Verilog:
overview
structural modeling,
dataflow modeling

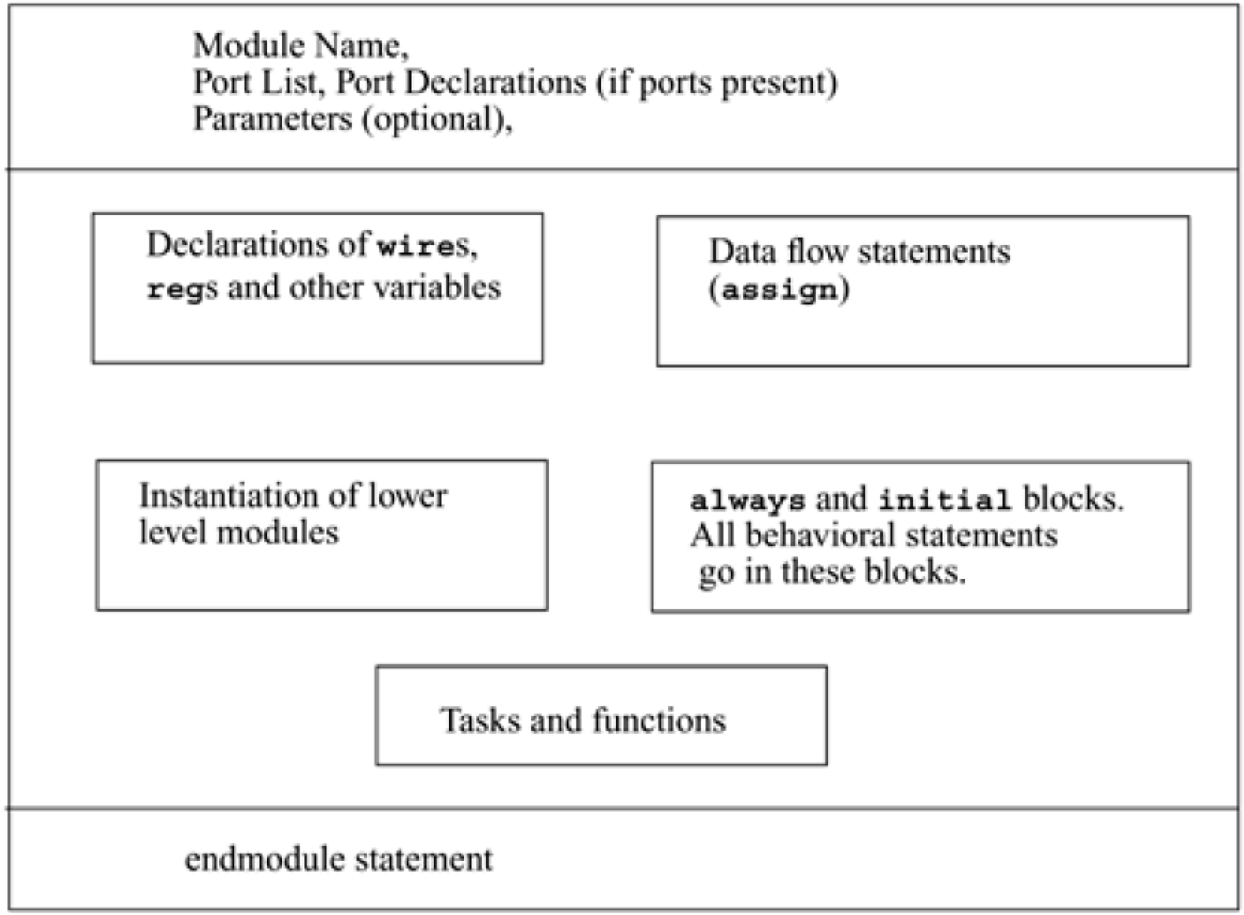
Outlines

- Overview
- Testbench
- System Task, Compiler Directives
- Verilog Operators, Data Types
- Structural-level Modeling
- Modules and Ports
- Dataflow Modeling

