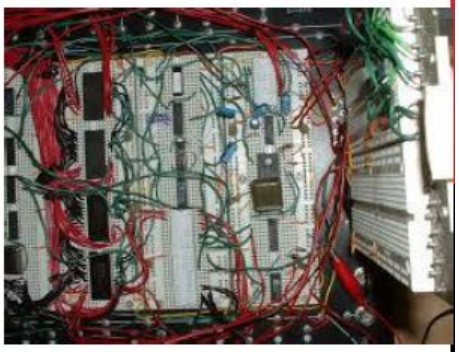
INTRODUCTION TO VERILOG

Prepared by
Dr. A. Maria Jossy

WHY WE NEED HARDWARE DESCRIPTION LANGUAGE (HDL)



- Testing of large circuits on the Printed Circuit Board is not possible, Each and every gate cannot be checked

- HDL – Easy to design through computer and then implement it on hardware.

Types of HDL

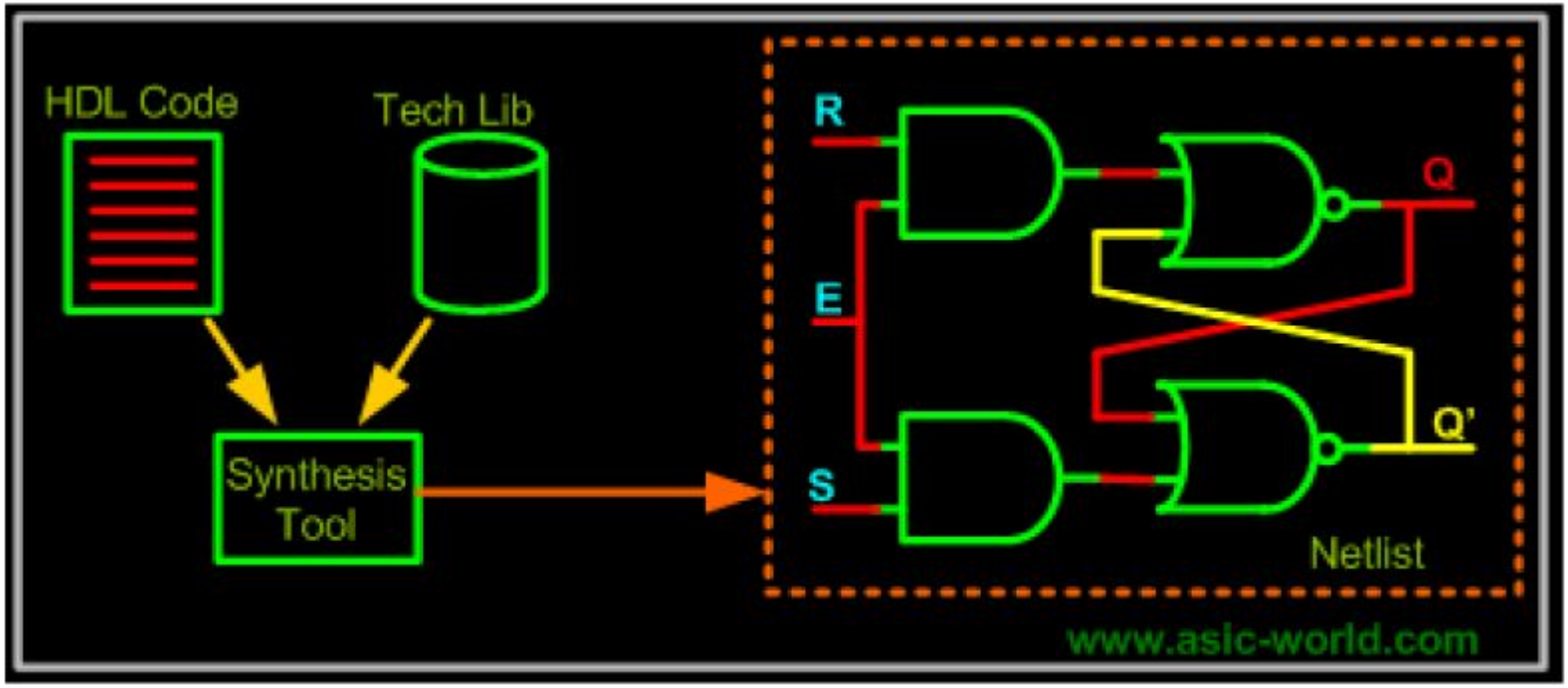
i. VHDL - VHSIC Hardware Description Language
(standardized as IEEE 1076-1987)

i. Verilog – Verify Logic
(standardized as IEEE 1364)

"Think Hardware and implement with HDL"

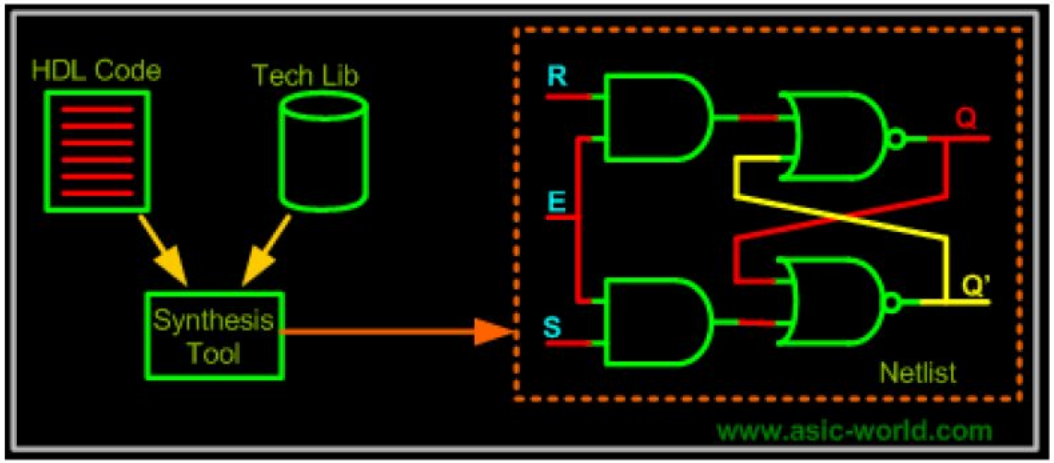
WHAT IS LOGIC SYNTHESIS?

- Logic synthesis is the process of converting a **high-level description of design into an optimized gate-level representation**.
- Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops.
- Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).



WHAT IS LOGIC SYNTHESIS?

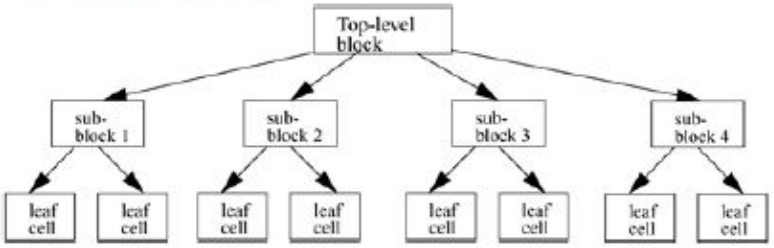
- Logic synthesis is the process of converting a **high-level description of design into an optimized gate-level representation**.
- Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops.
- Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).



1. DESIGN METHODOLOGIES

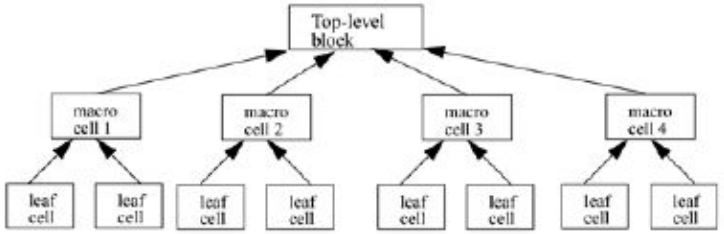
1. Top-down Design Methodology

In a *top-down design methodology*, we define the top-level block and identify the sub-blocks necessary to build the top-level block.



