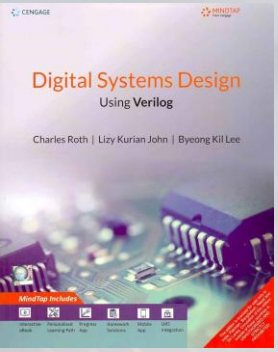
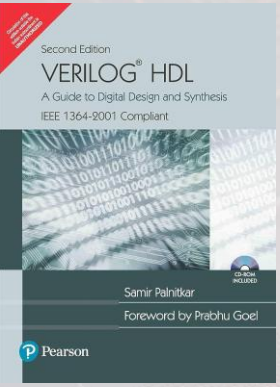
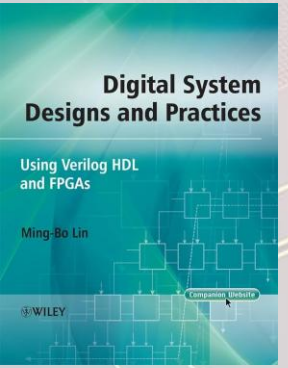
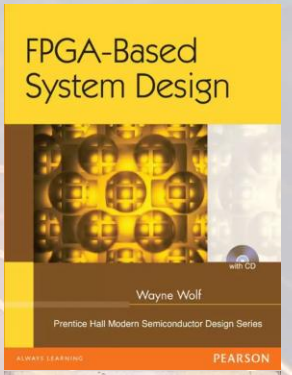
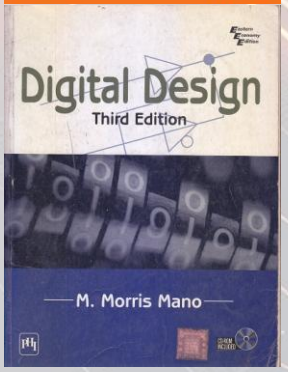






FPGA Based System Design using Verilog





COURSE INFORMATION

Verilog HDL

Coding Style: Lexical Conventions - Ports and Modules – Operators - Structural Modeling, Data Flow Modeling - Behavioral level Modeling - Tasks & Functions. System Tasks & Compiler Directives - Test Bench. **[10]**

Verilog Modelling of Combinational and Sequential Circuits:

Behavioral, Data Flow and Structural Realization – Adders – Multipliers- Comparators - Flip Flops - Realization of Shift Register - Realization of a Counter- Synchronous and Asynchronous. **[8]**

Synchronous Sequential Circuit:

State diagram-state table –state assignment-choice of flipflops – Timing diagram –One hot encoding Mealy and Moore state machines – Design of serial adder using Mealy and Moore state machines - State minimization – Sequence detection- Design examples: Sequence detector, Serial adder, Vending machine using One Hot Controller. **[8]**

COURSE INFORMATION

Overview of FPGA Architectures and Technologies
FPGA Architectural options, coarse vs fine-grained, vendor-specific issues (emphasis on Xilinx FPGA), Antifuse, SRAM and EPROM-based FPGAs, FPGA logic cells, interconnection network, and I/O Pad. **[5]**

System Design Examples using Xilinx FPGAs
Traffic light Controller, Real Time Clock - Interfacing using FPGA: VGA, Keyboard, LCD, Embedded Processor Hardware Design. **[5]**

***Self-directed Learning:** Simulation of various designs using Xilinx Software.

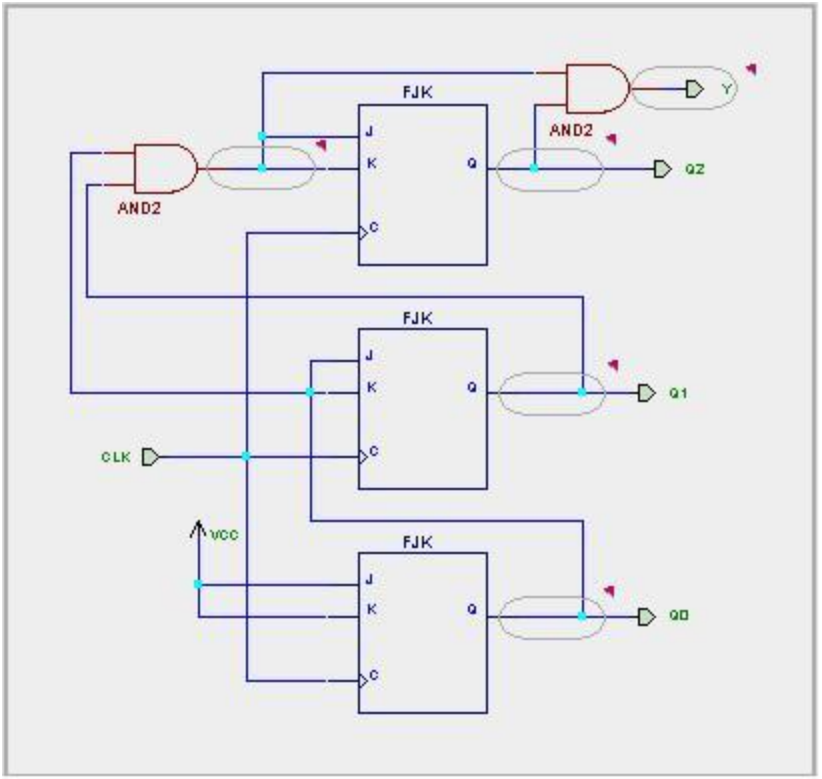
Course Books

1. M.J.S. Smith, *"Application Specific Integrated Circuits"*, Pearson, 2000.
2. Peter Ashenden, *"Digital Design using Verilog"*, Elsevier, 2007.
3. Clive Maxfield, *"The Design Warriors' Guide to FPGAs"*, Elsevier, 2004
4. Samir Palnitkar, *"Verilog HDL: A Guide to Digital Design and Synthesis"* Prentice Hall, Second Edition, 2003.
5. Wayne Wolf, *"FPGA Based System Design*", Prentices Hall Modern Semiconductor Design Series, Pearson, 2004.

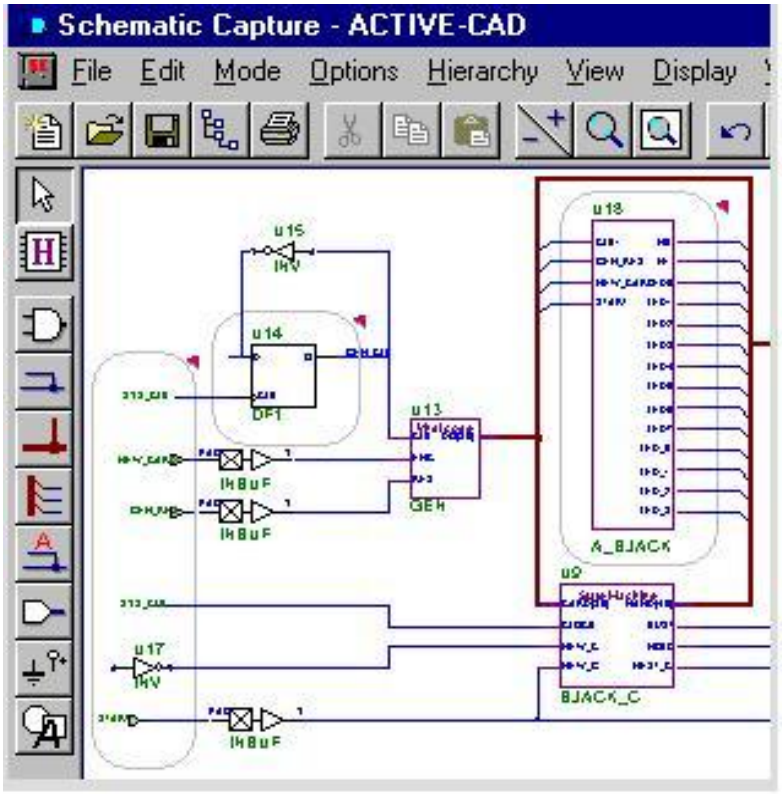
OBJECTIVES

CLO1	Explain the various modelling style in Verilog
CLO2	Explain the various FPGA architecture and technologies
CLO3	Develop the ability to model combinational and sequential digital circuits by Verilog HDL
CLO4	Implementation of the combinational and sequential digital circuits in FPGA
CLO5	Design and implement a system using FPGAs for real-world applications like a traffic light controller or real-time clock.

Traditional Design approaches



Gate Level Design



Schematic Design

Where is the problem?

- System specification is behavioral
- Manual Translation of design in Boolean equations
- Handling of large Complex Designs
- Can we still use SPICE for simulating Digital circuits?

Importance of HDLs

HDL: Hardware Description Language

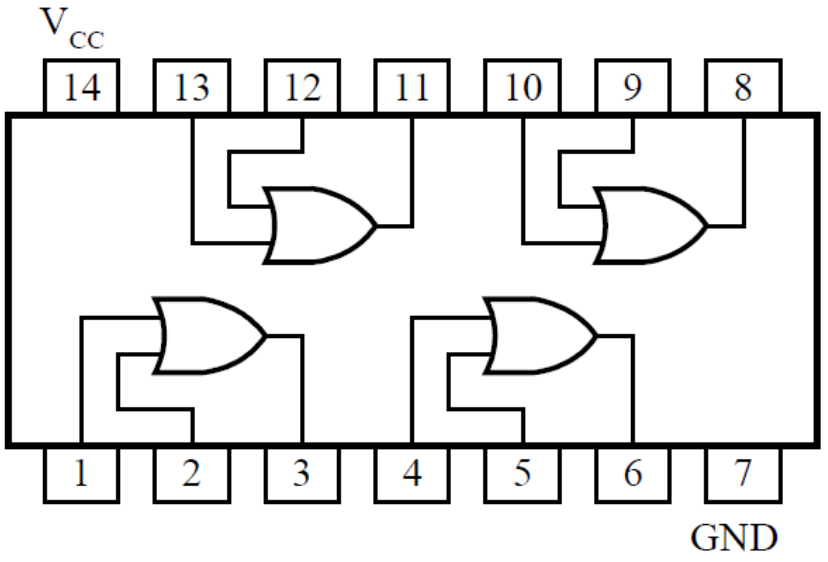
- Two most commonly used HDLs:
  - **VHDL** (Very high-speed integrated circuits HDL-VHSIC)
  - **Verilog HDL** (also called Verilog for short)
- Features of HDLs:
  - Design can be described at a very abstract level.
  - Functional verification can be done early in the design cycle.
  - Designing with HDLs is analogous to computer programming.

Importance of HDLs

- **Why use an HDL?**
  - Describe complex designs (millions of gates)
  - Input to synthesis tools (synthesizable subset)
  - Design exploration with simulation
- **Why not use a general-purpose language**
  - Support for structure and instantiation
  - Support for describing bit-level behavior
  - Support for timing
  - Support for concurrency
- **Verilog vs. VHDL**
  - Verilog is relatively simple and close to C
  - VHDL is complex and close to Ada
  - Verilog has 60% of the world digital design market (larger share in US)

Modules –Hardware Module Concept

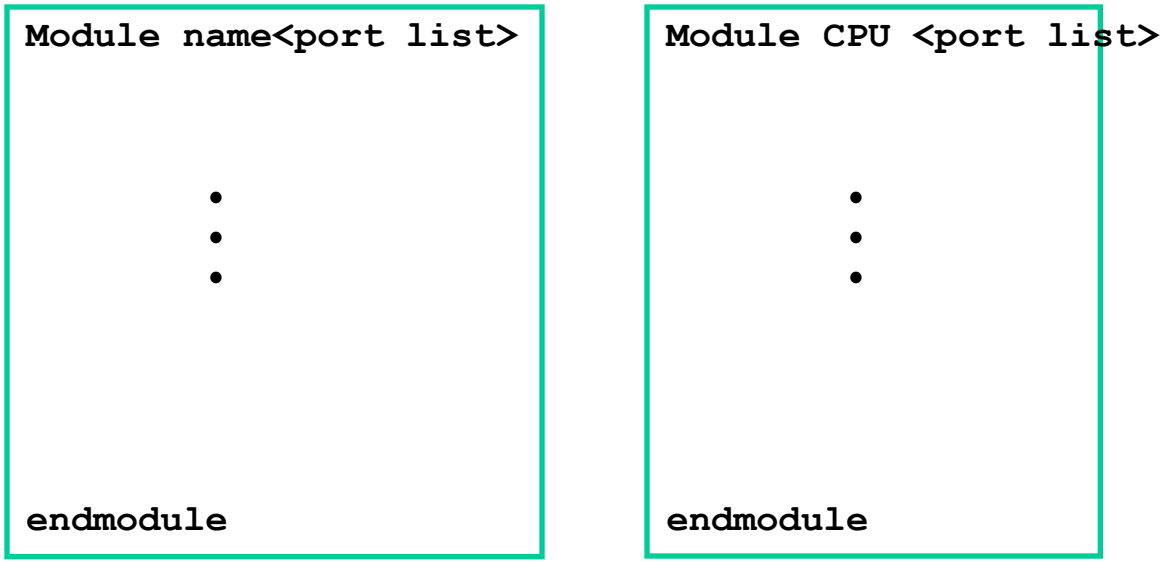
- The basic unit of a digital system is a module.
- Each module consists of:
  - a core circuit (called internal or body) --- performs the required function
  - an interface (called ports) --- carries out the required communication between the core circuit and outside.



Modules –Verilog HDL modules

The module is the basic building block in Verilog

- Modules can be interconnected to describe the structure of your digital system
- Modules start with keyword **module** and end with keyword **endmodule**



Modules have ports for interconnection with other modules

Modules –Verilog HDL modules

```
Module name<port list>

    .
    .
    .

endmodule
```

```
module and_1(a,b,c);
input a,b;
output c;
assign c=a&b;
endmodule
```

Modules must have a module_name (here:and_1), which is an identifier for the module.

The module can’t be nested

Value Set

- In Verilog HDL
  - 0 and 1 represent logic values low and high, respectively.
  - z indicates the high-impedance condition of a node or net.
  - x indicates an unknown value of a net or node.

Value	Meaning
0	Logic 0, false condition
1	Logic 1, true condition
x	Unknown logic value
z	High impedance

Lexical Conventions

- Verilog HDL uses almost the same lexical conventions as C language.
  - Identifiers consists of alphanumeric characters, _, and $.
    - Verilog is a case-sensitive language just like C.

Identifire's first character must be a letter or underscore, and not a digit or dollar sign

  - White space: blank space (\b), tabs (\t), and new line (\n).

Usually ignored by Verilog except in string

  - Sized number: <size>`<base format><number>
    - 4`b1001 --- a 4-bit binary number
    - 16`habcd --- a 16-bit hexadecimal number

Comments

1. “//” indicates a single-line comment.

2. A *multiple-line* comment starts with /* and ends with */ and cannot be nested.

```
// This is a single line comment

integer a;   // Creates an int variable called a

/*
This is a
multiple-line or
block comment
*/

/* This is /*
an invalid nested
block comment */
*/

/* However,
// this one is okay
*/

// This is also okay
////////////// Still okay
```

Lexical Conventions

- Unsized number: `<base format><number>
  - 2007 --- a 32-bit decimal number by default
  - `habc --- a 32-bit hexadecimal number
- x or z values: x denotes an unknown value; z denotes a high impedance value.
- Negative number: -<size>`<base format><number>
  - -4`b1001 --- a 4-bit binary number
  - -16`habcd --- a 16-bit hexadecimal number
- "_" and "?"
  - 16`b0101_1001_1110_0000
  - 8`b01??_11?? --- equivalent to a 8`b01zz_11zz

Lexical Conventions

- String: “Back to School and Have a Nice Semester”

- Coding style:
  - Use lowercase letters for all signal names, variable names, and port names.
  - Use uppercase letters for names of constants and user-defined types.
  - Use meaningful names for signals, ports, functions, and parameters.

Keywords

Note : All keywords are defined in lower case

Examples :

module, endmodule

input, output, inout

reg, integer, real, time

not, and, nand, or, nor, xor

parameter

begin, end

Data Types

- Nets
  - Nets are physical connections between devices
  - Nets always reflect the logic value of the driving device
  - Many types of nets, but all we care about is **wire**
- Registers
  - Implicit storage – unless variable of this type is modified it retains previously assigned value
  - Does not necessarily imply a hardware register
  - Register type is denoted by **reg**

- int is also used

Nets

- *Net* data type represent physical connections between structural entities.
- A *net* must be driven by a driver, such as a gate or a continuous assignment.
- Verilog automatically propagates new values onto a *net* when the drivers change value.

```
wire   : default
wor    : wire-ORed
wand   : wire-ANDed
trireg : with Capacitive Storage
```

```
supply1        ; power
supply0        ; groun
```

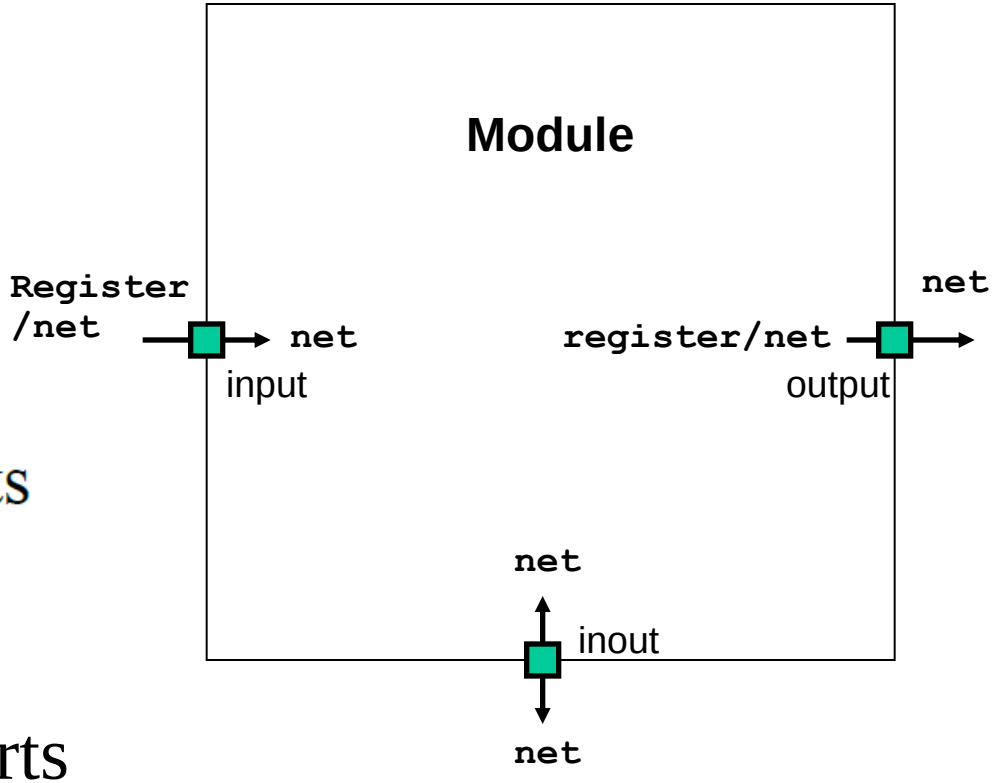
Registers

- *Registers* represent abstract storage elements.
- A *register* holds its value until a new value is assigned to it.
- *Registers* are used extensively in behavior modeling.

Scalars and Vectors

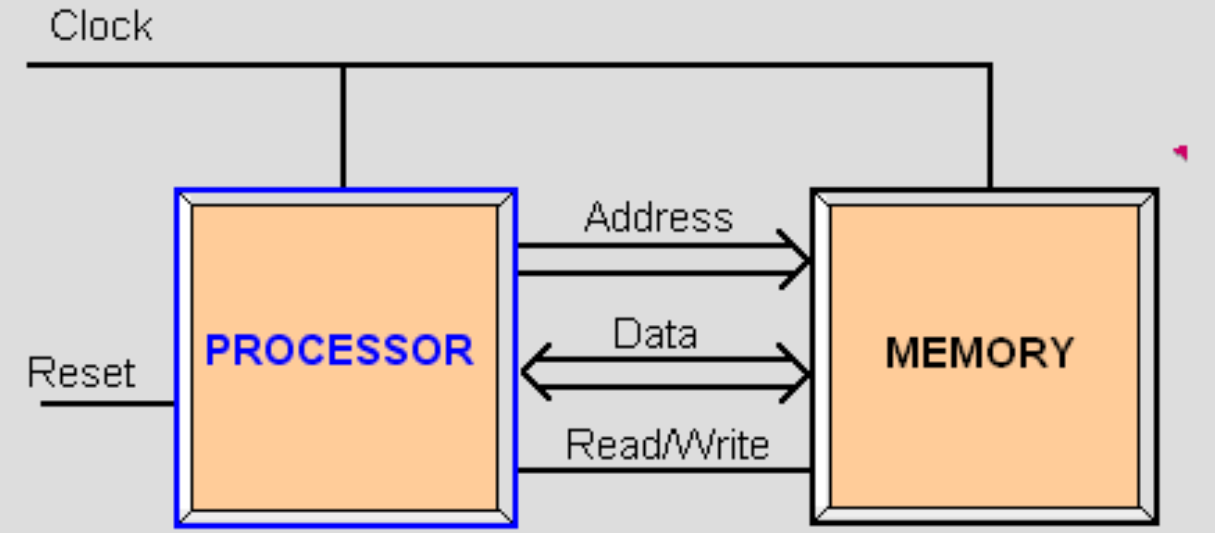
Ports and Data Types

- Port Declaration
  - input: input ports.
  - output: output ports.
  - inout: bidirectional ports



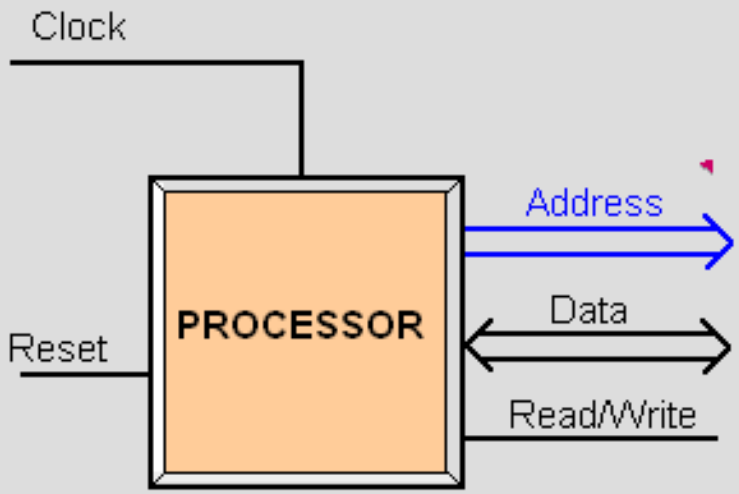
- Correct data types for ports

Specifications of Ports




```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
endmodule
```

Registered Output




```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
reg [19:0] Address;
endmodule
```

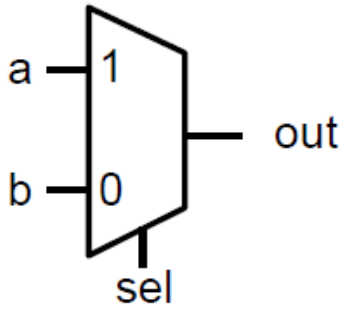
Output ports can be type *register* by adding a new declaration for the port, stating that it is a register type.