Overview

Components of a Verilog Module

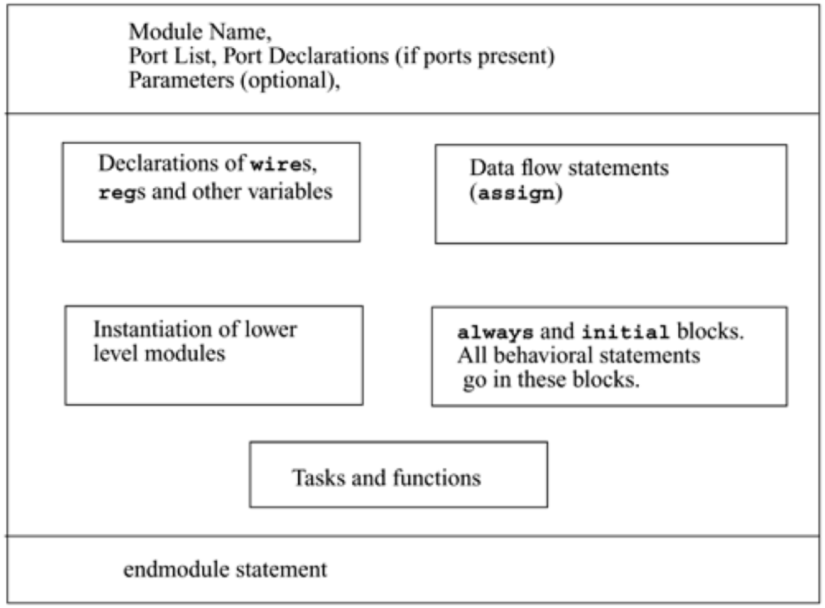
- declaration
  - inputs, outputs
  - parameters
  - data types
- **structure** model
- module *instantiation*
- **dataflow** model
  - *continuous assignment*
- behavior model
  - *procedural assignment*



Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

```
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;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end
```

```
// dataflow: continuous assignment
assign W1 = expr ;  // use operators

// module instances: structural
modeling
COMP U1 (.PP2(W2), .PP1(W1);
COMP U2 (W3, W4);

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction
```


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Parallel Coding in Hardware

- hardware programming is basically **parallel** coding
  - order of codes does NOT matter
  - e.g., describe two hardware units: adder and multiplier

```
…
assign c=a+b; // descrbe adder first
assign out=c*d; // then describe multiplier
…
```

```
…
assign out=c*d; // describe multiplier
assign c=a+b; // then describe adder
…
```

- software programming is usually **sequential**
  - order of codes matters

```
…
c=a+b;
out=c*d; // out = (a+b) *d
…
```

```
…
out=c*d; // out = old_c *d
c=a+b; // new_c = a+b
…
```

Combinational VS. Sequential Logic

- Combinational Logic
  - Logic circuits with pure *feed-forward* datapath
  - No flip flops or latches
  - e.g.

```
module adder (input wire [7:0], a, b, output  wire [7:0] c);
assign c = a + b;   // dataflow modeling
endmodule
```

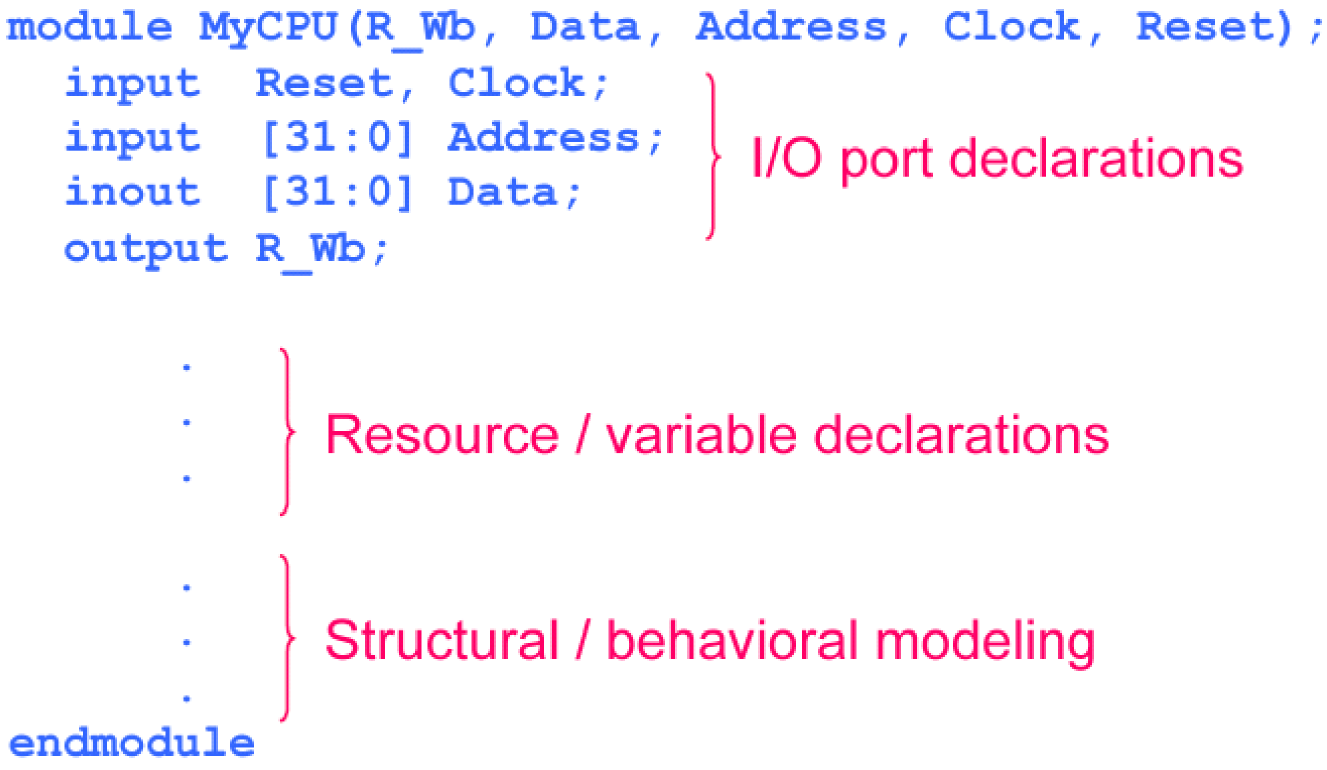
- Sequential Logic
  - Logic circuits with latches or flip-flop storage elements
  - e.g.

```
module lateched (input [7:0], a, b, input  clk, output reg [7:0] c);
always @ (poedege clk)    // behavioral modeling
   c = a + b;  // result c stored in a positive-edge-riggered register
endmodule
```