2. Bottom-up Design Methodology

In a *bottom-up design methodology*, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks.



EXAMPLE – DESIGN METHODOLOGY

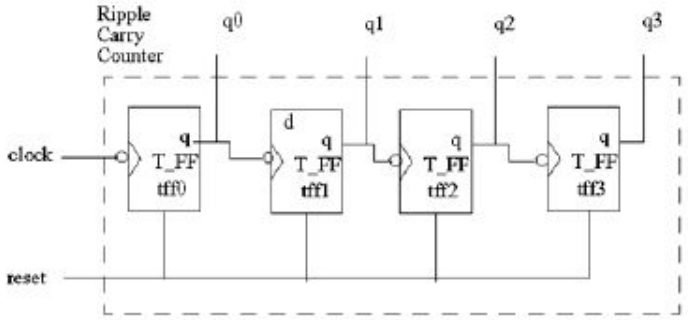


Fig 1: Ripple carry counter

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0

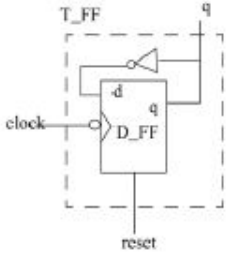


Fig 2: T flip flop

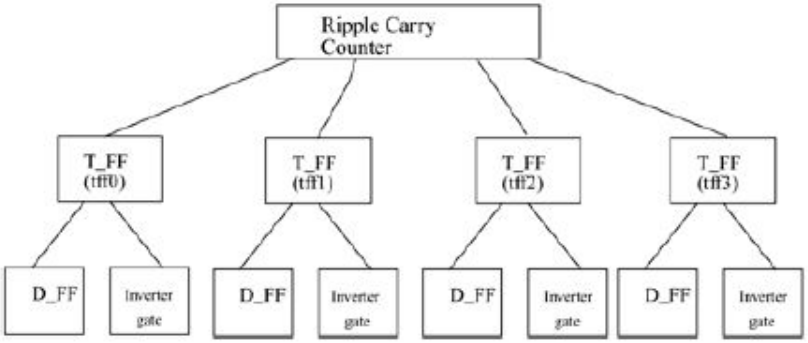
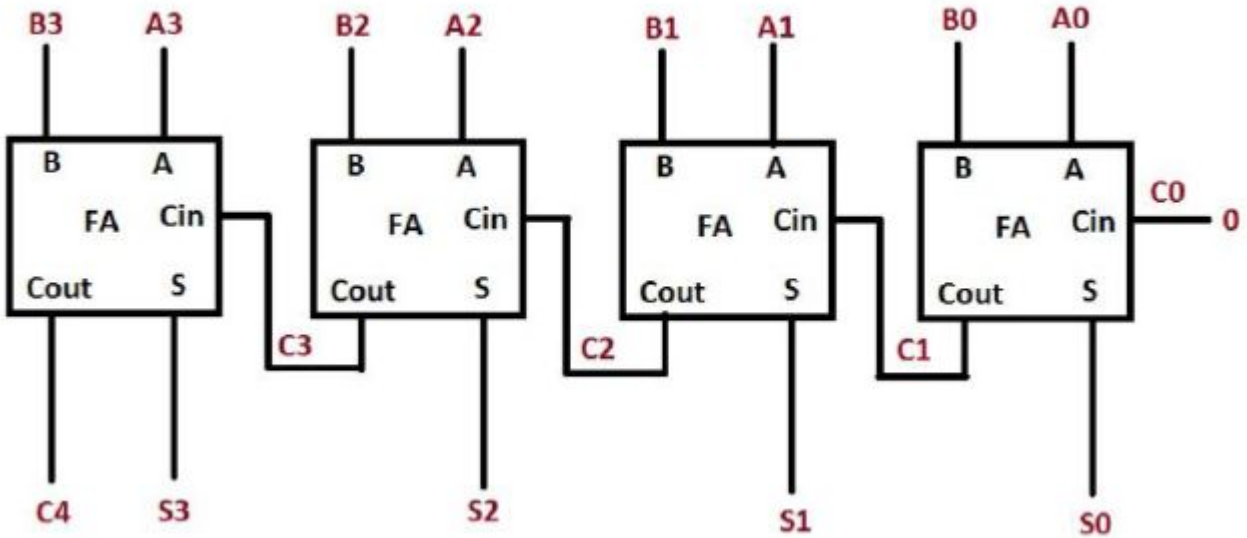


