

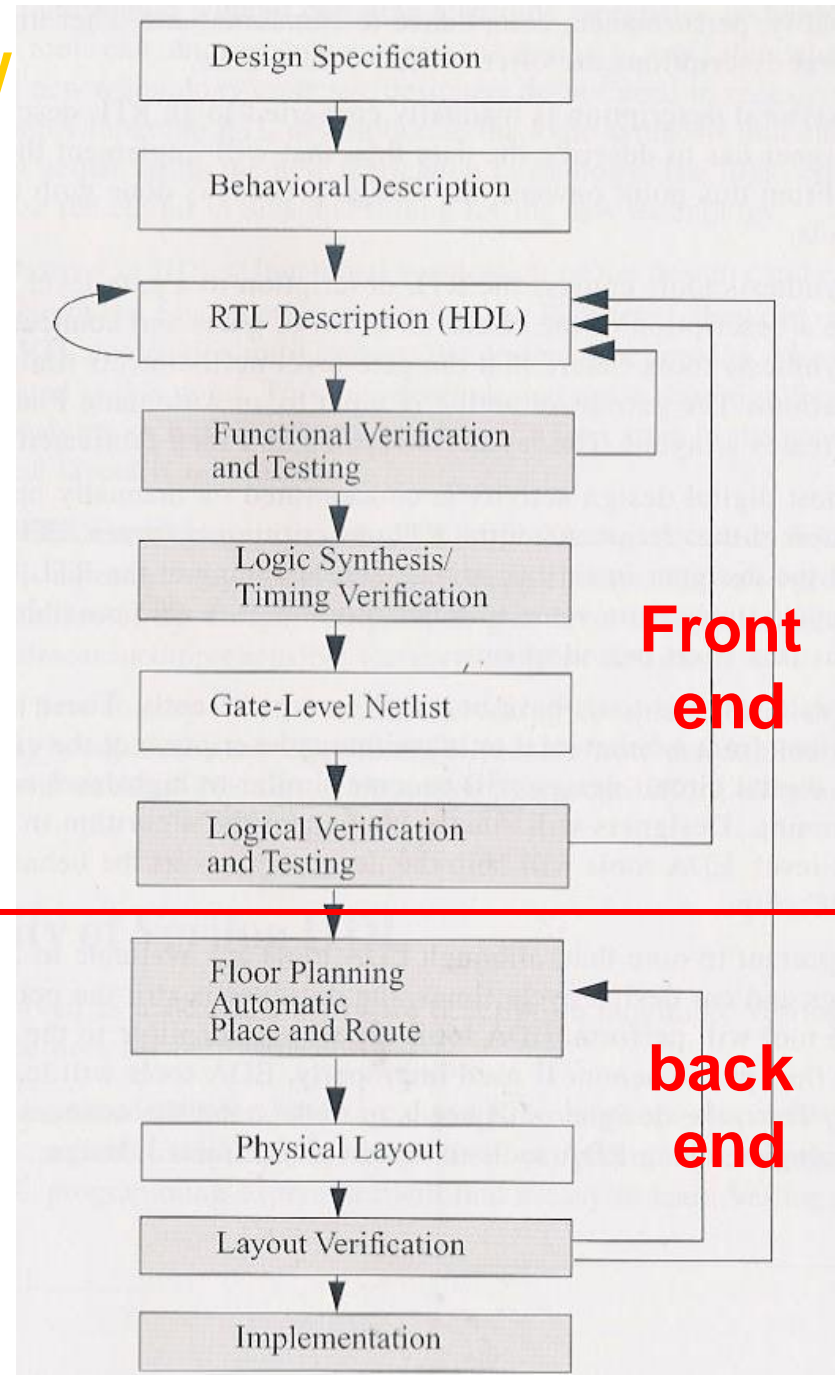
Verilog: structural modeling, dataflow modeling

S. Palnitkar, “***Verilog HDL, A Guide to Digital Design and Synthesis***”, 2nd ed., , Sun Microsystems, Inc, 2003.

Overview

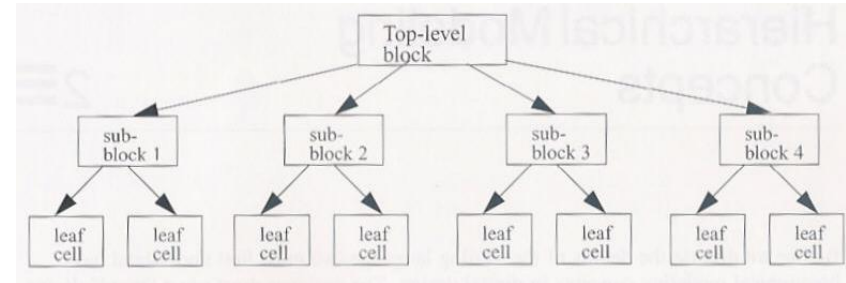
Typical Design Flow

- Front-end
 - RTL coding/simulation
 - Synthesis
 - Gate-level simulation (post-synthesis simulation)
- Back-end (ASIC)
 - Place and route (P&R)
 - Post-layout simulation
- Back-end (FPGA)
 - Mapping and implementation

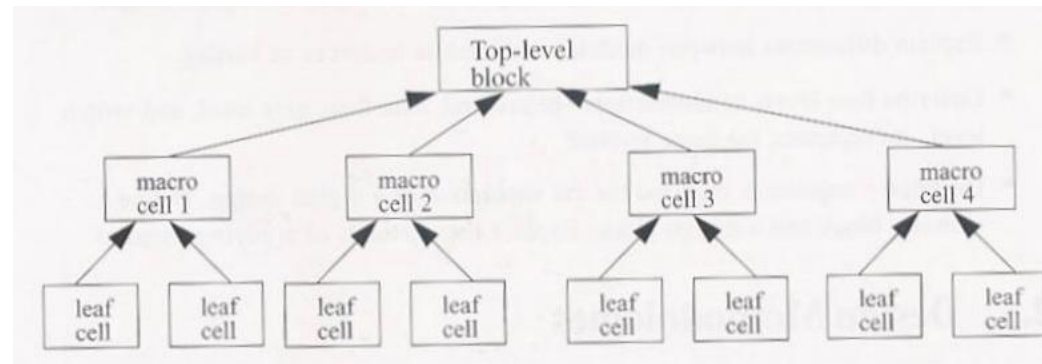


Design Methodologies

- top-down design
 - Start from RTL coding

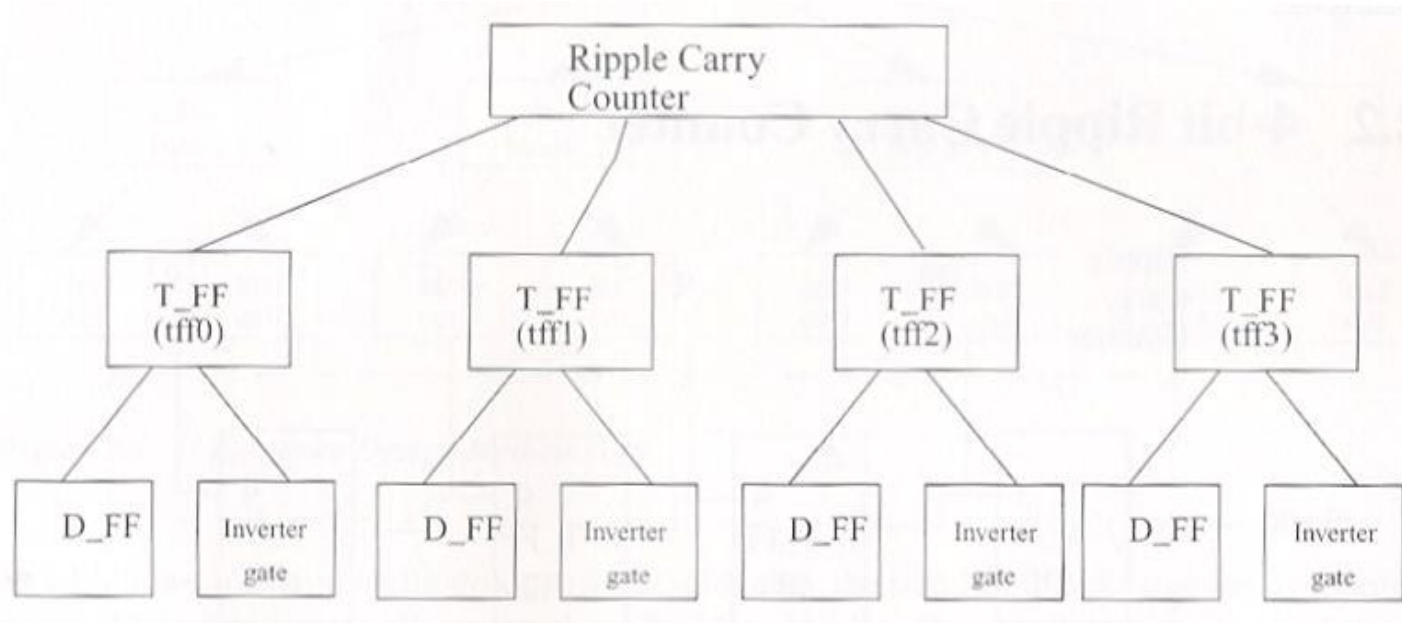
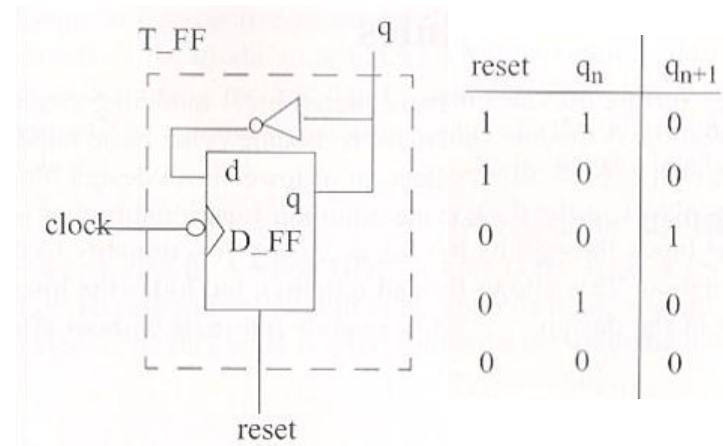
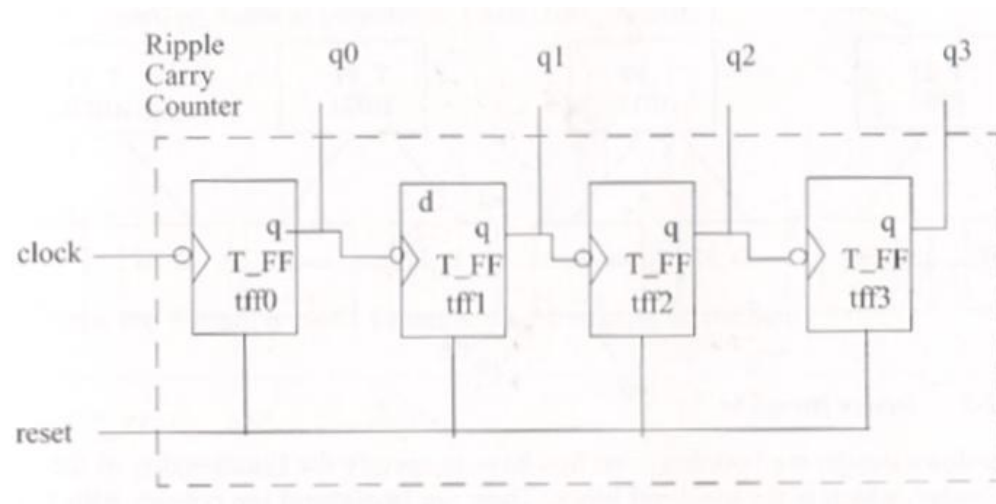