module definition

- declaration of I/O port and datatype could be merged with module declaration in the first line, e.g.,

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



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

        .
        .
        .

        .
        .
        .
endmodule
```

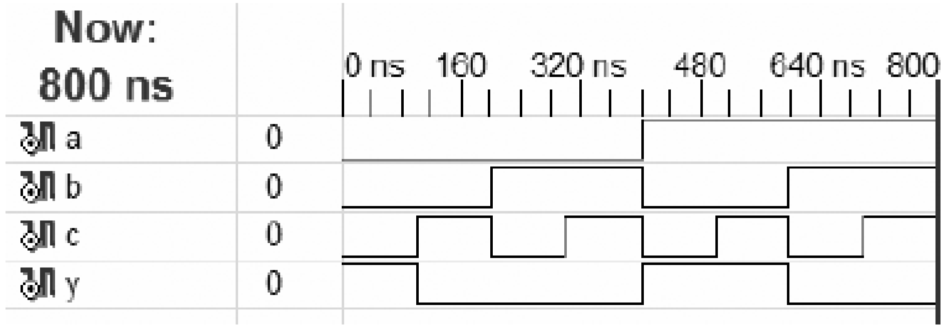
I/O port declarations
Resource / variable declarations
Structural / behavioral modeling

module

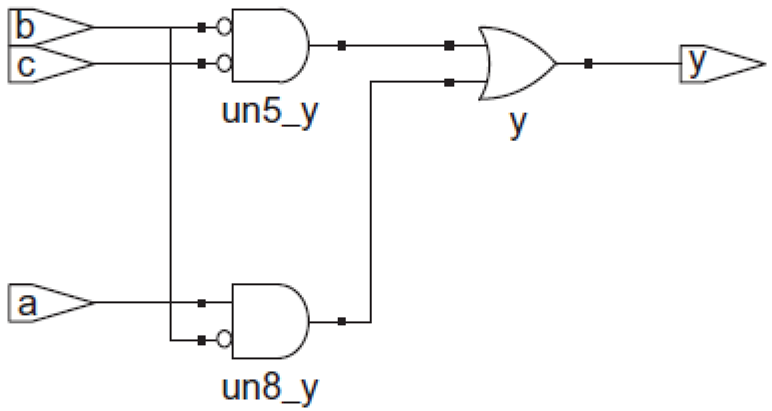
- define a hardware component and its input/output ports

```
module sillyfunction(input a, b, c,
                     output y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

- simulation



- synthesis results



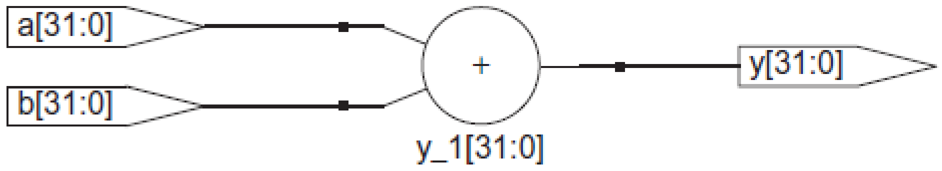
32-bit adder

- dataflow modeling (**assign** …)
  - real adder hardware circuit implementation depends on synthesis constraints (delay, area, power)
    - ripple carry adder (slow but smaller), or
    - carry lookahead adder (fast but larger)

