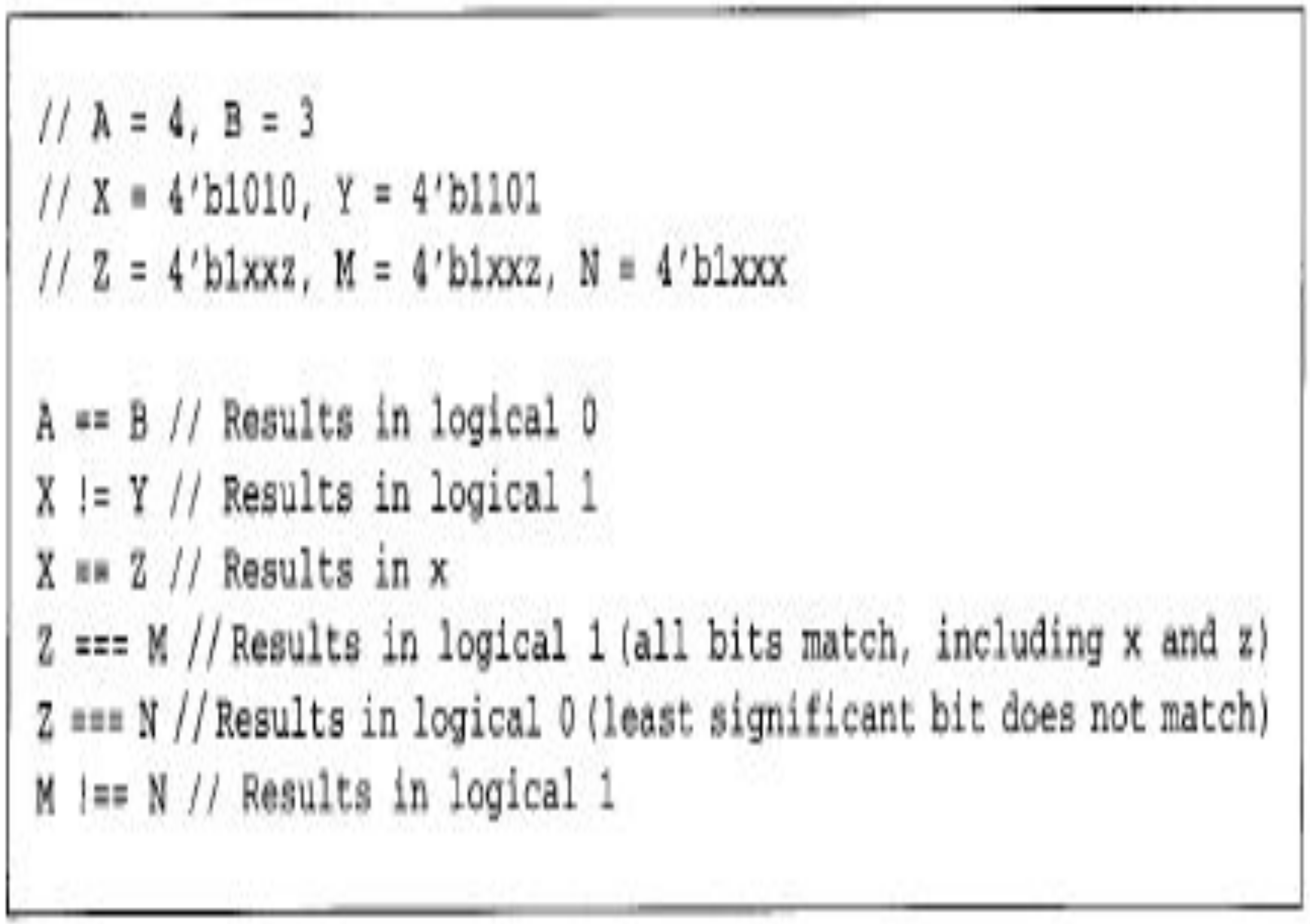
•If any operand is not equal to zero, it is equivalent to a logical 1. If it is equivalent To zero, it is equivalent to logical 0.

•If any operand bit is x or z, it is equivalent to x and is treated as false condition.

Relational Operators

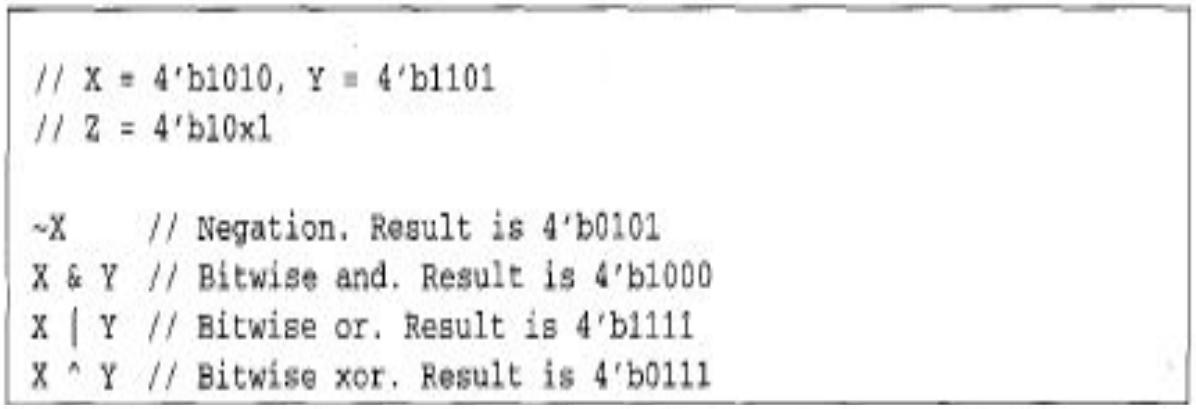
```
// A=4, B=3
//X=4'b1010   Y=4'b1101     Z=4'b1xxx
A<=B // logical 0
A>B  // logical 1
Y>=X // logical 1
Y<Z  // X
```

Equality Operators

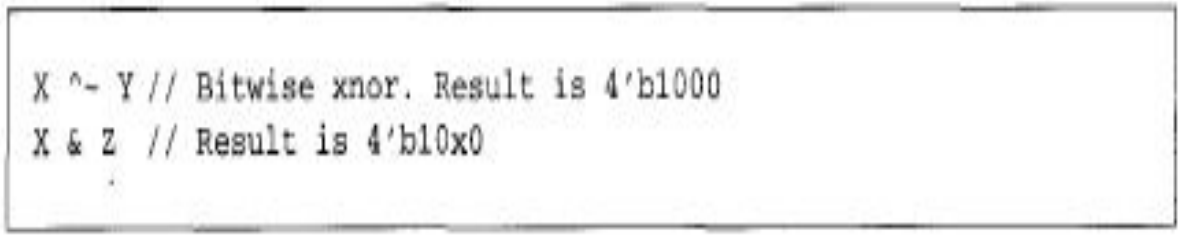


```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

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
```

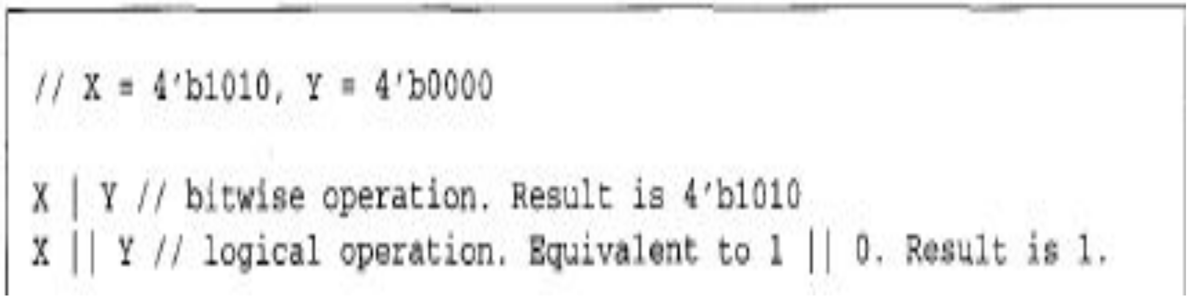
Bitwise Operators



```
// 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
```

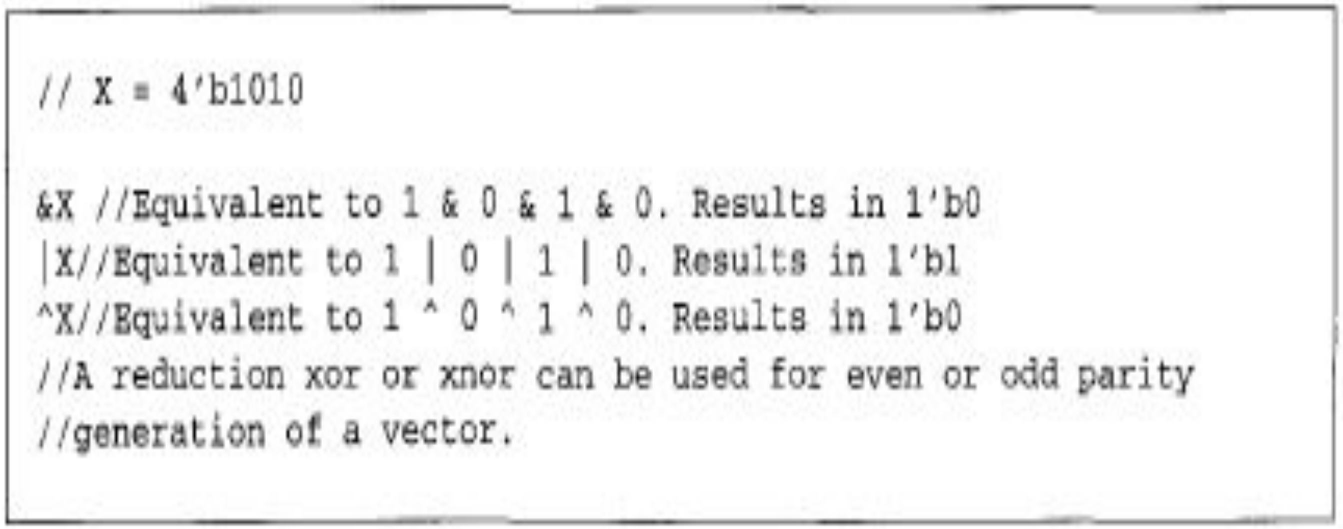


```
X ^~ Y // Bitwise xnor. Result is 4'b1000
X & Z  // Result is 4'b10x0
```

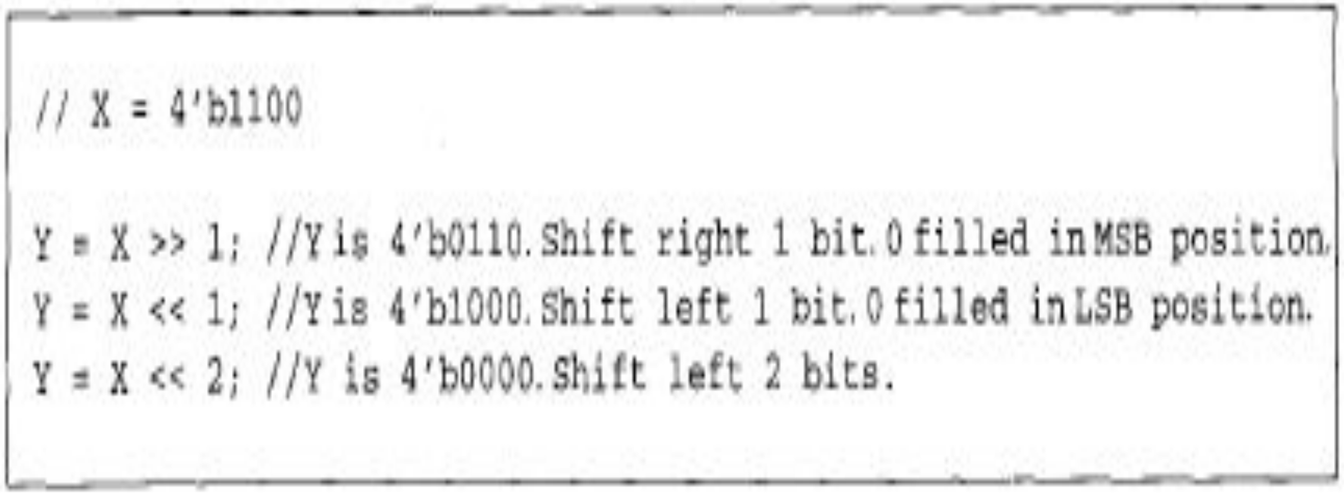


```
// X = 4'b1010, Y = 4'b0000

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

Reduction & Shift Operators

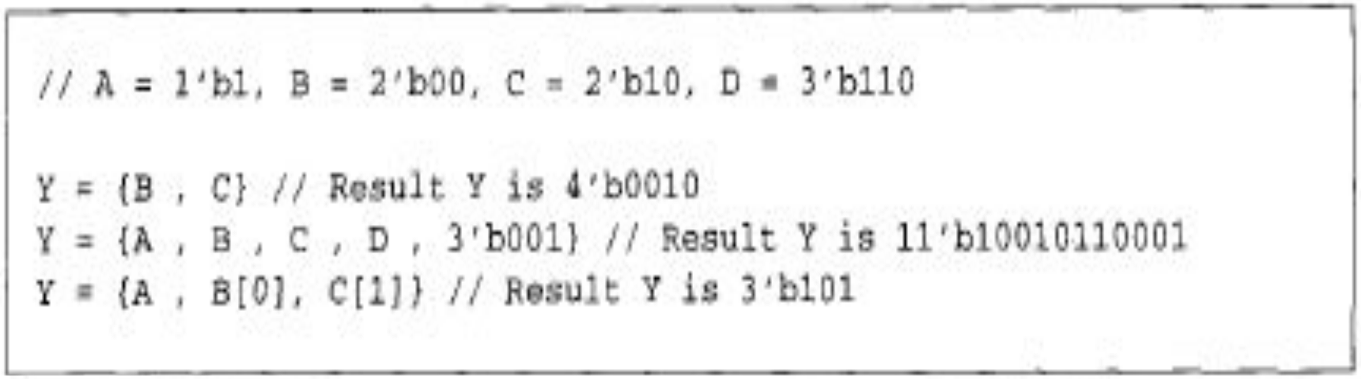