- bottom-up design
 - Start from design of basic components



- Most designs involve both top-down and bottom-up

Example: 4-b Ripple Carry Counter



Modules

- basic building block in Verilog
 - provide common functionality
- behavioral level
- structural level
- dataflow level
 - Continuous assignment
- gate level
- switch level (transistor-level)
- Verilog allows designs of mixed levels
- RTL (Register Transfer Level) Verilog
 - combination of behavioral, structural and dataflow levels
 - must be synthesizable (tool-dependent) for hardware modules

```
module T_FF (q, clock, reset);  
... <functionality of TFF>  
endmodule
```

module definition

```
module MyCPU(R_Wb, Data, Address, Clock, Reset);  
    input  Reset, Clock;  
    input  [31:0] Address;  
    inout  [31:0] Data;  
    output R_Wb;
```

} I/O port declarations

```
    .  
    .  
    .
```

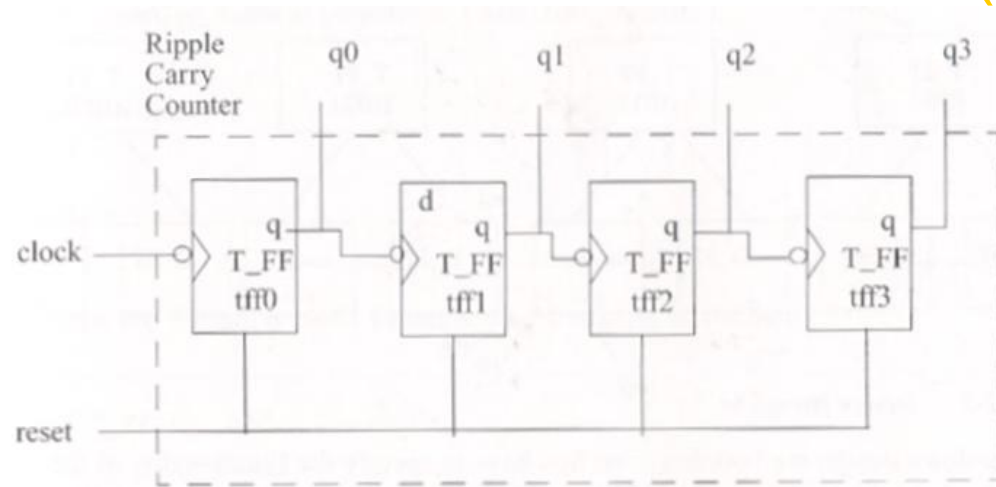
} Resource / variable declarations

```
    .  
    .  
    .
```

} Structural / behavioral modeling

```
endmodule
```

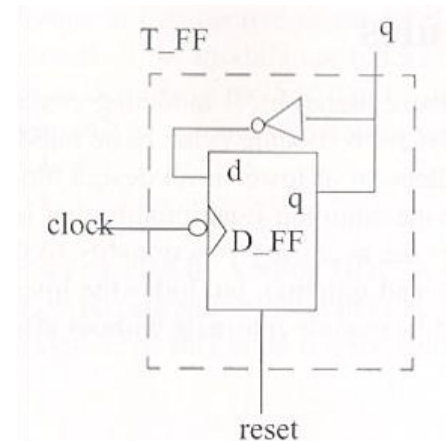
Module Instantiation (structural modeling)



```
module ripple_carry_counter (q, clk, rst);  
output [3:0] q;  
input clk, rst;
```

```
  T_FF tff0 (q[0], clk, rst);  
  T_FF tff1 (q[1], clk, rst);  
  T_FF tff2 (q[2], clk, rst);  
  T_FF tff3 (q[3], clk, rst);
```

```
endmodule
```



```
module T_FF (q, clk, rst);  
output q;  
input clk, rst;
```

```
wire d;
```

```
// instantiate a D_FF.  
// call it dff0
```

```
  D_FF dff0 (q, d, clk, rst);  
  not n1(d, q);  
endmodule
```


structural vs. behavioral modeling

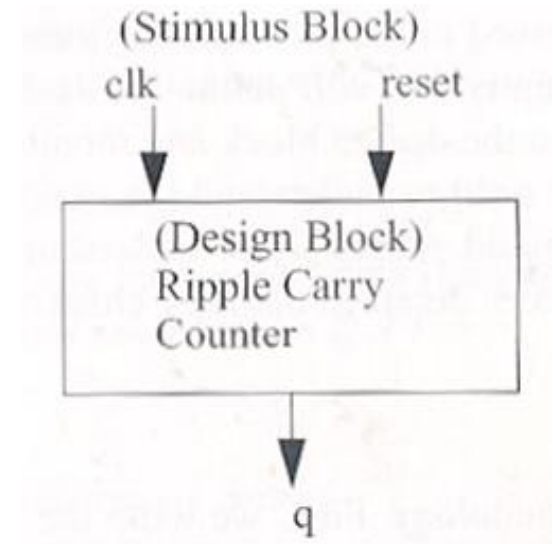
- structural modeling, eg. module T_FF
- behavioral modeling, eg. module D_FF

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

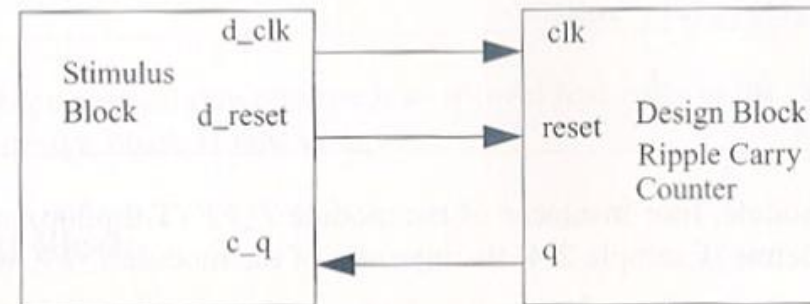
```
module T_FF (q, clk, rst);  
output q;  
input clk, rst;  
  
wire d;  
  
// instantiate a D_FF.  
// call it dff0  
D_FF dff0 (q, d, clk, rst);  
not n1(d, q);  
endmodule
```

two styles of simulation constructs

- simulation block (also called **test bench**, or **test fixture**) instantiates design block and then apply stimulus (test patterns or test vectors) and observe/compare the output
- stimulus block and design blocks are instantiated in a top-level module
 - stimulus block (for simulation only) can use *non-synthesizable* statements
 - design block (for hardware design) must use only *synthesizable* code