Assignments Statement

- Continuous Assignment

```
assign out = a&b;
```

- Procedural Assignment




```
module combinational(a, b, sel,
                                out);
  input a, b;
  input sel;
  output out;
  reg out;

  assign out = sel?a:b

endmodule
```

Blocking and Nonblocking Statement

- ***Blocking assignment:*** **evaluation and assignment are immediate**

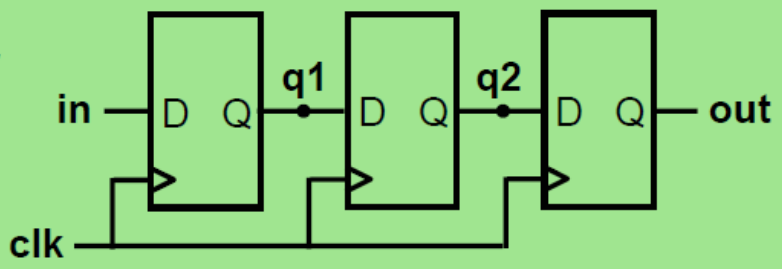
```
always @ (a or b or c)
begin
   x = a | b;          1. Evaluate a | b, assign result to x
   y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
   z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

- ***Nonblocking assignment:*** **all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)**

```
always @ (a or b or c)
begin
   x <= a | b;         1. Evaluate a | b but defer assignment of x
   y <= a ^ b ^ c;     2. Evaluate a^b^c  but defer assignment of y
   z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                    4. Assign x, y, and z with their new values
```

Blocking and Nonblocking Statement

Flip-Flop Based Digital Delay Line



- **Will nonblocking and blocking assignments both produce the desired result?**

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

Operators

There are three types of operators: *unary*, *binary*, and *ternary or conditional*.

- Unary operators shall appear to the left of their operand
- Binary operators shall appear between their operands
- Conditional operators have two separate operators that separate three operands

```
1 | x = ~y;
2 | x = y | z;
3 | x = (y > 5) ? w : z;
```

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Equality Operators	==, !=, ===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^, ~^
Unary Reduction	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	?:
Concatenations	{}

Arithmetic Operators

Operator	Description
a + b	a plus b
a - b	a minus b
a * b	a multiplied by b
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

Arithmetic Operators

```
1  module des;
2    reg [7:0]  data1;
3    reg [7:0]  data2;
4
5    initial begin
6      data1 = 45;
7      data2 = 9;
8
9      $display ("Add + = %d", data1 + data2);
10     $display ("Sub - = %d", data1 - data2);
11     $display ("Mul * = %d", data1 * data2);
12     $display ("Div / = %d", data1 / data2);
13     $display ("Mod %% = %d", data1 % data2);
14     $display ("Pow ** = %d", data2 ** 2);
15
16   end
17 endmodule
```

```
Add + =  54
Sub - =  36
Mul * = 149
Div / =   5
Mod % =   0
Pow ** =  81
```

Relational Operators

```
1  module des;
2    reg [7:0]  data1;
3    reg [7:0]  data2;
4
5    initial begin
6      data1 = 45;
7      data2 = 9;
8      $display ("Result for data1 >= data2 : %0d", data1 >= data2);
9
10     data1 = 45;
11     data2 = 45;
12     $display ("Result for data1 <= data2 : %0d", data1 <= data2);
13
14     data1 = 9;
15     data2 = 8;
16     $display ("Result for data1 > data2 : %0d", data1 > data2);
17
18     data1 = 22;
19     data2 = 22;
20     $display ("Result for data1 < data2 : %0d", data1 < data2);
21
22   end
23 endmodule
```

```
Result for data1 >= data2 : 1
Result for data1 <= data2 : 1
Result for data1 > data2 : 1
Result for data1 < data2 : 0
```

Equality Operators

Operator	Description
a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, result can be unknown
a != b	a not equal to b, result can be unknown

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length.
- For the == and != operators, the result is x, if either operand contains an x or a z

Logical Operators

Operator	Description
a && b	evaluates to true if a *and* b are true
a || b	evaluates to true if a *or* b are true

Logical operators often combine logical expressions formed from relational operators.

Example	Meaning
`(a == b) && (c < d)`	true if a is equal to b AND c is less than d
`a == b && c < d`	same as the above (see precedence rules)
`(a > b) || (c != d)`	true if a is greater than b OR if c is not equal to d
`a > b || c != d`	same as the above (see precedence rules)

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Equality Operators	==, !=, ===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^, ~^
Unary Reduction	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	?:
Concatenations	{}

Logical Operators

Operator	Description
a && b	evaluates to true if a *and* b are true
a || b	evaluates to true if a *or* b are true

Logical operators often combine logical expressions formed from relational operators.

Example	Meaning
`(a == b) && (c < d)`	true if a is equal to b AND c is less than d
`a == b && c < d`	same as the above (see precedence rules)
`(a > b) || (c != d)`	true if a is greater than b OR if c is not equal to d
`a > b || c != d`	same as the above (see precedence rules)

Operator Precedence

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
->	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
-	Unary minus (Negation)		
++	Increment		
--	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
-	Subtraction		

Operator Precedence

<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

Bit-wise Operators

Operator	Type
&	And
~&	Nand
|	Or
~|	Nor
^	Xor
~^	Xnor

Example

```
wire [3:0] a,b,c;
assign a = 4'b1010;
assign b = 4'b1100;
assign c = a & b;
```

c will now have the value 4'b1000

```
assign c = ~a;
```

Now **c** will have the value 4'b0101

Unary Reduction Operators

Operator	Type
&	And
~&	Nand
|	Or
~|	Nor
^	Xor
~^	Xnor

Example

```
wire [3:0] a;
wire b;
assign a = 4'b1010;
assign b = &a;
```

In this example **b** will be 0

```
assign b = a[0] & a[1] & a[2] & a[3];
```

Reduction Operators

Shift Operators

- **Logical shift operators** : << and >>
- **Arithmetic shift operators**: <<< and >>>

The basic shift operators are **zero-filling**

Example 1

```
wire [4:0] a,b,c;
assign a = 5'b10100;
assign b = a << 2;
assign c = a >> 2;
```

b: 5'b10000
c: 5'b00101

Example 2

```
wire [4:0] a,b,c;
assign a = 5'b10100;
assign b = a <<< 2;
assign c = a >>> 2;
```

b: 5'b10000
c: 5’b11101

Shift Operators

- These vary from the basic shift operators because they perform **sign-extended** shifts.
- When shifting left, it performs the same as the basic shift operator, but
- When shifting to the right, the most significant bit (the **sign bit**) plays a role.

If the sign bit is 0, then the arithmetic shift operator acts the same way as the basic shift operator. However, if the sign bit is 1, the shift will fill the left side with ones.

Concatenation and Replication Operators

Function		Operator
Concatenation	:	{ , }
Replication	:	{ { } }

The **concatenation operator** is used to merge two values together into a wider value.

The **replication operator** is used to duplicate a value multiple times.

Example 1

```
wire [3:0] a,b;
wire [7:0] c;
assign a = 4'b1100;
assign b = 4'b1010;
assign c = {a,b};
```

c: 8'b11001010

Example 2

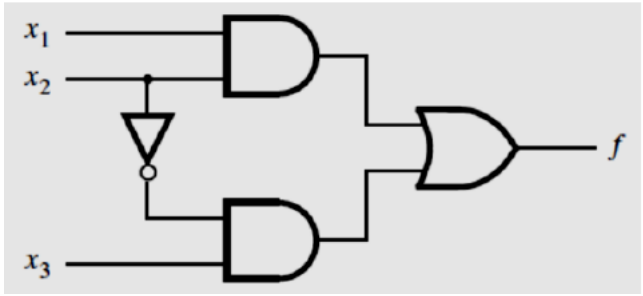
```
wire [1:0] a;
wire [7:0] b;
assign a = 4'b10;
assign b = {4{a}};
```

b: 'b10101010
(a is replicated four times)

Exercises

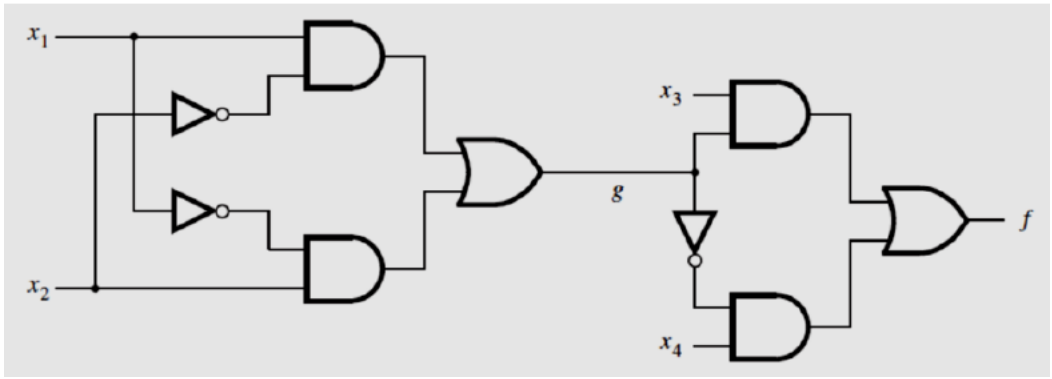
Write Verilog code to implement the following circuit:

(A)




```
module example1(x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

(B)




```
module example5(x1, x2, x3, x4, f);
        input x1, x2, x3, x4;
        output f;
        assign g = (x1 & ~x2) | (~x1 & x2);
        assign f = (g & x3) | (~g & x4);
endmodule
```

Exercises

(A) Write behavioral Verilog code for 2 to 1 multiplexer using always and conditional operators.

```
module mux2to1 (w0, w1, s, f);
        input w0, w1, s;
        output f;
        reg f;
        always @(w0 or w1 or s)
        f = s ? w1 : w0;
endmodule
```

(B) Write a Verilog code for a 4 to 1 multiplexer

```
module m41(input a, input b, input c, input d, input s0, s1, output out);
assign out = s1 ? (s0 ? d : c) : (s0 ? b : a);
endmodule
```

Exercises

(A) Write behavioral Verilog code for a 4 to 1 multiplexer using the **if-else** statement.

```
module multiplexer4_1( din, sel, dout );
output dout;
reg dout;
input [3:0] din;
input [1:0] sel;
always @ (din or sel)
begin
        if (sel= =0)
        dout = din[3];
        else if (sel= =1)
        dout = din[2];
        else if (sel= =2)
        dout = din[1];
        else
        dout = din[0];
end
endmodule
```

Exercises

(A) Write behavioral Verilog code for a 4 to 1 multiplexer using **case** statement.

```
module multiplexer4_1( din, sel, dout );
output dout;
reg dout;
input [3:0] din;
input [1:0] sel;
always @ (din or sel)
begin
 case (sel)
2'b00 : dout <= din[3];
2'b01 : dout <= din[2];
2'b10 : dout <= din[1];
default : dout <= din[0];
endcase
end
endmodule
```

```
2'b11 : dout <= din[0];
```

What if we have ‘x’ or ‘z’ ??

Module Modeling Style

- Structural Style
- DataFlow Style
- Behavioral Style
- Mixed Style

Module Modeling Styles

- Structural style
  - Gate level comprises a set of interconnected gate primitives.
  - Switch level consists of a set of interconnected switch primitives.
- Dataflow style
  - specifies the dataflow (i.e., data dependence) between registers.
  - is specified as a set of continuous assignment statements.

Module Modeling Styles

- Behavioral or algorithmic style
  - is described in terms of the desired design algorithm
  - is without concerning the hardware implementation details.
  - can be described in any high-level programming language.
- Mixed style
  - is the mixing use of above three modeling styles.
  - is commonly used in modeling large designs.
- In industry, RTL (register-transfer level) means
  - RTL = synthesizable behavioral + dataflow constructs

Structural Modeling Styles

- The **module is implemented** in terms of **logic gates** and interconnections between these gates.
- All the **basic gates** are available as ready modules **called "Primitives."**
- Design at this level is similar to describing a design in terms of a gate-level logic diagram.

12-Gate Primitives

16-Switch Primitives

Verilog Switch Primitives

- Ideal switches – without a prefixed letter "r"
- Resistive switches – with a prefixed letter "r"

- MOS switches
  - nmos
  - pmos
  - cmos
- Bidirectional switches
  - tran
  - tranif0
  - tranif1
- Power and ground nets
  - supply1
  - supply0

- Resistive switches
  - rnmos
  - rpmos
  - rcmos
- Resitive bidirectional switches
  - rtran
  - rtranif0
  - rtranif1
- Pullup and pulldown
  - pullup
  - pulldown

Verilog Gate Primitives

- and/or gates
  - have one scalar output and multiple scalar inputs
  - are used to realize the basic logic operations
  - include

    and or xor nand nor xnor
- buf/not gates
  - have one scalar input and one or multiple scalar outputs
  - are used to realize the not operation,
  - are used to buffer the output of an and/or gate,
  - are used as controlled buffers
  - include

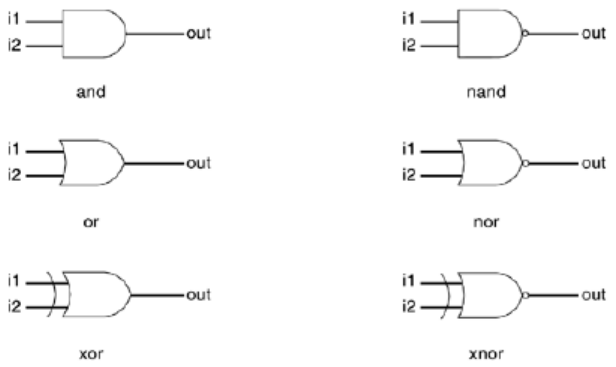
    buf not bufif0 notif0 bufif1 notif1

Verilog Gate Primitives

- Basic logic gates only
  - and
  - or
  - not
  - buf
  - xor
  - nand
  - nor
  - xnor
  - bufif1, bufif0
  - notif1, notif0

Verilog Primitives - Example

```
and i1 (output, input_1, input_2, …, input_n);
nand i2 (output, input_1, input_2, …, input_n);
or  i3 (output, input_1, input_2, …, input_n);
nor  i4 (output, input_1, input_2, …, input_n);
xor i5 (output, input_1, input_2, …, input_n);
xnor i6 (output, input_1, input_2, …, input_n);
```



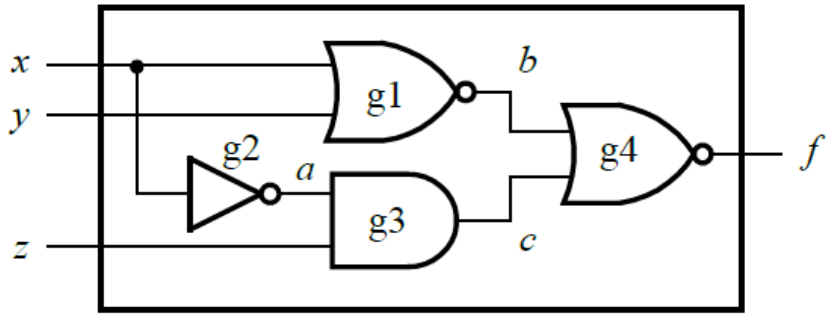
Verilog Gate Primitives

- To instantiate and/or gates

  gatename [*instance_name*](output, input1, input2, ..., input*n*);

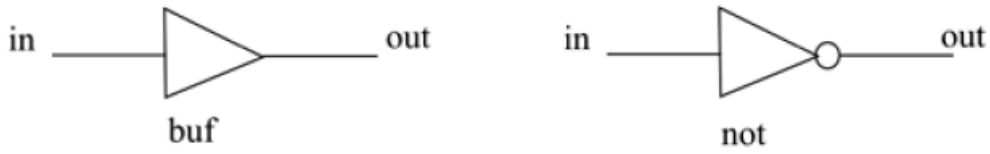
  - instance_name is optional.

```
module basic_gates (x, y, z, f) ;
input   x, y, z;
output f ;
wire a, b, c;  // internal nets
// Structural modeling using basic gates.
    nor  g1  (b, x, y);
    not  g2  (a, x);
    and g3  (c, a, z);
    nor g4  (f, b, c);
endmodule
```



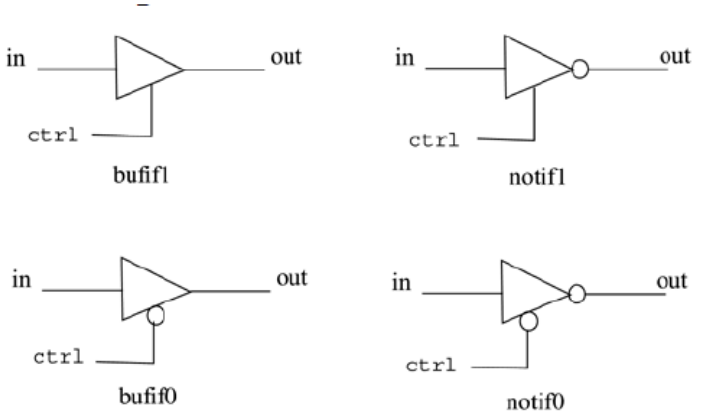
Verilog Primitives - Example

```
buf i1 (output_1, output_2, …, output_n, input);
not i2 (output_1, output_2, …, output_n, input);
```



Verilog Primitives - Example

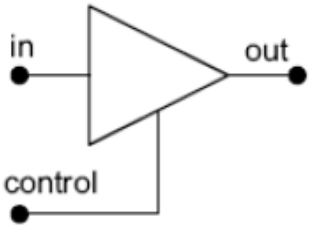
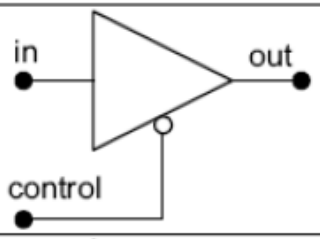
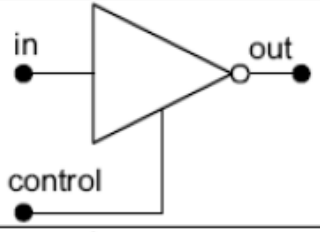
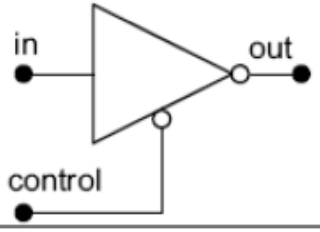
```
bufif0 i1 (output, data input, control input);
bufif1 i2 (output, data input, control input);
notif0 i3 (output, data input, control input);
notif1 i4 (output, data input, control input);
```



Verilog Gate Primitives Summary

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	**and** ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	**or** gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	**nand** gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	**nor** gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	**xor** gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	**xnor** gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	**buf** gb (o1, o2, ... i);	o1, o2, o3, . .	i
NOT	**not** gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Verilog Gate Primitives Summary

Typical instantiation	Functional representation	Functional description
bufif1 (out, in, control);		Out = in if control = 1; else out = z
bufif0 (out, in, control);		Out = in if control = 0; else out = z
notif1 (out, in, control);		Out = complement of in if control = 1; else out = z
notif0 (out, in, control);		Out = complement of in if control = 0; else out = z

Array of Instances of primitives

- The primitives available in Verilog can also be instantiated as arrays.
- A judicious use of such array instantiations often leads to compact design descriptions.
- A typical array instantiation has the form

  and gate [7 : 4] (a, b, c);