```
// X = 4'b1010

&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.
```

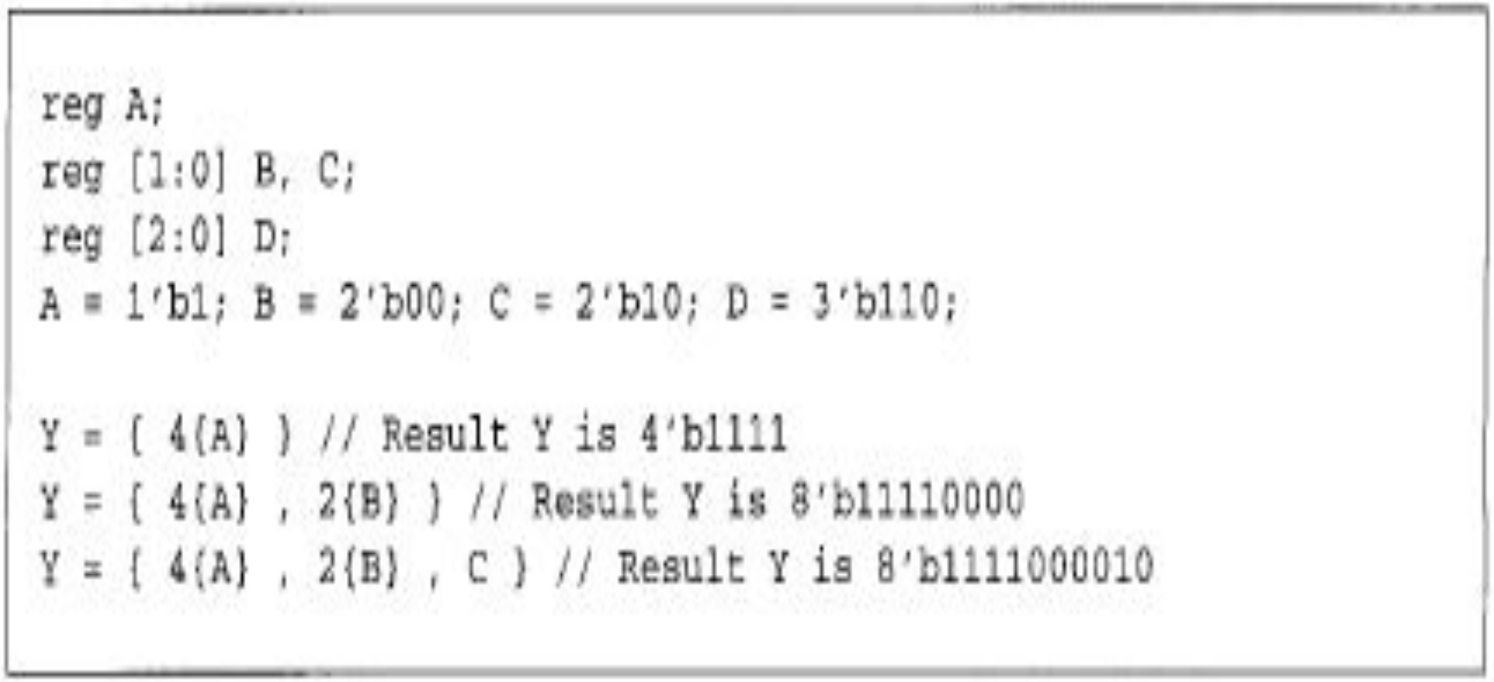


```
// X = 4'b1100

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 & Replication Operators



```
// 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
```



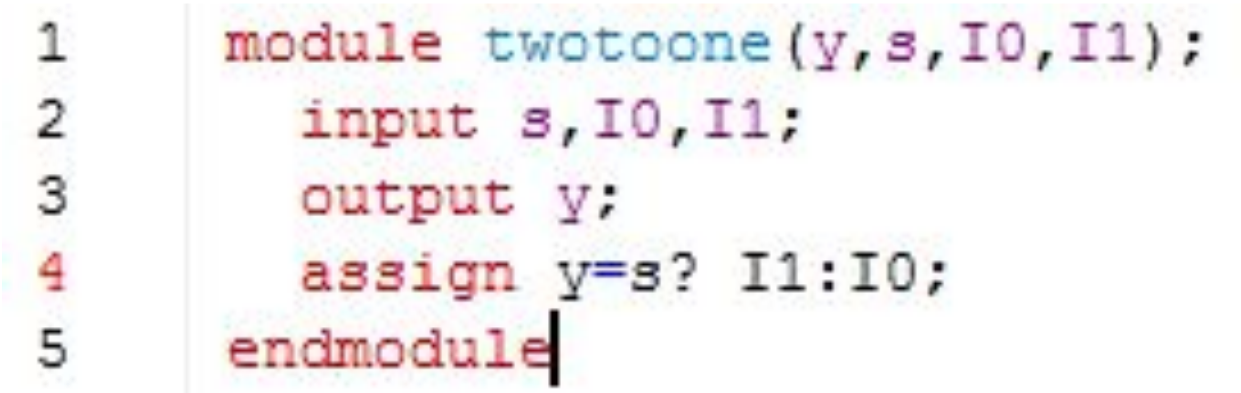
```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

Conditional Operator

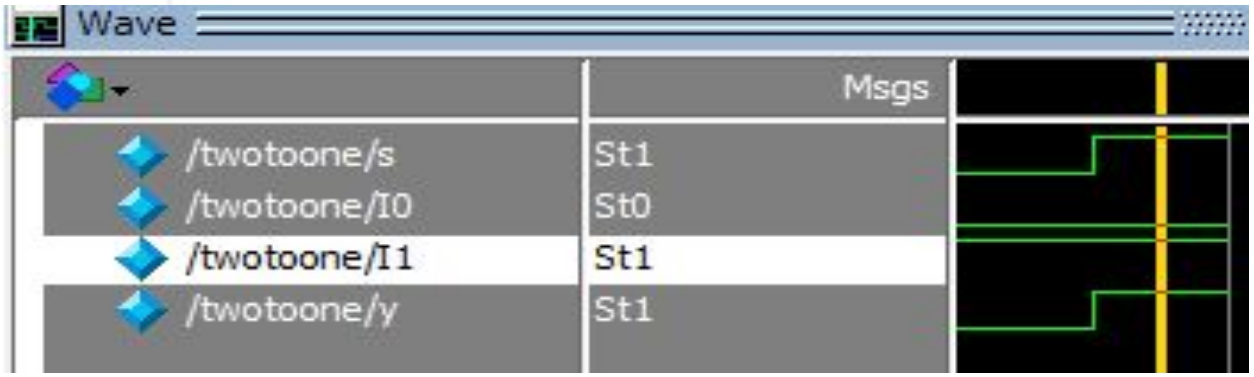
Usage:

Condition_exp ? True_exp : false_exp ;

2:1 Multiplexer Pgm using conditional operator

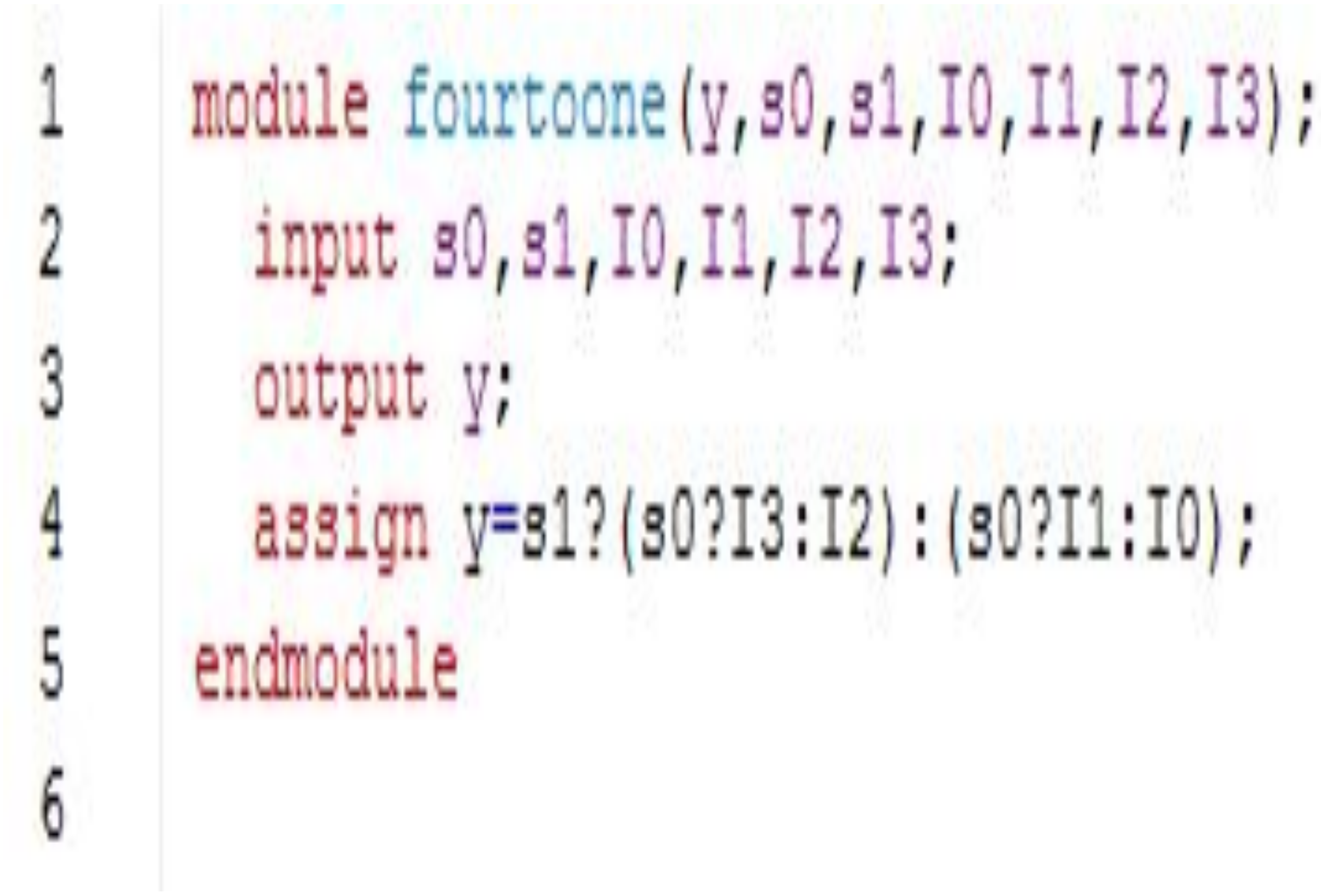


```
module twotoone(y,s,I0,I1);
  input s,I0,I1;
  output y;
  assign y=s? I1:I0;
endmodule
```



```
Wave
                      Msgs
/twotoone/s           St1
/twotoone/I0          St0
/twotoone/I1          St1
/twotoone/y           St1
```

4:1 Mux using Conditional Operator



```
module fourtoone(y,s0,s1,I0,I1,I2,I3);
  input s0,s1,I0,I1,I2,I3;
  output y;
  assign y=s1?(s0?I3:I2):(s0?I1:I0);
endmodule

```

Conditional Statements in Behavioral Modeling – IF statement

- If

1. **if** (<expression>) true-stmt;
2. **if**(<expression>)true-stmt; **else** false-stmt;
3. **if**(<expression>)true-stmt1

   else if (<expression>) true-stmt2;

   else if (<expression>) true-stmt3;

   else default-stmt;

- Multiple statements can be grouped using begin and end.

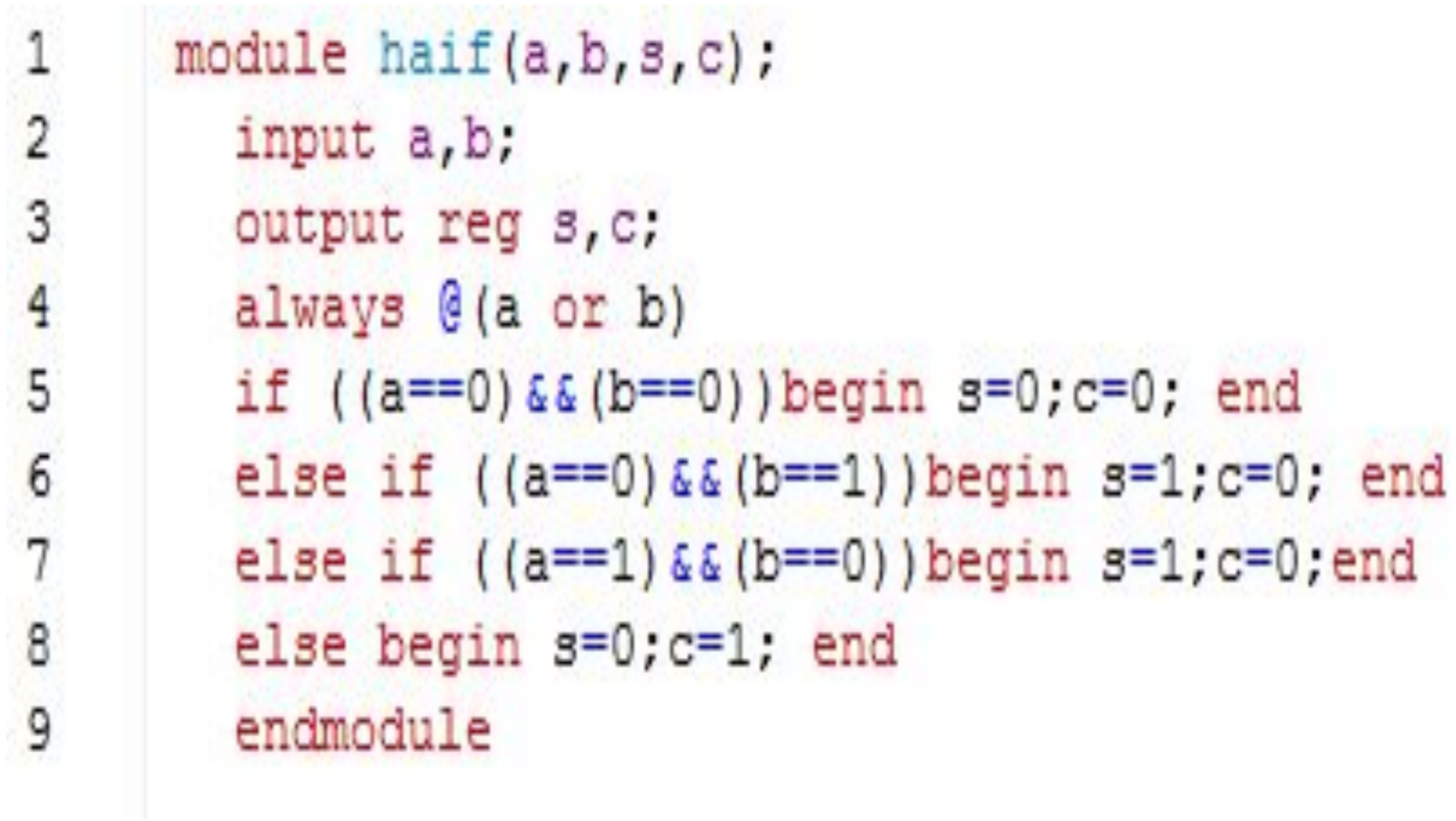
case statement

- Keywords are case, endcase and default.
- Syntax:

```
  case (expression)
  alternative1: statement 1;
  alternative2: statement 2;
  alternative3: statement 3;
  …………….
  default: default-statement;
endcase
```

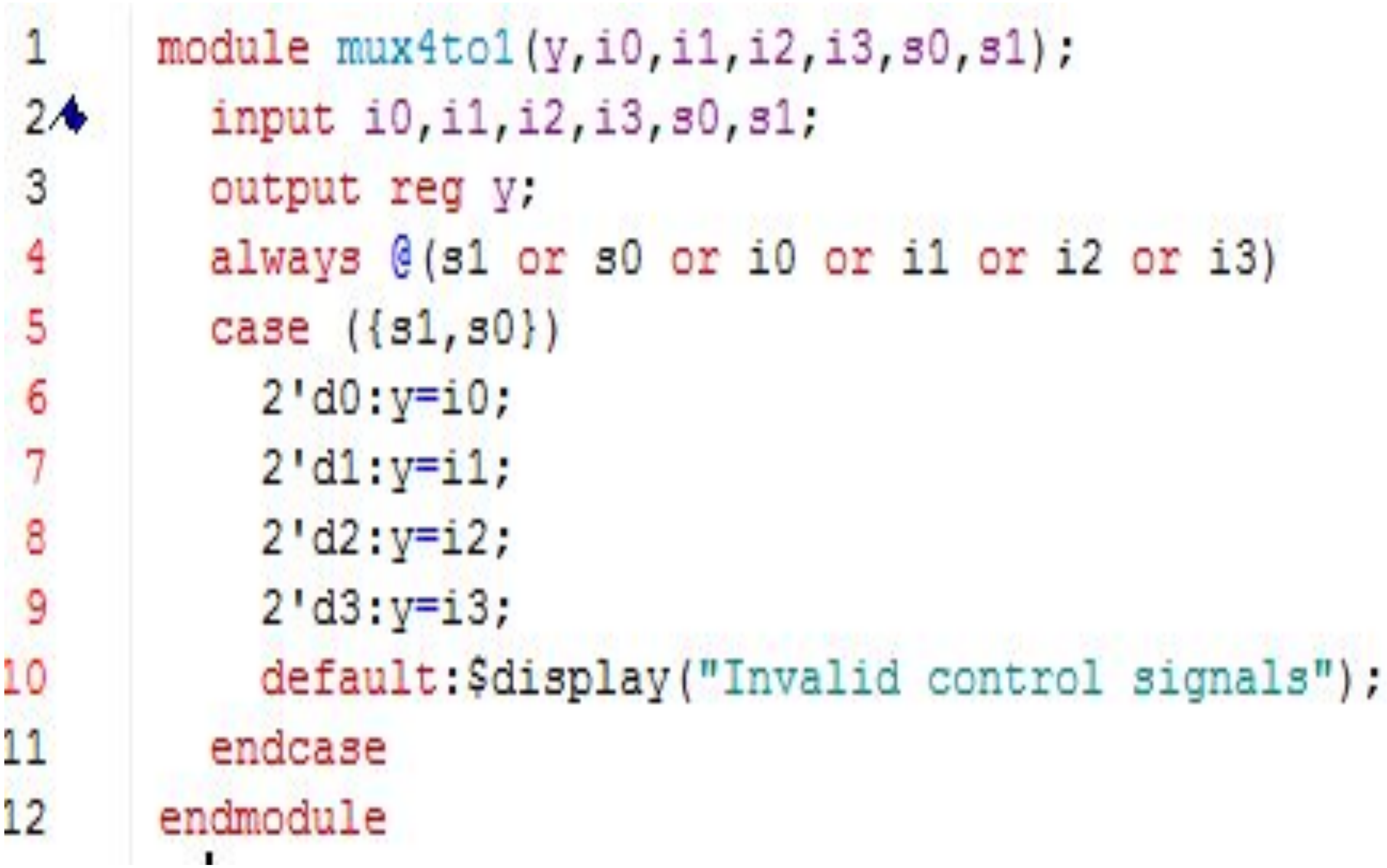
- Default statement is optional
- Case statements can be nested

Half Adder using if statement



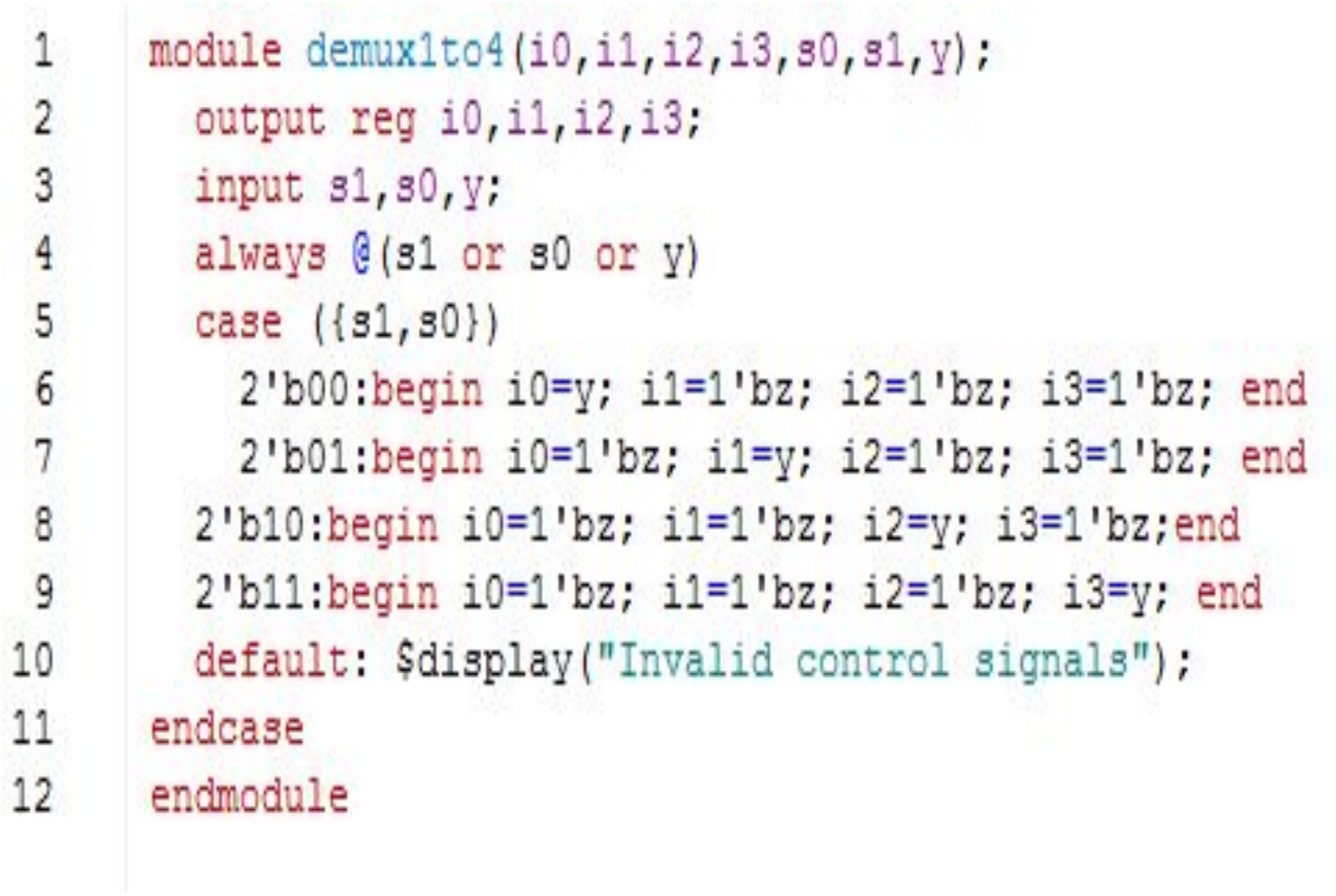
```
module haif(a,b,s,c);
  input a,b;
  output reg s,c;
  always @(a or b)
  if ((a==0)&&(b==0))begin s=0;c=0; end
  else if ((a==0)&&(b==1))begin s=1;c=0; end
  else if ((a==1)&&(b==0))begin s=1;c=0;end
  else begin s=0;c=1; end
  endmodule
```

4:1 MUX using case



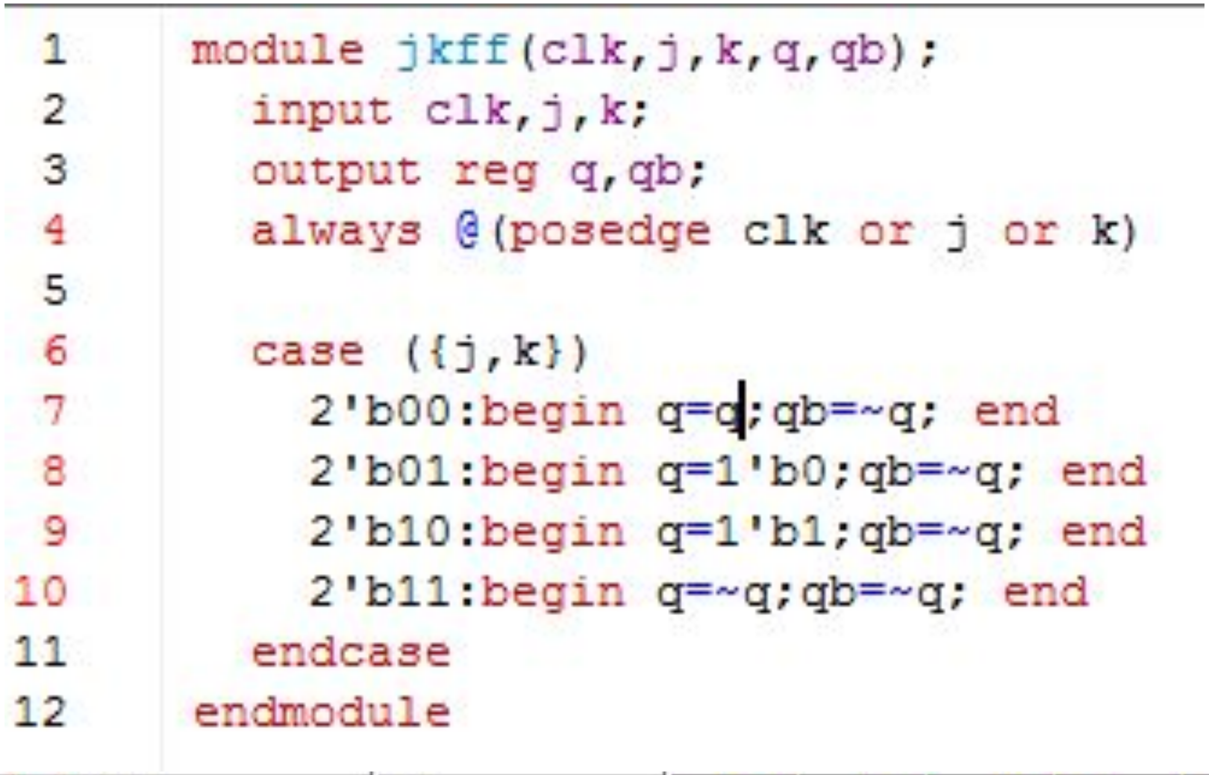
```
module mux4to1(y,i0,i1,i2,i3,s0,s1);
  input i0,i1,i2,i3,s0,s1;
  output reg y;
  always @(s1 or s0 or i0 or i1 or i2 or i3)
  case ({s1,s0})
    2'd0:y=i0;
    2'd1:y=i1;
    2'd2:y=i2;
    2'd3:y=i3;
    default:$display("Invalid control signals");
  endcase
endmodule
```

1:4 Demux using case

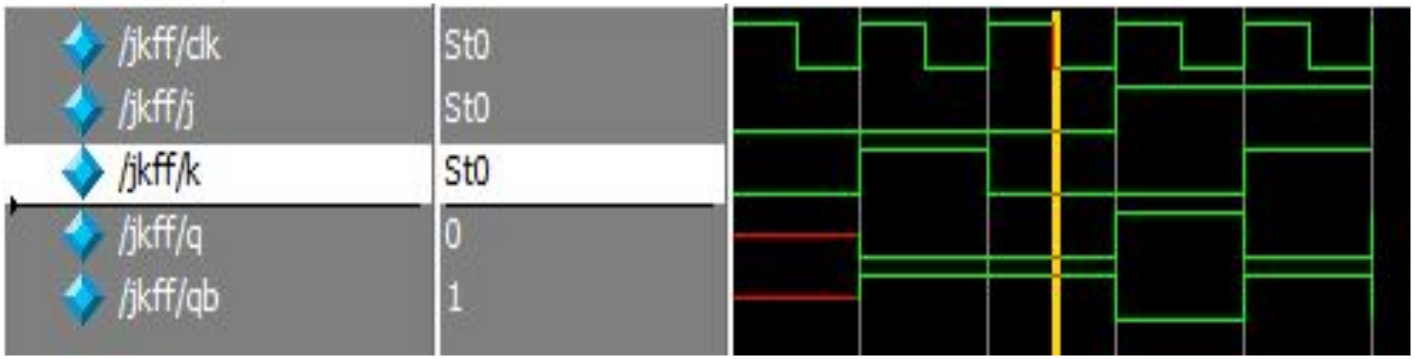