Fig 3: Design Hierarchy

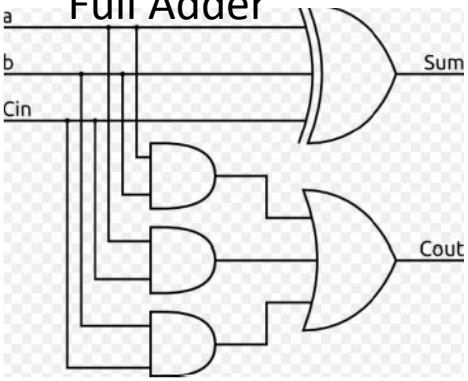
A 4 BIT RIPPLE CARRY ADDER



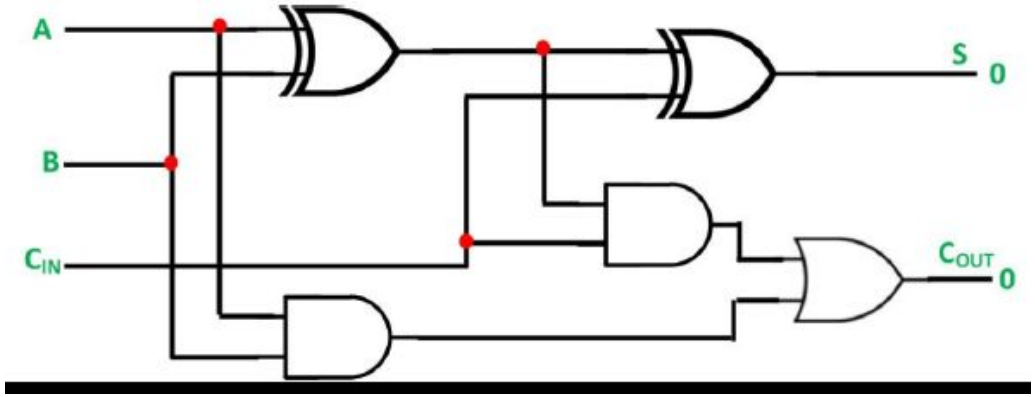
FA - Full Adders
B - Data 2
C - Carry

A - Data 1
S - Sum

Full Adder

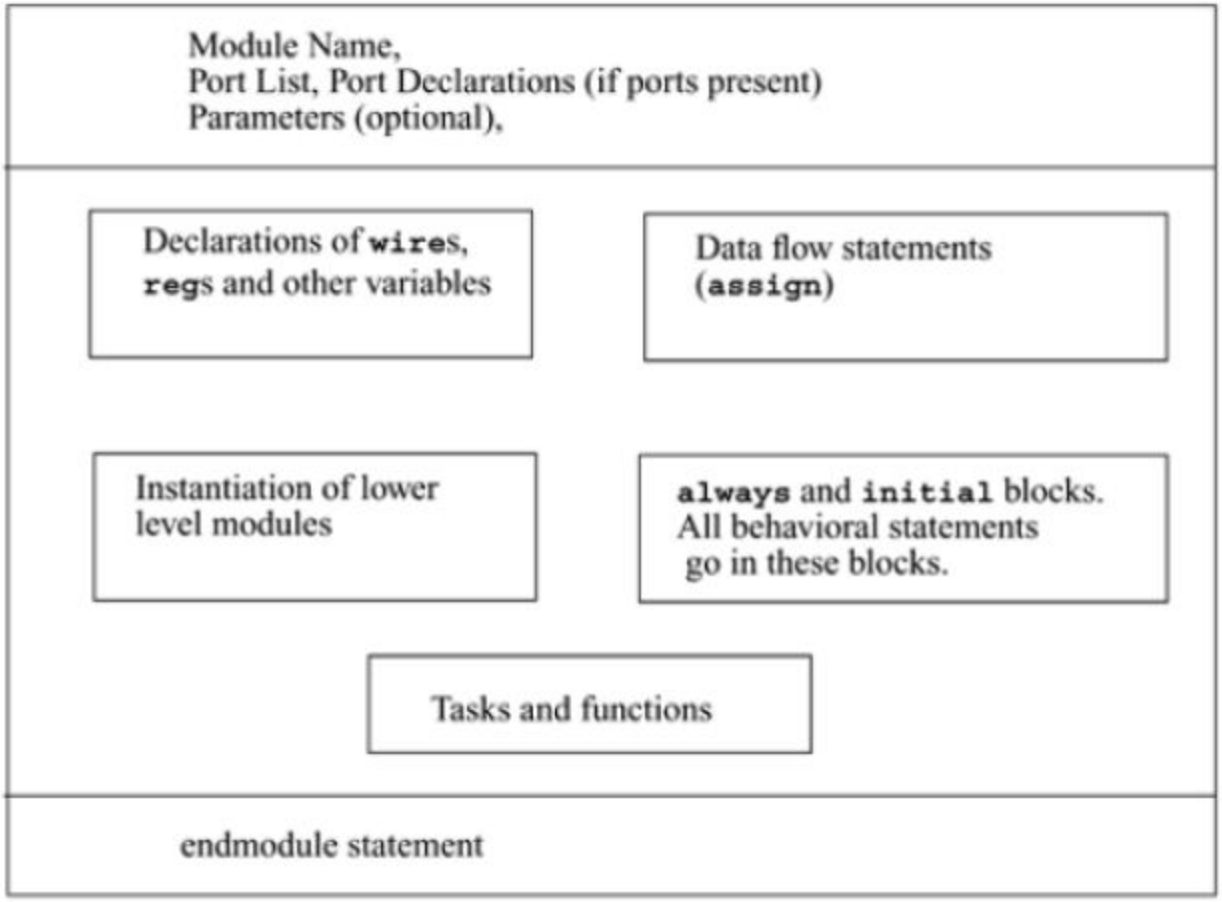


Full Adder using half adder

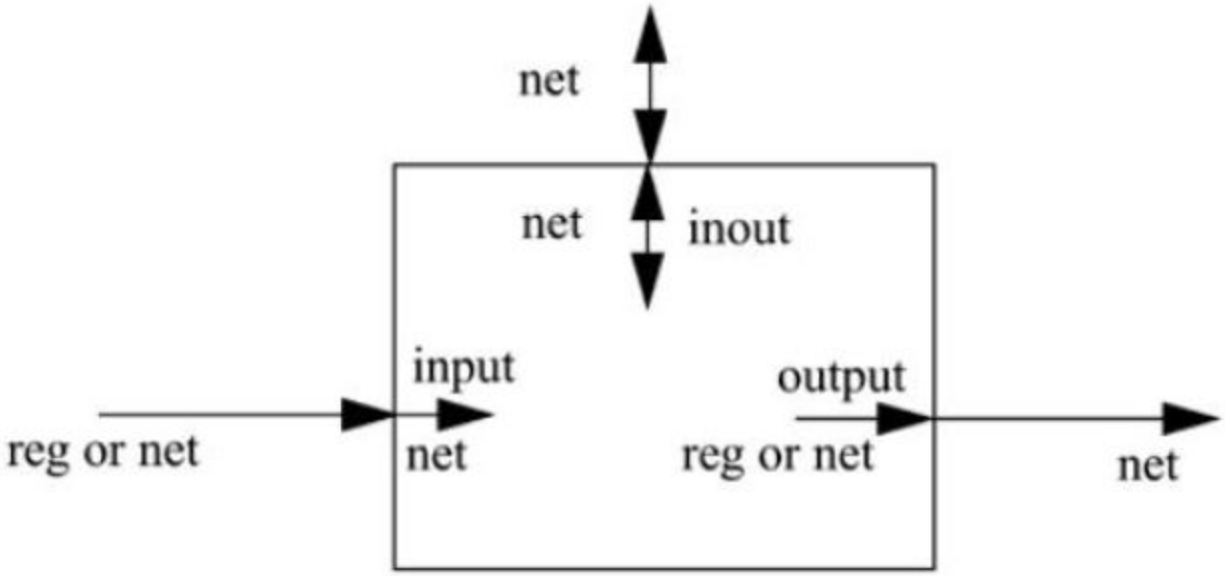


COMPONENTS OF VERILOG

Components of a Verilog Module



PORT CONNECTION RULES

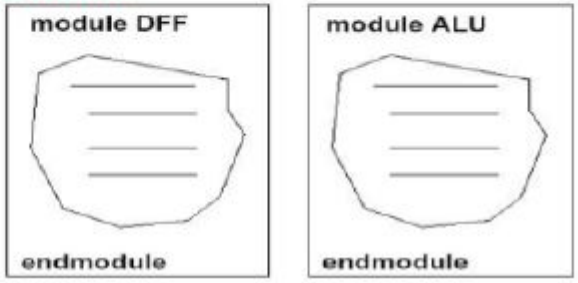


2. MODULES

- Verilog provides the concept of a module. **A module is the basic building block in Verilog.**
- A module can be an element or a collection of lower-level design blocks. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.
- Each **module** must have a **module_name**, which is the identifier for the module, and a **module_terminal_list**, which describes the input and output terminals of the module.

Basic unit --*Module*

```
module module_name
  (port_name);
        port declaration
      data type declaration
  module functionality or structure
Endmodule
```



LEVELS OF ABSTRACTION

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design.

i. **Behavioral or algorithmic level**
ii. **Dataflow level**
iii. **Gate level**
iv. **Switch level**

In the digital design community, the term **register transfer level (RTL)** is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.

LEVELS OF ABSTRACTION

1. Dataflow level

The designer is aware of how data flows between hardware registers and how the data is processed in the design. In dataflow modeling most of the design is implemented using continuous assignments, which are used to drive a value onto a net. **The continuous assignments are made using the keyword *assign*.**

```
assign [delay] net_lvalue= expression;
```

2. Gate level

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

3. Behavioral or algorithmic level

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.

4. Switch level

This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details

3. INSTANCES

- module provides a template from which you can create actual **objects**. When a module is invoked, Verilog creates a unique object from the template. **Each object has its own name, variables, parameters, and I/O interface.**
- **The process of creating objects from a module template is called instantiation, and the objects are called instances.**

Module Instantiation – Example

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations
              //will be explained later.
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

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

//Declarations to be explained later
output q;
input clk, reset;
wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive. Explained later.

endmodule
```

ILLEGAL MODULE NESTING - EXAMPLE

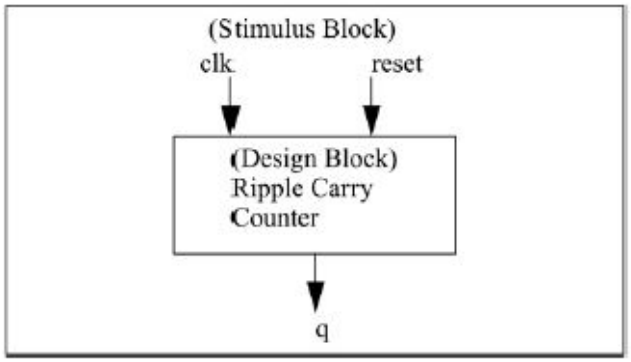
An illegal module nesting where the module T_FF is defined inside the module definition of the ripple carry counter.