Top-Level Block



```
module stimulus;
```

```
reg clk;
```

```
reg rst;
```

```
wire [3:0] q;
```

```
/* instantiate the design block */
```

```
/* positional port mapping */
```

```
ripple_carry_counter r1(q, clk, rst);
```

```
/* generate the input waveform */
```

```
initial clk = 1'b0;
```

```
always #5 clk = ~clk;
```

```
initial
```

```
begin
```

```
    rst = 1'b1;
```

```
    #15 rst = 1'b0;
```

```
    #180 rst = 1'b1;
```

```
    #10 rst = 1'b0;
```

```
    #20 $finish;
```

```
end
```

```
/* display the simulation results */
```

```
initial $monitor ($time, "Output q = %d", q);
```

```
endmodule
```

Simulation Example

```
module testfixture;
```

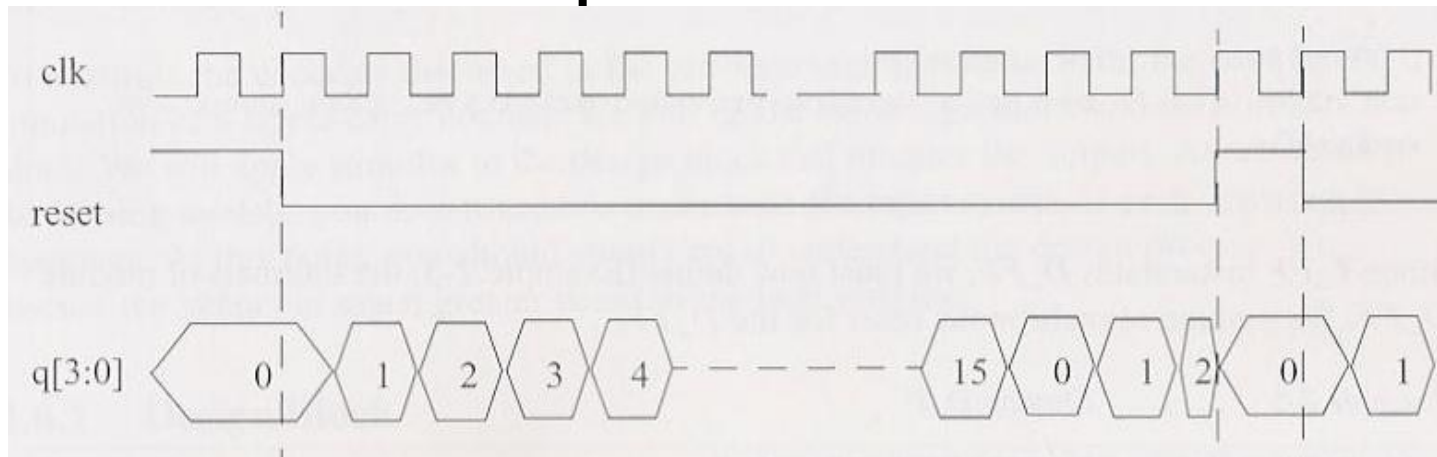
```
// data type declaration
```

```
// instantiate design block
```

```
// apply stimulus
```

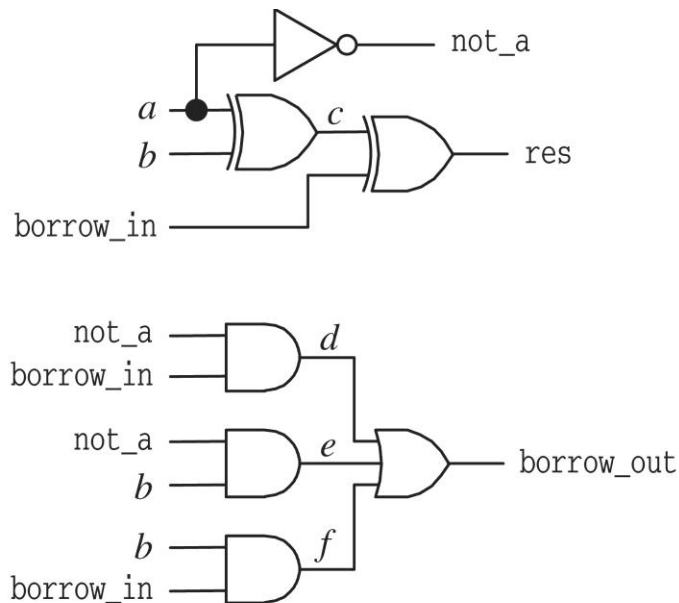
```
// display results
```

```
endmodule
```



Another example (full subtractor)

```
module full_sub_behave (res, borrow_out, a, b, borrow_in);  
  output res;          // result of the subtraction  
  output borrow_out; // the borrow-out value  
  input a;             // the value to subtract  
  input b;             // the value to be subtracted  
  input borrow_in;    // the borrow-out of the "previous" stage  
  
  assign {borrow_out, res} = a - b - borrow_in;  
          // result of subtraction is two bits; the  
          // msb is "borrow_out" and the lsb is "res".  
endmodule
```



```
module full_sub_struct (res, borrow_out, a, b, borrow_in);  
  output res;          // subtraction result  
  output borrow_out; // the borrow-out value  
  input a;             // the input to subtract  
  input b;             // the input to be subtracted  
  input borrow_in;    // the borrow-out from the previous stage  
  
  xor (c, a, b);      // uses XOR gate library part  
  xor (res, c, borrow_in); // res = a XOR b XOR borrow_in  
  not (not_a, a);  
  and (d, not_a, borrow_in);  
  and (e, not_a, b);  
  and (f, b, borrow_in);  
  or (borrow_out, d, e, f); // borrow_out = a'w + a'b + bw,  
endmodule
```

```
`timescale 1ns/100ps // time unit = 1ns, precision = 100ps
```

```
module test_sub();  
  reg a, b, borrow_in; // inputs to full_sub_struct  
  wire res, borrow_out; // outputs of full_sub_struct  
  reg expected_res, expected_bo; // expected results  
  integer error_count; // number of errors
```

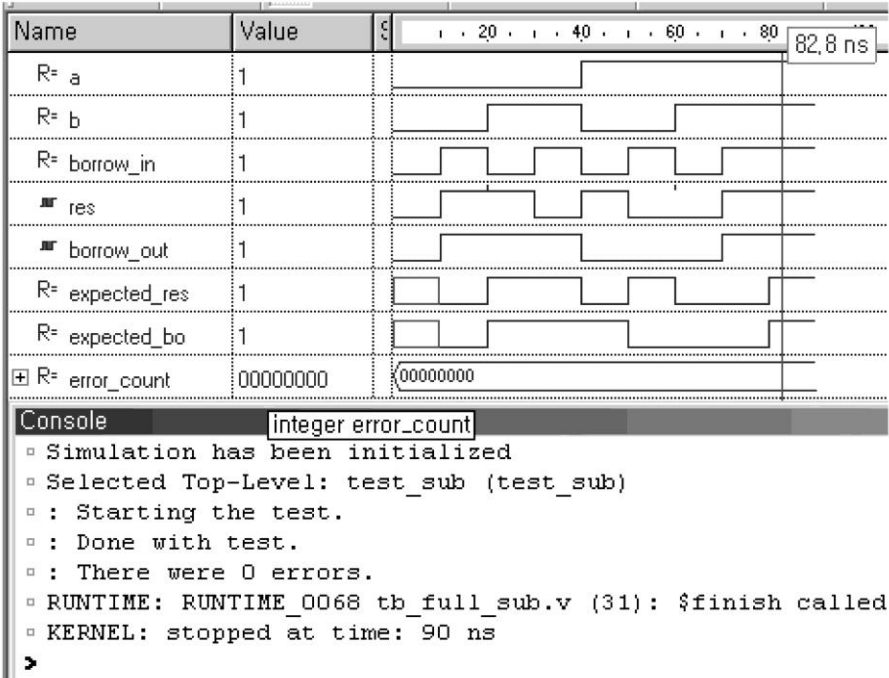
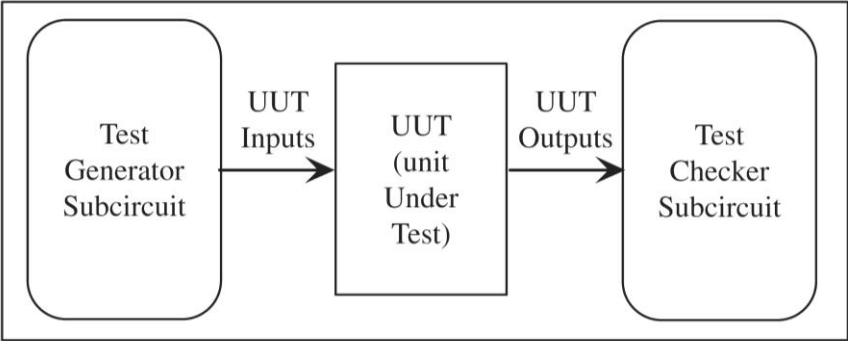
```
// make connections to the unit under test (UUT)  
full_sub_struct U1 (res, borrow_out, a, b, borrow_in);
```

```
initial begin // one-time execution block  
  $display("Starting the test.");  
  error_count = 0;  
  {a, b, borrow_in} = 3'b000;  
end  
  
always begin // repeated execution block  
  #10; // wait for 10 * 1ns = 10ns  
  {expected_bo, expected_res} = a - b - borrow_in;  
  if ({expected_bo, expected_res} != {borrow_out, res}) begin  
    $display("Expected (%b, %b) != actual (%b, %b) at time %0t.",  
      expected_bo, expected_res, borrow_out, res, $time);  
    error_count = error_count + 1;  
  end  
  if ({a, b, borrow_in} === 3'b111) begin  
    #10 $display("Done with test.");  
    $display("There were %0d errors.", error_count);  
    $finish;  
  end  
  else // compute next test vector  
    {a, b, borrow_in} = {a, b, borrow_in} + 1;  
end // of main always block
```

```
endmodule
```