```
module demux1to4(i0,i1,i2,i3,s0,s1,y);
  output reg i0,i1,i2,i3;
  input s1,s0,y;
  always @(s1 or s0 or y)
  case ({s1,s0})
    2'b00:begin i0=y; i1=1'bz; i2=1'bz; i3=1'bz; end
    2'b01:begin i0=1'bz; i1=y; i2=1'bz; i3=1'bz; end
  2'b10:begin i0=1'bz; i1=1'bz; i2=y; i3=1'bz;end
  2'b11:begin i0=1'bz; i1=1'bz; i2=1'bz; i3=y; end
  default: $display("Invalid control signals");
endcase
endmodule
```

JK Flip Flop



```
module jkff(clk,j,k,q,qb);
  input clk,j,k;
  output reg q,qb;
  always @(posedge clk or j or k)

  case ({j,k})
    2'b00:begin q=q;qb=~q; end
    2'b01:begin q=1'b0;qb=~q; end
    2'b10:begin q=1'b1;qb=~q; end
    2'b11:begin q=~q;qb=~q; end
  endcase
endmodule
```



Structured Procedures

- Initial statement

  --initial block starts at time 0, executes exactly once during a simulation

  --if there are multiple blocks, each block starts to execute concurrently at time 0

  --multiple statements can be grouped using begin and end

  --initial blocks are used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run

Structured Procedures

- Always statement

  --starts at time 0 and executes the statements in the always block continuously in a looping fashion

  --used to model a block of activity that is repeated continuously in a digital circuit

Example for initial and always

```
module clk_gen(output reg clock);
//initialize clock at time zero
initial
   clock=1’b0;
//toggle clock every half-cycle(time period=20)
always
   #10 clock = ~clock;
initial
   #1000 $finish;
endmodule
```

Procedural Assignments

- Procedural assignments update values of integer, real or time variables
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.
- Two types of procedural assignments are

  - blocking assignments

  - nonblocking assignments

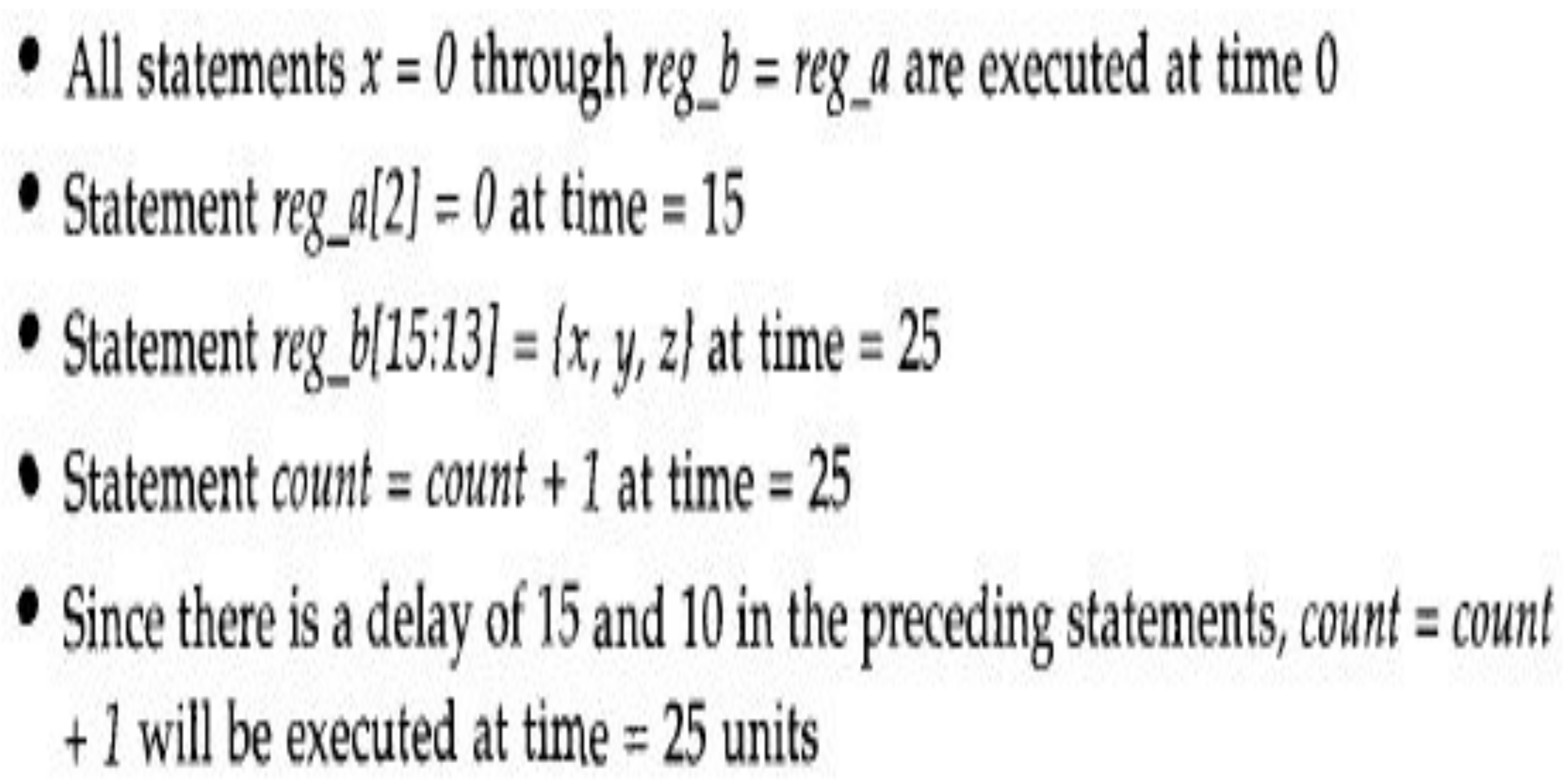
Blocking Assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
```

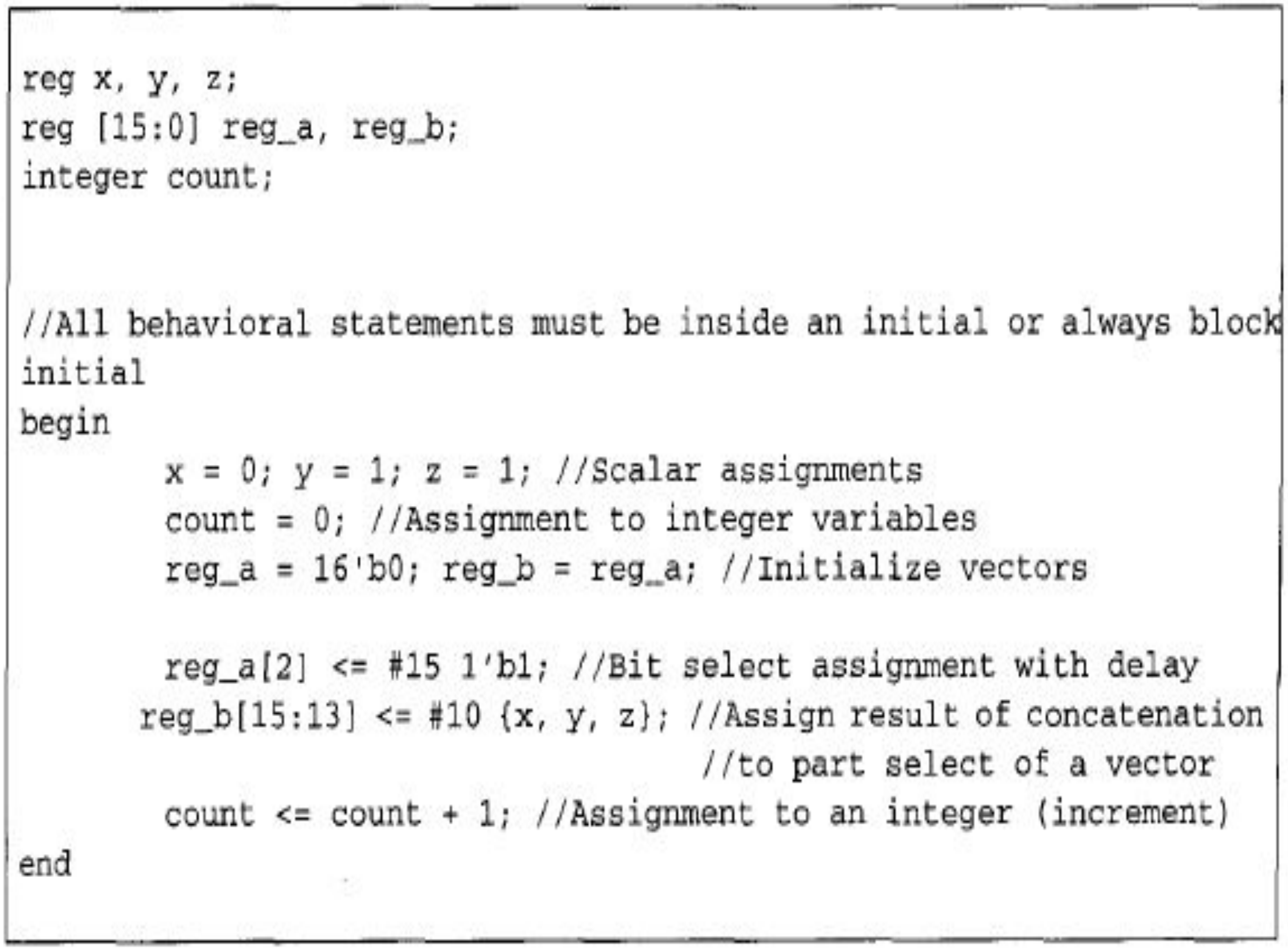
```
//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //initialize vectors

        #15 reg_a[2] = 1'b1; //Bit select assignment with delay
      #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to
                                       // part select of a vector
        count = count + 1; //Assignment to an integer (increment)
end
```

Blocking Assignments



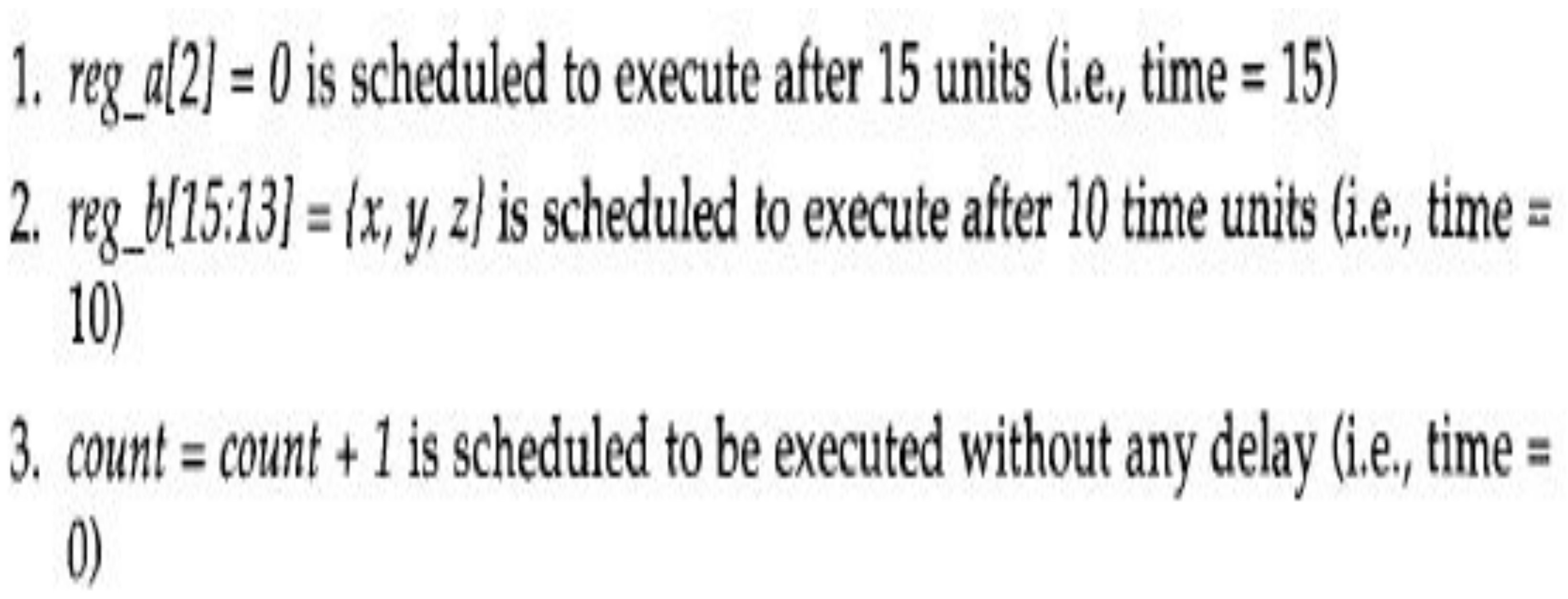
- All statements *x = 0* through *reg_b = reg_a* are executed at time 0
- Statement *reg_a[2] = 0* at time = 15
- Statement *reg_b[15:13] = {x, y, z}* at time = 25
- Statement *count = count + 1* at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, *count = count + 1* will be executed at time = 25 units