```
// Define the top-level module called ripple carry counter.
// It is illegal to define the module T_FF inside this module.
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;

   module T_FF(q, clock, reset);// ILLEGAL MODULE NESTING
   ...
   <module T_FF internals>
   ...
   endmodule // END OF ILLEGAL MODULE NESTING

endmodule
```

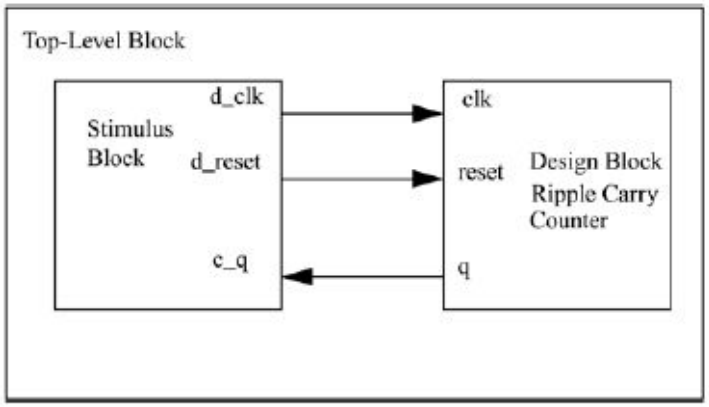
COMPONENTS OF A SIMULATION

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block. The stimulus block is also commonly called a test bench

Stimulus Block Instantiates Design Block



Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module



LEXICAL CONVENTIONS

LEXICAL CONVENTIONS

- Verilog contains a stream of tokens.
- Tokens can be comments, delimiters(a blank space, comma, or other character or symbol that indicates the beginning or end of a character string, word, or data item), numbers, strings, identifiers(user provided names to identify the objects), and keywords(predefined identifiers to define the language constructs).
- Verilog is case sensitive and are in lower case.

1. White space, namely, spaces, tabs and new-lines are ignored
2. Verilog has two commands :

1. One line comments start with `//` and end at the end of the line
2. Multi-line comments start with `/*` and end with `*/`

LEXICAL CONVENTIONS (CONT)

3. Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized Number

Sized numbers are represented as ***<size> '<base format> <number>.***

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

Unsized Number

- Numbers that are specified without a <base format> specification are decimal numbers by default.
- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

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

LEXICAL CONVENTIONS(CONT..)

X or Z values

- ✓ An unknown value is denoted by an x.
- ✓ A high impedance value is denoted by z.

```
12'h13x // This is a 12-bit hex number; 4 least significant bits
unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number
```

Negative numbers

- ✓ Negative numbers can be specified by putting a minus sign before the size for a constant number

```
-6'd3 // 8-bit   negative number stored as 2's complement of 3
-6'sd3 // Used for performing signed integer math
4'd-2 // Illegal specification
```

A question mark "?" is the Verilog HDL **alternative for z** in the context of numbers.

Strings

A string is a sequence of characters that are enclosed by double quotes.

"Hello Verilog World" // is a string

LEXICAL CONVENTIONS(CONT..)

Identifiers and Keywords

- ✓ **Keywords** are special identifiers reserved to define the language constructs.
- ✓ **Identifiers** are names given to objects

reg value; // reg is a keyword; value is an identifier

input clk; // input is a keyword, clk is an identifier and be referenced in the design.

VALUE SET IN VERILOG

Value Set - Verilog supports four values and eight strengths to model the functionality of real hardware.

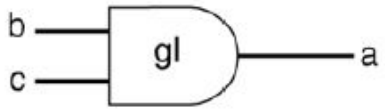
Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

DATA TYPES

DATA TYPES



Net : Connection between Hardware elements.
Take a value from a driver such as gate. e.g Wire a

- Nets
  - Any hardware connection points
- Variables
  - Any data storage elements
    - ✓ **In Verilog, the term register merely means a variable that can hold a value.**
    - ✓ Behave exactly like memory in a computer
    - ✓ Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do.
    - ✓ Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Types of Variable Data Types

- reg
- integer
- time
- real
- realtime

The reg Variable

- Hold a value between assignments
- May be used to model hardware registers
- Examples

```
reg a, b;
reg [7:0] data_a;
reg [0:7] data_b;
reg signed [7:0] d;
```

The integer Variable

- Contains integer values
- Has at least 32 bits
- Is treated as a signed reg variable with the LSB being bit 0

```
integer i, j;
integer data[7:0];
```

The time Variable

- Used for storing and manipulating simulation time
- Used in conjunction with the $time system task
- Only unsigned value and at least 64 bits, with the LSB being bit 0

```
time events;
time current_time;
```

Memory

- Memory
  - model ROM, RAM, or register files
- Reference
  - a whole word or
  - a portion of a word of memory

```
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)
membyte[511] // Fetches 1 byte word whose address is 511.
```

Parameters

Verilog allows constants to be defined in a module by the keyword parameter.

```
parameter port_id = 5; // Defines a constant port_id
parameter cache_line_width = 256; // Constant defines width of cache
line
parameter signed [15:0] WIDTH; // Fixed sign and range for parameter
                               // WIDTH
```

Strings

Strings can be stored in reg.

```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide
initial
   string_value = "Hello Verilog World"; // String can be stored
                                         // in variable
```

OPERATORS

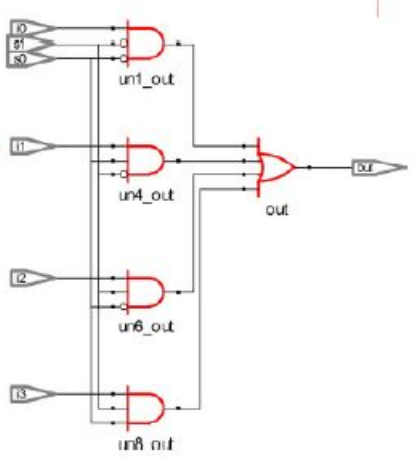
Operators

Arithmetic	Bitwise	Reduction	Relational
+: add - : subtract * : multiply / : divide % : modulus **: exponent	~ : NOT &: AND | : OR ^: XOR ~^, ^~: XNOR	&: AND |: OR ~&: NAND ~|: NOR ^: XOR ~^, ^~: XNOR	>: greater than <: less than >= : greater than or equal <=: less than or equal
Shift	**case equality**	**Logical**	**Miscellaneous**
<< : left shift >> : right shift <<< : arithmetic left shift >>>: arithmetic right shift	===: equality !==: inequality	&&: AND || : OR ! : NOT	{ , }: concatenation {c{ }}: replication ? : conditional
	Equality		
	==: equality !=: inequality		

Bitwise Operators

- A bit-by-bit operation
  - A z is treated as x
  - The shorter operand is zero-extended