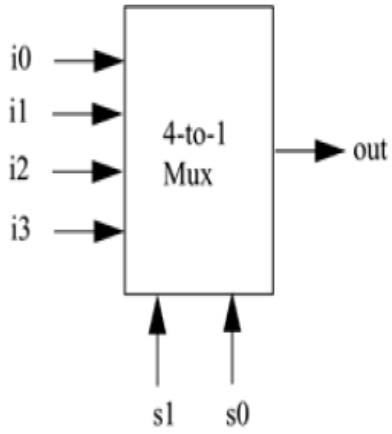
  where a, b, and c are to be 4 bit vectors.
- The above instantiation is equivalent to combining the following 4 instantiations:
- **and** gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);

Array of Instances

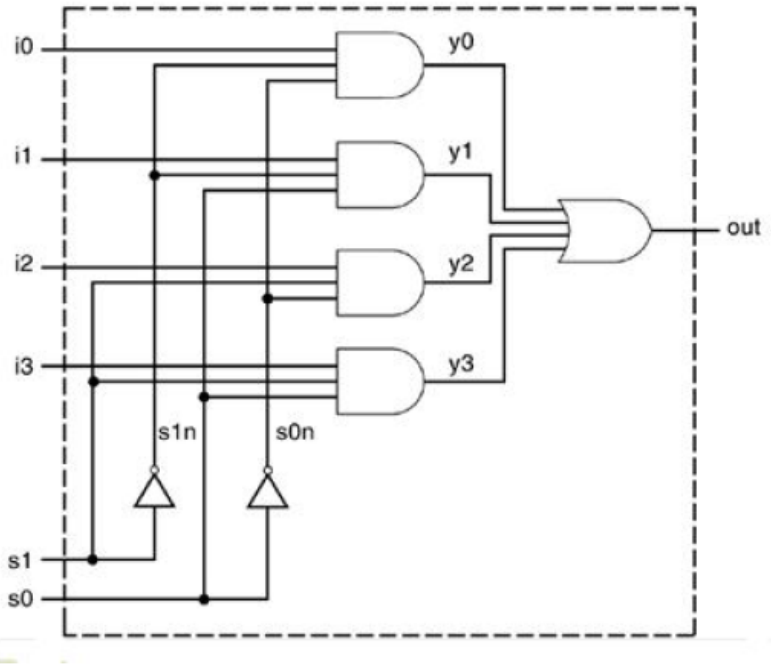
- Array instantiations may be a synthesizer dependent!
  - Suggestion: you had better to check this feature before using the synthesizer.

```
wire [3:0] out, in1, in2;
// basic array instantiations of nand gate.
nand n_gate[3:0] (out, in1, in2);

// this is equivalent to the following:
nand n_gate0 (out[0], in1[0], in2[0]);
nand n_gate1 (out[1], in1[1], in2[1]);
nand n_gate2 (out[2], in1[2], in2[2]);
nand n_gate3 (out[3], in1[3], in2[3]);
```

Example - Primitive Instantiation

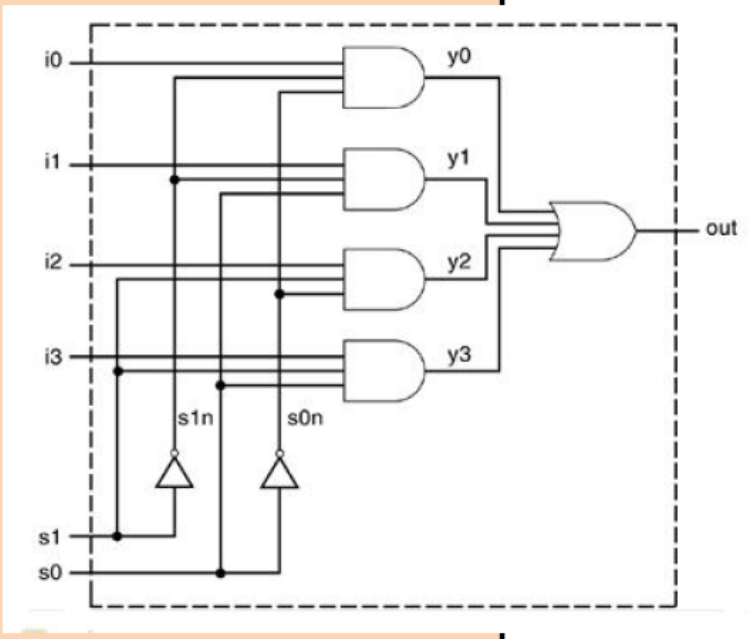


s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



Primitive Instantiation - Example

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
wire s1n, s0n;
wire y0, y1, y2, y3;
not (s1n, s1);
not (s0n, s0);
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);
endmodule
```



Module Instantiation - Example

```
module twmux (a,b,s,y);
input a,b,s;
output y;
wire y,s1,w1,w2;
not n1(s1,s);
and a1(w1,a,s);
and a2 (w2,b,s1);
or o1(y,w1,w2);
endmodule
```

```
module  frmux (a,b,c,d,se1,se2,y);
input a,b,c,d,se1,se2;
output y;
wire y,se1,se2,w1,w2;
twmux t1(a,b,se2,w1);
twmux t2(c,d,se1,w2);
twmux t3(w1,w2,se1,y);
endmodule
```

Connecting Ports to External Signals

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed.

- Connecting by ordered list
- Connecting ports by name

Connecting by ordered list

```
module twmux (a,b,s,y);
input a,b,s;
output y;
wire y,s1,w1,w2;
not n1(s1,s);
and a1(w1,a,s);
and a2 (w2,b,s1);
or o1(y,w1,w2);
endmodule
```

- Connecting by ordered list is the most intuitive method for most beginners.

```
module  frmux (a,b,c,d,se1,se2,y);
input a,b,c,d,se1,se2;
output y;
wire y,se1,se2,w1,w2;
twmux t1(a,b,se2,w1);
twmux t2(c,d,se1,w2);
twmux t3(w1,w2,se1,y);
endmodule
```

- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition

Connecting by Port list

❖ What if number of ports are higher?

```
module twmux (a,b,s,y);
input a,b,s;
output y;
wire y,s1,w1,w2;
not n1(s1,s);
and a1(w1,a,s);
and a2 (w2,b,s1);
or o1(y,w1,w2);
endmodule
```

```
module  frmux (a,b,c,d,se1,se2, y);
input a,b,c,d,se1,se2;
output y;
wire y, w1,w2;
twmux t1(.a(a), .b(b), .s(se1), .y(w1) );
twmux t2(.a (c), .b(d), .s(se1), .y(w2) );
twmux t3(.a(w1), .b(w2), .s(se2), .y(y));
endmodule
```

Verilog Switch Primitives

- Ideal switches – without a prefixed letter "r"
- Resistive switches – with a prefixed letter "r"

- MOS switches
  - nmos
  - pmos
  - cmos
- Bidirectional switches
  - tran
  - tranif0
  - tranif1
- Power and ground nets
  - supply1
  - supply0

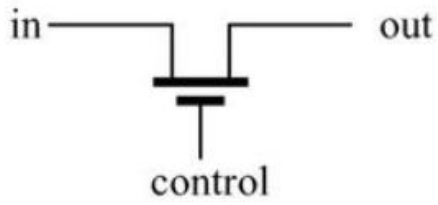
- Resistive switches
  - rnmos
  - rpmos
  - rcmos
- Resitive bidirectional switches
  - rtran
  - rtranif0
  - rtranif1
- Pullup and pulldown
  - pullup
  - pulldown

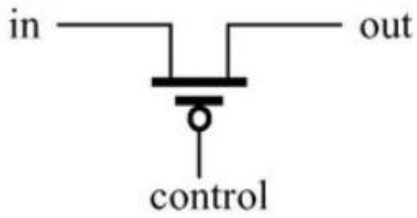
MOS Switch Primitives

- To instantiate switch elements

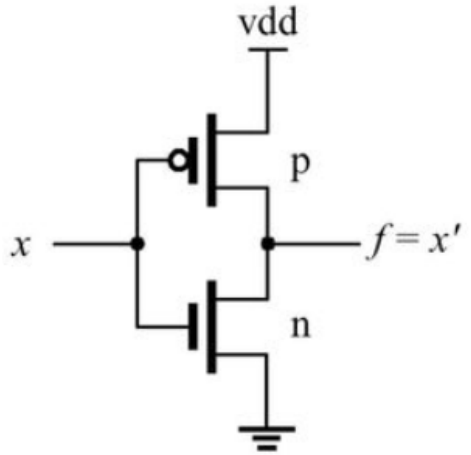
  switch_name [*instance_name*] (output, input, control);

  - The instance_name is optional

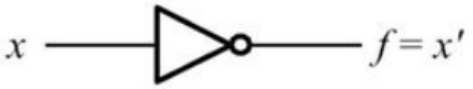




Example: CMOS Inverter



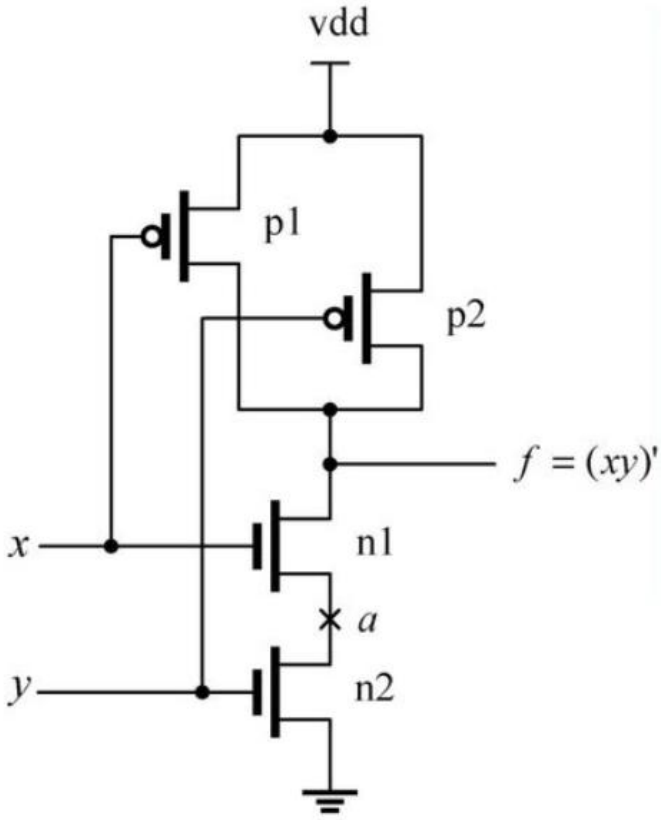
(a) Circuit



(b) Logic symbol

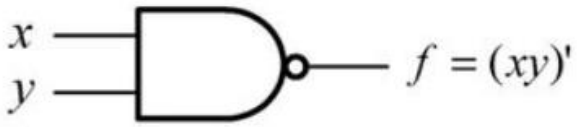
```
module mynot (input x, output f);
// internal declaration
supply1  vdd;
supply0  gnd;
// NOT gate body
    pmos p1 (f, vdd, x);
    nmos n1 (f, gnd, x);
endmodule
```

Example: CMOS NAND Gate



(a) Circuit

```
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
```

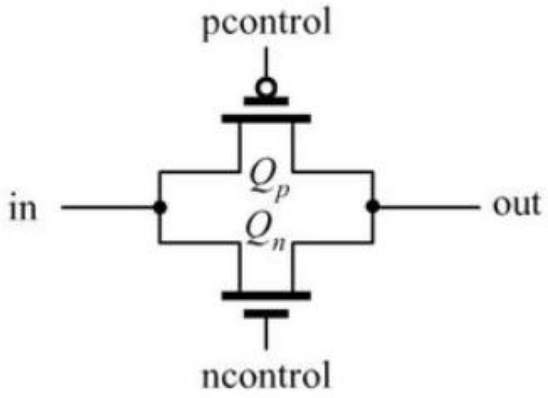


(b) Logic symbol

CMOS Switch

- To instantiate CMOS switches

  cmos [*instance_name*]
  (output, input, ncontrol, pcontrol);



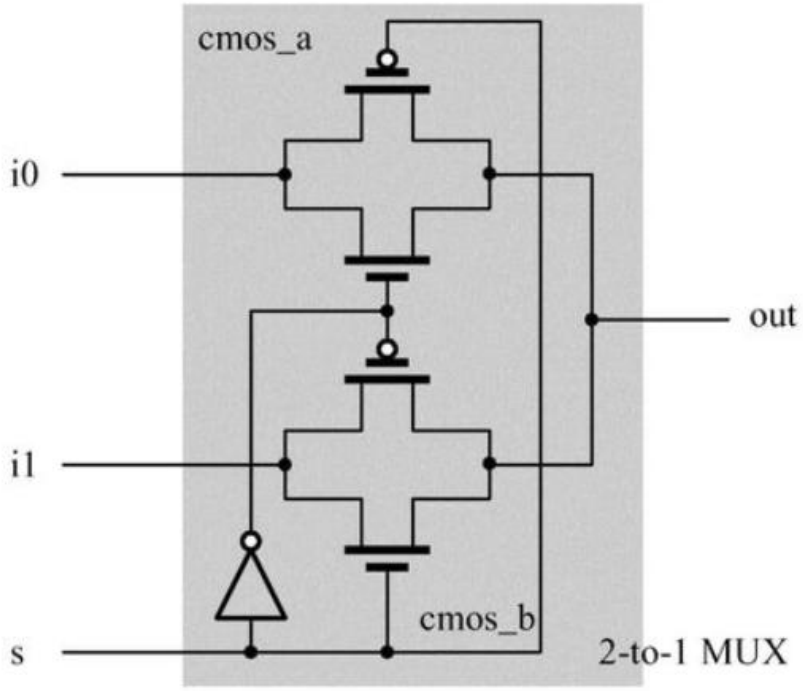
(a) Symbol

- The instance_name is optional

control		data			
n	p	0	1	x	z
0	0	0	1	x	z
0	1	z	z	z	z
0	x	L	H	x	z
0	z	L	H	x	z
1	0	0	1	x	z
1	1	0	1	x	z
1	x	0	1	x	z
1	z	0	1	x	z
x	0	0	1	x	z
x	1	L	H	x	z
x	x	L	H	x	z
x	z	L	H	x	z
z	0	0	1	x	z
z	1	L	H	x	z
z	x	L	H	x	z
z	z	L	H	x	z

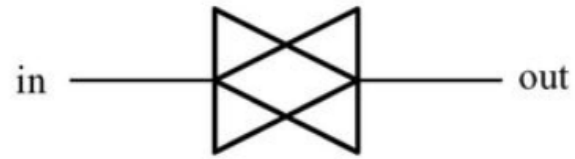
(b) Truth table

Example: CMOS 2:1 MUX

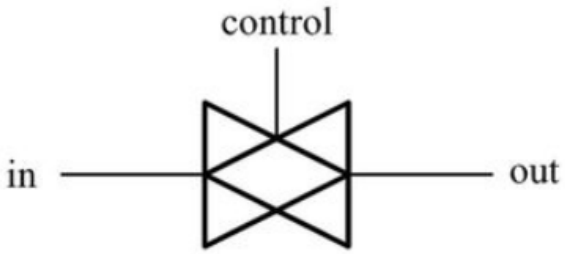



```
module my_mux (out, s, i0, i1);
output out;
input    s, i0, i1;
//internal wire
wire sbar; //complement of s

not (sbar, s);
//instantiate cmos switches
cmos (out, i0, sbar, s);
cmos (out, i1, s, sbar);
endmodule
```

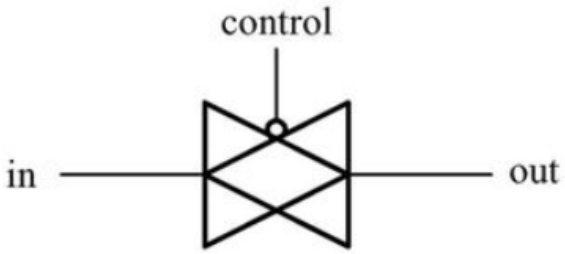
Bidirectional Switch



(a) tran (rtran)



(b) tranif1 (rtranif1)



(c) tranif0 (rtranif0)

- To instantiate bidirectional switches:

```
tran     [instance_name] (in, out);
tranif0  [instance_name] (in, out, control);
tranif1  [instance_name] (in, out, control);
```

  - instance_name is optional

Delay in Structural Style

Delay

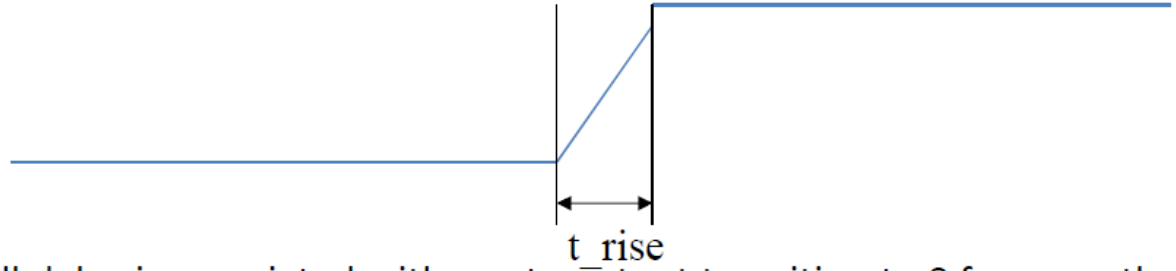
- Verilog can account for different types of propagation delays of circuit elements.
- Any connection can cause a delay due to the distributed nature of its resistance and capacitance.
- Due to the manufacturing tolerances, these can vary over a range in any given circuit. Similar delays are present in gates too.
- These manifest as propagation delays in the 0 to 1 transitions and 1 to 0 transitions from input to the output.
- Sometimes manufacturers adjust input and output impedances of circuit elements to specific levels and exploit them to reduce interface hardware.
- A variety of such delays can be accommodated in Verilog.

Delay

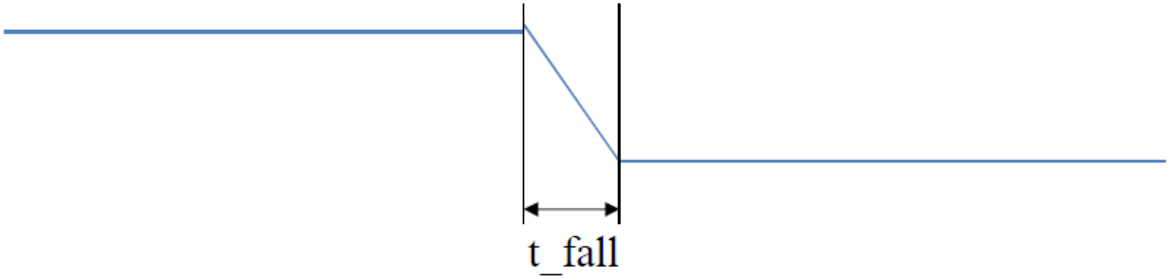
- Inertial delay model
  - The signal events do not persist long enough will not be propagated to the output.
  - It is used to model gate delays (RC delays).
  - It is the default delay model for HDL (Verilog HDL and VHDL).
- Transport delay model
  - Any signal events will be propagated to the output.
  - It is used to model net (i.e. wires) delays.
  - The default delay of a net is zero.

Gate Delays

- Rise delay is associated with a gate output transition to 1 from another value



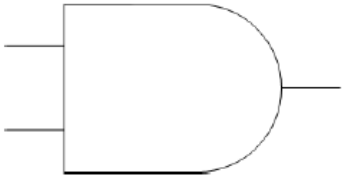
- Fall delay is associated with a gate output transition to 0 from another value



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

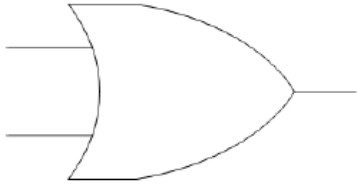
Inertial Delay

Gate delay specifications



- One delay specification: If specified, it is used for all transitions.

```
and #(delay time) a1 (out, i1, i2);
and #(4)  a1 (out, i1, i2);
```

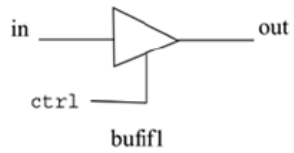


- Two delay specification: If specified, they refer to rise and fall times.

```
or #(rise_del, fall_del) o1 (out, i1, i2);
or #(5, 6)  o1 (out, i1, i2);
```

- Three delay specification: If specified, they refer to rise, fall and turn-off times.