Test bench

Test Bench Circuit



Basic Concept

Verilog Operators

`a = ~b; // not b (unary operator)`

only one operand: b

`a = b && c; // logical operation (binary operator)`

two operands: b, c

`a = b ? c : d; // (ternary operator)`

`// a=c if b=1,`

`// a=d if b=0`

three operands: b, c, d

Other operators

- arithmetic operators (+, -, *, /, %, **)
- relational operators (<, >, <=, >=)
- equality operators (==, !=, ===, !==)
- logical operators (&&, ||, !)
- bitwise operators (~, &, |, ^, ~^)
- reduction operators (&, ~&, , |, ~|, , ^, ~^)
- shift operators (>>, <<, >>>, <<<)
- concatenation operator ({ })
- conditional operator (? :)

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

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	Three

Equality Operator

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

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

reduction operator

- perform bitwise operator on a single vector operand and yield a 1-bit result

- eg.

$X = 4'b1010$

$\&X \quad // = 1\&0\&1\&0 = 1'b0$

$|X \quad // = 1|0|1|0 = 1'b1$

$\^X \quad // = 1\^0\^1\^0 = 1'b0$

Concatenation Operator

Replication Operator

- $A=1'b1$, $B=2'b01$, $C=2'b10$
- concatenation operator
 $Y=\{B,C\} = 4'b0110$
- replication operator
 $Y=\{ 4\{A\}, 2\{B\} \}=8'b11110101$

Number

unsized numbers (decimal by default)

sized numbers:

<size>'<base format> <number>

<size>: the number of bits

'<base>: 'd (decimal),

'h (hexadecimal),

'b (binary),

'o (octal)

' is the single quote symbol, **not** back quote symbol ` which will be used in compiler directive

e.g.

4'**b**1111 // 4-bit binary number $1111_2 = 15_{10}$


12'**h**abc // 12-bit hexadecimal number = $1010_1011_1100_2$

16'**d**255 // 16-bit decimal number = 0000000011111111_2

Data types: value set

- 4 values, 8 strengths

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

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

Four-Valued Logic System

- Signal and variable values represented using a 4-valued logic system
 - 1'b0: 1-bit binary logic 0 value
 - 1'b1: 1-bit binary logic 1 value
 - 1'bx: 1-bit binary 'x' or 'X' value
 - Unknown value, produced for uninitialized values or values driven to conflicting values by more than one signal source
 - 1'bz: 1-bit binary 'z' or 'Z' value
 - High impedance value, produced when a wire is **disconnected** from all signal sources driving that wire

data types (**wire** and **reg**)

- Nets (**wire**): connections between hardware elements
 - nets have values continuously by the outputs of devices that they are connected toeg.: **wire** a, b, c;
- Registers (**reg**): data storage element
 - a variable that can hold a value
 - *not necessarily* mean a flip-flop in real circuiteg.: **reg** reset;
- SystemVerilog use **logic** to replace **wire** and **reg** to avoid confusion
 - not all **reg** data types synthesize into hardware registers

data types (vectors)

- Vectors: multiple bit widths (default is scalar)
wire [7:0] bus; // 8-bit bus
wire [31:0] word; // 32-bit word
reg [255:0] data; // 256-bit word register, data[255] is MSB
reg [0:40] v_addr; // v_addr[0] is MSB
- Vector Part Select
word[7:0] // the least significant byte of the 32-bit vector word
v_addr[0:1] // two most significant bits of the vector v_addr
- variable vector part select
[<starting_bit> +: width] : part-select *increments*
[<starting_bit> -: width] : part-select *decrements*
starting bit can be varied, but the width must be constant

eg1.: **reg** [255:0] data1;
 reg [0:255] data2;
 reg [7:0] byte;

byte=data1[31 -:8] // data1[31:24]
byte=data1[24 +:8] // data1[31:24]
byte=data2[31 -:8] // data2[24:31]
byte=data2[24 +:8] // data2[24:31]

eg2. **reg** [255:0] data1;
 for (j=0; j<=31; j=j+1) byte = **data1[(j*8)+:8]** ;
 // sequence is [7:0], [15:8], ..., [255:248]

register data types (**integer**, **real**, **time**)

- **integer**: register data type used for quantity
 - **reg** store *unsigned* quantity while **integer** store *signed* quantity
 - eg., **integer** counter; counter = -1; // used as a counter
- **real**: real number constant or register data type
 - eg. **real** delta;
 - delta = 4e10; delta = 2.14; // delta is a real variable
- **time**: register data type to store simulation time
 - eg. **time** save_sim_time;
 - save_sim_time = **\$time**; // define a time variable
 - // system task **\$time** is invoked to get current simulation time

signed vs. unsigned

- unsigned

```
wire [3:0] x;  
wire [7:0] y;  
  
assign y = {4'b0000, x};
```

```
wire [3:0] x;  
wire [7:0] y;  
  
assign y = x; // zero padded
```

- signed

```
wire signed [3:0] x;  
wire signed [7:0] y;  
  
assign y = {4{x[3]}, x};
```

```
wire signed [3:0] x;  
wire signed [7:0] y;  
  
assign y = x; // sign-extended
```

```
wire [11:0] s1;  
wire signed [11:0] s2;  
  
assign s2 = $signed(s1); // convert to signed number  
// assign s1 = $unsigned(s2); // convert to unsigned number
```

Arithmetic (unsigned vs. signed)

- unsigned

```
wire [7:0] a, b, s;  
wire c;
```

```
assign s = a + b;    // no overflow detection  
// assign {c, s} = {1'b0, a} + {1'0, b};  
// assign {c, s} = a + b; // implicit extension by Verilog
```

- signed

```
wire signed [7:0] a, b;  
wire signed [8:0] s;
```

```
assign s = a + b;    // no overflow detection  
// assign s = {a[7], a} + {b[7], b};  
// assign s = a + b; // implicit sign extension by Verilog
```

data types (array and memory)

- Arrays

```
reg [4:0] port_id [0:7];
```

```
// array of 8 port_ids, each port_id is 5-bit wide
```

```
port_id[6] = 5`b00000;
```

```
reg [63:0] array_4d [15:0] [7:0] [7:0] [255:0];
```

```
// 4-d array, new Verilog, IEEE1364-2001
```

- Memory (register files, RAM, ROM)

```
reg [7:0] membyte [0:1023]; // 1K bytes memory
```

```
data = membyte[511] // fetch 1 byte whose address id=511
```

- In real implementation, use tool-supported memory generator to create hardware memory

data type (parameter)

- Parameters
 - allows constants to be defined in a module
 - parameter values for each module instance can be overridden individually at compile time

```
parameter cache_line_width = 256;  
// constant defined inside a module  
// parameter values can be changed  
// at module instantiation #() or by using defparam
```

localparam

```
state1=4'b0001,  
state2=4'b0010,  
state3=4'b0100,  
state4=4'b1000;  
// parameter values cannot be changed
```

System Tasks **\$task_name**

- **\$display**

- display values of variables or strings or expressions

\$display("at time **%d**, address is **%h**", **\$time**, addr);

- **\$monitor**

- continuously monitor a signal when its value changed
- unlike \$display, \$monitor only needs to be invoked once
- use **\$monitoroff** and **\$monitoron** to disable/enable

\$monitor (**\$time**, "clock = **%b** reset = **%b**", clk, rst);

- **\$stop**

- suspend the simulation for debug

- **\$finish**

- terminate simulation

Escape character and string format

Escaped Characters	Character Displayed
<code>\n</code>	newline
<code>\t</code>	tab
<code>%%</code>	%
<code>\\</code>	\
<code>\"</code>	"
<code>\ooo</code>	Character written in 1–3 octal digits