```
// using basic and, or , not logic operators
assign out =   (~s1 & ~s0 & i0) |
               (~s1 & s0 & i1)  |
               (s1 & ~s0 & i2)  |
               (s1 & s0 & i3) ;
```



Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Reduction Operators

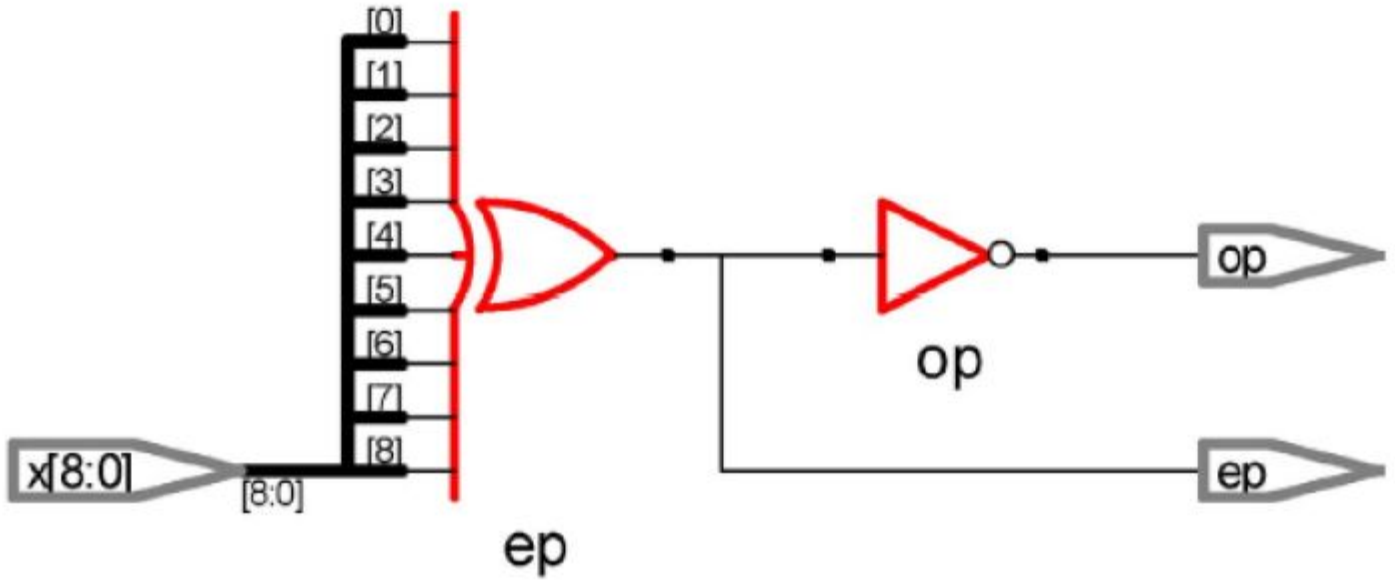
- Perform only on one vector operand
  - Carry out a bit-wise operation
  - Yield a 1-bit result
  - Work bit by bit from right to left

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

```
// X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.
```

A 9-Bit Parity Generator

```
// dataflow modeling using reduction operator
assign ep = ^x;    // even parity generator
assign op = ~ep;   // odd parity generator
```



Logical Operators

- Always evaluate to a 1-bit value, 0, 1, or x
- The result is x (a false condition) if any operand bit is x or z

Symbol	Operation
!	Logical negation
&&	Logical and
\|\| | Logical or |

Expressions connected by && and || are evaluated from left to right

DIFFERENCE – BITWISE, REDUCTION, LOGICAL

Bitwise Operator	Reduction Operator	Logical Operator
perform a bit-by-bit operation on two operands	Reduction operators take only one operand.	Logical operators take variables or expressions as operands. Expressions connected by && and || are evaluated from left to right
// X = 4'b1010, Y = 4'b0000 X | Y // bitwise operation. Result is 4'b1010	// X = 4'b1010 |X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1	X || Y // logical operation. Equivalent to 1 || 0. Result is 1. If (J=1'b0 && K=1'b0)

Arithmetic Operators

- The result is x if any operand bit has a value x
- Negative numbers are represented as 2's complement

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

CONCATENATION/REPLICATION OPERATORS

- **Concatenation operator**

y = {a, b[0], c[1]};

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

- **Replication operator -** Repetitive concatenation of the same number can be expressed by using a replication constant.

y = {a, {4{b[0]}}, c[1]};

A replication constant specifies how many times to replicate the number inside the brackets ({ }).

REPLICATION OPERATOR - EXAMPLE

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

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

Symbol	Operation
{ , }	Concatenation
{const_expr{}}	Replication

Relational Operators

- The result is 1 if the expression is true and 0 if the expression is false
  - Return x if any operand bit is x or z

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Equality Operators

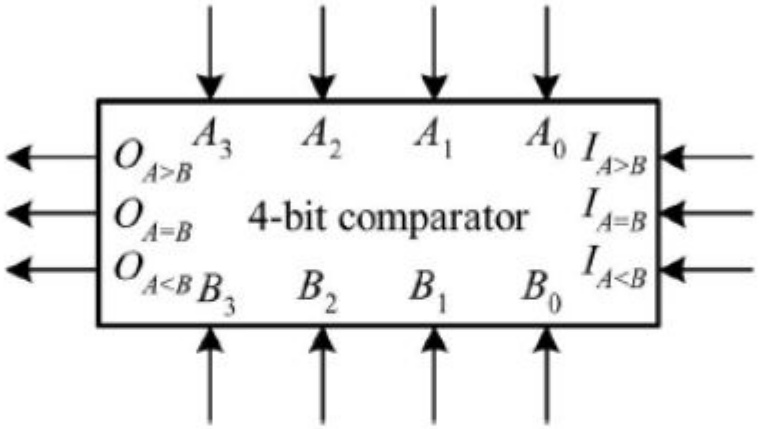
- Compare the two operands bit by bit
  - The shorter operand is zero-extended
  - Return 1 if the expression is true and 0 if the expression is false

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

DIFFERENCE BETWEEN THE LOGICAL EQUALITY OPERATORS AND CASE EQUALITY OPERATORS

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

A 4-B Magnitude Comparator




```
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);                  // =
   assign Oagtb  = (a > b) || ((a == b)&& (Iagtb == 1));     // >
   assign Oaltb   = (a < b) || ((a == b)&& (Ialtb == 1));    // <
```

Shift Operators

- Logical shift operators
- Arithmetic shift operators

When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around.

sign-extended shifts - **only** actually performs an arithmetic shift if **the first operand is signed.** If the first operand is unsigned, the operator actually performs a *logical* right shift.

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

ARITHMETIC SHIFT OPERATORS

- These vary from the basic shift operators because they perform **sign-extended** shifts.
- When **shifting left,** it performs exactly 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.**

Take a look at this example.

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

Now **b will have the value 5'b10000** just like before,

but **c will have the value 5'b11101.**

WHAT IS THE NEED OF ARITHMETIC OPERATOR ?

- The reason is because shifting bits is a very cheap way to perform multiplication and division by powers of two.
- **Take the value 6 (4'b0110)**

  If we shift it to the left one bit we get 4'b1100 or 12(i.e 6 * 2 = 12) and

  If we shift it to the right one bit we get 4'b0011 or 3(i.e 6/2 = 3).

- **what about the value -4 (4'b1100)?**

  If we shift it to the left one bit we get 4'b1000 or -8.

  However if we shift it to the right one bit we get 4'b0110 or 6!

but if we use the arithmetic shift we get 4'b1110 or -2!

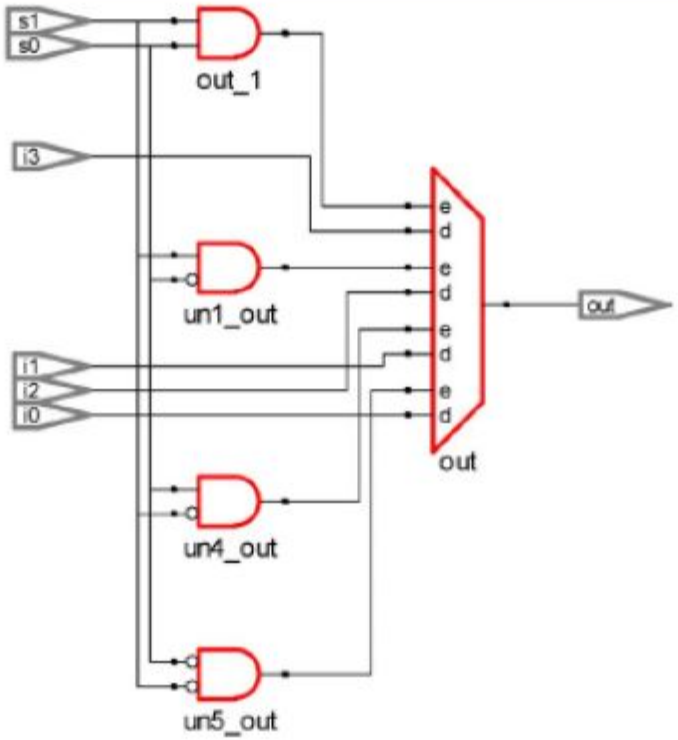
The Conditional Operator

- Usage: condition_expr ? true_expr: false_expr;
  - If condition_expr = x or z: the result = true_expr & false_expr (0 and 0 gets 0, 1 and 1 gets 1, others gets x)
- For example