```
bufif1 #(rise_del, fall_del, turn_off_del) b1 (out, in, cnt);
bufif1 #(2, 3, 5)  b1 (out, in, crtl);
```



Important Point: Delay

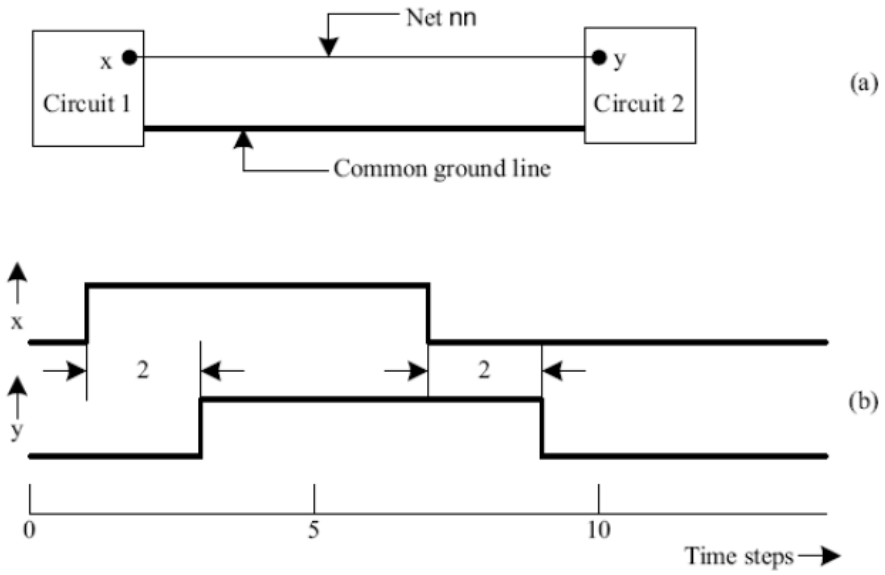
- Delays and storage times can be specified on the gate primitives and the nets but **not on regs**.
- All 3-delay values are separately specified in the most versatile case.
- If only **two time-values** are specified, Verilog interprets these as the rise and fall time, respectively. The turn-off time (delay) is taken as the smaller of these two.
- If only **1-delay** value is specified, it is taken as the rise time, the fall time, and the turn-off time.
- If **no time value** is specified, the rise and fall times at the output are taken as zero. The turn-off is taken as instantaneous.

Net Delay

One of the simplest delays is that of a direct connection – a net. It can be part of the declaration statement

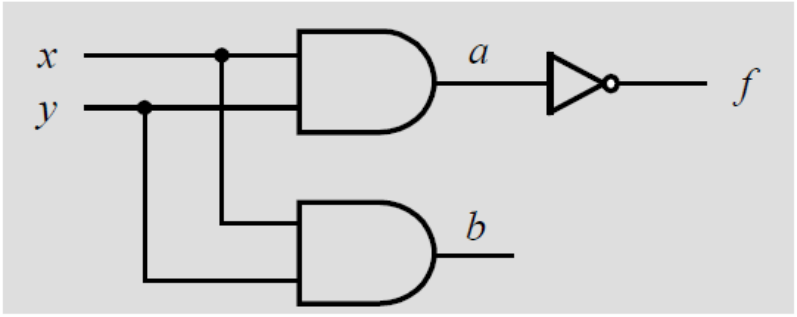
wire #2 nn; // nn is declared as a net with a propagation delay of 2 time steps

wire # (2, 1) nm; //nm is declared as net with rise delay of 2t.u and fall delay of 1t.u

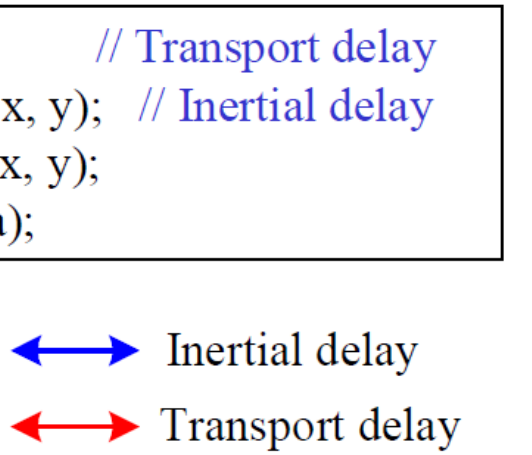


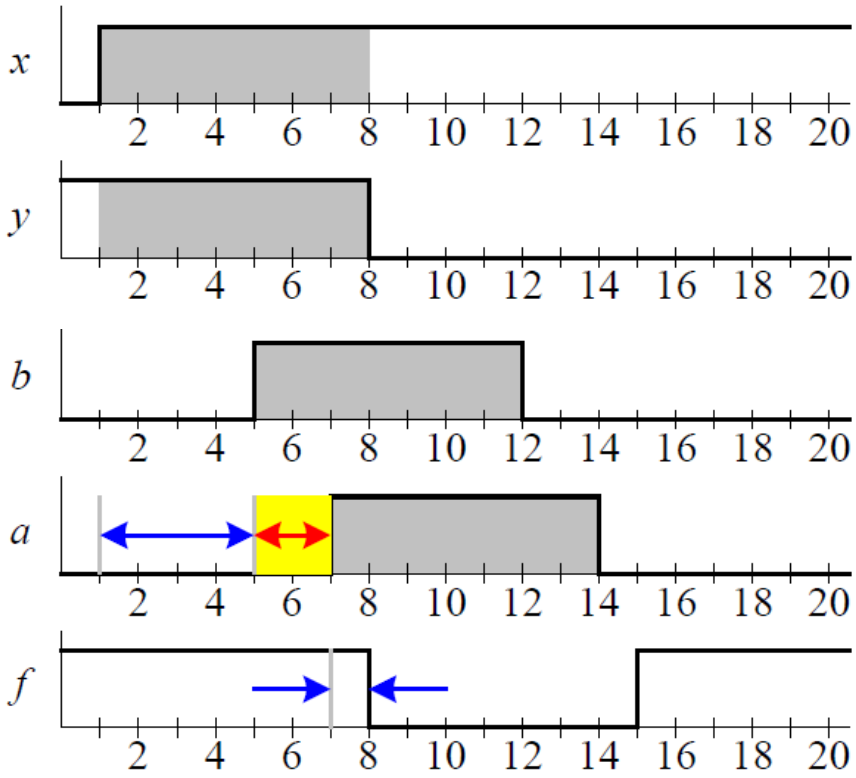
Transport Delay

Transport and Inertial Delays




```
wire #2 a;          // Transport delay
and  #4 (b, x, y);  // Inertial delay
and  #4 (a, x, y);
not  #1 (f, a);
```





Lecture 6

Module Modeling Style

- Structural Style
- **DataFlow Style**
- Behavioral Style
- Mixed Style

Data Flow Style

- Gate level Modeling approach works well for small circuits but not for complex designs.
- Designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than the 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 the instantiation of individual gates.

Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called **logic synthesis**.

Data Flow Style

- This level of abstraction level resembles that of the Boolean equation representation of digital systems.
- The dataflow assignments are called “Continuous Assignments”.
- Continuous assignments are always active.
- **Syntax: assign #(delay) target = expression;**
- A Verilog module can contain any number of continuous assignment statements, and all statements execute concurrently.

LHS target -> always a net, not a register.
RHS -> registers, nets or function calls.

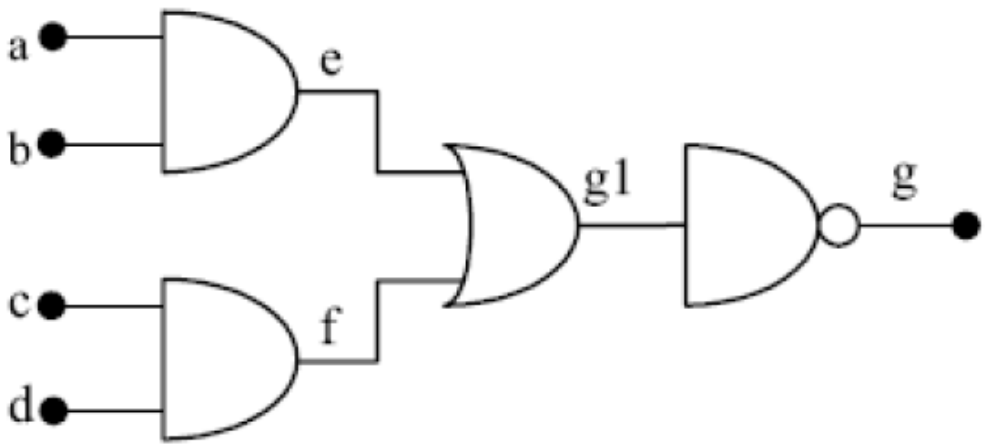
Data Flow Style

- Two basic forms of assignments
  - Continuous assignment: assign values to nets
  - Procedural assignment: assign values to variables
- Two additional forms of assignments: procedural continuous assignments
  - assign/deassign
  - force/release
- An assignment consists of two parts: a LHS and a RHS separated by = or <=
  - RHS: any expression that evaluates to a value to which the LHS value is to be assigned.

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Equality Operators	==, !=, ===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^, ~^
Unary Reduction	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	?:
Concatenations	{}

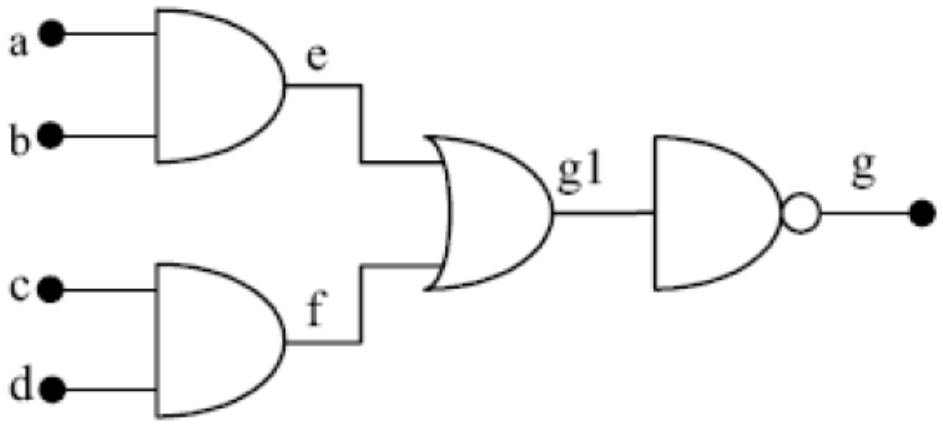
Example: Data Flow Style




```
module aoi2(g,a,b,c,d);
output g;
input a,b,c,d;
wire e,f,g1,g;
assign e = a & b;
assign f = c & d;
assign g1 = e | f;
assign g = ~g1;
endmodule
```

Using Multiple Assign Statements

Example: Data Flow Style



Using Multiple Assign Statements

```
module aoi2(g,a,b,c,d);
output g;
input a,b,c,d;
wire e,f,g1,g;
assign e = a & b;
assign f = c & d;
assign g1 = e | f;
assign g = ~g1;
endmodule
```

```
module aoi2(g,a,b,c,d);
output g;
input a,b,c,d;
wire e,f,g1,g;
assign e = a & b,f = c & d, g1 = e|f, g=~g1;
endmodule
```

Using Single Assign Statements

Concatenation of Vectors

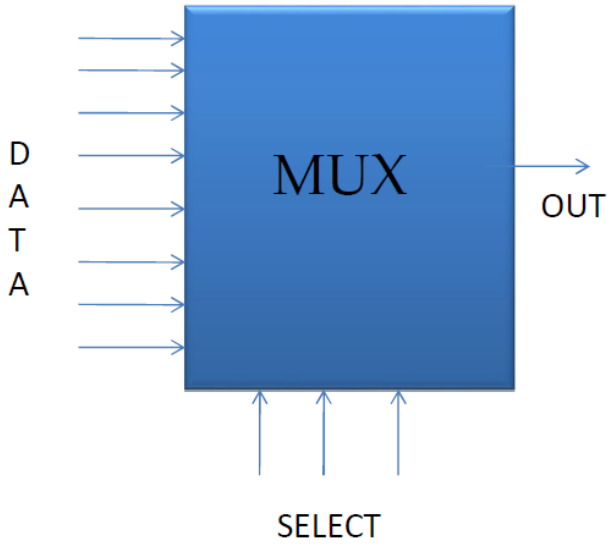
- One can concatenate vectors, scalars, and part vectors to form other vectors.
- The concatenated vector is enclosed within braces.
- Commas separate the components –scalars, vectors, and part vectors.

Example: Full Adder

```
assign {cout, sum} = a + b + cin;
```

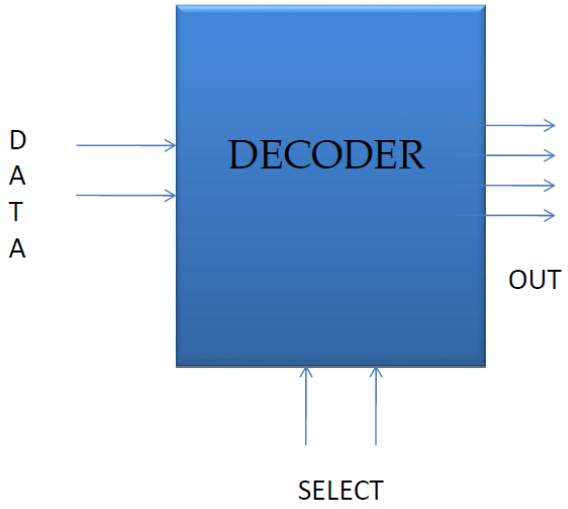
```
module add_8_c(c,cco,a,b,cci);
input[7:0]a,b;
output[7:0]c;
input cci;
output cco;
assign {cco,c} = (a + b + cci);
endmodule
```

Example Concatenation: Data Flow Style



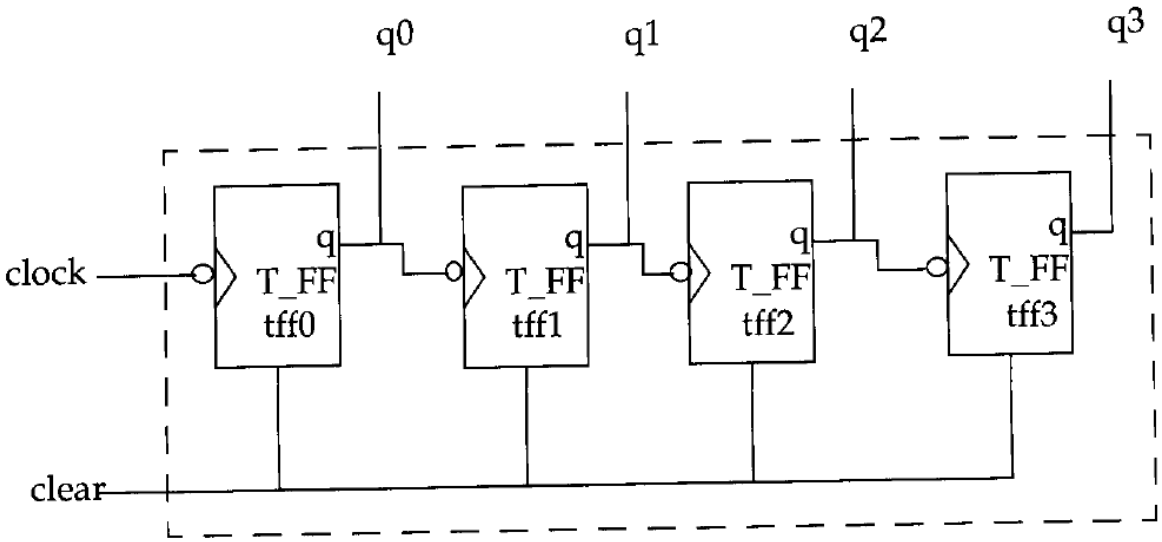

```
module generate_mux
   (data,select,out);
input [0:7] data;
input [0:2]select;
output out;
wire out;
assign out = data[select];
endmodule
```

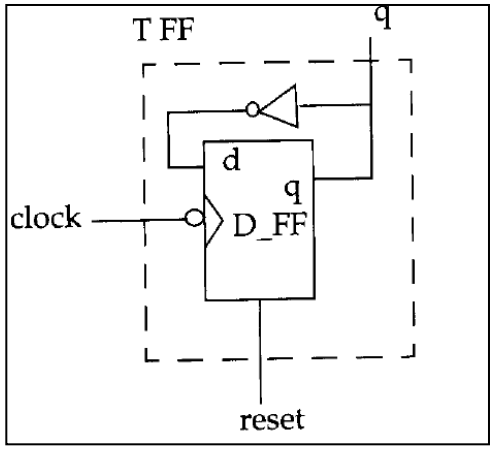
```
module generate decoder
   (data,select,out);
input  data;
input [0:1]select;
output [0:3] out;
wire [0:3] out;
assign out[select]=data;
endmodule
```



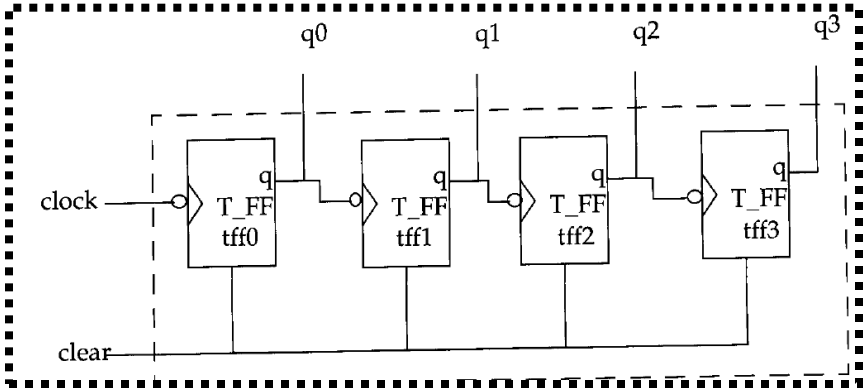
Example : Data Flow Style

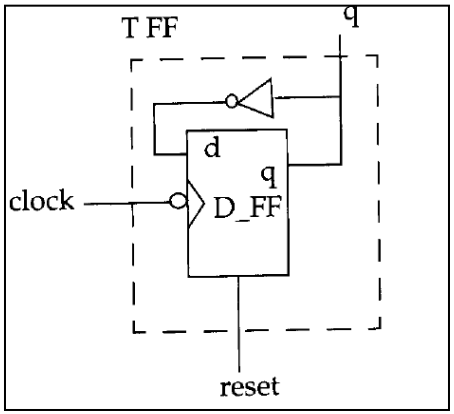
Design a 4-bit ripple counter using a negative edge-triggered D flip-flop considering dataflow modeling.

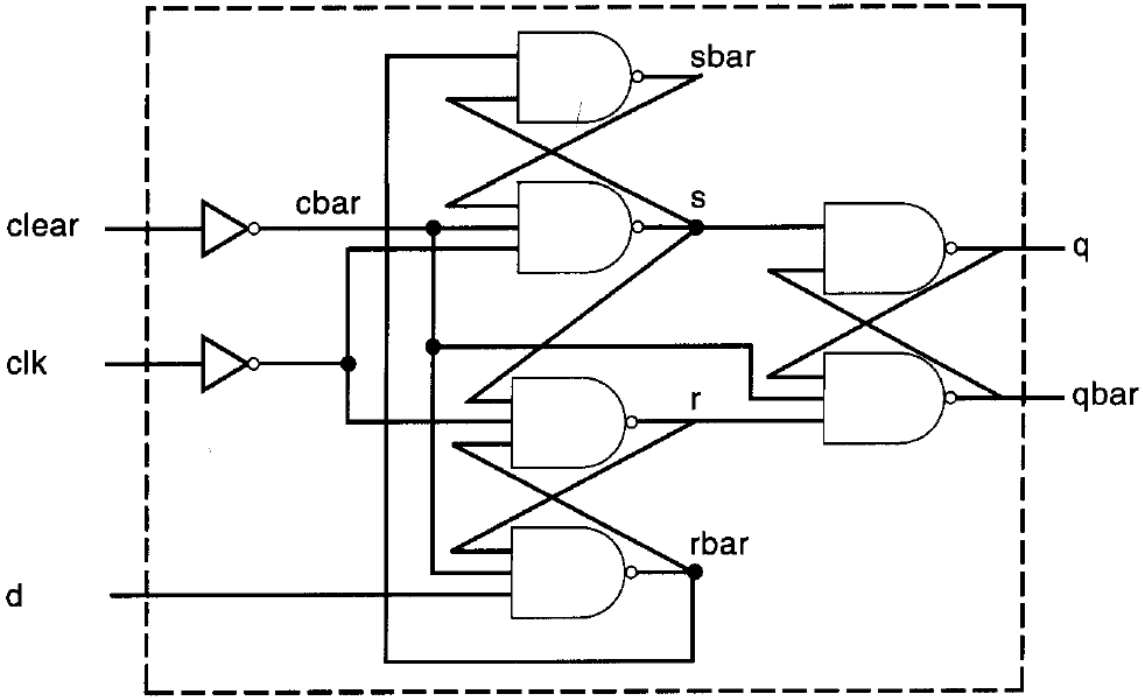




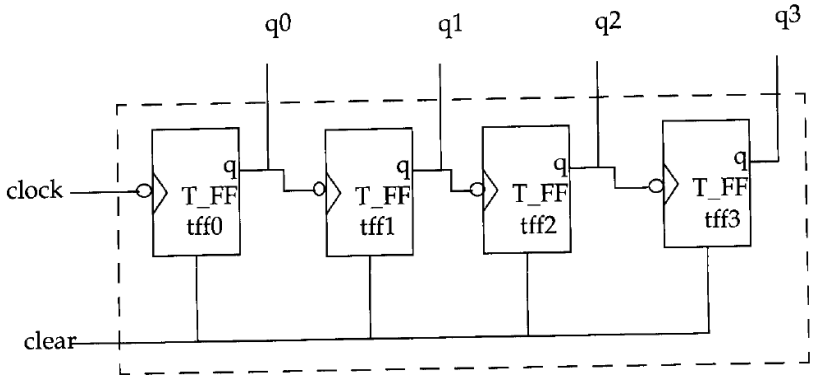
Example : Data Flow Style





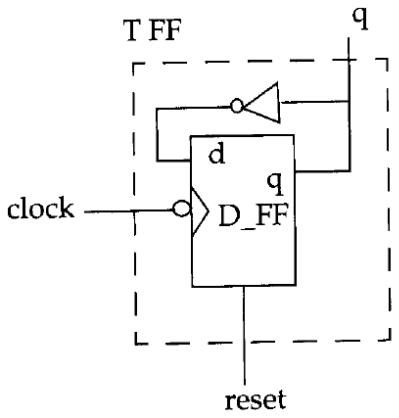


Example : Data Flow Style



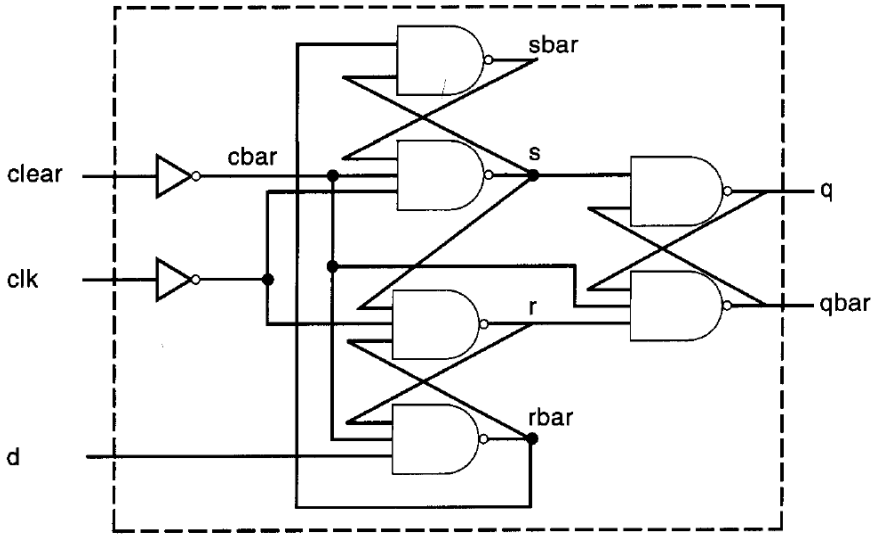