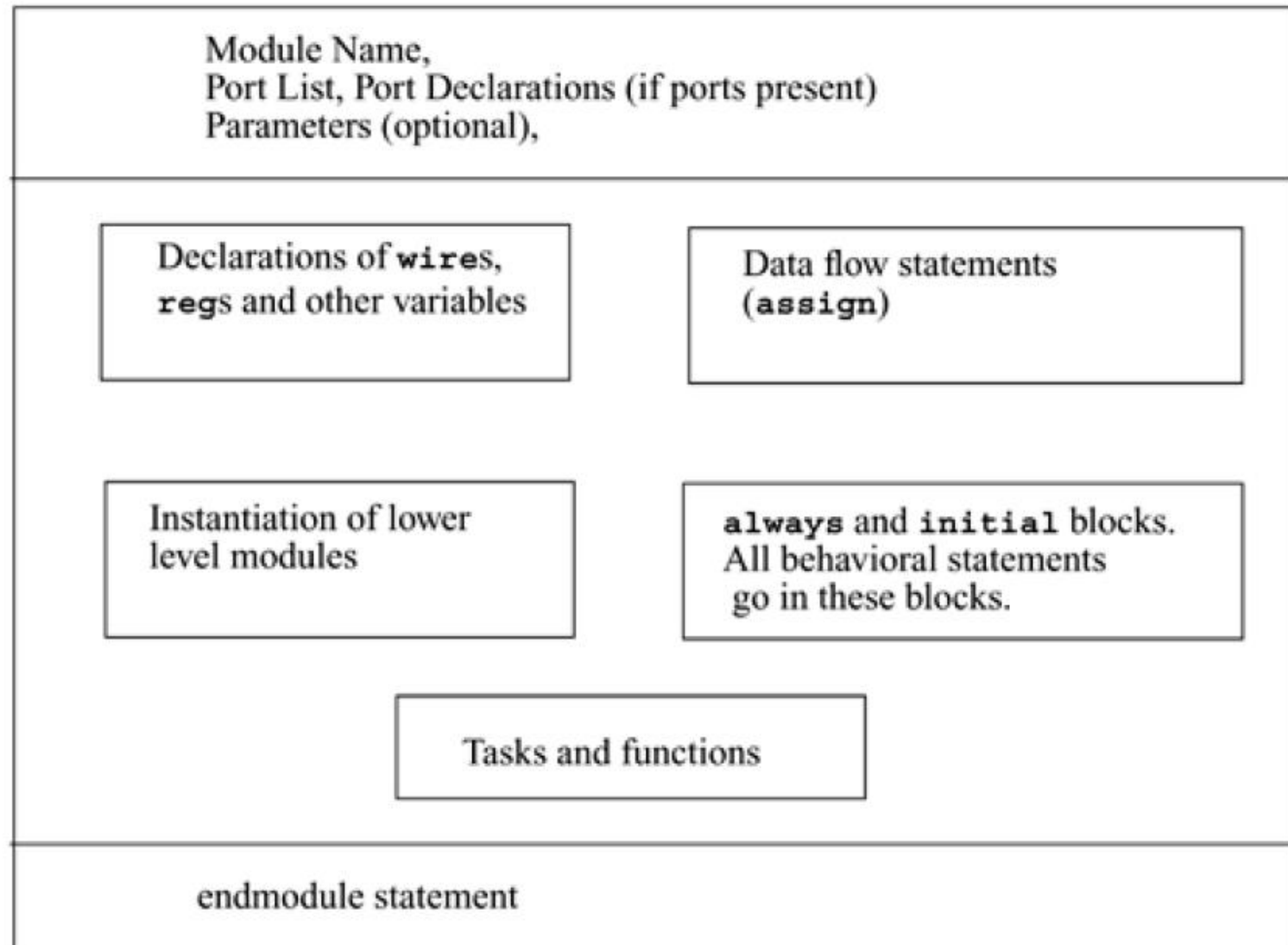
Format	Display
<code>%d</code> or <code>%D</code>	Display variable in decimal
<code>%b</code> or <code>%B</code>	Display variable in binary
<code>%s</code> or <code>%S</code>	Display string
<code>%h</code> or <code>%H</code>	Display variable in hex
<code>%c</code> or <code>%C</code>	Display ASCII character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

Compiler Directives ` (back quote)

- **`define**
 - define text macro for later text macro substitution
 - `define** WORD_SIZE 32
 - // used as **`WORD_SIZE** in the code
- **`include**
 - include a Verilog source file in another file
 - `include** header.v //include the file header.v
- **`ifdef**
 - conditional compilation
 - `ifdef** TEST **module** test;
 - // compile module test only if text macro TEST is defined using **`define** TEST
- **`timescale**
 - `timescale** 100ns/1ns

Modules and Ports

Components of a Verilog Module



module M (P1, P2, P3, P4);

input P1, P2;

output [7:0] P3;

inout P4;

reg [7:0] R1, M1[0:1023];

wire W1, W2, W3, W4;

wire [3:0] W5;

parameter C1=const;

initial

begin : blockname

 // statements

end

always

begin

 // statements

end

// continuous assignment

assign W1 = expr ;

// module instances

COMP U1 (.PP2(W2), .PP1(W1);

COMP U2 (W3, W4);

task T1;

input A1;

inout A2;

output A3;

begin

 // statements

end

endtask

function [7:0] F1;

input A1;

begin

 // statements

end

endfunction

endmodule

Statements inside Behavioral Modeling

#delay

wait (expression)

@(A **or** B **or** C)

@(**posedge** clk)

Reg = expression; // blocking assignment

Reg <= expression; // non-blocking assignment

if (condition1) ... **else if** (condition2) ... **else** ...

case (selection)

choice1 : **begin** ... **end**

choice2, choice 3: ...

...

default : ...

endcase

for (i=1; i<max; i=i+1) **begin** ... **end**

repeat (8) ...

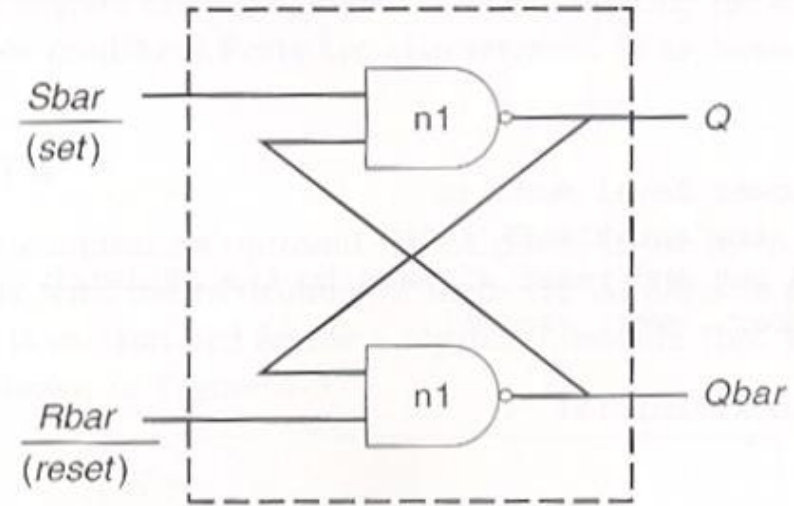
while (condition) ...

```

module SR_latch (Q, Qbar, Sbar, Rbar);
output Q, Qbar;
input Sbar, Rbar;
    nand n1(Q, Sbar, Qbar);
    nand n2(Qbar, Rbar, Q);
endmodule

```

Example: SR latch



```

module top;
wire q, qbar;
reg set reset;

```

```

SR_latch m1(q, qbar, ~set, ~reset);

```

```

initial
begin

```

```

    $monitor ($time, "set= %b, reset = %b, q= %b\n", set, reset, q);

```

```

    set = 0; reset = 0;

```

```

    #5 reset = 1;

```

```

    #5 reset = 0;

```

```

    #5 set = 1;

```

```

end
endmodule

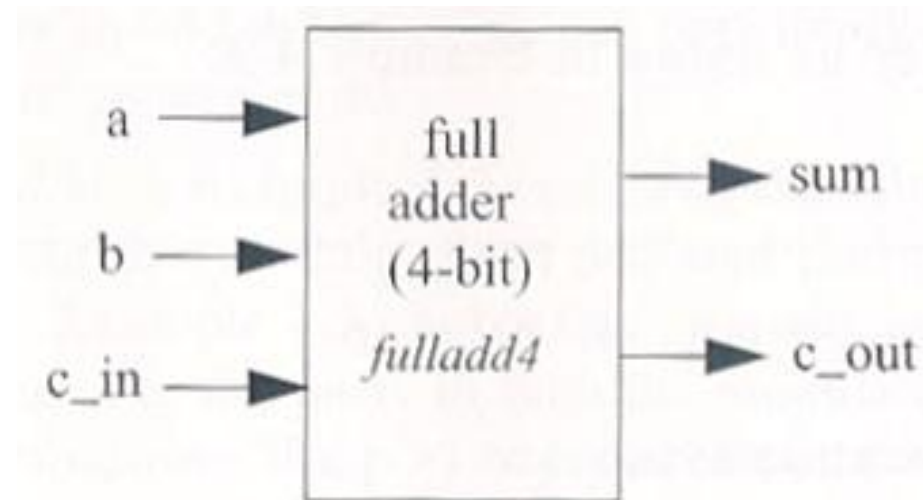
```

- hierarchical name
 - top.qbar
 - top.m1
 - top.m1.n1

Port Declaration (old and new)