```
assign out = selection ? in_1: in_0;
```

An Example --- A 4-to-1 MUX

```
// using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
```



STRUCTURAL MODELING

INSTANTIATING A MODULE

Instances of

module mymod(y, a, b);

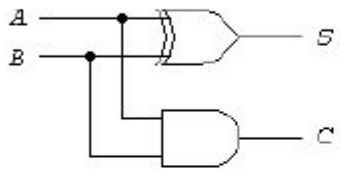
Instantiating the component mymod in our programe

```
mymod  g1(y1, a1, b1);          // Connect-by-position : Positional Association
mymod (y2, a1, b1),
      (y3, a2, b2);             // Instance names omitted
```

or

```
mymod g1(.a(a2), .b(b2), .y(c2));  // Connect-by-name : Named Association
```

HALF ADDER DESIGN - VERILOG



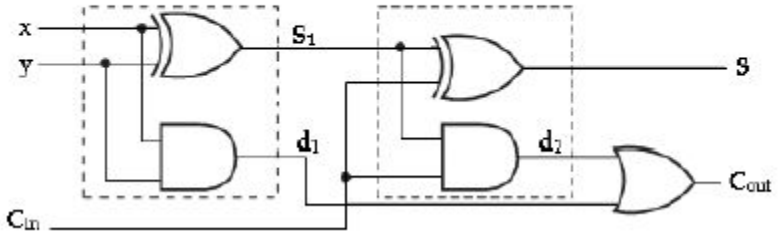
Dataflow model

```
module halfadder(s,c,a,b);
input b;
assign s= a ^ b;
assign c= a & b;
endmodule
```

Gate-level description Half Adder

```
module halfadder(s,co,x,y);
input x,y;
output s,co;

//Instatiate primitive gates

xor (s,x,y);
and (co,x,y);
endmodule
```

FULL ADDER



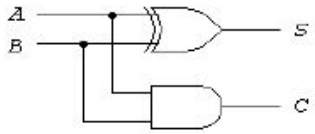

```
//Description of Full Adder
module fulladder(s,co,x,y,ci);
input x,y,ci;
output s,co;
wire s1,d1,d2;

//Instantiate Half Adder
halfadder ha_1(s1,d1,x,y);
halfadder ha_2(s,d2,s1,ci);
or or_gate(co,d2,d1);

endmodule
```

```
//Stimulus for testing Full Adder
module Fa_tb;
reg x,y,ci;
wire s,co;

//Instantiate Full Adder
fulladder f1(s,co,x,y,ci);
initial
begin
x=1'b0; y=1'b0; ci=1'b0;
#100 x=1'b0; y=1'b0; ci=1'b1;
#100 x=1'b0; y=1'b1; ci=1'b0;
#100 x=1'b0; y=1'b1; ci=1'b1;
#100 x=1'b1; y=1'b0; ci=1'b0;
#100 x=1'b1; y=1'b0; ci=1'b1;
#100 x=1'b1; y=1'b1; ci=1'b0;
#100 x=1'b1; y=1'b1; ci=1'b1;
end
endmodule
```

TEST BENCH – HALF ADDER

Verilog Code – Half Adder

```
module halfadder(s,c,a,b);
input b;
assign s= a ^ b;
assign c= a & b;
endmodule;
```

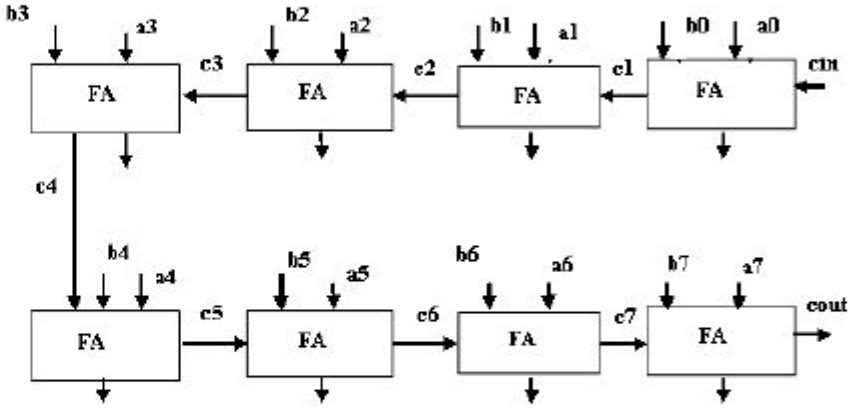


Test Bench – Half Adder (stimulus block)

```
module ha_tb;
reg a;      // input is declared as reg data type
reg b;
wire sum;    // output is declared as wire data type
wire carry;
halfadder m1( sum,carry,a,b);
Initial
Begin
a=1'b0; b=1'b0;
#10 a=1'b0; b=1'b1;
#10 a=1'b1; b=1'b0;
#10 a=1'b1; b=1'b1;

$finish;
end
endmodule
```

RIPPLE CARRY ADDER

```
module ripplecarryadder(s,cout,a,b,cin);
output[7:0]s;
output cout;
input[7:0]a,b;
input cin;
wire c1,c2,c3,c4,c5,c6,c7;
fulladd fa0(s[0],c1,a[0],b[0],cin);
fulladd fa1(s[1],c2,a[1],b[1],c1);
fulladd fa2(s[2],c3,a[2],b[2],c2);
fulladd fa3(s[3],c4,a[3],b[3],c3);
fulladd fa4(s[4],c5,a[4],b[4],c4);
fulladd fa5(s[5],c6,a[5],b[5],c5);
fulladd fa6(s[6],c7,a[6],b[6],c6);
fulladd fa7(s[7],cout,a[7],b[7],c7);
endmodule
```



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

Types of UDP

1. Combinational UDP

2. Sequential UDP
    i) Level Sensitive UDP
    ii) Edge Sensitive UDP

COMBINATIONAL UPD

```
//To generate a Carry for an Adder
primitive carry(out, a, b, c);
output out;
input a, b, c;
table
  00? : 0;
  0?0 : 0;
  ?00 : 0;
  11? : 1;
  1?1 : 1;
  ?11 : 1;
endtable
endprimitive
```

Always have exactly one output

Truth table may include don't-care (?) entries

SEQUENTIAL UPD – LEVEL SENSITIVE