Nonblocking Assignments



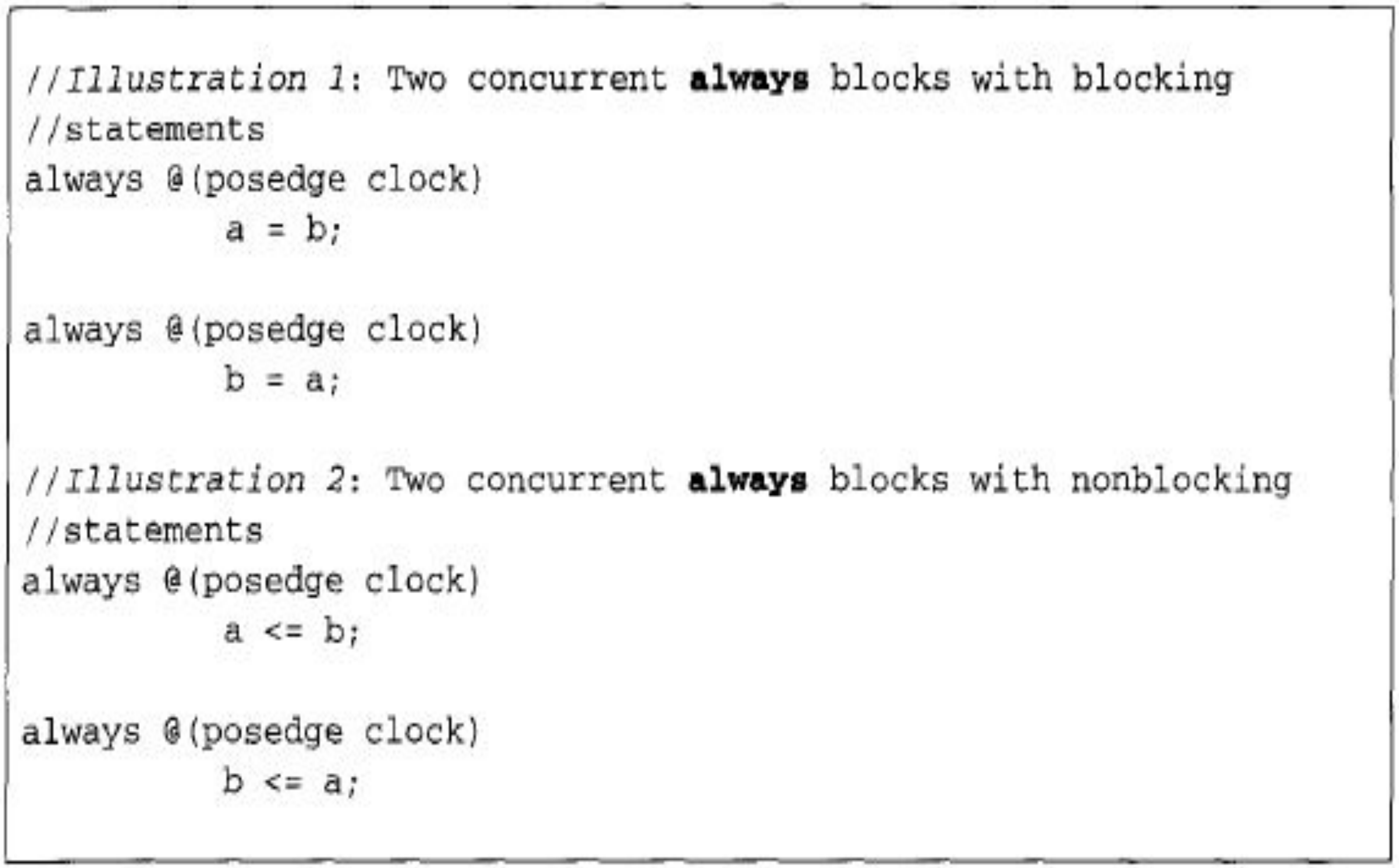
```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;


//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //Initialize vectors

        reg_a[2] <= #15 1'b1; //Bit select assignment with delay
       reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
                                       //to part select of a vector
        count <= count + 1; //Assignment to an integer (increment)
end
```

Nonblocking Assignments



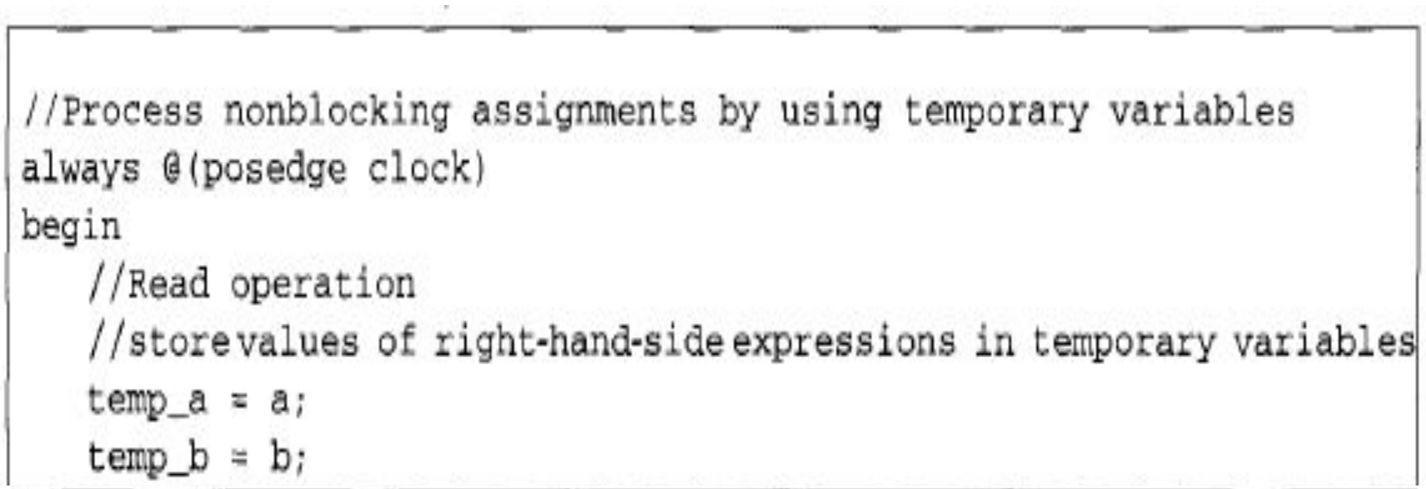
1. *reg_a[2] = 0* is scheduled to execute after 15 units (i.e., time = 15)
2. *reg_b[15:13] = {x, y, z}* is scheduled to execute after 10 time units (i.e., time = 10)
3. *count = count + 1* is scheduled to be executed without any delay (i.e., time = 0)

Nonblocking Assignments to Eliminate Race Condition

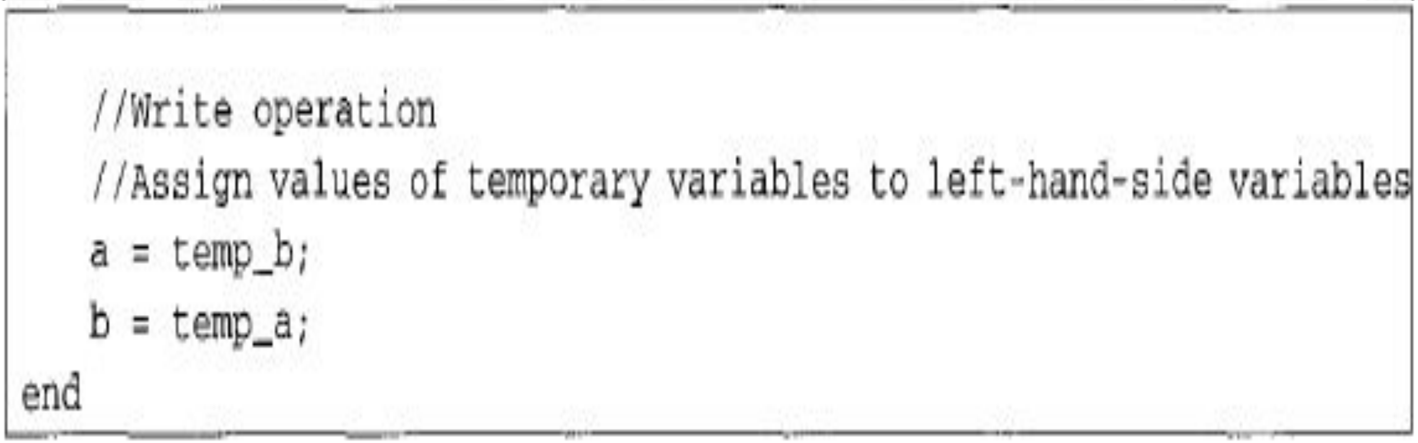


```
//Illustration 1: Two concurrent always blocks with blocking
//statements
always @(posedge clock)
          a = b;

always @(posedge clock)
          b = a;

//Illustration 2: Two concurrent always blocks with nonblocking
//statements
always @(posedge clock)
          a <= b;

always @(posedge clock)
          b <= a;
```

Implementing Nonblocking using blocking assignments

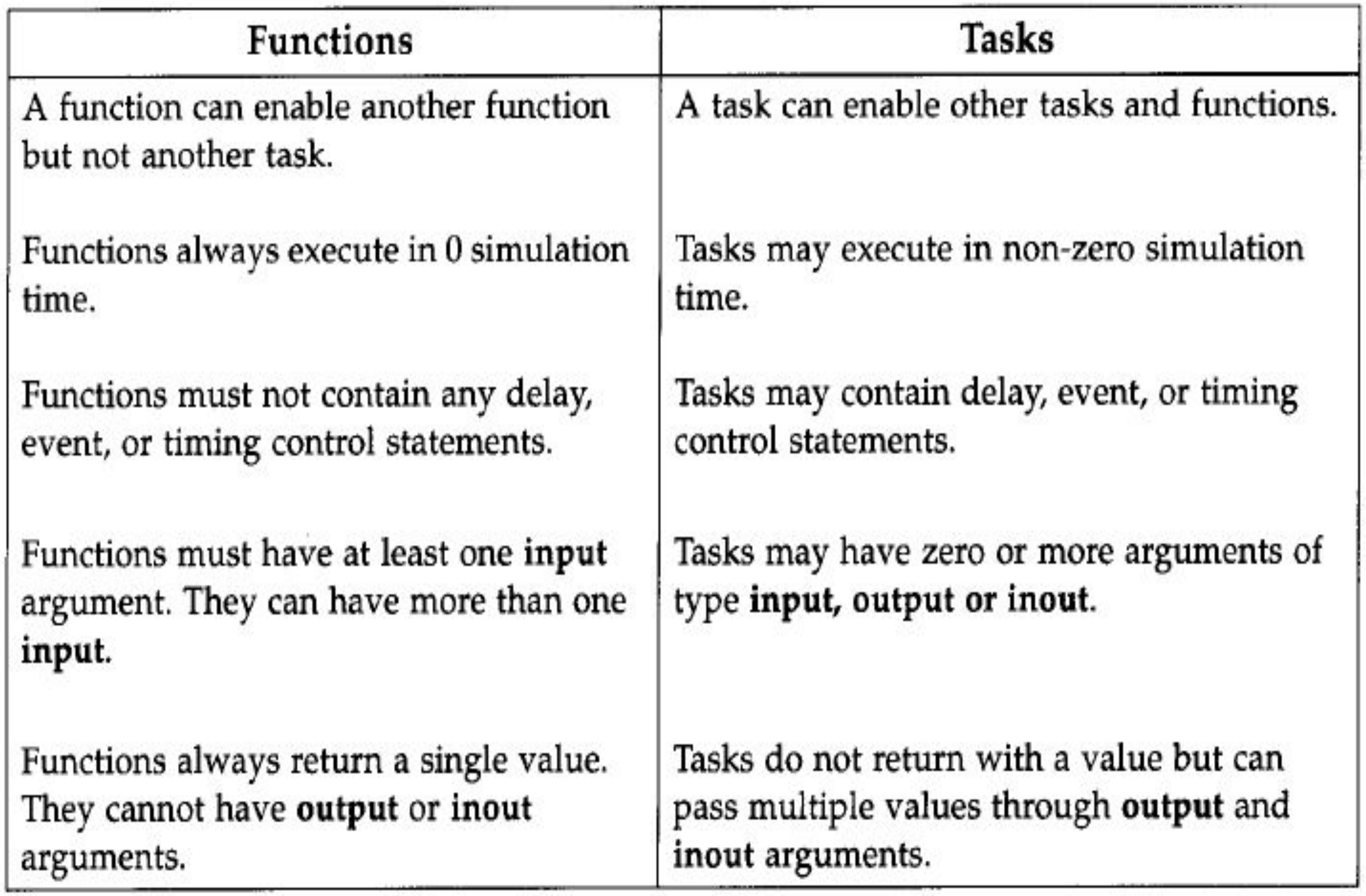


```
//Process nonblocking assignments by using temporary variables
always @(posedge clock)
begin
   //Read operation
   //store values of right-hand-side expressions in temporary variables
   temp_a = a;
   temp_b = b;
```



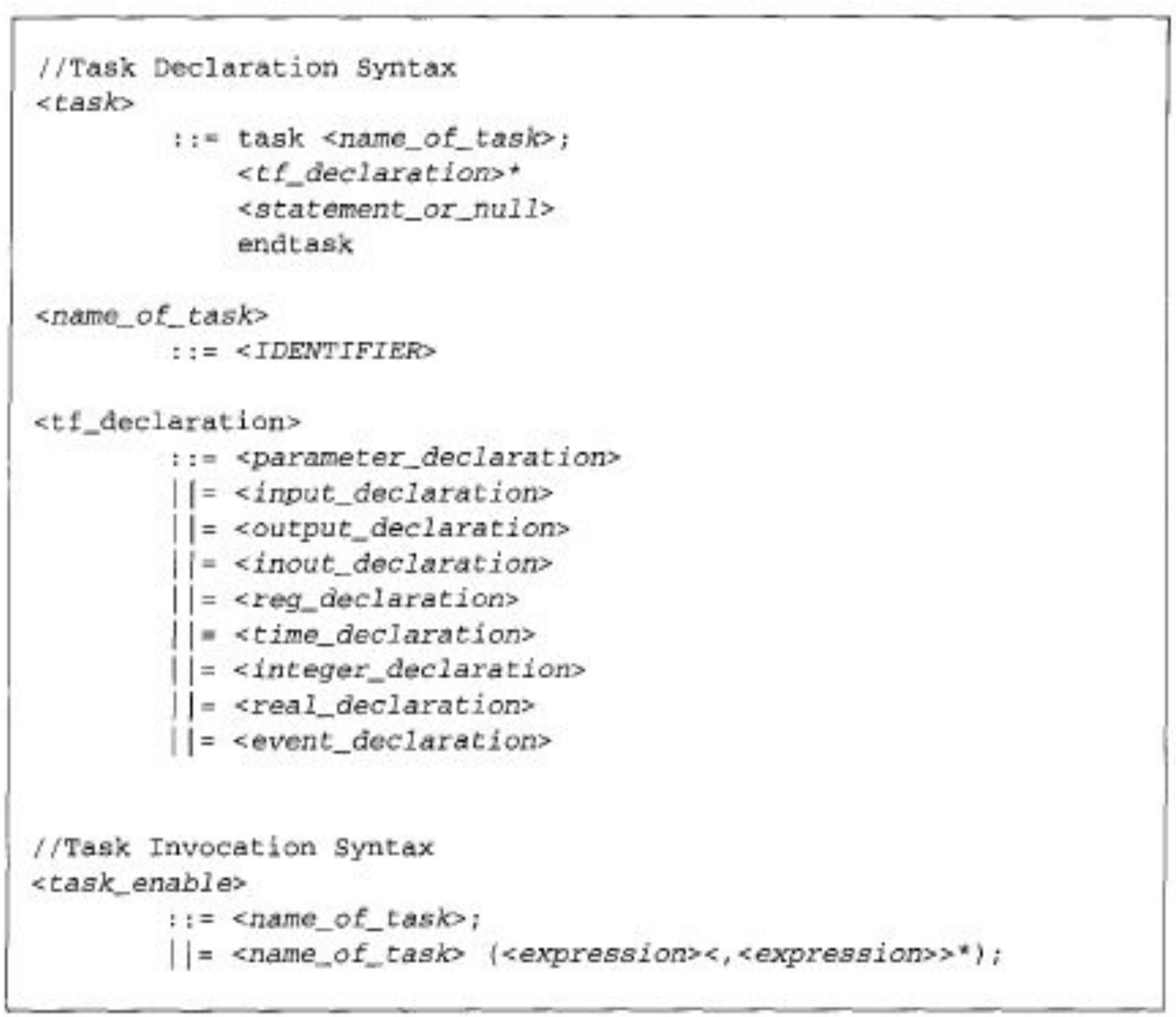
```
   //Write operation
   //Assign values of temporary variables to left-hand-side variables
   a = temp_b;
   b = temp_a;
end
```