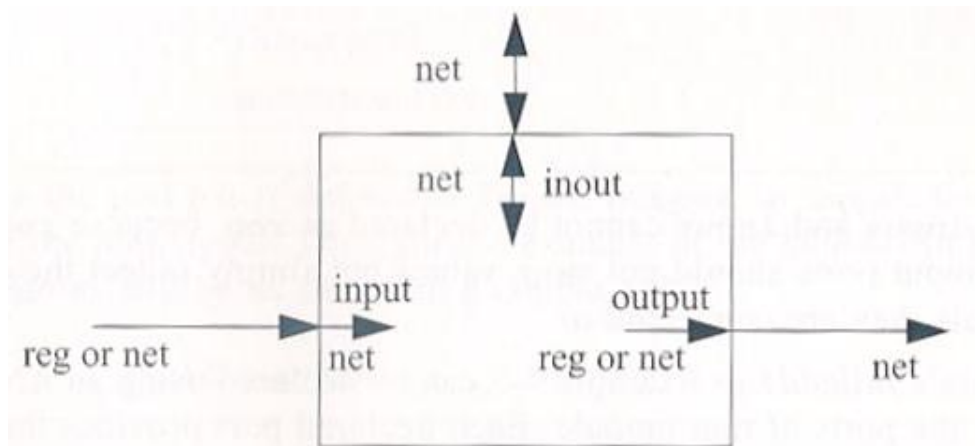
```
// IEEE 1364-1995 (old)
module fulladd4(sum, c_out, a, b, c_in);
parameter width = 4;
output [width-1:0] sum;
output c_out;
input [width-1:0] a, b;
input c_in;
reg [width-1:0] sum;
reg c_out;
...
endmodule
```

```
// ANSI C style (IEEE 1364-2001)
module
  # (parameter width =4)
  fulladd4 (
    output reg [width-1:0] sum,
    output reg c_out;
    input [width-1:0] a, b;
    input c_in);
...
endmodule
```



Port Connection Rules

- all port declarations are implicitly declared as net **wire**
- **input** and **inout** cannot be declared as **reg**
- **input**
 - internally, always be of the type **wire**
 - externally, can be connected to **reg** or **wire**
- **output**
 - internally, can be **reg** or **wire**
 - externally, always connected to **wire**
- **inout**
 - always be **wire** internally or externally



Connecting Ports to External Signals

- Module instantiation
- connecting by order list

```
// instantiate fulladd4 (sum, c_out, a, b, c_in)
```

```
// call it fa_ordered, signals are connected to ports in order (by position)
```

```
fulladd4 fa_ordered (EXT_SUM, EXT_C_OUT, EXT_A, EXT_B, EXT_C_IN);
```

- connecting by name

```
// instantiate module fa_byname and connect signals to ports by name
```

```
fulladd4 fa_byname
```

```
(.c_out(EXT_C_OUT), .sum(EXT_SUM), .b(EXT_B), .c_in(EXT_C_IN), .a(EXT_A));
```

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

```
output [3:0] sum;
```

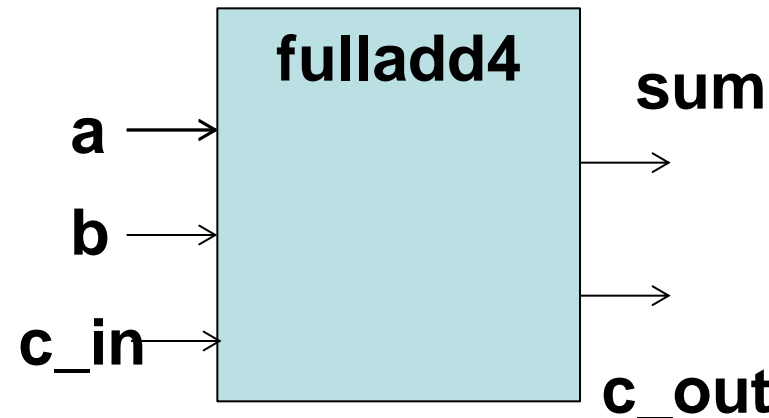
```
output c_cout;
```

```
input [3:0] a, b;
```

```
input c_in;
```

```
....
```

```
endmodule
```



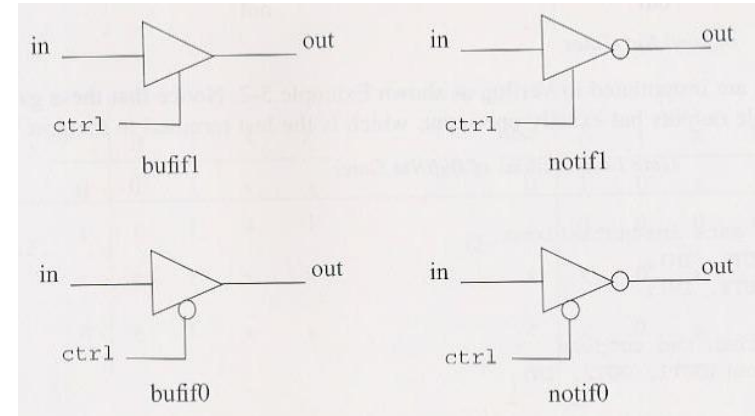
structural modeling

- component instantiation
 - Verilog *built-in* gate primitives (**not, and, or, nand, nor, xor, xnor, ...**)
 - module instantiation (from previously defined modules)
- component connectivity
 - connecting by ordered lists (ordered by position)
fulladd4 fa_ordered (**EXT_SUM, EXT_C_OUT, EXT_A, EXT_B, EXT_C_IN**)
 - connecting by name (order by name)
fulladd4 fa_byname
(**.c_out(EXT_C_OUT), .sum(EXT_SUM), .b(EXT_B), .c_in(EXT_C_IN), .a(EXT_A)**);

Structural-Level (Gate-Level) Modeling

gate types

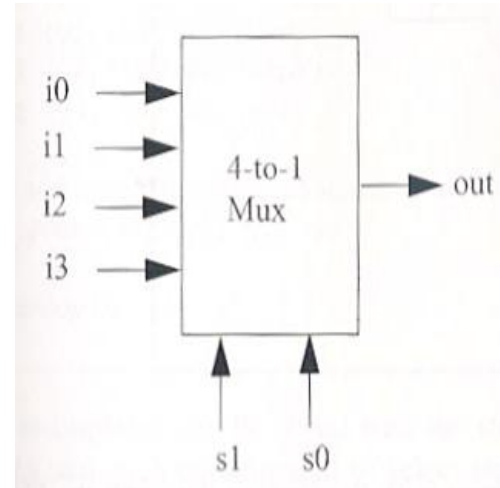
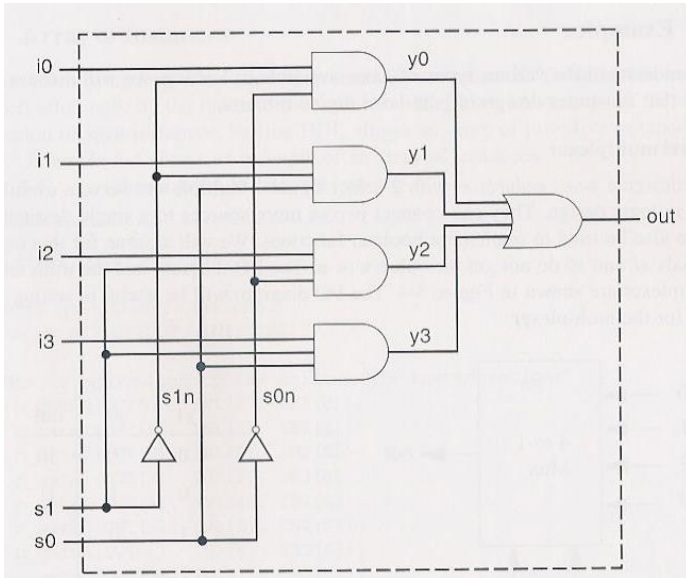
- basic logic gate as predefined (built-in) primitives
 - **and, or, nand, not, xor, xnor**
 - eg., **and** a1(out, in1, in2);
 - **buff, not**
 - eg., **buff** b1(out1, in1);
 - **bufif1, bufif0** // tristate buffers
 - eg., **bufif1** b1(out, in, ctrl);
- array of instances



```
wire [7:0] out, in1, in2;
```

```
nand n_gate [7:0] (out, in1, in2);  
// equivalent to following 8 instances  
nand n_gate0 (out[0], in1[0], in2[0];  
nand n_gate1 (out[1], in1[1], in2[1];  
...  
nand n_gate7 (out[7], in1[7], in2[7];
```

Gate-Level 4:1 Multiplexer



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

// structural level modeling of MUX

module mux4_to_1

(out, i0, i1, i2, i3, s1, s0);

output out;

input i0, i1, i2, i3, s0, s1;

wire s1n, s0n, 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

// behavioral level modeling of MUX

module mux4_to_1

(output reg out, input i0, i1, i2, i3, s1, s0);

// output wire out, **assign** out = s1 ? (s0 ? I3 : I2) : (s0 ? I1 : I0);

//

always @ (i0, i1, i2, i3, s0, s1)

begin

case ({s1, s0})

2'b00: out = i0;

2'b01: out = i1;

2'b10: out = i2;

2'b11: out = i3;

default: \$display("invalid control signals");

endcase

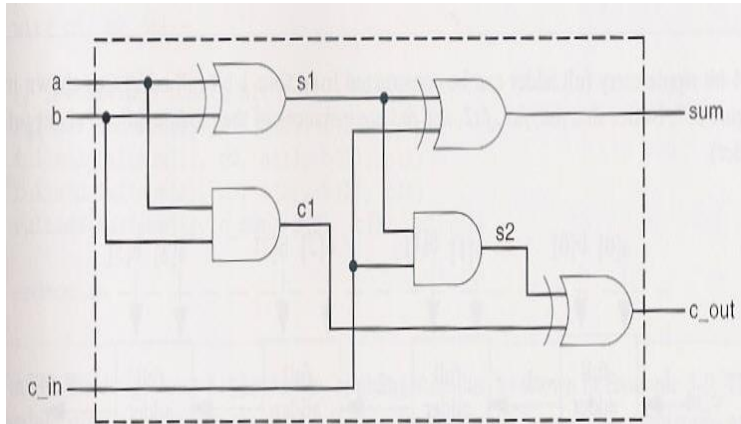
end

endmodule

Gate-Level Ripple Carry 1-bit Full Adder

// *structural* level modeling

```
module fulladd(sum, c_out, a, b, c_in);  
output sum, c_out;  
input a, b, c_in;  
wire s1, c1, c2;  
  
    xor (s1, a, b);  
    and (c1, a, b);  
    xor (sum, s1, c_in);  
    and (c2, s1, c_in)  
    xor (c_out, c2, c1);  
  
endmodule
```