```
//Define level-sensitive latch by using UDP.
primitive latch(q, d, clock, clear);
output q;
reg q;
input d, clock, clear;
initial
q = 0;
table
//d clock clear : q : q+ ;
  ?   ?      1   : ? : 0 ; //clear condition;
//q+ is the new output value
   1  1      0  : ? : 1 ; //latch q = data = 1
   0  1      0  : ? : 0 ; //latch q = data = 0
   ?  0      0 : ? : - ; //retain original state if clock = 0
endtable
endprimitive
```

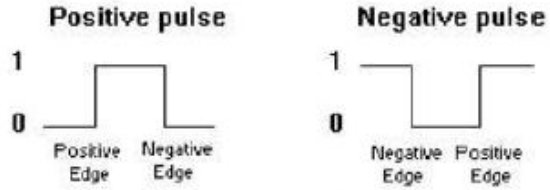
(the dash "-" symbol is used to denote no change in the state of the latch)

SEQUENTIAL UPD – EDGE SENSITIVE

//To design D Flip Flop for positive edge trigger Clock input and active low Clear

```
Primitive dff( q, clk, data);
output q; reg q;
input clk, data;
table
// clk data q   new-q
  (01)   0  : ? :   0;      // Latch a 0
  (01)   1  : ? :   1;      // Latch a 1
  (0x)   1  : 1 :   1;      // Hold when d and q both 1
  (0x)   0  : 0 :   0;      // Hold when d and q both 0
  (?0)   ?  : ? :   -;      // Hold when clk falls
  ?    (??) : ? :   -;      // Hold when clk stable
endtable
endprimitive
```

(01) – Positive Edge Triggered clock
(10) – Negative Edge Triggered clock



Definition of clock pulse transition

GENERATE BLOCK

- **generate blocks** are simply a convenient way of replacing multiple repetitive Verilog statements with a single statement inside a loop.
- **genvar** is a keyword used to declare variables that are used only in the evaluation of generate block. genvars do not exist during simulation of the design.
- Generate loops can be nested. However, two generate loops using the same genvar as an index variable cannot be nested.
- The name xor_loop assigned to the generate loop is used for hierarchical name referencing of the variables inside the generate loop. Therefore, the relative hierarchical names of the xor gates will be xor_loop[0].g1, xor_loop[1].g1, ..., xor_loop[31].g1.

FULL ADDER USING GENERATE STATEMENT

```
module fa(cin,x,y,s,cout
    );
parameter n=32;
input cin; //carry input
input [n-1:0]x,y;//numbers to be added
output [n-1:0]s;//sum
output cout;//carry out
wire [n:0]c;

genvar k; // generate 32 full adders with k as variable
assign c[0]=cin;
assign cout=c[n];

generate // generate 32 full adders
for(k=0;k<n;k=k+1)
begin:fa
wire z1,z2,z3;
xor(s[k],x[k],y[k],c[k]);
and(z1,x[k],y[k]);
and(z2,x[k],c[k]);
and(z3,c[k],y[k]);
or(c[k+1],z1,z2,z3);
end
endgenerate
endmodule
```

genvar is a keyword used to declare variables that are used only in the evaluation of generate block.

TEST BENCH FOR FULL ADDER

```
module test_gen;

        // Inputs
        reg cin;
        reg [31:0] x;
        reg [31:0] y;

        // Outputs
        wire [31:0] s;
        wire cout;

        // Instantiate the Unit Under Test (UUT)
        fa uut (
                .cin(cin),
                .x(x),
                .y(y),
                .s(s),
                .cout(cout)
        );
```

```
initial begin
        // Initialize Inputs
        cin = 0;
        x = 0;
        y = 0;
        #100;


        cin = 0;
        x = 32'd20;
        y = 32'd30;
        #100;

        cin = 0;
        x = 32'd25;
        y = 32'd56;
        #100;

          cin = 0;
          x = 32'd25;
          y = 32'd56;
          #100;

          cin = 0;
          x = 32'd74;
          y = 32'd46;
          #100;

          // Add stimulus here

    end

endmodule
```

GENERATE BLOCK STRUCTURE

- The keywords used
  - generate and endgenerate

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
genvar i;
generate for (i = 0; i < SIZE; i = i + 1) begin: bit
     assign bin[i] = ^gray[SIZE-1:i];
end endgenerate
```

N-BIT ADDER USING GENERATE

genvar is a keyword used to declare variables that are used only in the evaluation of generate block.

```
// define a full adder at dataflow level.
module full_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;
// Specify the function of a full adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar  i;
wire  [N-2:0] c;      // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
  if (i == 0)          // specify LSB
    full_adder fa (x[i], y[i], c_in, sum[i], c[i]);
  else if (i == N-1) // specify MSB
    full_adder fa (x[i], y[i], c[i-1], sum[i], c_out);
  else                  // specify other bits
    full_adder fa (x[i], y[i], c[i-1], sum[i], c[i]);
end endgenerate
```

BEHAVIORAL MODEL

PROCEDURAL CONSTRUCTS – INITIAL AND ALWAYS BLOCKS

Basic components for behavioral modeling

initial

```
begin
  … imperative statements …
end
```

- ✓ Runs when simulation starts
- ✓ Terminates when control reaches the end
- ✓ Good for providing stimulus

always

```
begin
  … imperative statements …
end
```

- ✓ Runs when simulation starts
- ✓ Restarts when control reaches the end
- ✓ Good for modeling/specifying hardware

IMPERATIVE STATEMENTS

```
if (select == 1)  y = a;
else                          y = b;


case (op)
  2'b00: y = a + b;
  2'b01: y = a – b;
  2'b10: y = a ^ b;
  default: y = 'hxxxx;
endcase
```

WHILE LOOPS

A increasing sequence of values on an output

```
reg [3:0] i, output;

i = 0;
while (I <= 15) begin
  output = i;
  #10 i = i + 1;
end
```

FOR LOOP

A increasing sequence of values on an output

```
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end
```

MODELING A FLIP-FLOP WITH ALWAYS

Very basic: an edge-sensitive flip-flop

```
reg q;

always @(posedge clk)
  q = d;
```

q = d assignment runs when clock rises: exactly the behavior you expect

BLOCKING VS. NONBLOCKING ASSIGNMENTS

Verilog has two types of procedural assignment

Fundamental problem:

- In a synchronous system, all flip-flops sample simultaneously
- In Verilog, always @(posedge clk) blocks run in some undefined sequence

A FLAWED SHIFT REGISTER

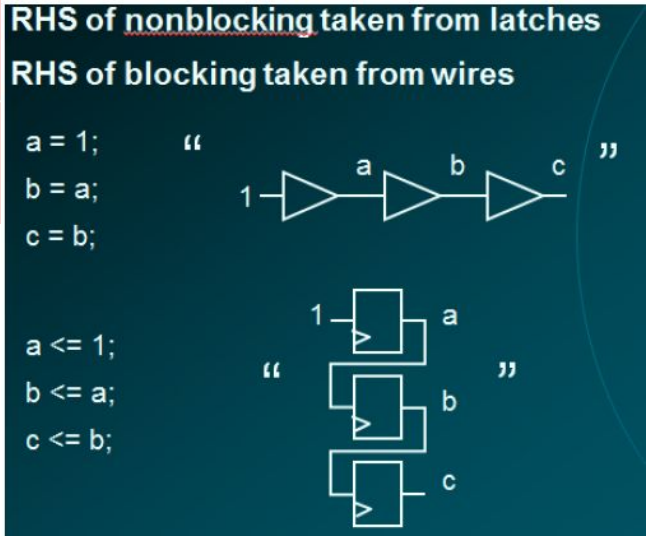
This doesn't work as you'd expect:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