Tasks and Functions



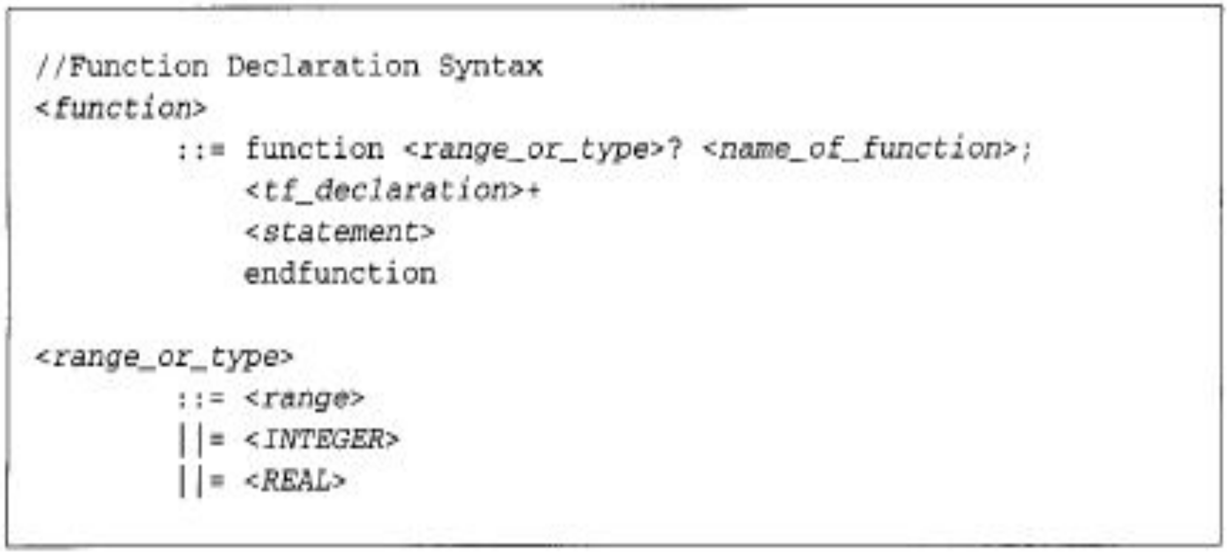
Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output or inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value but can pass multiple values through **output** and **inout** arguments.

Task Declaration and Invocation

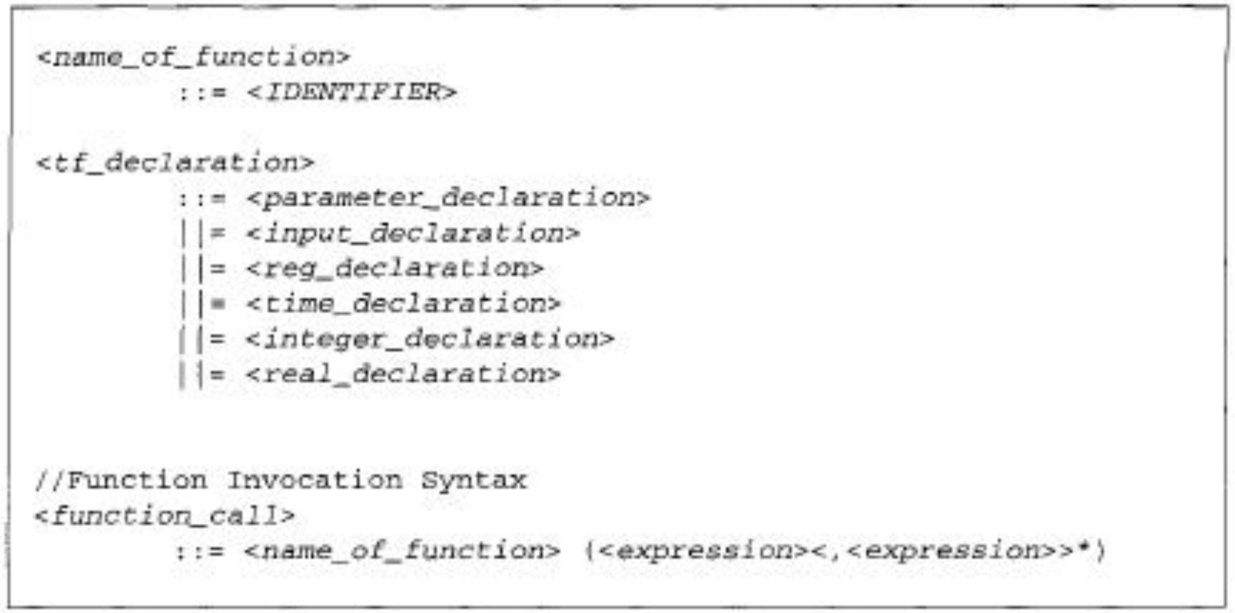


```
//Task Declaration Syntax
<task>
        ::= task <name_of_task>;
            <tf_declaration>*
            <statement_or_null>
            endtask

<name_of_task>
        ::= <IDENTIFIER>

<tf_declaration>
        ::= <parameter_declaration>
        ||= <input_declaration>
        ||= <output_declaration>
        ||= <inout_declaration>
        ||= <reg_declaration>
        ||= <time_declaration>
        ||= <integer_declaration>
        ||= <real_declaration>
        ||= <event_declaration>


//Task Invocation Syntax
<task_enable>
        ::= <name_of_task>;
        ||= <name_of_task> (<expression><,<expression>>*);
```

Function Declaration and Invocation

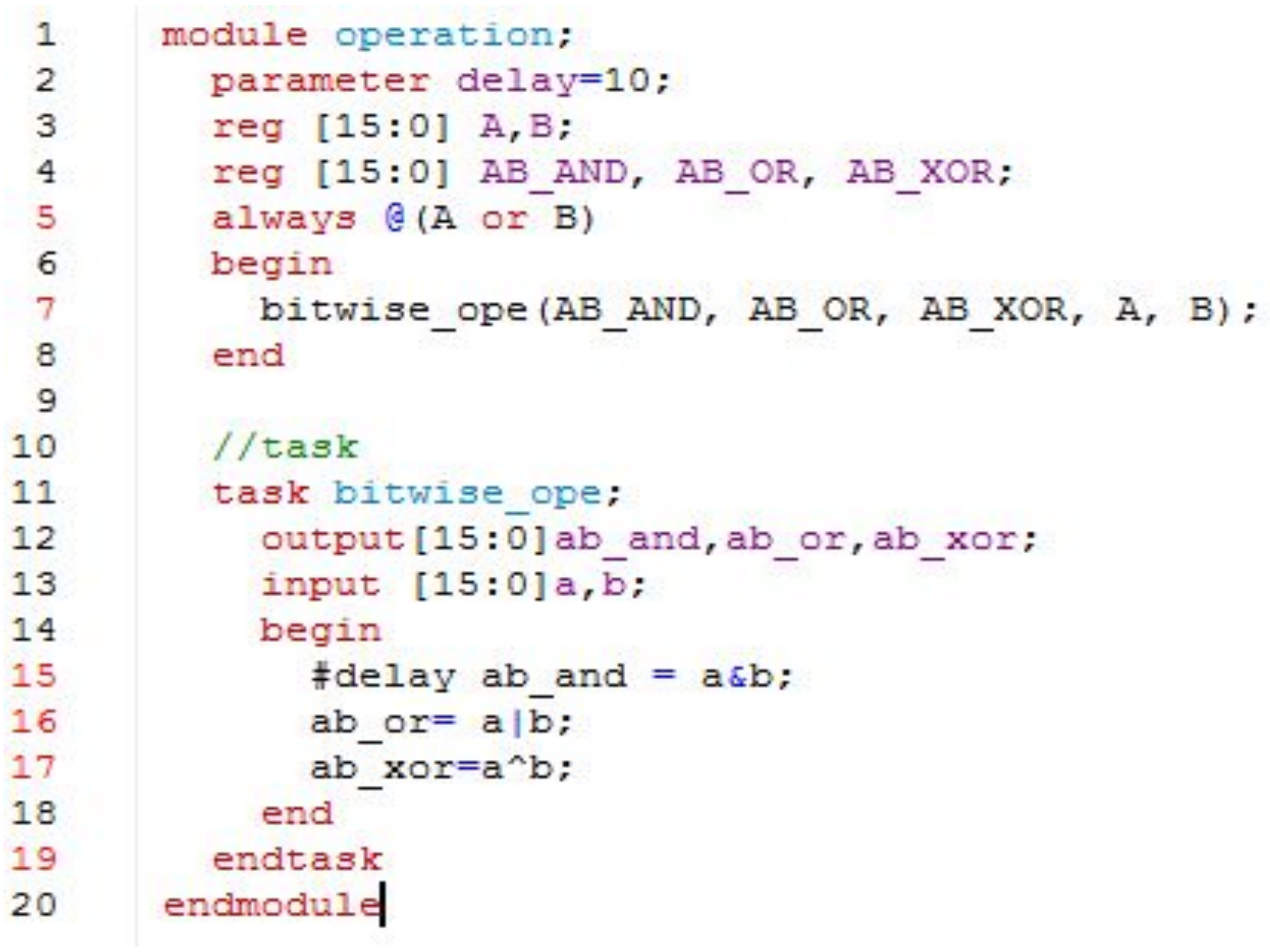


```
//Function Declaration Syntax
<function>
        ::= function <range_or_type>? <name_of_function>;
            <tf_declaration>+
            <statement>
            endfunction

<range_or_type>
        ::= <range>
        ||= <INTEGER>
        ||= <REAL>
```



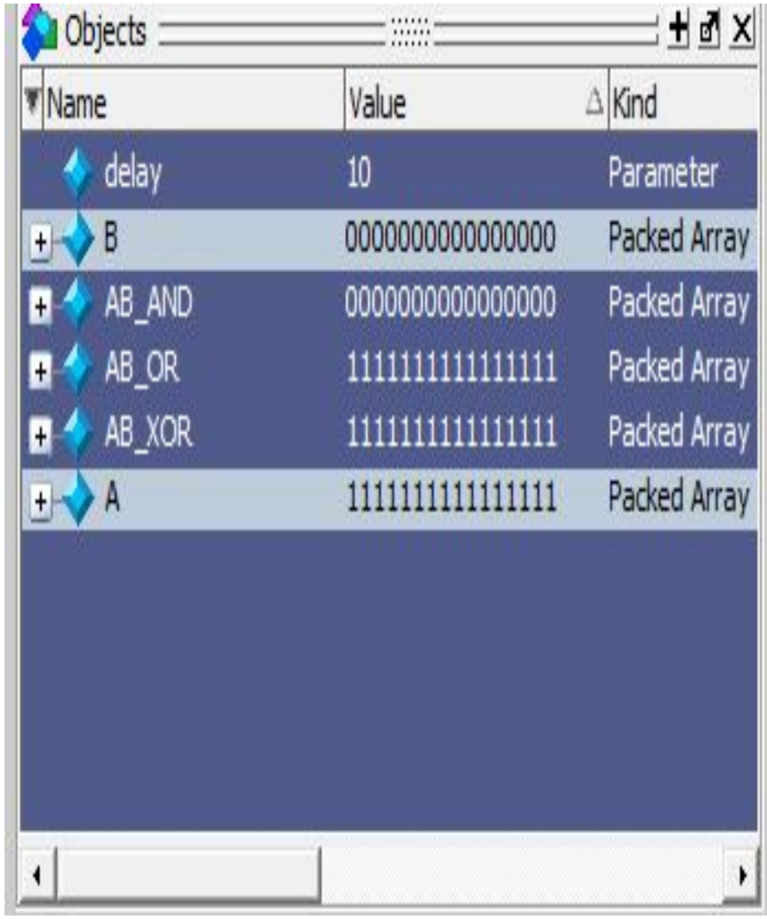
```
<name_of_function>
        ::= <IDENTIFIER>

<tf_declaration>
        ::= <parameter_declaration>
        ||= <input_declaration>
        ||= <reg_declaration>
        ||= <time_declaration>
        ||= <integer_declaration>
        ||= <real_declaration>


//Function Invocation Syntax
<function_call>
        ::= <name_of_function> (<expression><,<expression>>*)
```