// *dataflow* modeling, continuous assignment

```
module fulladd(sum, c_out, a, b, c_in);  
output sum, c_out;  
input a, b, c_in;  
  
    assign {c_out, sum} = a + b + c_in;  
  
endmodule
```

// *behavioral* level modeling

```
module fulladd(sum, c_out, a, b, c_in);  
output reg sum, c_out;  
input a, b, c_in;  
  
    always @ (a or b or c_in)  
        {c_out, sum} = a + b + c_in;  
  
endmodule
```

4-bit Ripple Carry Adder

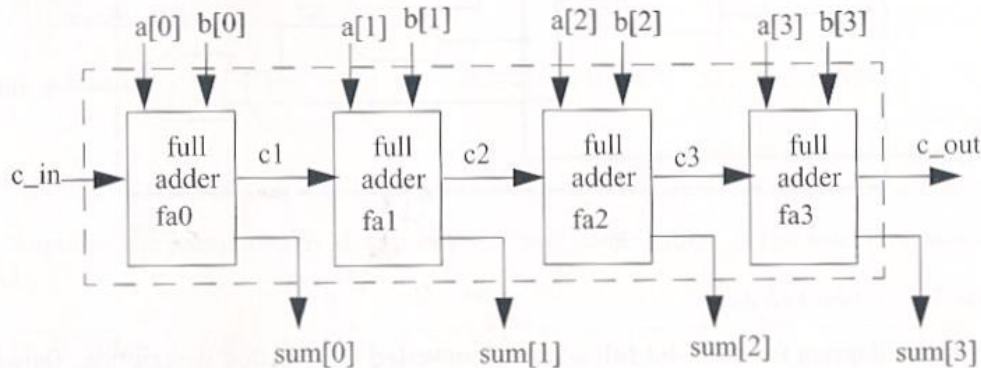
```
// structural level modeling
module fulladd4 (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in

wire c1, c2, c3;

    fulladd fa0 (sum[0], c1, a[0], b[0], c_in);
    fulladd fa1 (sum[1], c2, a[1], b[1], c1);
    fulladd fa2 (sum[2], c3, a[2], b[2], c2);
    fulladd fa3 (sum[3], c_out, a[3], b[3], c3);

endmodule
```



```
// continuous assignment
module FA_dataflow
    (output [3:0] sum,
     output c_out,
     input [3:0] a, b,
     input c_in);

    assign {c_out, sum} = a+b+c_in;

endmodule
```

```
// behavioral level modeling
module FA_behavior
    (output reg [3:0] sum,
     output reg c_out,
     input [3:0] a, b,
     input c_in);

    always @(a, b, c_in)
        {c_out, sum} = a+b+c_in;

endmodule
```


Stimulus for 4-bit RCA

```
module stimulus;  
reg [3:0] A, B;  
reg C_IN;  
wire [3:0] SUM;  
wire C_OUT;
```

```
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN); // module instance
```

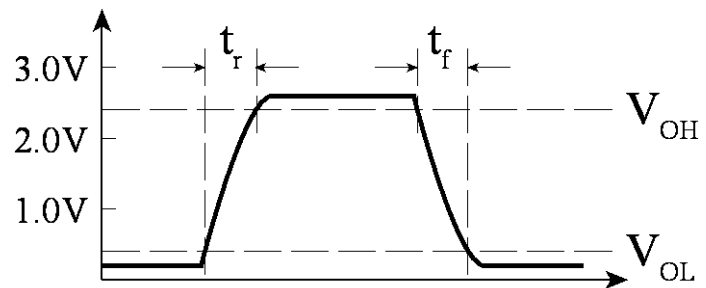
```
initial  
begin  
$monitor ($time, "A=%b, B=%b, C_IN=%b, C_OUT=%b, SUM=%b", A, B, C_IN, C_OUT, SUM);  
end
```

```
initial  
begin  
A = 4'd0; B=4'd0; C_IN=1'b0;  
#5 A=4'd3; B=4'd4;  
#5 A=4'd2; B=4'd5;  
#5 A=4'd9; B=4'd9;  
#5 A=4'd10; B=4'd15;  
#5 A=4'd10; B=4'd5; C_IN=1'b1;  
end  
endmodule
```

Gate Delays

- rise delay (t_{pdH})
 - output transition to 1
- fall delay (t_{pdL})
 - output transition to 0
- turn-off delay
 - output transition to z
- for simulation only
- gate delay will be ignored during logic synthesis
 - real timing delay depends on the hardware components from real standard cell library

```
and #(5) a1(out, i1, i2);  
// delay of 5 time units for all  
// transitions  
  
and #(4, 6) a2(out, i1, i2);  
// rise delay =4, fall delay=6  
  
bufif0 #(3,4,5) b1(out, in, cntl);  
// rise=3, fall=4, turn-off=5
```



Rise time: shape of a single waveform
Rise delay (propagation L->H delay):
Input to output delay

min/typ/max delays

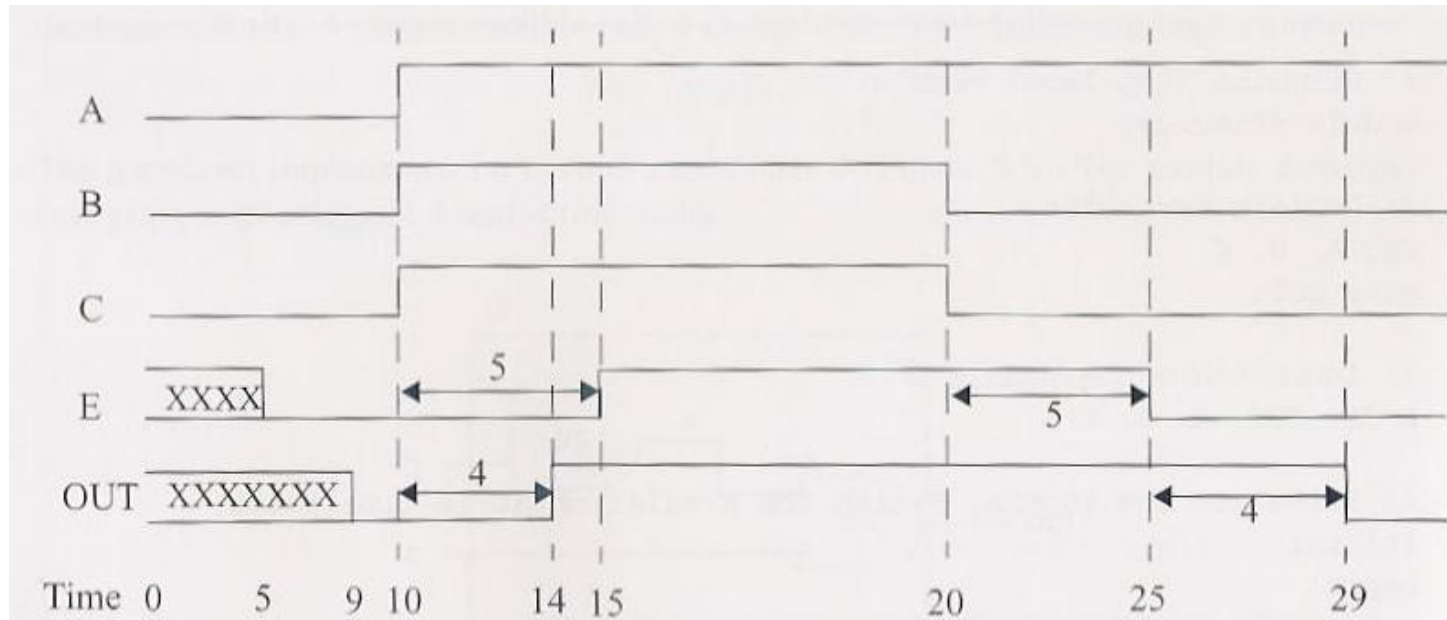
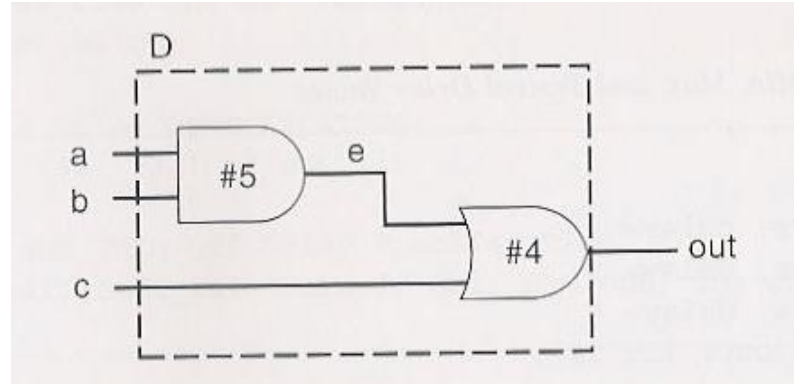
```
// with Verilog-XL option +mindelays, delay=4  
// with Verilog-XL option +typdelays, delay=5  
// with Verilog-XL option +maxdelays, delay=6  
and #(4:5:6) a1(out, i1, i2);
```

```
// if +mindelays, rise=3, fall=5, turn-off=min(3,5)  
// if +typdelays, rise=4, fall=6, turn-off=min(4,6)  
// if +maxdelays, rise=5, fall=7, turn-off=min(5,7)  
and #(3:4:5, 5:6:7) a2(out, i1, i2);
```

```
// in fact, #() overwrite default parameters in a module  
during module instantiation
```