These run in some order, but you don't know which

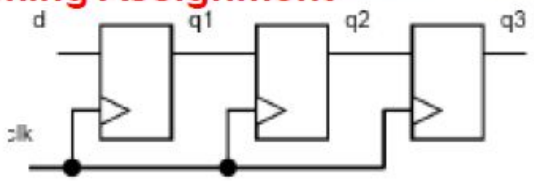
DIFFERENCE BETWEEN BLOCKING AND NON BLOCKING

Blocking Assignments	Non Blocking Assignments
Blocking assignment statements are executed in the order they are specified in a sequential block.	Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
It executes only after the preceding statement has completed execution.	It executes concurrently(parallely), so that order in which they are listed implies no precedence of any kind and has no effect.
The = operator is used to specify blocking assignments.	The <= operator is used to specify blocking assignments.
RHS of blocking taken from wires	RHS of nonblocking taken from latches



BLOCKING AND NON BLOCKING ASSIGNMENTS - EXAMPLE

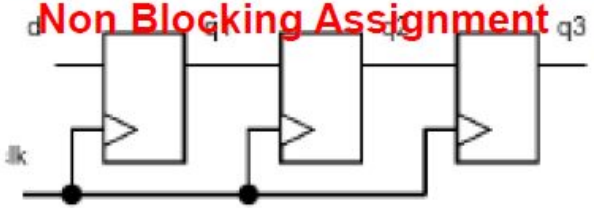
Blocking Assignment




```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Simulation – √
Synthesis – √

Non Blocking Assignment




```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

Simulation – √
Synthesis – √

GUIDELINES FOR BLOCKING AND NON BLOCKING ASSIGNMENTS

If blocking assignment is used to model a D-FF:

Does it work?
It does. But it is not good habit.

It is much safer to use nonblocking assignment to do this:

```
module dffb (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q = 1'b0;
    else     q = d;
endmodule

module dffx (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q <= 1'b0;
    else     q <= d;
endmodule
```

Guideline #1: When modeling sequential logic, use nonblocking assignments
Guideline #2: When modeling latches, use nonblocking assignments

FSM WITH COMBINATIONAL LOGIC

```
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;

always @(a or b or state)
 case (state)
   2'b00: begin
     nextState = a ? 2'b00 : 2'b01;
     o = a & b;
   end
   2'b01: begin nextState = 2'b10; o = 0; end
 endcase
```

Output o is declared a reg because it is assigned procedurally, not because it holds state

Combinational block must be sensitive to any change on any of its inputs (Implies state-holding elements otherwise)

TASKS AND FUNCTIONS

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

TASK - EXAMPLE

```
//Defining a task to do bitwise operation
task bitwise_oper (output [15:0] ab_and, ab_or,
ab_xor, input [15:0] a, b);
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
```

```
// Calling a task in the Top module
module operation;
...
...
parameter delay = 10;

reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) //whenever A or B changes in value

begin
bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);

end
enmodule
```

FUNCTION - EXAMPLE

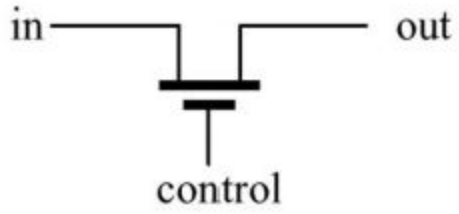
```
//Function Declaration
function calc_parity;
input [31:0] address;
begin
calc_parity = ^address; //Return the xor
of all address bits.
end
endfunction
```

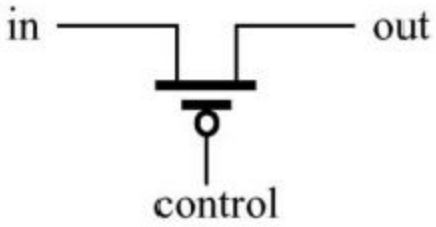
```
//Calling a function in a top module

module parity;
...
reg [31:0] addr;
reg parity;

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

SWITCH LEVEL MODELLING

The nmos and pmos Switches

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



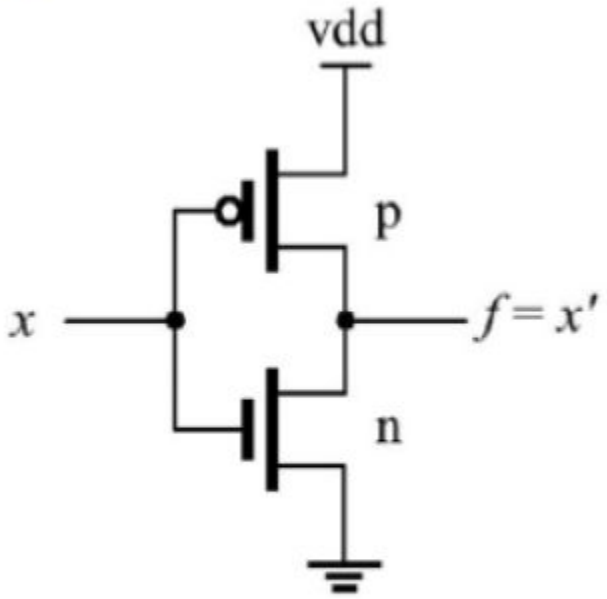


nmos		control			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

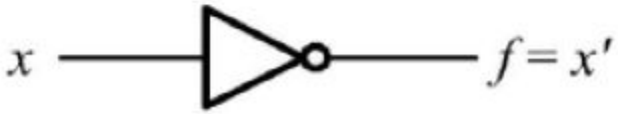
pmos		control			
		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

Example : The CMOS Inverter

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



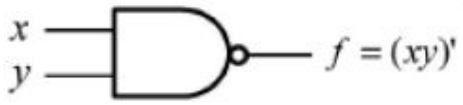
(a) Circuit



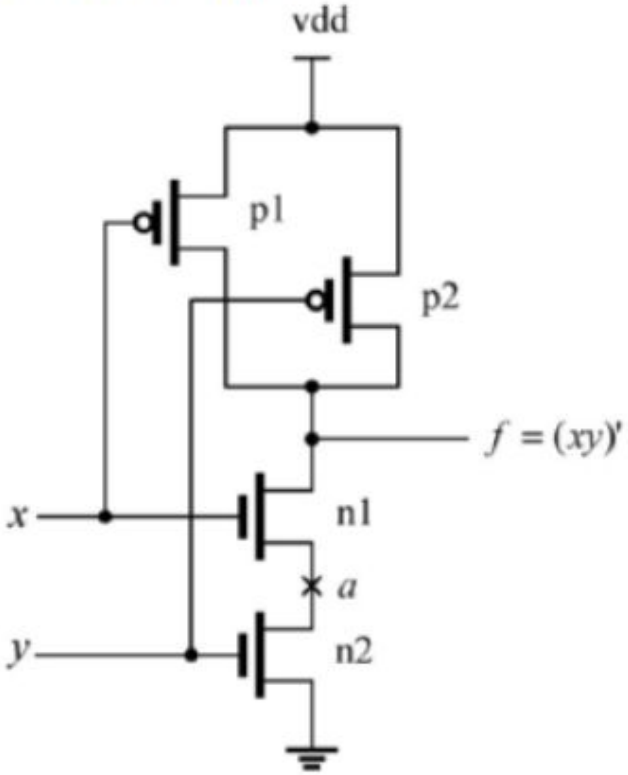
(b) Logic symbol

Example : CMOS NAND Gate

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



(b) Logic symbol



(a) Circuit