```
// Ripple counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```

```
module T_FF(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);

endmodule
```



Example : Data Flow Style

```
module T_FF(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);

endmodule
```




```
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q,qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar,cbar;

// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;

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
```

Example : Data Flow Style

```
// Ripple counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```

```
module T_FF(q, clk, clear);

// I/O ports
output q;
input clk, clear;

// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);

endmodule
```

```
module edge_dff(q, qbar, d, clk, clear);

// Inputs and outputs
output q,qbar;
input d, clk, clear;

// Internal variables
wire s, sbar, r, rbar,cbar;

// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;

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
```

Module Modeling Style

- Structural Style
- DataFlow Style
- **Behavioral Style**
- Mixed Style

Behavioral Style

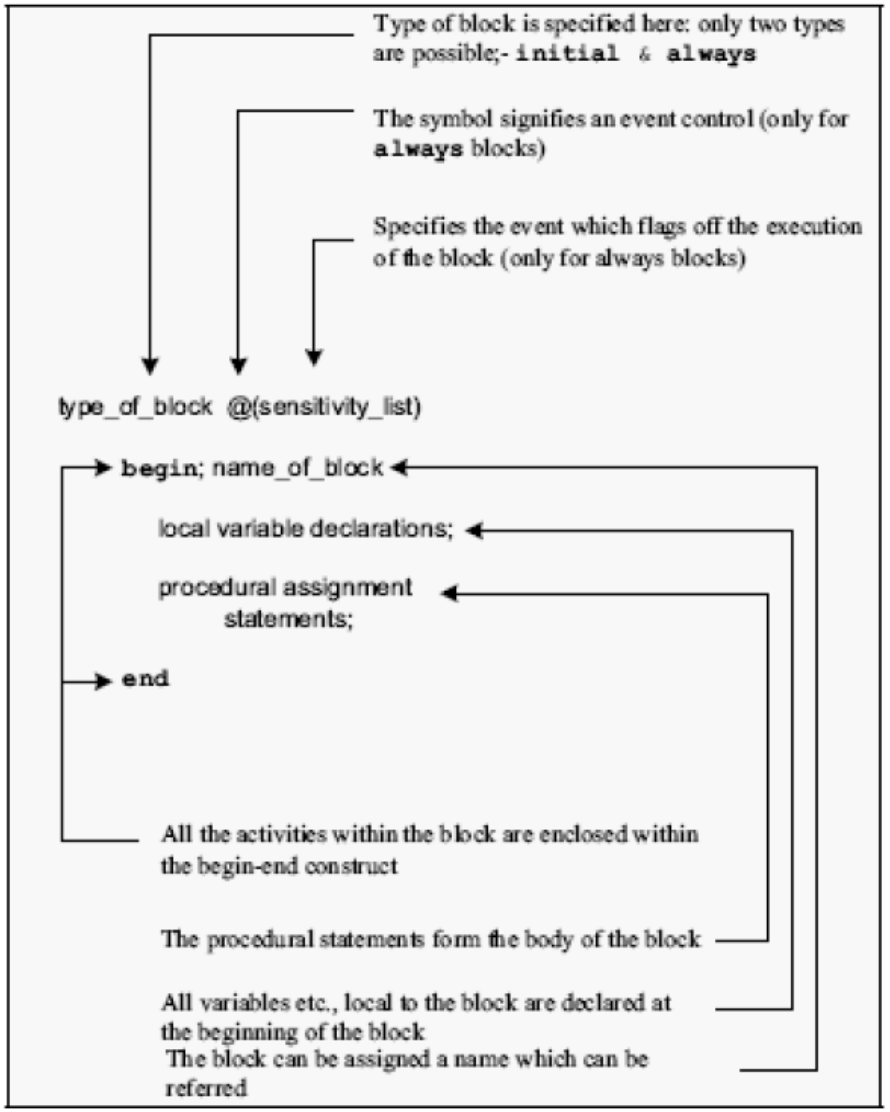
- This is the highest level of abstraction provided by Verilog HDL.
- A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details.
- Designing at this level is very similar to C programming.

Behavioral Style

- There are two structured procedures in Verilog:
  - **initial**
  - **always**
- **Concurrent execution** is observed in these procedures.
- Sequential / concurrent execution can be realized within these procedures.
- **Only registers** can be assigned in these procedures.
- The assignments in these procedures are called “procedural assignments”.

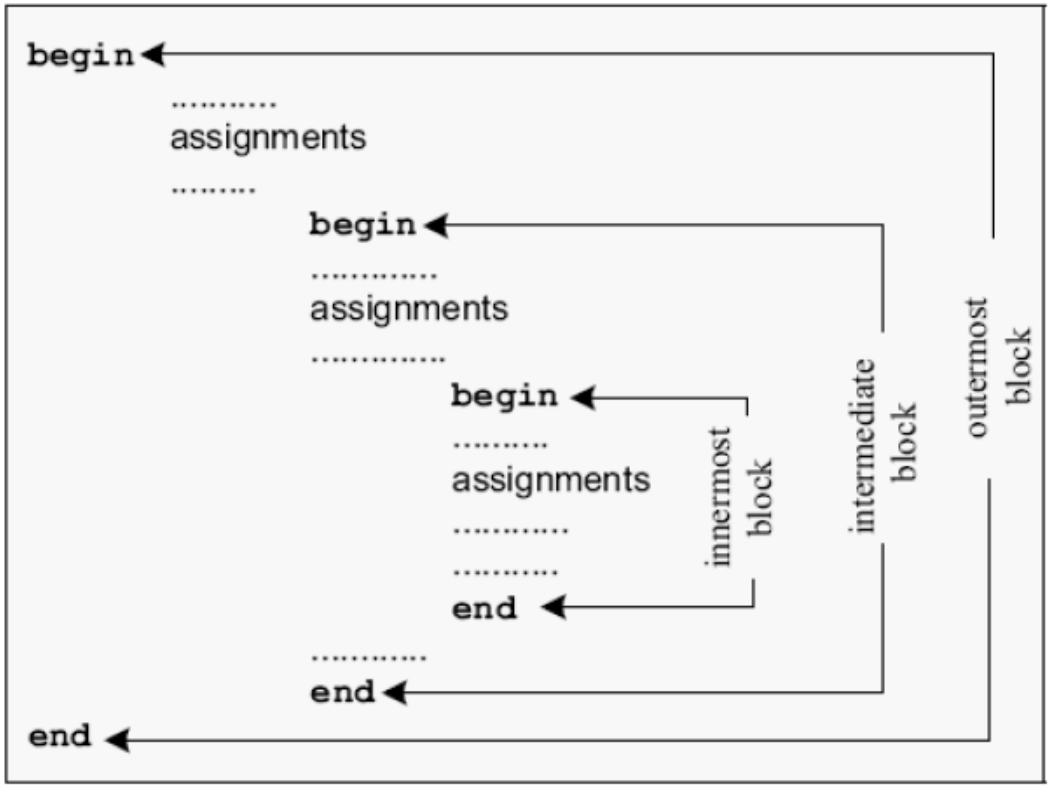
Behavioral Style

Procedural Block



Behavioral Style

Nesting of begin-end block



Behavioral Style

- Starts execution at ‘0’ simulation time and executes only once during the entire simulation.
- Multiple statements in initial block can be grouped with (**begin** & **end**) or (**fork** & **join**) keywords.
- These blocks are not synthesizable.
- **“initial”** blocks cannot be nested.
- Each initial block represent a separate and independent activity.
- “initial” blocks are used in generating test benches.

Behavioral Style

- A module can have as many **initial blocks as desired.**
- **All of them are** activated at the start of simulation.
- The time delays specified in one **initial** block are exclusive of those in any other block.

```
module nil1;
reg a, b;
initial
begin
a = 1'b0; b = 1'b0;
#2 a = 1'b1; #3 b = 1'b1; #1 a = 1'b0;
end
initial
begin
$monitor ($time, "monitor: a = %b, b = %b", a, b);
end
initial
begin
#100$stop;
end
endmodule
```

Behavioral Style

initial block - Example

```
initial
 xor_out = in1 ^ in2;
```

```
initial
begin
   and_out = a_in & b_in;
end
```

```
initial
begin
     enable = 1'b0;
     rst    = 1'b0;
     #100   rst = 1'b1;
     #20 enable = 1'b1;
end
```

```
initial
begin
    clk = 1'b0;
    reset   = 1'b0;
 initial
 begin
    #100  reset = 1'b1;
    #20 clk = 1'b1;
    end
    end
```

Behavioral Style

always statement

- Starts execution at '0' simulation time and is active all through out the entire simulation.
- Multiple statements inside always block can be grouped with (**begin** & **end**) or (**fork** & **join**) keywords.
- Execution of always blocks is controlled by using the timing control.
- always blocks cannot be nested.

Behavioral Style

- An always block without any sensitivity control will create an infinite loop and execute forever.
- Each always block represent a separate and independent activity.
- These blocks can synthesize to different hardware depending on their usage.
- always block with timing control are synthesizable.

Behavioral Style

always block – Example

```
always
   xor_out = in1 ^ in2;
```

```
always @(a_in or b_in)
  begin
    and_out = a_in & b_in;
  end
```

```
always @(posedge reset)
  begin
    if (reset == 1'b1)
    q_out =  1'b0;
    else
    q_out =  d_in;
  end
```

```
always
  begin
     cnt = 1'b0;
     reset   = 1'b0;
    always @(posedge clk)
     begin
       #100  cnt = 1'b1;
       #20 enable = 1'b1;
     end
 end
```

Behavioral Style

always block- example

```
module regis (A,B,Y);
output  Y;
input   A,B;
reg  Y;
always @(A,B)
begin
if (A==1 && B==1)
Y=1'b1;
else
Y=1'b0;
end
endmodule
```

```
module dff (clk,
rst,din,dout);
input clk, rst, din;
output dout;
reg dout;
always @(posedge clk)
begin
if (rst)
dout = 1'b0;
else
dout=din;
end
endmodule
```

if-else statements

- The if construct checks a specific condition and decides execution based on the result.

```
assignment 1;
if(condition)
begin
assignment2;
end
assignment3;
assignment4;
```

- After execution of **assignment1**, the condition specified is checked. If it is satisfied , **assignment2** is executed; if not it is skipped.
- In either case the execution continues through **assignment3, assignment4**, etc.
- Execution of **assignment2** alone is dependent on the condition. The rest of the sequence remains.

Behavioral Style

if-else and if-elseif-else statements

```
if (<expression>)
begin
   true_statements;
end
```

```
if (<expression>)
begin
    true_statements;
end
else
begin
    false statements;
end
```

```
if (<expression 1>)
begin
    true_statements 1;
end
                :
                :
else if (<expression
n>)
begin
    true_statements n;
end
else
begin
    default_statements;
end
```

Behavioral Style

Conditional statement - Examples

```
if (!ena == 1'b1)
begin
   out = in1 &
   in2;
end
```

```
if (sel == 1'b0)
begin
   mux_out = in0;
end
else
begin
   mux_out = in1;
end
```

```
if (in1 == 1'b0 && in2 == 1'b0)
dec_o1 = 1'b1;

else if (in1 == 1'b0 && in2 == 1'b1)
dec_o2 = 1'b1;

else if (in1 == 1'b1 && in2 == 1'b0)
dec_o3 = 1'b1;

else
dec_o4 = 1'b1;
```

Behavioral Style

case statement

- This is a multiway decision statement that tests whether an expression matches one of a number of other alternatives.
- Doesn't treat 'x' & 'z' as don't cares.

```
case (expression)
alternative 1:begin
  end
alternative 2:begin
  end
alternative 3:begin
  end
default: begin
  end
endcase
```

Behavioral Style

case statement - Example

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0); output
   out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

- **casez**: same as **case** statement but treats ‘z’ as don’t cares.
- All bit positions with ‘z’ can also be represented by ‘?’

Behavioral Style

casez - Examples

```
module casez_ex(counter,cmd);
input[0:3] counter;
output cmd;
reg cmd;
always@(counter)
casez(counter)
4'b???1:cmd=0;
4'b??10:cmd=1;
4'b?100:cmd=2;
4'b1000:cmd=3;
default : cmd=0;
endcase
endmodule
```

Behavioral Style

casex - Example

```
module casex_ex(sel,Bitposition);
input [5:1] sel;
output [2:0] Bitposition;
reg [2:0] Bitposition;
always @ (sel)
casex(sel)
5'bxxxx1 : Bitposition=1 ;
5'bxxx1x : Bitposition=2;
5'bxx1xx  : Bitposition=3;
5'bx1xxx : Bitposition=4;
5'b1xxxx : Bitposition=5;
default : Bitposition=0;
endcase
endmodule
```

Write a sequential Verilog code for an 8-to-3 priority encoder with active high enable input

Behavioral Style

Looping statements

- Verilog supports four types of loops:

– **while** loop

– **for** loop

– **forever** loop

– **repeat** loop

- Many Verilog synthesizers supports only ‘for’ loop for synthesis:

– Loop bound must evaluate to a constant.

– Implemented by unrolling the ‘for’ loop, andreplicating the statements.

Behavioral Style

for loop construct

The for loop in Verilog is quite similar to the for loop in C
Format:

```
for (initialization; expression; inr/dcr)
   begin
      statements
   end
```

The sequence of execution as follows

1. Execute **initialization**
2. Evaluate **expression**
3. If the **expression** evaluates to the true state(1), carry out **statements.** Go to step 5.
4. If the **expression** evaluates to false state (0), exit the loop.
5. Execute **incr/dcr**. Go to step 2.

Behavioral Style

Implement 8-bit Adder using For Loop

```
module addfor (s,co,a,b,cin,en);
output [7:0] s;
output co;
input [7:0] a,b;
input en,cin;
reg [8:0] c;
reg co;
reg [7:0] s;
integer i;
always @ (posedge en)
begin
c[0] = cin;
```

```
for (i=0; i<=7; i=i+1)
begin
{c[i+1],s[i]} = (a[i]
+ b[i] + c[i]);
end
co=c[8]
end
endmodule
```

Behavioral Style

repeat construct

- The repeat construct is used to repeat specified block a specified number of times.
- The format is show below , the quantity 'a ' can be a number or an expression evaluated to a number.
- As soon as the repeat statement is encountered, a is evaluated, the block will be executed 'a' times.
- If 'a' evaluates to 0 or x or z, the block is not executed.

```
Format:
repeat (a)
    begin
        statements
    end
```

Behavioral Style

While loop

The ***format*** of while loop is shown below.

- The **expression** is evaluated. If it is ***true*** , the **statements** executed and **expression** evaluated and checked again.
- If the expression evaluates to ***false***, the loop is terminated and the following statement is taken for execution.
- If the expression evaluates to ***true***, execution of **statement** is repeated.
- The loop is terminated and broken only if the expression evaluates to false.

Format:

```
while (expression)
   begin
      statements
   end
```

Behavioral Style

forever loop

Repeated execution of a block in an endless manner is best done with the **forever** loop (compare with repeat where the repetition is for a fixed number of times)

```
module clkgen (clk);
output clk;
reg clk;
initial
begin
    clk=1'b0;
    forever #10 clk=~clk;
end
initial
#1000 $finish;

endmodule
```

Format:

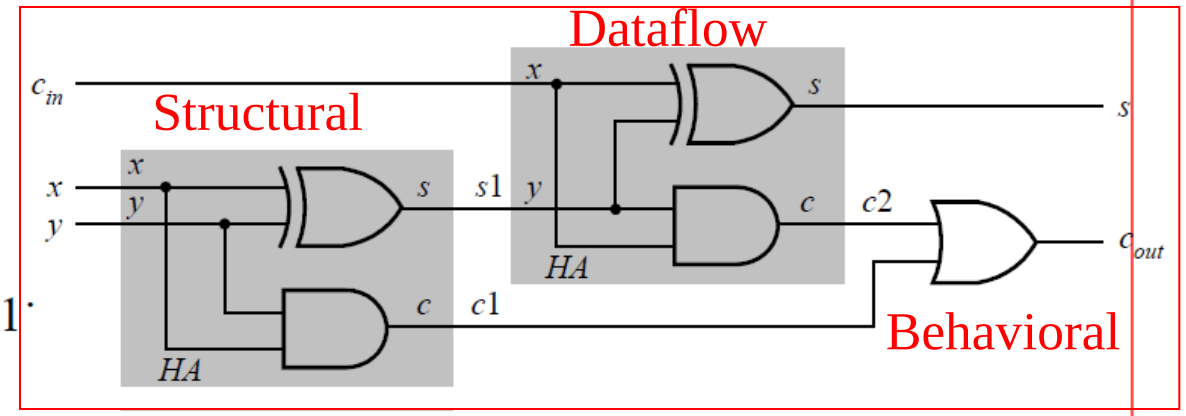
```
forever
  begin
      statements;
  end
```

Structural vs Behavioral HDL Constructs

- *Structural* constructs specify actual hardware structures
  - Low-level, direct correspondence to hardware
    - Primitive gates (e.g., and, or, not)
    - Hierarchical structures via modules
  - Analogous to programming software in assembly
- *Behavioral* constructs specify an operation on bits
  - High-level, more abstract
    - Specified via equations, e.g., out = (a & b) | c
- Not all behavioral constructs are synthesizable
  - We've already talked about the pitfalls of trying to "program"
  - But even some combinational logic won't synthesize well
  - out = a % b // modulo operation – what does this synthesize to?
  - We will *not* use: + - * / % > >= < <= >> <<

Mixed-signal Modeling Styles

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input x, y, c_in;
output s, c_out;
reg c_out;
wire s1, c1, c2;
// structural modeling of HA 1.
xor xor_ha1 (s1, x, y);
and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
assign s = c_in ^ s1;
assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(c1, c2) // can also use always @(*)
c_out = c1 | c2;
endmodule
```



Examples: Modeling Styles

AND Gate:

Structural Model	Data Flow Model	BehaviouralModel

moduleandstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
and g1(z,x,y);
 	
endmodule
	moduleanddf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z=(x&y);
	
	endmodule
		module andbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=x&y;
		
		endmodule

Examples: Modeling Styles

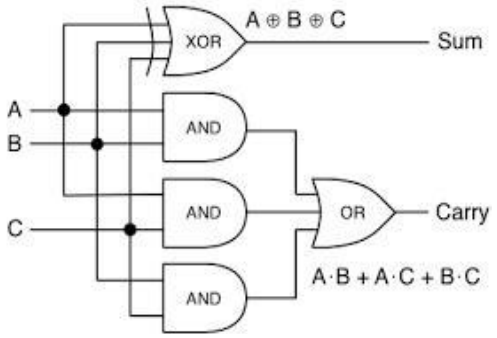
XNOR Gate:

Structural Model	Data Flow Model	BehaviouralModel

modulexnorstr(x,y,z);
 	
inputx,y;
 	
output z;
 	
xnor g1(z,x,y);
 	
endmodule
	modulexnordf(x,y,z);
	
	inputx,y;
	
	output z;
	
	assign z= !(x^y);
	
	endmodule
		module xnorbeh(x,y,z);
		
		input x,y;
		
		output z;
		
		reg z;
		
		always @(x,y)
		
		z=!(x^y);
		
		endmodule

Examples: Modeling Styles

FULL ADDER:



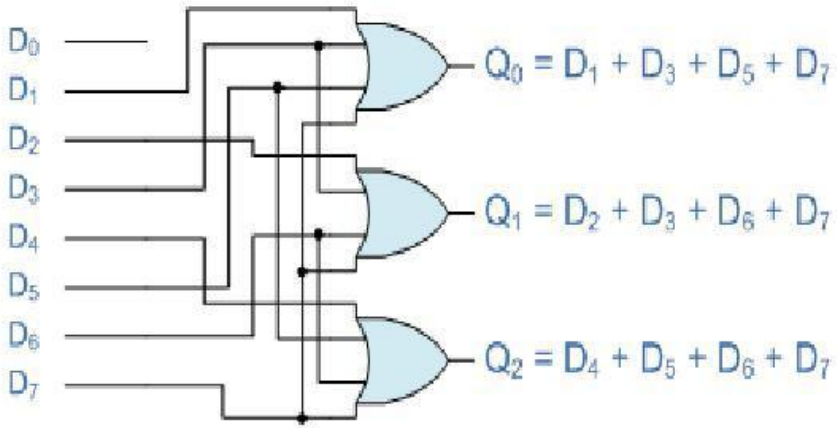
Structural model	Dataflow model	Behaviouralmodel

module fulladdstr(sum,carry,a,b,c);
 	
outputsum,carry;
 	
inputa,b,c;
 	
xor g1(sum,a,b,c);
 	
and g2(x,a,b);
 	
and g3(y,b,c);
 	
and g4(z,c,a);
 	
or g5(carry,x,z,y);
 	
endmodule
	modulefulladddf(sum,carry,a,b,c);
	
	outputsum,carry;
	
	inputa,b,c;
	
	assign sum = a ^ b^c;
	
	assign carry=(a&b) | (b&c) | (c&a);
	
	endmodule
		modulefulladdbeh(sum,carry,a,b,c);
		
		outputsum,carry;
		
		inputa,b,c;
		
		regsum,carry;
		
		always @ (a,b,c)
		
		sum = a ^ b^c;
		
		carry=(a&b) | (b&c) | (c&a);
		
		endmodule

Examples: Modeling Styles

8:3 Encoder:

Structural Model	Data Flow Model	BehaviouralModel
Module enc83str(d0,d1,d2,d3,d4,d5,d6,d7,q0,q1,q2); Input d0,d1,d2,d3,d4,d5,d6,d7; Output q0,q1,q2; Or g1(q0,d1,d3,d5,d7); Or g2(q1,d2,d3,d6,d7); Or g3(q2,d4,d5,d6,d7); Endmodule	Module enc83df(d0,d1,d2,d3,d4,d5,d6,d7,q0,q1,q2); Input d0,d1,d2,d3,d4,d5,d6,d7 Output q0,q1,q2; Assign q0=d1|d3|d5|d7; Assign q1=d2|d3|d6|d7; Assign q2=d4|d5|d6|d7; Endmodule	module enc83beh (din,a,b,c); input [0:7]din; outputa,b,c; rega,b,c; always@(din) case(din) 8'b10000000:begin a=1'b0;b=1'b0,c=1'b0;end 8'b01000000:begin a=1'b0;b=1'b0;c=1'b1;end 8'b00100000:begin a=1'b0;b=1'b1;c=1'b0;end 8'b00010000:begin a=1'b0;b=1'b1;c=1'b1;end 8'b10001000:begin a=1'b1;b=1'b0,c=1'b0;end 8'b10000100:begin a=1'b1;b=1'b0,c=1'b1;end 8'b10000010:begin a=1'b1;b=1'b1,c=1'b0;end 8'b10000001:begin a=1'b1;b=1'b1,c=1'b1;end endcase endmodule



Task and Functions

Task and Function

- Repeatedly used functionality/logic makes the description verbose.
- Commonly used logic can be separated and can be implemented with task/function.
- Task and function are used to abstract verilog code that is used in many places in the design.
- Task & function provide the ability to execute common logic/functionality from several different places in a description.

Task and Function

- Tasks and functions provide the ability to reuse the common piece of code from several different places in a design.
- Comparison of tasks and functions

Item	Tasks	Functions
Arguments	May have zero or more *input*, *output,* and *inout*.	At least one *input*, and cannot have *output* and *inout*.
Return value	May have multiple values via *output* and *inout*.	Only a single value via function name.
Timing control statements	Yes	No
Execution	In non-zero simulation time.	In 0 simulation time.
Invoke functions/tasks	Functions and tasks.	Functions only.

When to use task ?

- Use task when:
  - there are delay, timing, or event control constructs in the procedure.

OR

  - the procedure has zero or more than one output arguments.

OR

  - the procedure has no input arguments.

When to use function ?

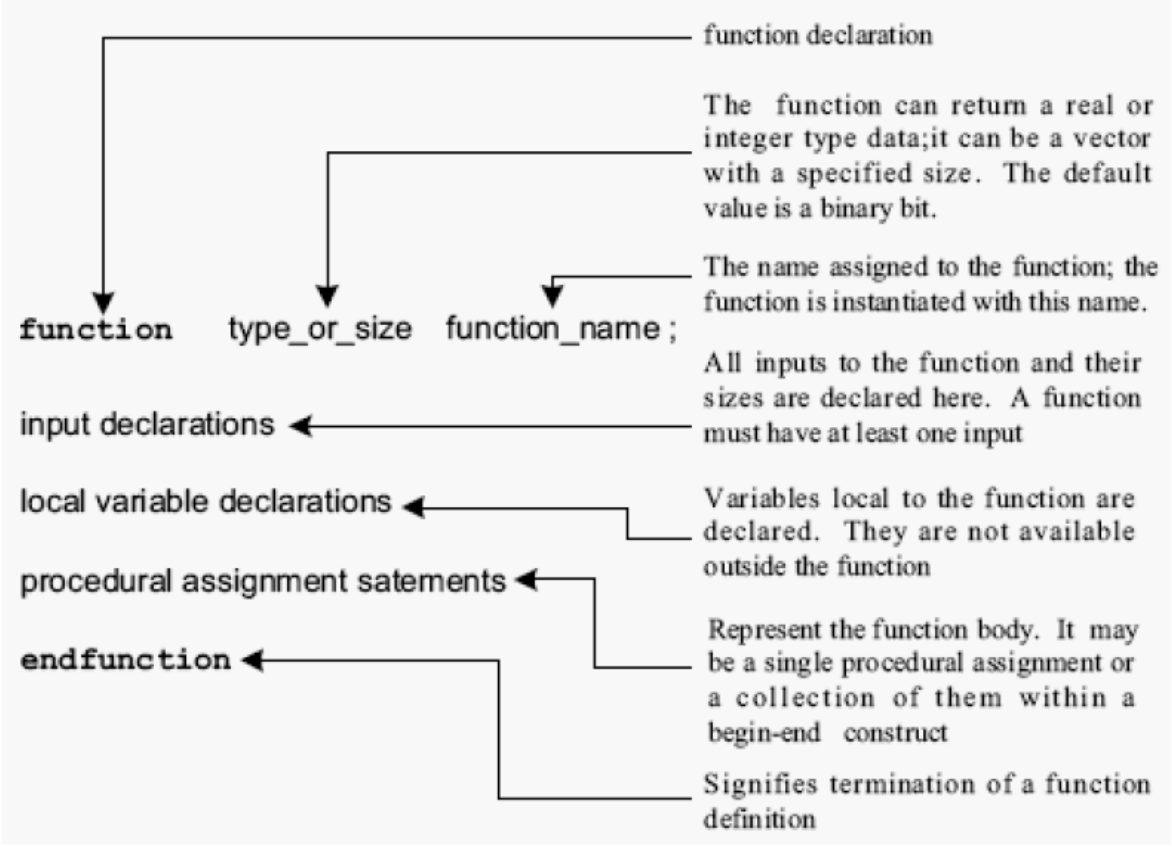
- Functions are declared with the keywords function and endfunction.
- When to use functions if the procedure
  - has no delay or timing control constructs (any statement introduced with #, @, or wait).
  - returns a single value.
  - has at least one input argument.
  - has no output or inout arguments.
  - has no nonblocking assignments.

function

- A function is like a subroutine or a procedure in a program.
- It is defined separately within a module and can be called whenever necessary.
- When a function is declared with a function name, the system allocates a register for it.
- The name of the register is that of the function; and its type (as well as size) is also that of the function.
- When a function is called, the system executes the functional activity and generates the output.
- output is assigned to the register identified for the function.
- The quantity returned by the function can be used as an operand in an assignment or in an expression

Functions (cont'd)

Structure for function definition



Functions (cont'd)

function properties

Only Combinational Logic

- function should not contain delay information.
- function must contain at least one **input**.
- Supports only **input** arguments.
- A function can call other function only.
- Can return only one value.
- function are declared with keyword **function** and **endfunction**.

Example

Implement a Parity Checker

```
module parity_chk(a,en,chk);
input [7:0] a;
input en;
output chk;
reg chk;
always@(posedge en)
begin
chk=pb(a);
end
function pb;
input [7:0]a;
pb=^a;
endfunction
endmodule
```

Example

Implement a Left/Right Shifter

Left Shift: Control =0
Right Shift: Control = 1

```
module shifter (addr, laddr, raddr)
input [31:0] addr;
output [31:0] laddr, raddr;
reg laddr, raddr;
reg addr, control;
‘define lshift 1’b0;
‘define rshift 1’b1;
always @(addr)
begin
laddr  = shift(addr, lshift);
raddr = shift(addr, rshift);
end

function [31:0]shift;
input [31:0]addr;
input control;
begin
shift  = (control==lshift)?(addr<<1):(addr >>1);
end

endmodule
```

Automatic (Recursive) Functions

- Functions are normally used non-recursively .
- If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space.
- However, the keyword **automatic** can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive calls.
- Each call to an automatic function operates in an independent variable space.
- Automatic functions can be invoked through the use of their hierarchical name.

Example: Automatic (Recursive) Functions

// Computes the factorial of the *n* numbers.

❖ The term "**automatic**" in the function declaration statement ensures recursive computation.

```
module factorial(input  [7:0] n, output [15:0] result);
// instantiate the fact function
   assign result = fact(7);
                                                  // define fact function
   function automatic [15:0]  fact;
   input [7:0] N;

   // the body of function
      if (N == 1) fact = 1;
      else fact = N * fact(N - 1);
   endfunction
endmodule
```

Example: Automatic (Recursive) Functions

// Computes the sum of the squares the first *n* natural numbers.

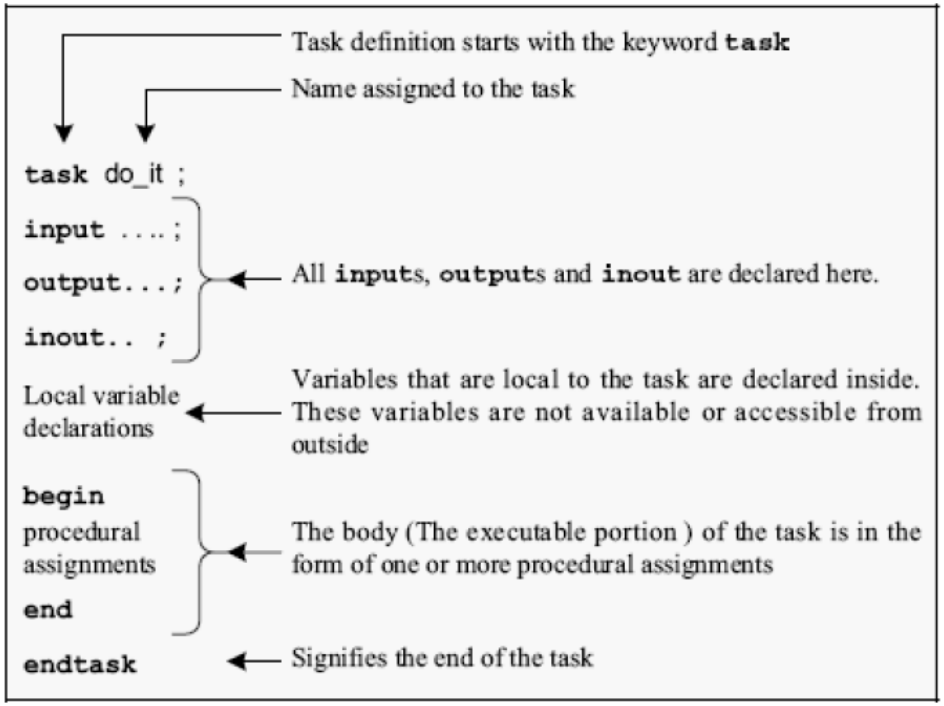
```
function automatic integer sum_sq;
input n;
begin
if(n==1) sum_sq =1;
else sum_sq = sum_sq + n*n;
end
Endfunction
```

- Thus, if *n* is assigned the value 4, during compilation sum_sq (4) will be successively replaced by

sum_sq (3) + 4^2,
sum_sq (2) + $3^2 + 4^2$,
sum_sq (1) + $2^2 + 3^2 + 4^2$ and
finally by $1^2 + 2^2 + 3^2 + 4^2$.

TASK

- The role of a task in a module is similar to that of a subroutine in a program.
- Its scope and role are wider than those of a function.



- It is defined within a module and can be called as many times as desired within a procedural block.

task properties

- task can be used with delay, timing or event control constructs.
- task can have zero or more arguments.
- Supports **input, output** and **inout** arguments.
- Passes the values through **output** and **inout** arguments to the task call.
- A task can call other tasks and functions as well.
- task are declared with keyword **task** and **endtask**.

Write a Verilog code to count the zeros in a byte using Task.

```
module zero_count_task (data, out);
input   [7:0] data;
output reg [3:0] out;  // output declared as register
always @(data)
    count_0s_in_byte(data, out);

// task declaration from here.
task count_0s_in_byte(input [7:0] data, output reg [3:0] count);
integer i;
begin   // task body
   count = 0;
for (i = 0; i <= 7; i = i + 1)
     if (data[i] == 0) count= count + 1;
end endtask
endmodule
```

System Task and Functions

System Function or Special Language Tokens

- $<identifier>: System tasks and functions
  - $time
  - $stop
  - $finish
  - $monitor
  - $ps_waves
- #<delay specification>
  - used in
    - gate instances and procedural statements
    - unnecessary in RTL specification

Special Language Tokens

- These system functions will return the integer value of simulation time at which they are executed.
  - **$time** : returns time (integer) in 64 bits
  - **$stime** : returns time (integer) in 32 bits
  - **$realtime** : returns time (real)

Time Scale

- Time scale compiler directive

  `timescale time_unit / time_precision

  - The time_precision must not exceed the time_unit.
  - For instance, with a timescale 1 ns/1 ps, the delay specification #15 corresponds to 15 ns.
  - It uses the same time unit in both behavioral and gate-level modeling.
  - For FPGA designs, it is suggested to use ns as the time unit.

Special Language Tokens

- $display displays values of variables, string, or expressions
  - $display(*ep*1, *ep*2, …, *epn*);

    *ep*1, *ep*2, …, *epn*: quoted strings, variables, expressions.
- $monitor monitors a signal when its value changes.
  - $monitor(*ep*1, *ep*2, …, *epn*);
- $monitoton enables monitoring operation.
- $monitotoff disables monitoring operation.
- $stop suspends the simulation.
- $finish terminates the simulation.

$display

- These system tasks automatically inserts a new line at the end after printing the information.

- If no format specification exists for an argument then:
  - **$display** : displays argument values in decimal.
  - **$displayb** : displays argument values in binary.
  - **$displayh** : displays argument values in hexadecimal.
  - **$displayo** : displays argument values in octal.

$display

String format specifications

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

$display

$display - Example

```
module and_gate
   (a,b,y);
input a,b;
output y;
assign y= a & b;
endmodule
```

TEST BENCH:

```
module and_test ();
reg a,b;
wire y;
and_gate a1(a,b,y);
initial
begin
a=1'b0;b=1'b0;
#100;
a=1'b0; b=1'b1;
#100;
a=1'b1; b=1'b0;
#100;
a=1'b1; b=1'b1;
#100;
$display($time ,
"The value of a=%b,
b=%b,  y=%b", a,b,y);
end
endmodule
```

```
ModelSim> vsim -gui work.display_tst
# vsim -gui work.display_tst
# Loading work.display_tst
# Loading work.display_ex
add wave -r /*
VSIM 4> run
run
run
VSIM 5> run
#                  400
#  The value of a is '1', b is '1', y is '1'

VSIM 5>
```

$monitor

- These system tasks displays the data whenever there is a change of value in an argument in the argument list and displays at the end of the simulation time at which they are executed.

- If no format specification exists for an argument then:
  - **$monitor** : displays argument values in decimal.
  - **$monitorb** : displays argument values in binary.
  - **$monitorh** :: displays argument values in hexadecimal.
  - **$monitoro** : displays argument values in octal.

$monitor

```
module monitorex(q, d, clk,
reset);
 output q;
 input d, clk, reset;
reg q;
always @(posedge clk)
if (reset)
 q <= 1'b0;
else
q <= d;
endmodule
```

```
module monitorex_tb;
reg clk;
reg reset,d;
wire q;
monitorex d1(q,d, clk, reset);
 initial clk = 1'b0;
always #5 clk = ~clk;
initial
begin
#5 reset = 1'b1;
#5 d = 1'b0;
end
initial
begin
#50 d=1'b1;
#50 reset = 1'b0;
#1000 $stop;
end
 initial
$monitor ($time, " Output q = %d", q);
endmodule
```

```
  0 Output q = x
  5 Output q = 0

105 Output q = 1
```

$monitoroff - on

- These are two additional tasks allied to the **$monitor** task.
- They are useful to enable and disable the monitoring activity.
- **$monitoroff** turns off the monitoring at the specified time, while **$monitoron** turns it on at the specified time.

$monitoroff - on

```
module monitoronex
(s,ca,a,b);
  input a,b;
  output s,ca;
  xor   (s,a,b);
  and   (ca,a,b);
endmodule
```

```
module monitoronex_tb;
  reg a,b;
  wire s,ca;
  monitoronex s1(s,ca,a,b);
always
  begin
    #5 a=1; b=0;
    #5 a=0; b=1;
    #5 a=1; b=1;
    #5 a=0; b=0;
  end
initial
    $monitor($time,"a=%b,b =%b, outcarry=%
      outsum=%b",a,b,ca,s);
    initial
    begin
      #30 $monitoroff;
      #30 $monitoron;
      #30 $monitoroff;
    end
```

```
 0a=x,b =x, outcarry=x, outsum=x
 5a=1,b =0, outcarry=0, outsum=1
10a=0,b =1, outcarry=0, outsum=1
15a=1,b =1, outcarry=1, outsum=0
20a=0,b =0, outcarry=0, outsum=0
25a=1,b =0, outcarry=0, outsum=1
60a=0,b =0, outcarry=0, outsum=0
65a=1,b =0, outcarry=0, outsum=1
70a=0,b =1, outcarry=0, outsum=1
75a=1,b =1, outcarry=1, outsum=0
80a=0,b =0, outcarry=0, outsum=0
85a=1,b =0, outcarry=0, outsum=1
```

$strobe

- These system tasks display the data at the end of the simulation time at which they are executed.

- If no format specification exists for an argument then:
  - **$strobe** : displays argument values in decimal.
  - **$strobeb** : displays argument values in binary.
  - **$strobeh** : displays argument values in hexadecimal.
  - **$strobeo** : displays argument values in octal.

$strobe

```
module strobe_ex
(a,b,y);
input a,b;
output y;
assign y= a & b;
endmodule
```

```
module strobe_tst ();
reg a,b;
wire y;
strobe_ex a1(a,b,y);
initial
begin
 a=1'b0;b=1'b0;
#100
a=1'b0; b=1'b1;
#100
a=1'b1; b=1'b0;
#100
a=1'b1; b=1'b1;
#100;
end
initial
#225 $strobe ("The value of a is
'%b', b is '%b' y is '%b'",a,b,y);
endmodule
```

```
225The value of a is '1', b is '0' y is '0'
```

$fopen - $close

To carry out any file-based task, the file has to be opened, read, written, and the file closed.

```
module
   ha_1(s,ca,a,b);
input a,b;
output s,ca;
xor  (s,a,b);
and   (ca,a,b);
endmodule
```

```
module fclose_tb;
  reg a,b;
  wire s,ca;
  integer info;
  fclose s1(s,ca,a,b);
  initial
  begin
   a=0;
    b=0;
    info=$fopen("file.txt");
  end
  always
  begin
    #5      a=1;     b=0;
    #5      a=0;    b=1;
    #5      a=1;    b=1;
    #5      a=0;      b=0;
    end
```

```
  initial
begin
  $fmonitor(info,$time,"a=%b,b =%b,
outcarry=%b, outsum=%b",a,b,ca,s);
   #1000 $stop;
$fclose(info);
end
  endmodule
```

$random

- Random number generation capabilities are required for generating a random set of test vectors.
- Random testing is important because it often catches hidden bugs in the design.
- Random vector generation is also used in performance analysis of chip architectures.
- The system task **$random** is used for generating a random number.