Task - Example

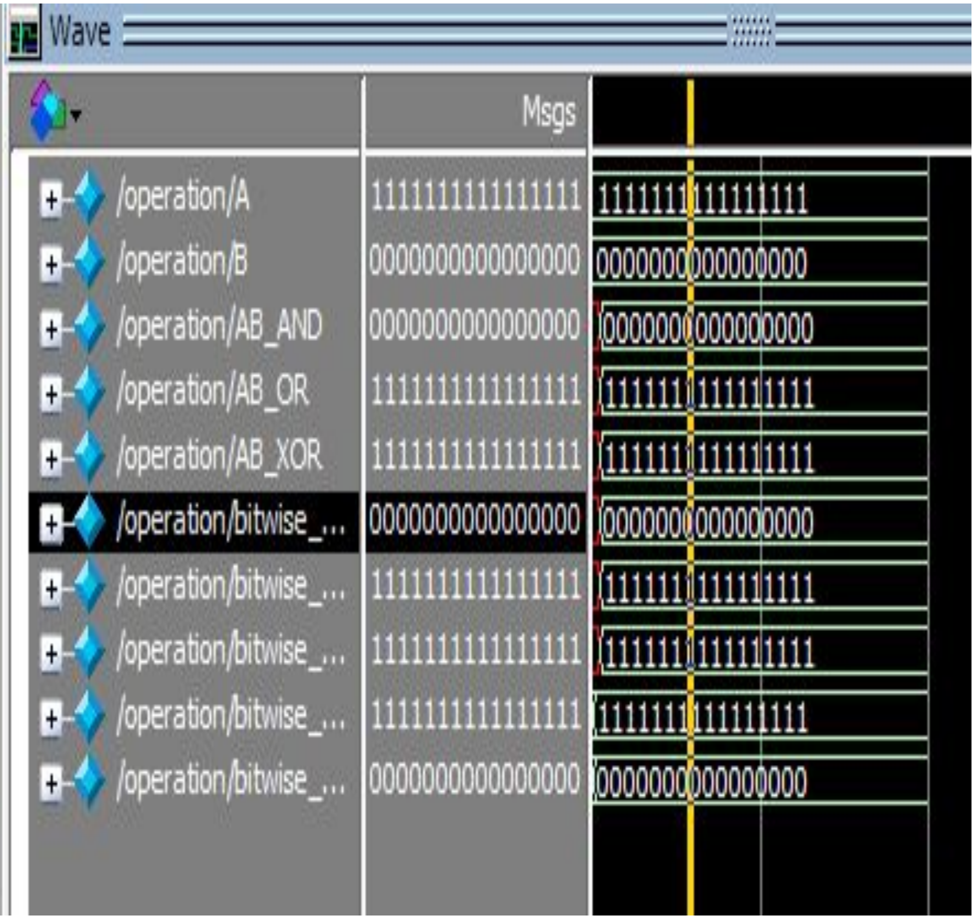


```verilog
module operation;
  parameter delay=10;
  reg [15:0] A,B;
  reg [15:0] AB_AND, AB_OR, AB_XOR;
  always @(A or B)
  begin
    bitwise_ope(AB_AND, AB_OR, AB_XOR, A, B);
  end

  //task
  task bitwise_ope;
    output[15:0]ab_and,ab_or,ab_xor;
    input [15:0]a,b;
    begin
      #delay ab_and = a&b;
      ab_or= a|b;
      ab_xor=a^b;
    end
  endtask
endmodule
```

Task

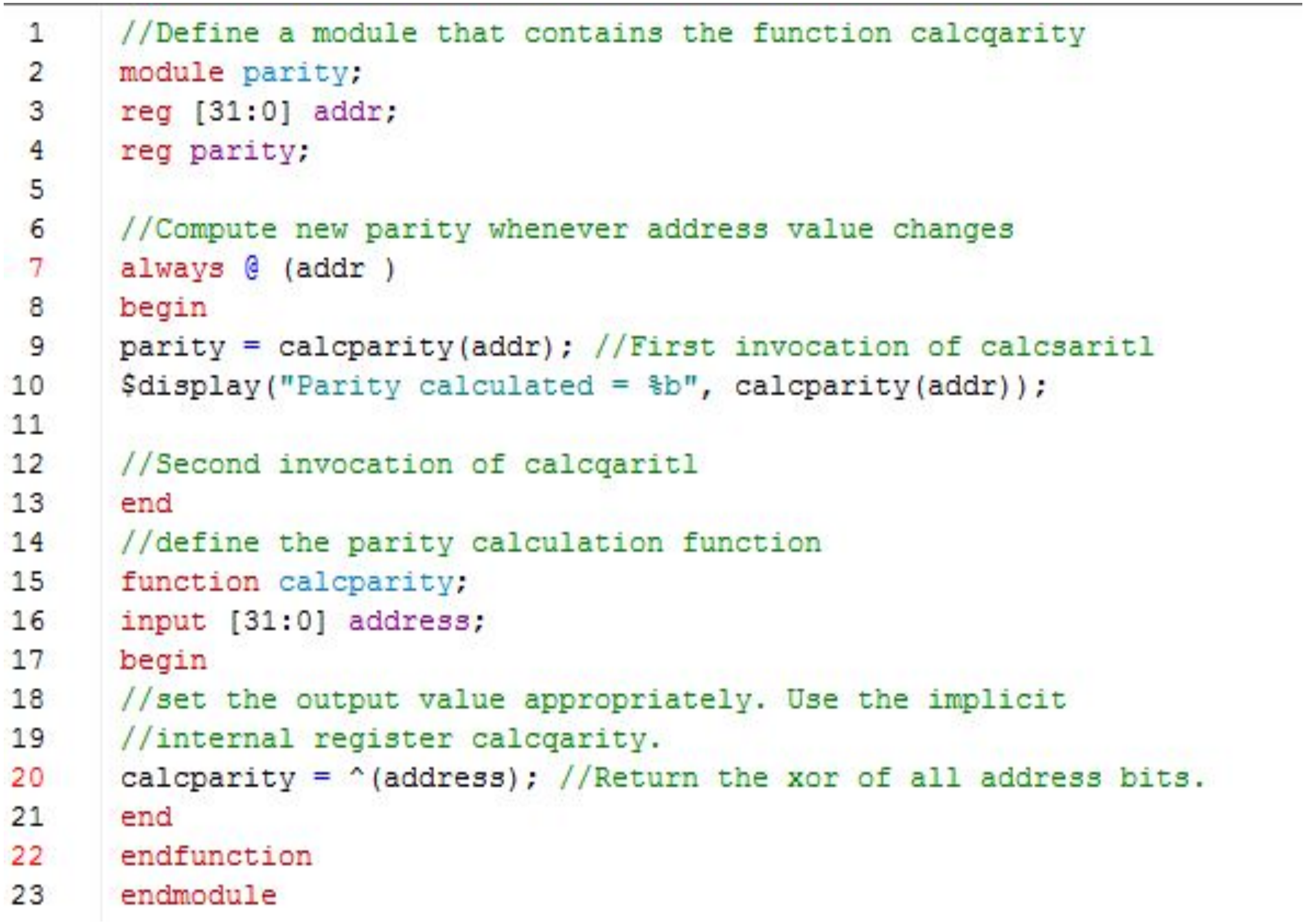


Name	Value	Kind
delay	10	Parameter
B	0000000000000000	Packed Array
AB_AND	0000000000000000	Packed Array
AB_OR	1111111111111111	Packed Array
AB_XOR	1111111111111111	Packed Array
A	1111111111111111	Packed Array



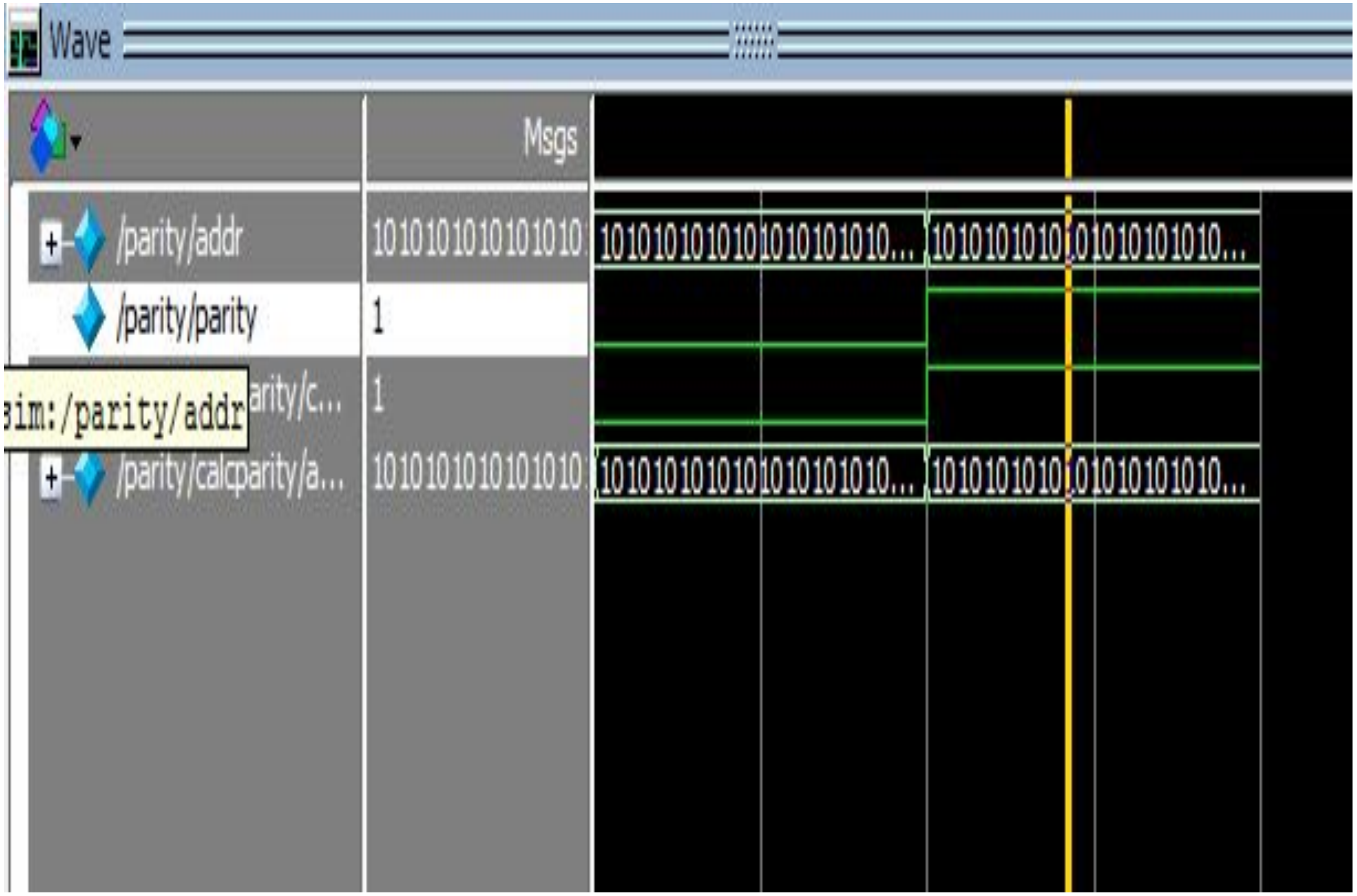
	Msgs	
/operation/A	1111111111111111	1111111111111111
/operation/B	0000000000000000	0000000000000000
/operation/AB_AND	0000000000000000	0000000000000000
/operation/AB_OR	1111111111111111	1111111111111111
/operation/AB_XOR	1111111111111111	1111111111111111
/operation/bitwise_...	0000000000000000	0000000000000000
/operation/bitwise_...	1111111111111111	1111111111111111
/operation/bitwise_...	1111111111111111	1111111111111111
/operation/bitwise_...	1111111111111111	1111111111111111
/operation/bitwise_...	0000000000000000	0000000000000000

Function Example



```
//Define a module that contains the function calcqarity
module parity;
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @ (addr )
begin
parity = calcparity(addr); //First invocation of calcsaritl
$display("Parity calculated = %b", calcparity(addr));

//Second invocation of calcqaritl
end
//define the parity calculation function
function calcparity;
input [31:0] address;
begin
//set the output value appropriately. Use the implicit
//internal register calcqarity.
calcparity = ^(address); //Return the xor of all address bits.
end
endfunction
endmodule
```

Function


Wave
Msgs
/parity/addr
/parity/parity
sim:/parity/addr
arity/c...
/parity/calcparity/a...
1010101010101010
1
1
1010101010101010
101010101010101010101010...
101010101010101010101010...
101010101010101010101010...
101010101010101010101010...

Compiler Directives

- `` `define `` – (Similar to #define in C) used to define global parameter
- Example:

```
 `define BUS_WIDTH 16
 reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
```

- `` `undef `` – Removes the previously defined directive
- Example:

```
 `define BUS_WIDTH 16
   …
   reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
   …
 `undef BUS_WIDTH
```

Compiler Directives (cont.)

- `include – used to include another file
- Example

```
`include "./fulladder.v"
```

System Tasks

- Display tasks
  - $display : Displays the entire list at the time when statement is encountered
  - $monitor : Whenever there is a change in any argument, displays the entire list at end of time step
- Simulation Control Task
  - $finish : makes the simulator to exit
  - $stop : suspends the simulation
- Time
  - $time: gives the simulation

Loop Statements

- Loop Statements
  - Repeat
  - While
  - For
- Repeat Loop
  - Example:

```
repeat (Count)
  sum = sum + 5;
```

  - If condition is a x or z it is treated as 0

Loop Statements (cont.)