Delay Example

```
module D (out, a, b, c);  
  output out;  
  input a, b, c;  
  wire e;  
  and #(5) a1(e, a, b);  
  or #(4) o1(out, e, c);  
endmodule
```



Dataflow Modeling

(continuous assignment: **assign**)

continuous assignment (**assign**)

- most basic statement in dataflow modeling
- Continuously drive a value onto a **net (wire)**
 - Compared with *procedural assignment* in a behavioral block (**always**)
- replace gates in the description of circuits at a higher level of abstraction (dataflow model)
 - Describe **combinational logic**

```
wire [31:0] i1, i2, out;  
assign out = i1 | i2;  
// out must be wire  
// operator | denotes bitwise OR operation  
// out must be a net (wire)
```

Continuous Assignment (**assign**)

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

```
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];  
// addr is a vector net; addr1 and addr2 are vector registers
```

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;  
// left-hand side is a concatenation of a scalar net and a vector net  
// {c_out, sum[3:0]} is equal to c_out & sum(3 downto 0) in VHDL
```

- *the left-hand side must always be scalar or vector net (**wire**) (cannot be **reg**)*
- *evaluated as soon as the right-hand-side operands changes (unlike the behavioral procedural assignments)*
- *expressions combine operators and operands*

Implicit Continuous Assignments

- // regular continuous assignment
wire out; // wire is default data type
assign out = in1 & in2;
- // implicit continuous assignment
wire out = in1 & in2; // no keyword **assign**
- // implicit net declaration
wire in1, in2;
assign #10 out = in1 & in2;
// implicit net declaration of out as wire
// wait for 10 time units (#10, delay control), then
// sample values of in1, in2, calculate in1&in2,
// and assign to out

Operator Types

- arithmetic (+, -)
- logical (!, &&, ||)
- relational (>, <)
- equality (==, !=)
- bitwise (~, &, |, ^)
- reduction (&, ~&)
- shift (<<, >>)
- concatenation ({ })
- conditional (? :)

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

Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	Three

Examples

- if A=4'b0011; B=4'b0000
- A + B; // adding A and B equals 4'b1100
- A && B; // evaluate to 0 if A=3; B=0 (logical AND)
- A || B; // evaluate to 1 if A=3; B=0;
- A <= B; // evaluate to a logical 0 if A=3; B=0
- A == B; // results in logical 0 if A=3; B=0
- A & B; // bitwise AND operation, result is 4'b0000
- &A; // equivalent to 0&0&1&1. results in 1'b0;
- C = A >> 1; // logical right shift. results in 4'b0001
- C = {A, B}; // concatenation. C=8'b00110000
- C = {2{A}}; // replication. C=8'b00110011
- **assign** out = cntl ? in1 : in2; // conditional operator

Case equality operators (**===**, **!==**)

// A = 4, B=3

// X=4'b1010, Y=4'b1101

// Z=4'b1xxz, M=4'b1xxz, N=1'b1xxx

A == B // results in logical 0

X != Y // results in logical 1

Z === M // results in logical 1 (all bits match, including x and z)

Z === N // results in logical 0 (LSB does not match)

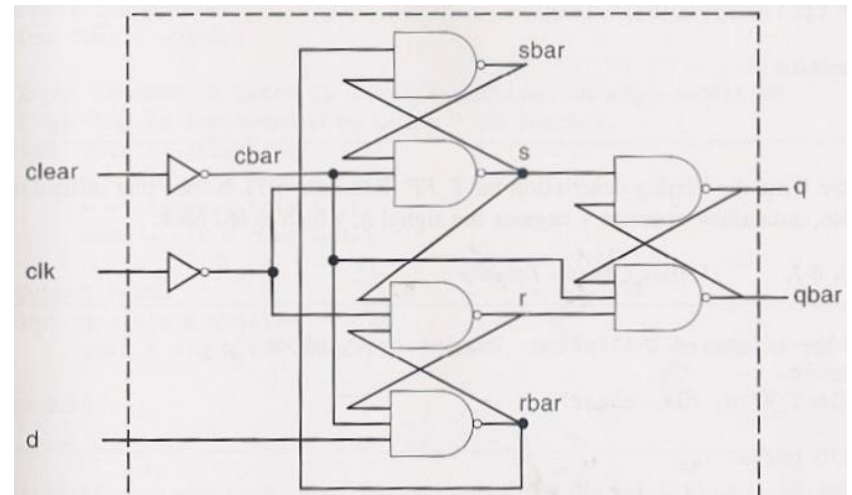
M !== N // results in logical 1

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

behavioral (**always**) vs. dataflow (**assign**)

```
module D_FF (q, qbar, d, clk, clr);  
output q, qbar;  
input d, clk, clr;  
reg q, qbar;  
  
// DFF with asynchronous clear (reset)  
always @ (posedge clr or negedge clk)  
if (clr)  
    begin  
        q <= 1'b0;  
        qbar <= 1'b1;  
    end  
else  
    begin  
        q <= d;  
        qbar <= ~d;  
    end  
endmodule
```

```
module dff (q, qbar, d, clk, clear);  
output q, qbar;  
input d, clk, clear;  
wire s, sbar, r, rbar, cbar;  
  
assign cbar = ~clear;  
assign sbar = ~(rbar & s),  
        s = ~(sbar & cbar & ~clk),  
        r = ~(rbar & ~clk & s),  
        rbar = ~(r & cbar & d);  
assign q = ~(s & qbar),  
        qbar = ~(q & r & cbar);  
  
endmodule
```



Both **always** and **assign**
belong to RTL coding style

Example (4-to-1 Multiplexer)

```
// using logic equations  
module mux4_logic (out, i0,  
    i1, i2, i3, s1, s0);
```

```
output out;  
input i0, i1, i2, i3, s0, s1;
```

```
assign out =  
    (~s1 & ~s0 & i0) |  
    (~s1 & s0 & i1) |  
    (s1 & ~s0 & i2) |  
    (s1 & s0 & i3);
```

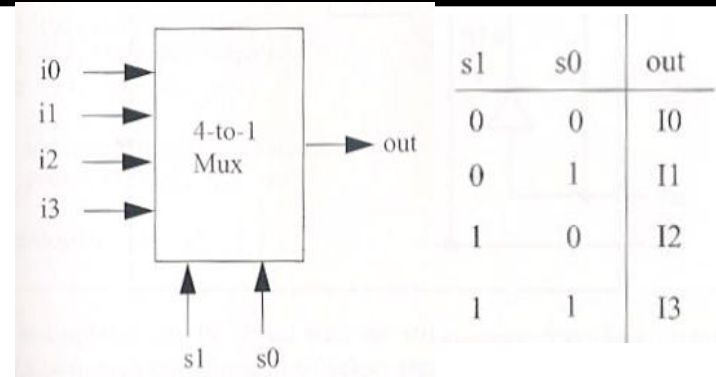
```
endmodule
```

```
// using conditional operator  
module mux4_cond (out, i0,  
    i1, i2, i3, s0, s1);
```

```
output out;  
input i0, i1, i2, i3, s0, s1;
```

```
assign out = s1 ?  
    (s0 ? i3 : i2) : (s0 ? i1 : i0);
```

```
endmodule
```



Example (4-bit Adder)

```
// using dataflow
// with proper synthesis constraint

module add4 (sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

// gate-level structure
// depends on
// synthesis constraints
assign {c_out, sum}
    = a + b + c_in;

endmodule
```

$$p_i = a_i \oplus b_i, \quad g_i = a_i b_i, \quad c_{i+1} = g_i + p_i c_i$$

$$c_1 = g_0 + p_0 c_{in}$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$

$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

$$s_i = p_i \oplus c_i$$

```
// explicit carry lookahead adder
```

```
module add4 (sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;

assign p0=a[0]^b[0], p1=a[1]^b[1], p2=a[2]^b[2],
    p3=a[3]^b[3];
assign g0=a[0]&b[0], g1=a[1]&b[1], g2=a[2]&b[2],
    g3=a[3]&b[3];
assign c1 = g0|(p0&c_in),
    c2=g1|(p1&g0)|(p1&p0&c_in),
    c3=g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&c_in),
    c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)
        |(p3&p2&p1&p0&c_in);
assign sum[0] = p0 ^ c_in, sum[1] = p1 ^ c1,
    sum[2]=p2 ^ c2, sum[3] = p3 ^ c3;
assign c_out = c4;

endmodule
```