$random

- Usage:

  $random;

  $random(<seed>);

- The value of <seed> is optional and is used to ensure the same random number sequence each time the test is run.
- The <seed> parameter can either be a reg, integer, or time variable.
- The task **$random** returns a 32-bit signed integer.
- All bits, bit-selects, or part-selects of the 32-bit random number can be used

$random

```
module adder (out, cout, a, b);
input [7:0] a, b;
output [7:0] out;
output cout;
assign #5 {cout,out} = a + b;
endmodule
```

```
module test_adder;
reg [7:0] a, b;
wire [7:0] sum; wire cout;
integer myseed;
adder ADD (sum, cout, a, b);
initial myseed = 15;
initial
begin
repeat (5)
begin
a = $random(myseed);
b = $random(myseed); #10;
$display ("T: %3d, a: %h, b: %h,
sum: %h", $time, a, b, sum);
end
end
endmodule
```

```
T:  10, a: 00, b: 52, sum: 52
T:  20, a: ca, b: 08, sum: d2
T:  30, a: 0c, b: 6a, sum: 76
T:  40, a: b1, b: 71, sum: 22
T:  50, a: 23, b: df, sum: 02
```

User Defined Primitives - UDP

User-Defined Primitives

- Way to define gates and sequential elements using a truth table.
- Often simulate faster than using expressions, collections of primitive gates, etc.
- Gives more control over behavior with ‘X’ inputs.
- Most often used for specifying custom gate libraries.

User-Defined Primitives

They are used to define custom Verilog primitives by the use of lookup tables.

They can specify:

– Truth table for combinational functions.

– State table for sequential functions.

– Don't care, rising and falling edges, etc. can be specified.

• For combinational functions, truth table entries are specified as:

<input1> <input2> … <inputN> : <output>;

• For sequential functions, state table entries are specified as:

<input1> <input2> … <inputN> : <present_state> : <next_state>

User-Defined Primitives

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
reg output_port;                    // only for sequential UDP
initial output-port = expression;   //only for sequential UDP
table                               // define the behavior of UDP
<table rows>
endtable
endprimitive
```

User-Defined Primitives

```
// port list declaration style
primitive udp_name(output output_port, input input_ports);
reg output_port;                          // only for sequential UDP
initial output-port = expression;         //only for sequential UDP
table                                     // define the behavior of UDP
<table rows>
endtable
endprimitive
```

UDP – Basic Rules

- Inputs
  - can have scalar inputs.
  - can have multiple inputs.
  - are declared with input.
- Output
  - can have only one scalar output.
  - must appear in the first terminal list.
  - is declared with the keyword output.
  - must also be declared as a reg in sequential UDPs.
- UDPs
  - do not support inout ports.
  - are at the same level as modules.
  - are instantiated exactly like gate primitives.

UDP – Combinational Logic

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
                                // UDP state table
table                           // the state table starts from here
<table rows>
endtable
endprimitive
```

UDP – Combinational Logic

Implement a UDP for 'AND' Gate

```
primitive udp_and(out, a, b);
// declarations
output out; // must not be declared as reg for combinational UDP
input a, b;  // declarations for inputs.
table        // state table definition; starts with keyword table
//    a   b   :   out;
      0   0   :   0;
      0   1   :   0;
      1   0   :   0;
      1   1   :   1;
endtable          // end state table definition
endprimitive      // end of udp  and definition
```

```
a b : out;
1 1 : 1;
0 ? : 0;
? 0 : 0;          // ? expanded to 0, 1, x
```

UDP – Combinational Logic

Instantiation of UDPs

```
// an example illustrates the instantiations of UDPs.
module UDP_full_adder(sum, cout, x, y, cin);
output sum, cout;
input x, y, cin;
wire s1, c1, c2;
// instantiate udp primitives
udp_xor (s1, x, y);
udp_and (c1, x, y);
udp_xor (sum, s1, cin);
udp_and (c2, s1, cin);
udp_or (cout, c1, c2);
endmodule
```

Some synthesizers might not synthesize UDPs.

UDP – Combinational Logic

Implement a UDP for '2:1 MUX'

```
// Output should always be the first signal in port list
primitive mux (out, sel, a, b);
    output  out;
    input   sel, a, b;

    table
        // sel  a   b       out
            0   1   ?   :   1;
            0   0   ?   :   0;
            1   ?   0   :   0;
            1   ?   1   :   1;
            x   0   0   :   0;
            x   1   1   :   1;
    endtable
endprimitive
```

UDP – Sequential Logic

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
reg output_port;                        // unique for sequential UDP
initial output-port = expression;       // optional for sequential UDP
// UDP state table
table // keyword to start the state table
   <table rows>
endtable
endprimitive
```

UDP – Sequential Logic

- State table entries

  <input1><input2>……<input*n*> : <current_state> : <next_state>;

  - The output is always declared as a reg.
  - An initial statement can be used to initialize output.
  - Inputs, current state, and next state are separated by a colon ":".
  - The input specification can be input levels or edge transitions.
  - All possible combinations of inputs must be specified to avoid unknown output value.

UDP – Sequential Logic

Level-Sensitive UDPs

```
1  primitive d_latch (q, clk, d);
2      output  q;
3      input   clk, d;
4      reg     q;
5
6      table
7          // clk  d       q   q+
8              1   1   :   ? : 1;
9              1   0   :   ? : 0;
10             0   ?   :   ? : -;
11     endtable
12 endprimitive
```

User-Defined Primitives

Symbols	Meaning	Explanation
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state value	Can use only in a sequential UDP output field
r	(01)	Rising edge of a signal
f	(10)	Falling edge of a signal
p	(01), (0x), or (x1)	Potential rising edge of a signal
n	(10), (1x), or (x0)	Potential falling edge of a signal
*	(??)	Any value change in signal

UDP – Sequential Logic

```
// define a level-sensitive latch using UDP.
primitive d_latch(q, d, gate, clear);
output q;
input d, gate, clear;
reg q;
initial q = 0;          // initialize output q
                        //state table
table
// d gate clear : q : q+;
   ?   ?    1   : ? : 0 ;   // clear
   1   1    0   : ? : 1 ;   // latch q = 1
   0   1    0   : ? : 0 ;   // latch q = 0
   ?   0    0   : ? : -;    // no state change
endtable
endprimitive
```

Represent No State Change

UDP – Sequential Logic

Edge-Sensitive UDPs

```
primitive d_flop (q, clk, d);
    output  q;
    input   clk, d;
    reg     q;

    table
        // clk      d       q       q+
            // obtain output on rising edge of clk
            (01)    0   :   ?   :   0;
            (01)    1   :   ?   :   1;
            (0?)    1   :   1   :   1;
            (0?)    0   :   0   :   0;

            // ignore negative edge of clk
            (?0)    ?   :   ?   :   -;

            // ignore data changes on steady clk
            ?       (??):   ?   :   -;
    endtable
endprimitive
```

UDP – Sequential Logic

Instantiation of UDPs

```
// an example of sequential UDP instantiations
module ripple_counter(clock, clear, q);
input   clock, clear;
output [3:0] q;

// instantiate the T_FFs.
T_FF tff0(q[0], clock, clear);
T_FF tff1(q[1], q[0],  clear);
T_FF tff2(q[2], q[1],  clear);
T_FF tff3(q[3], q[2],  clear);
endmodule
```

UDP – Sequential Logic

```
// define a positive-edge triggered T-type flip-flop using UDP.
primitive T_FF(q, clk, clear);
output q;
input clk, clear;
reg q;
// define the behavior of edge-triggered T_FF
table
// clk clear : q : q+;
    ?      1   : ? : 0 ;  // asynchronous clear
    ?   (10) : ? : - ;   // ignore negative edge of clear
   (01)    0 : 1 : 0 ;  // toggle at positive edge
   (01)    0 : 0 : 1 ;  // of clk
   (1?)    0 : ? : - ;   // ignore negative edge of clk
endtable
endprimitive
```

Writing Test Bench

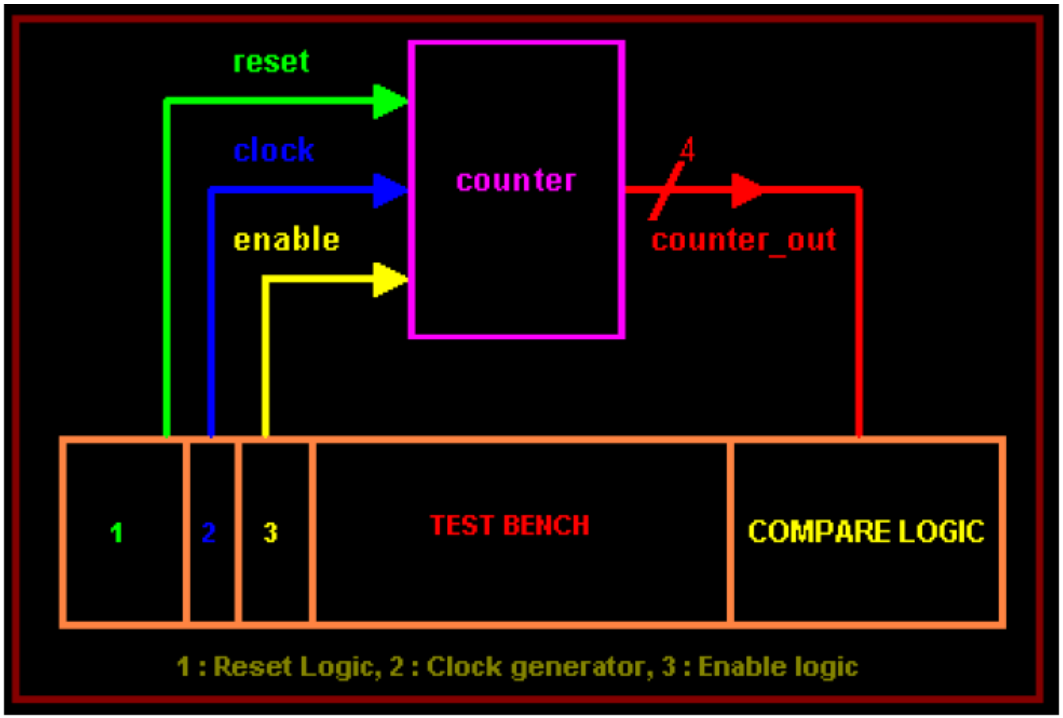
Writing Test Bench

- Writing Testbench it is essential to have the design specification of "design under test" or simply DUT.
- Specs need to be understood clearly and test plan is made.
- This basically documents the test bench architecture and the test scenarios (test cases) in detail.

Example - Counter

```
module counter (clk, reset, enable, count);
   input clk, reset, enable;
   output [3:0] count;
   reg [3:0] count;

   always @ (posedge clk)
   if (reset == 1'b1)
      count <= 0;
   else if ( enable == 1'b1)
      count <= count + 1;

endmodule
```

Example - Counter



DUT is instantiated in the testbench

Testbench will contain a clock generator, reset generator, enable logic generator, and compare logic.

Example - Counter

- The first step of any testbench creation is to create a template.
- This basically declares inputs to DUT as reg and outputs from DUT as wire.
- There is no port list for the test bench.

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));



endmodule
```

- Next step would be to add clock generator logic.

Example - Counter

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));

initial      // Executed only once
begin
clk = 0;
reset = 0;                    // Initially all zero
enable = 0;
end
always
#5 clk = !clk;

endmodule
```

The Simulator does not come out, or print anything on screen or does it dump any waveform.

Example - Counter

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));
initial
begin clk = 0;
reset = 0;
enable = 0; end
always
#5 clk = !clk;

initial
#100 $finish;

endmodule
```

// $finish is used for terminating simulation after #100 time units

The Simulator does come out at 100 time unit, but doesn't print anything on screen or does it dump any waveform.

Example - Counter

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));
initial
begin clk = 0;
reset = 0;
enable = 0; end
always
#5 clk = !clk;
initial
#100 $finish;
begin
$dumpfile ("counter.vcd");
$dumpvars; end
endmodule
```

// **$dumpfile** is used for specifying the file that simulator will use to store the waveform, that can be used later to view using waveform viewer.

//$dumpvars basically instructs the Verilog compiler Dump all the variables to counter.vcd file.

The Simulator doesn't print anything on scree.

Example - Counter

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));
initial
begin clk = 0;
reset = 0;
enable = 0; end
always
#5 clk = !clk;
initial
#100 $finish;
begin
$dumpfile ("counter.vcd");
$dumpvars; end
Initial
begin
$display("\t\ttime,\tclk,\treset,\tenable,\tcount");
$monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,count);
end
endmodule
```

//$finish is used for terminating simulation after #100 time units.

//$dumpfile specifying the file that can be used later to view using waveform viewer.

//$dumpvars dump all the variables to counter.vcd file.

//$display is used for printing text or variables.

//$monitor keeps track of changes to the variables.

Example - Counter

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (.clk(clk), .reset(reset), .enable(enable), .count (count));
initial
begin clk = 0;
reset = 0;
enable = 0; end
always
#5 clk = !clk;
initial
#100 $finish;
endmodule
```

// repeat (n)
// forever
// loop

Example - Adder

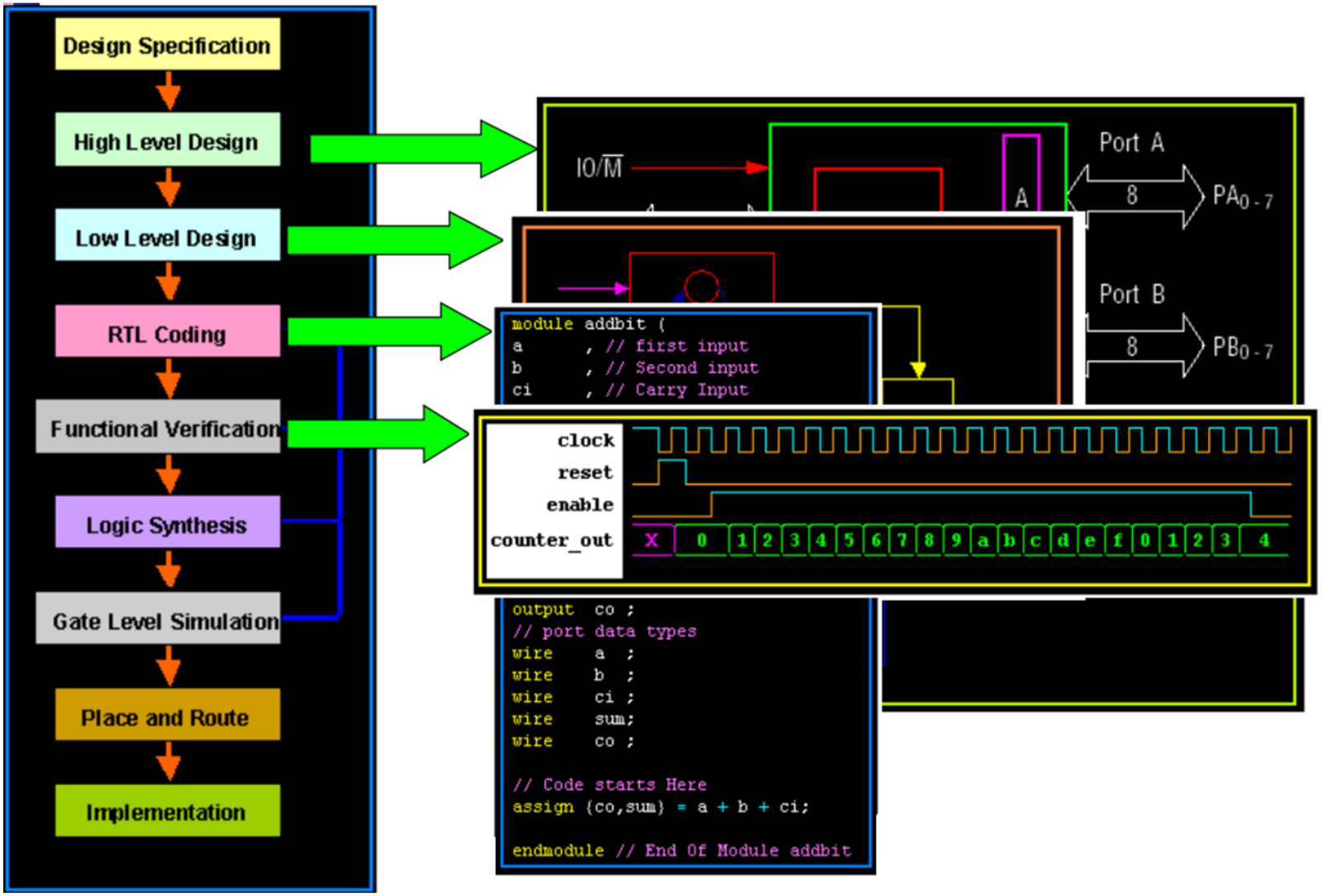
Write a testbench for the following full adder code

```
module fulladder(a,b,c,y,cout);
input a,b,c;
output y,cout;
assign y = a^b^c;
assign cout = a&b|b&c|a&c;
endmodule
```

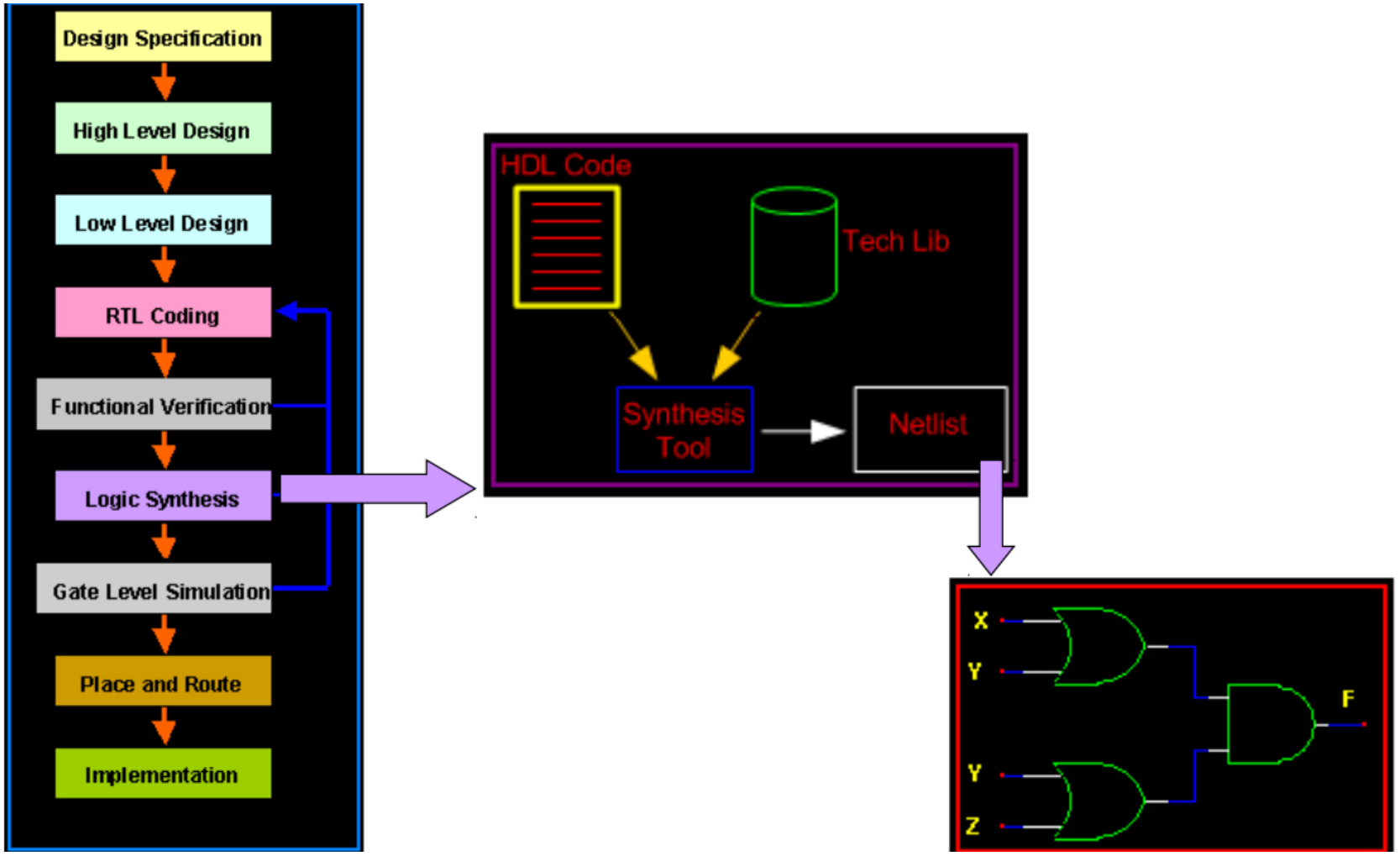
```
module fulladder_tb();
reg a,b,c;
wire y,cout;
fulladder uut(a,b,c,y,cout);


initial begin
a=0;b=0;c=0;
#5 a=0;b=0;c=1;
#5 a=0;b=1;c=0;
#5 a=0;b=1;c=1;
#5 a=1;b=0;c=0;
#5 a=1;b=0;c=1;
#5 a=1;b=1;c=0;
#5 a=1;b=1;c=1;
end

initial begin
$monitor($time,"a=%b,b=%b,c=%b",a,b,c);
#40 $finish;
end

endmodule
```

Coding Style for Synthesis

VLSI Design Flow



VLSI Design Flow



Coding Style for Synthesis

- Modeling style has a marked impact on the type of netlist generated after logic synthesis.
- Current synthesis tools can accept only a subset of constructs of Verilog or VHDL.
- The subset may vary from one synthesis tool to another.
- Functionally equivalent simulation models need not necessarily produce the same synthesis outputs.

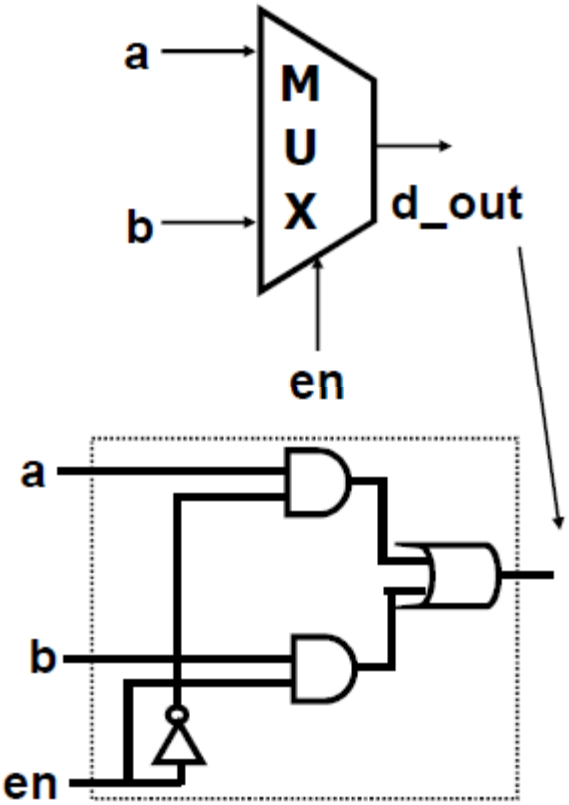
Synthesizing Combinational Logic

- Combinational logic can be inferred from Verilog codes in several ways.
- Based on the type of variable declaration (reg or wire), always block and assign statements can be used to infer COMBO.
- Care should be taken in the choice of the Verilog constructs used to infer COMBO.

Synthesizing Combinational Logic

Inferring a Multiplexer using if-else statement