- While Loop
  - Example:

```
while (Count < 10) begin
  sum = sum + 5;
  Count = Count +1;
end
```

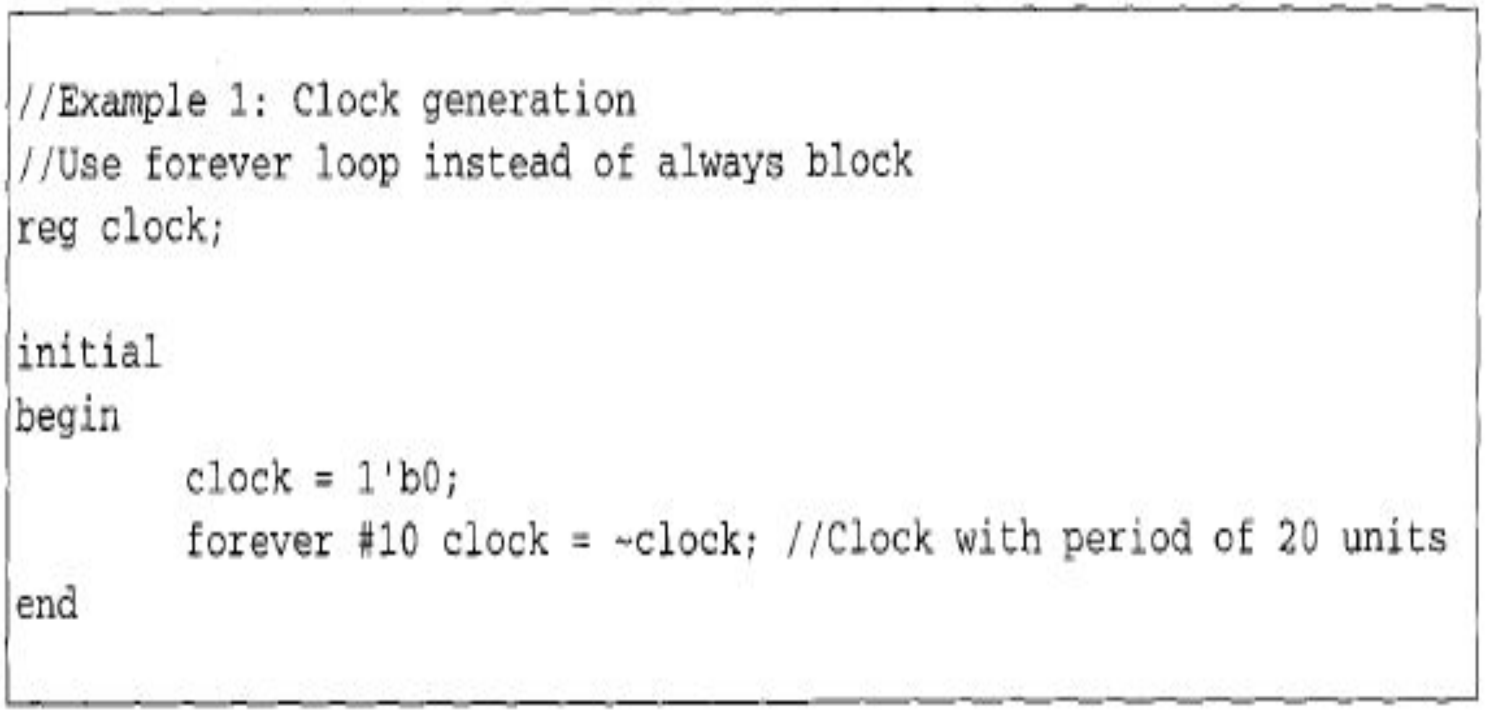
  - If condition is a x or z it is treated as 0

- For Loop
  - Example:

```
for (Count = 0; Count < 10; Count = Count + 1) begin
  sum = sum + 5;
end
```

Loop Statements (cont.)

Forever Loop:



```
//Example 1: Clock generation
//Use forever loop instead of always block
reg clock;

initial
begin
        clock = 1'b0;
        forever #10 clock = ~clock; //Clock with period of 20 units
end
```

Data Types

- Net Types: Physical Connection between structural elements
- Register Type: Represents an abstract storage element.
- Default Values
  - Net Types : z
  - Register Type : x
- Net Types: wire, tri, wor, trior, wand, triand, supply0, supply1
- Register Types : reg, integer, time, real, realtime

Data Types

- Net Type: Wire

  wire [msb : lsb] wire1, wire2, …

  - Example

    wire Reset; // A 1-bit wire

    wire [6:0] Clear; // A 7-bit wire

- Register Type: Reg

  reg [msb : lsb] reg1, reg2, …

  - Example

  reg [3: 0] cla; // A 4-bit register

  reg cla; // A 1-bit register

Restrictions on Data Types

- Data Flow and Structural Modeling
  - Can use only ***wire*** data type
  - Cannot use ***reg*** data type
- Behavioral Modeling
  - Can use only ***reg*** data type (within initial and always constructs)
  - Cannot use ***wire*** data type

Memories

- An array of registers

```
reg [ msb : lsb ] memory1 [ upper : lower ];
```

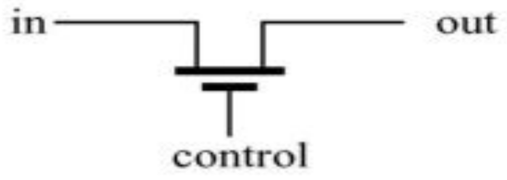
- Example

```
reg [ 0 : 3 ] mem [ 0 : 63 ];
// An array of 64 4-bit registers
reg mem [ 0 : 4 ];
// An array of 5 1-bit registers
```

Switch Level Modeling

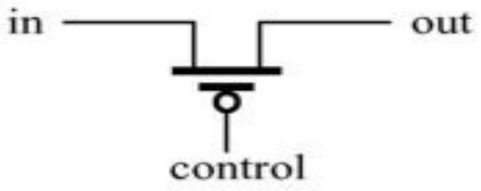
The nmos and pmos Switches

- To instantiate switch elements:
  - switch_name [*instance_name*] (output, input, control);
- The instance_name is optional



nmos		control			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

(a) nMOS switch



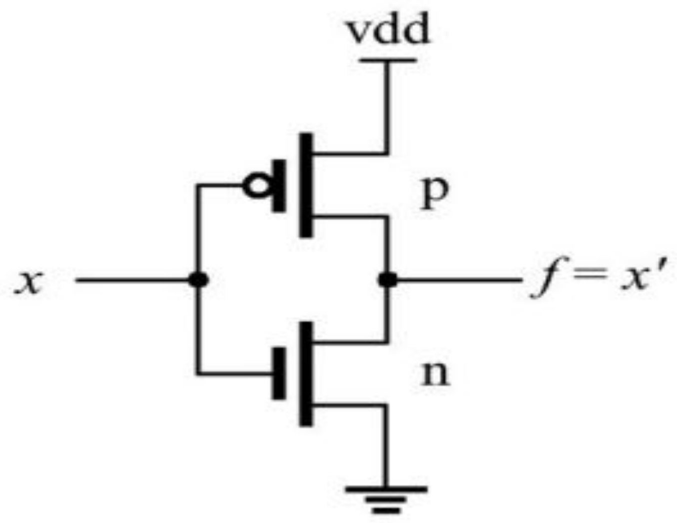
pmos		control			
		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

(b) pMOS switch

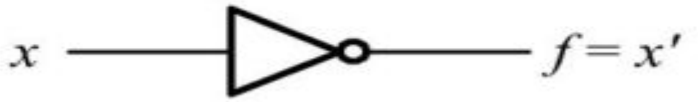
Switch Level Modeling

Example : The CMOS Inverter

```
module my_not (input x, output f);
// internal declaration
supply1 vdd;
supply0 gnd;
// NOT gate body
pmos p1 (f, vdd, x);
nmos n1 (f, gnd, x);
endmodule
```



(a) Circuit

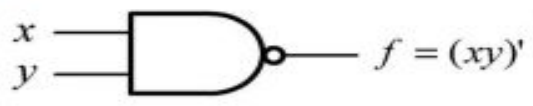


(b) Logic symbol

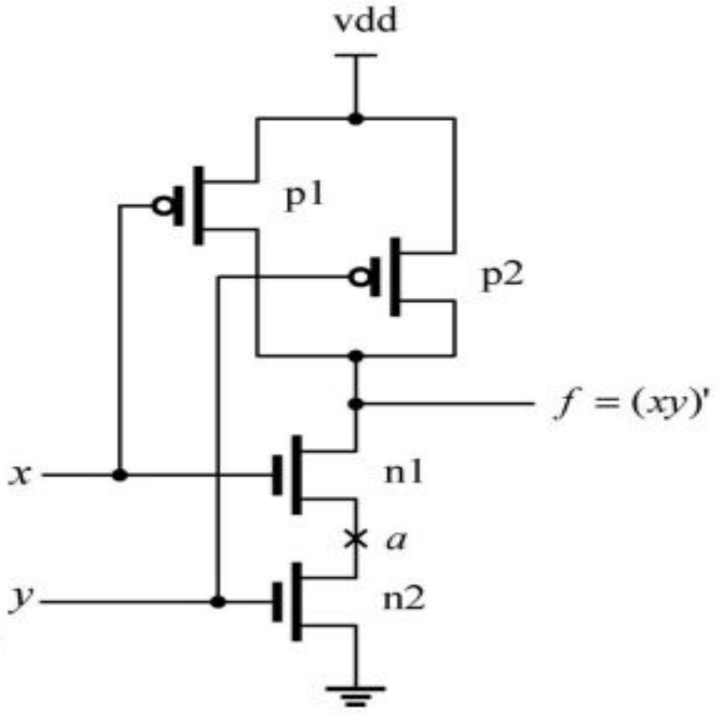
Switch Level Modeling

Example : CMOS NAND Gate

```
module my_nand (input x, y, output f);
supply1 vdd;
supply0 gnd;
wire a;
// NAND gate body
pmos p1 (f, vdd, x);
pmos p2 (f, vdd, y);
nmos n1 (f, a, x);
nmos n2 (a, gnd, y);
endmodule
```



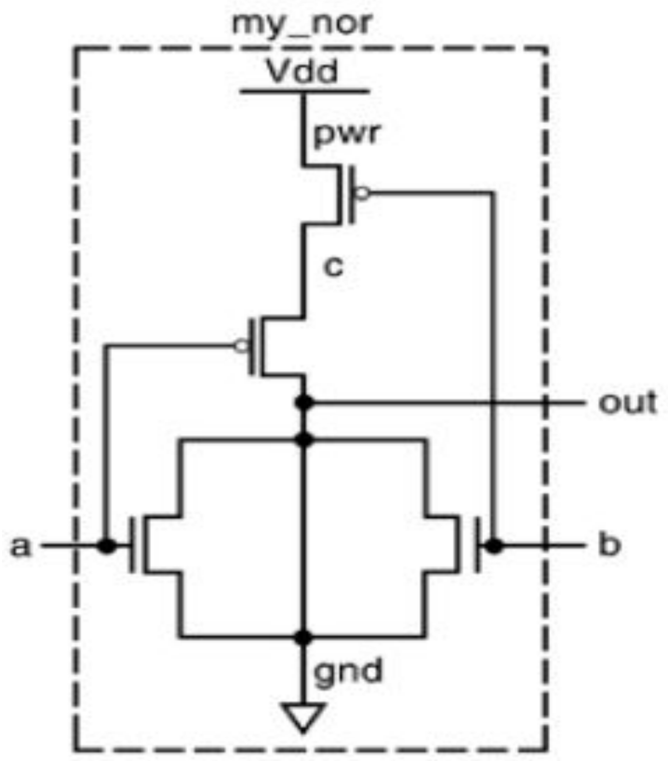
(b) Logic symbol



(a) Circuit

Switch Level Modeling

```
module my_nor(out,a,b);
output out;
Input a,b;
supply1  vdd;
supply0 gnd;
wire c;
pmos p1(c,vdd,b);
pmos p2(out,c,a);
nmos n1(out,gnd,a);
nmos n2(out,gnd,b);
endmodule
```



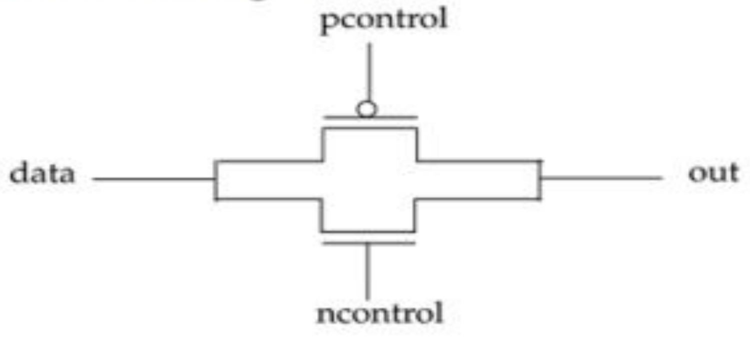
CMOS NOR Gate

Switch Level Modeling

CMOS Switch

- Is designed to propagate both 0 and 1 well
  - Nmos is only able to propagate 0 well
  - Pmos is only able to propagate 1 well
- To instantiate CMOS switches :
  - cmos [*instance_name*] (output, data, ncontrol, pcontrol);
- The instance_name is optional
- Exactly equivalent to the following :

```
nmos (out, data, ncontrol);
pmos (out, data, pcontrol);
```

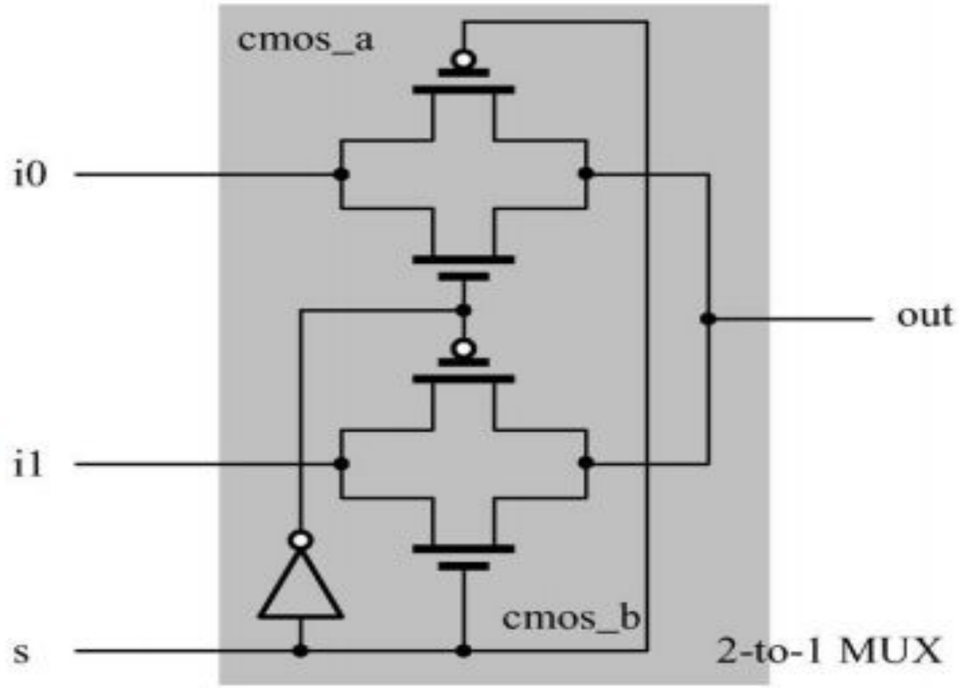


CMOS

Switch Level Modeling

Example : A 2-to-1 Multiplexer

```
module my_mux (out, s, i0, i1);
output out;
input s, i0, i1;
//internal wire
wire sbar;
not (sbar, s);
//cmos switches
cmos (out, i0, sbar, s);
cmos (out, i1, s, sbar);
endmodule
```


cmos_a
i0
out
i1
cmos_b
s
2-to-1 MUX