```
reg d_out ;
   always @ (a or b or en)
      if (en)
        d_out = a ;
  else
        d_out = b;
```



- The if ... else construct is completely specified.
- The modeled logic is a combinational multiplexer.

Synthesizing Combinational Logic

Priority encoded Multiplexer using if-else if statement

- Depending on the choice of "select" inputs, if - else if can b effectively used to model priority encoded "cascading multiplexers.

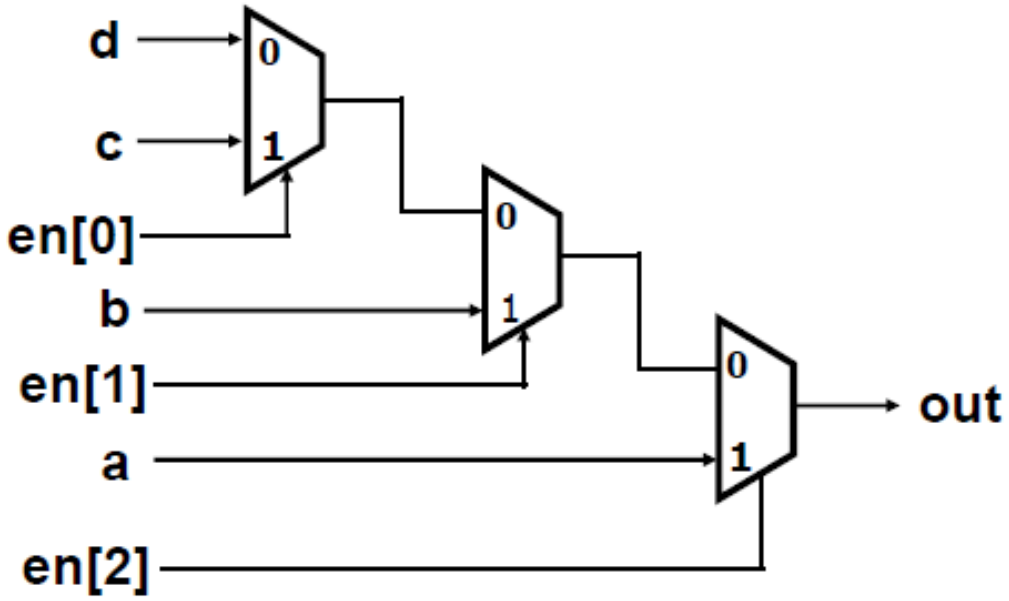
```
always @(a or b or c or d
or en)
   if (en[2] == 1'b1)
        out = a;
  else if (en[1] == 1'b1)
        out = b;
   else if (en[0] == 1'b1)
        out = c;
   else
        out = d;
```

en[2]	en[1]	en[0]
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Synthesizing Combinational Logic

Priority encoded Multiplexer using if-else if statement

- Priority encoded "cascading" multiplexers.



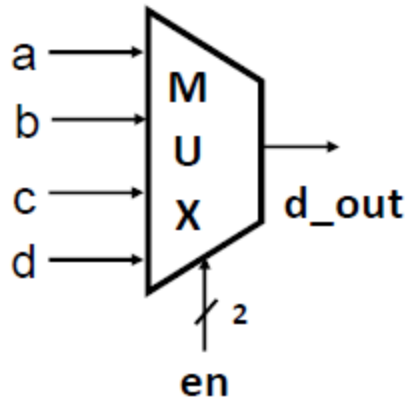
Synthesis of Case statements

- Case statements can be synthesized into combinational or sequential logic.
- This depends upon whether the case alternatives are completely specified or not.
- A completely specified case construct can infer purely combinational multiplexers with its select lines activated on a priority basis.
- Incomplete case specification infer latches.

Synthesis of Case statements

- Case statements infer faster and "large" (occupy more silicon area) multiplexers.

```
always @(a or b or c or d or en)
  begin
     case (en)
        2'b00 : d_out = a;
        2'b01 : d_out = b;
        2'b10 : d_out = c;
        default : d_out = d;
     endcase
  end
```



Synthesis of Case statements

Incomplete Case Statement inferred latches.

- When a case statement specifies only a partial list of alternatives, a latch is inferred.

```
always @(a or b or c or en)
  begin
     case (en)
        2'b00 : d_out = a;
        2'b01 : d_out = b;

        2'b10 : d_out = c;
     endcase
  end
```

- In this sample code, the case alternative for en = 11 is not specified.
- This makes the synthesis tool to infer a latch to store the previous value of d_out when en = 11 is encountered.

Synthesis of ‘if-else’ and ‘Case’ Statement

- For both Case and if-else statements, it is necessary to specify all clauses (alternatives).
- It is also necessary to specify all outputs for every alternative in Case and if - else if statements.
- Multiplexer is the preferred synthesis.

Synthesis of ‘For’ loop

- For loops can be used to infer cascaded combinational logic blocks.
- For loops cannot contain any timing or event control constructs.

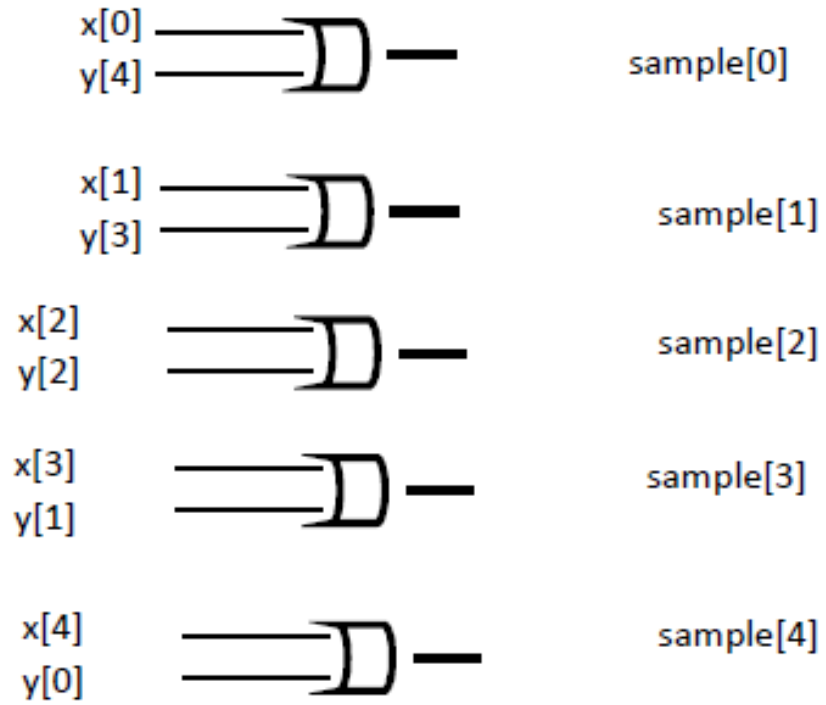
```
integer k;
always @ (x or y)
  begin
 for ( k = 0; k < 5; k = k + 1)
 sample[k] = x[k] | y[4 - k];
end
```

will be synthesized as a series of sequential assignments as follows:

```
sample[0] <= x[0] or y[4];
sample[1] <= x[1] or y[3];
sample[2] <= x[2] or y[2];
sample[3] <= x[3] or y[1];
sample[4] <= x[4] or y[0];
```

Synthesis of ‘For’ loop

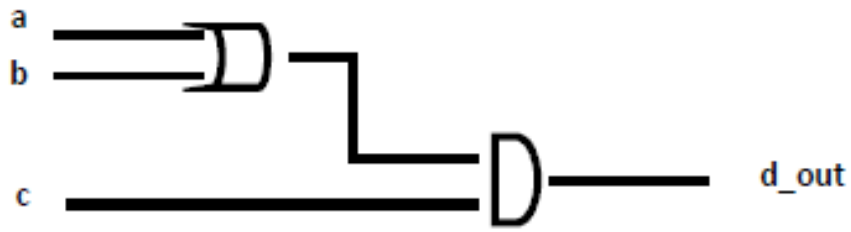
Equivalent gate level representation



Synthesis of ‘Assign’ statement

- The assign statement in Verilog can be used to synthesize combinational logic.

assign d_out = (a | b) & c; translates to:

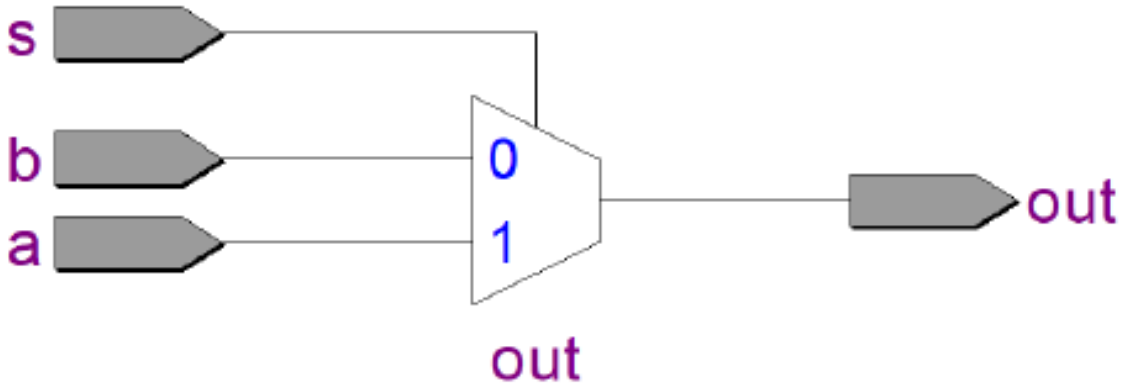


If a, b, c and d_out are multi-bit vectors, say 3-bit vectors, then, three identical circuits as above would be generated.

Synthesis of ‘Ternary’ operator

- The ternary conditional operator ? typically infers a multiplexer circuit.

```
assign out = (s ? b : a);
```



Guidelines for efficient synthesis

- Do not mix positive and negative edge-triggered flip-flops in the same design.
- ‘assign’ statements typically result in random COMBO logic, ‘gate instantiations’ in structured COMBO logic.
- Choice between assign & gate instantiations depends on optimization strategy, concise representation, complexity of design and technology re-targeting.

Guidelines for efficient synthesis

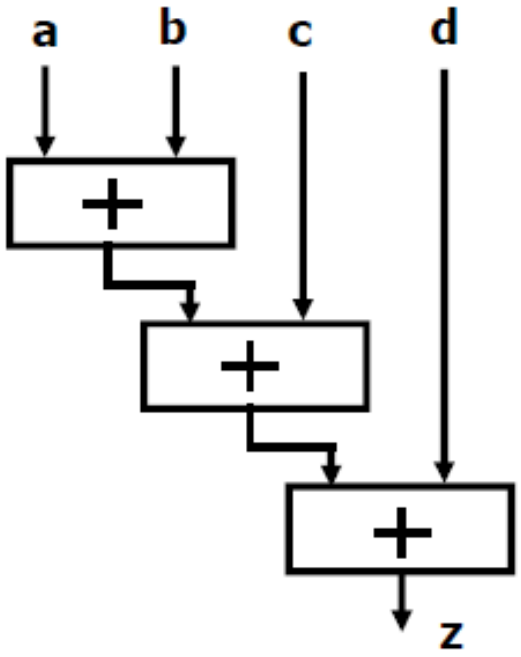
Parenthesis:

- Parentheses plays vital role in the type of logic implemented.

F = a + b + c + d;

would typically be implemented as:

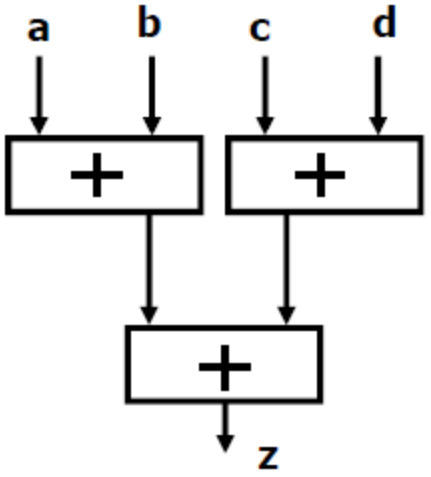
(a + b) is grouped together by default,
then c and d are added one at a time.



Guidelines for efficient synthesis

Parenthesis:

F = (a + b) + (c + d); // would typically be implemented as:

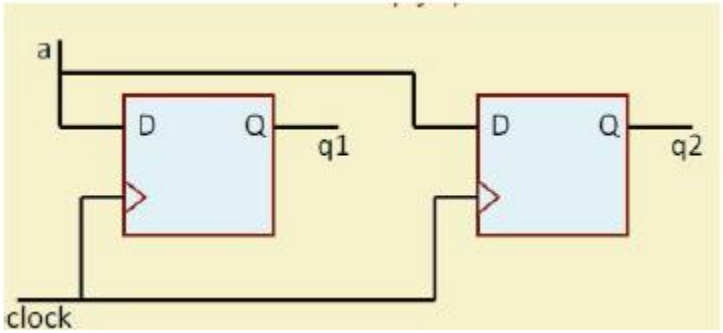


Results in a faster implementation

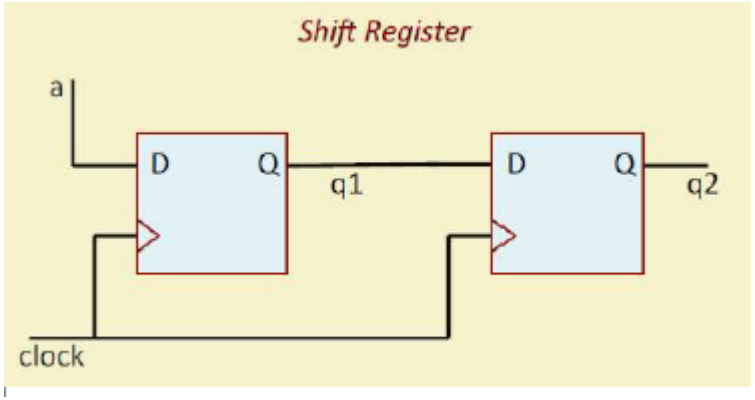
Guidelines for efficient synthesis

Blocking and Non-blocking Statement:

```
always @(posedge clock)
begin
q1 = a;
q2 = q1;
end
```



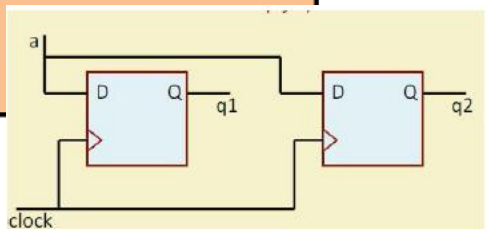

```
always @(posedge
clock)
begin
q1 <= a;
q2 <= q1;
end
```



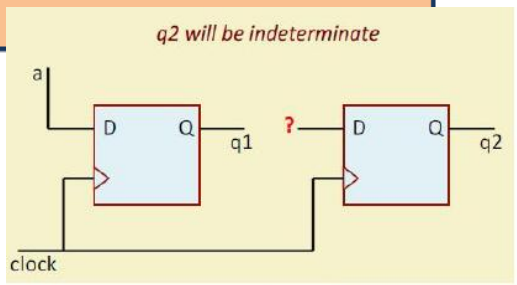
Guidelines for efficient synthesis

Blocking and Non-blocking Statement:

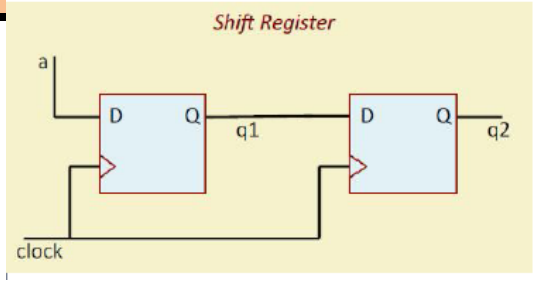
```
always @(posedge clock)
begin
q1 = a;
q2 = q1;
end
```



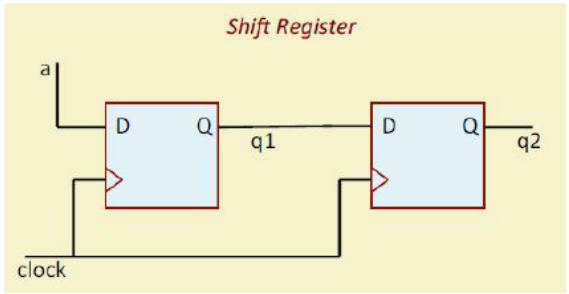

```
always @(posedge clock)
q1 = a;
always @(posedge clock)
q2 = q1;
```




```
always @(posedge
clock)
begin
q1 <= a;
q2 <= q1;
end
```

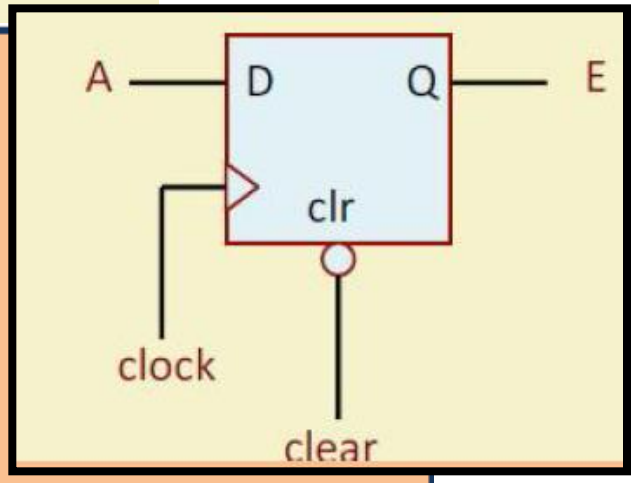


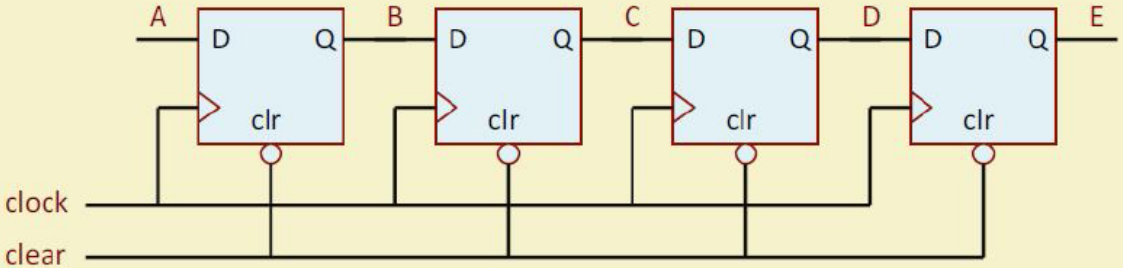

```
always @(posedge clock)
q2 <= q1;
always @(posedge clock)
q1 <= a;
```



Guidelines for efficient synthesis

```
module shiftreg_4bit (clock, clear, A, E);
input clock, clear, A;
output reg E;
reg B, C, D;
always @(posedge clock or negedge clear)
begin
if (!clear) begin B=0; C=0; D=0; E=0; end
else begin
E = D;
D = C;
C = B;
B = A;
end
end
endmodule
```





Guidelines for efficient synthesis

```
module shiftreg_4bit (clock, clear, A, E);
input clock, clear, A;
output reg E;
reg B, C, D;
always @(posedge clock or negedge clear)
begin
if (!clear) begin B<=0; C<=0; D<=0; E<=0; end
else begin
E <= D;
D <= C;
C <= B;
B <= A;
end
end
endmodule
```