SystemVerilog

```
module adder(input  logic [31:0] a,
             input  logic [31:0] b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

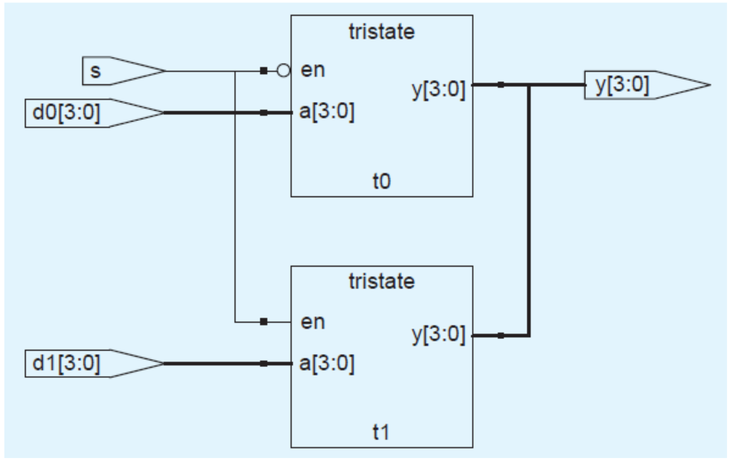
logic in SystemVerilig corresponds to**wire** (default) **, reg** in Verilog



4:1 multiplexer

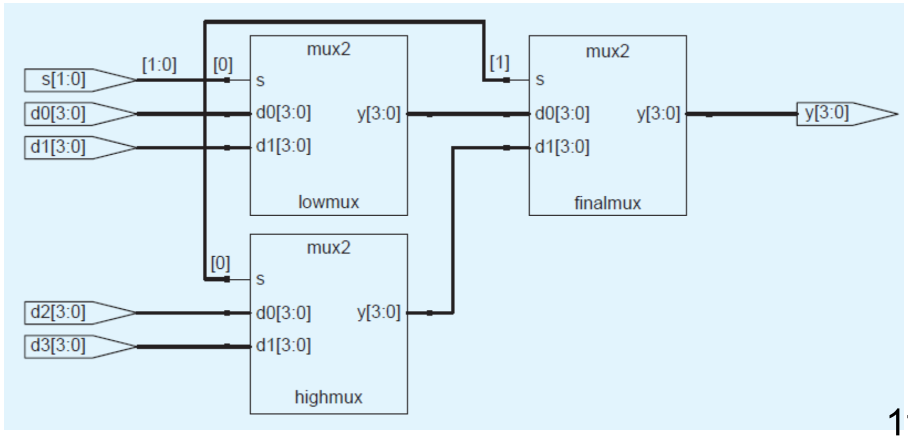
- *structure* of hardware is described
- Construct lower layer components (mux2) into higher ones (mux4)

```
module mux4 (input  [3:0] d0, d1, d2, d3,
             input  [1:0] s,
             output [3:0] y);

  wire [3:0] low, high;

  mux2 lowmux (d0, d1, s[0], low);
  mux2 highmux (d2, d3, s[0], high);
  mux2 finalmux (low, high, s[1], y);
endmodule
```



mux2 is composed of two tri-state buffers
// built-in tri-state inverts or buffers
notif1, notif0, bufif1, bufif0

```
module mux2 (input  [3:0] d0, d1,
             input        s,
             output [3:0] y);

  tristate t0 (d0, ~s, y);
  tristate t1 (d1, s, y);
endmodule
```



other modeling methods for mux4

```
// dataflow using
// nested
// conditional
// operator
…
assign y = s[1] ?
(s[0] ? d3 : d2) :
 (s[0] ? d1 : d0) ;
…
```

```
// behavior using case
…
 always @ (d0, d1, d2, d3, s)
begin
case (s[1], s[0])
   2'b00: y=d0;
   2'b01: y=d1;
   2'b10: y=d2;
   2'b11: y=d3;
endcase
end
…
```

```
// behavior using if else
…
 always @ (*)
begin
 if (s=2'b00)  y=d0;
   else if (s=2'b01) y=d1;
   else if (s=2'b10) y=d2;
   else y=d3);
…
```

half adder (HA)

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$sum = a \oplus b$$

$$cout = ab$$

```
// structural description
module HA (a, b, sum, cout);
input a, b;
output sum, cout;

// use built-in gates
xor X1(sum, a, b);
and a1(cout, a, b);

endmodule
```

```
// dataflow
module halfadder (a,b,sum,cout);
input a, b;
output sum; cout;

assign s = a ^ b ;   // ^ is XOR operator
assign c = a & b;  // & is AND operator
// assign {c, s} = a + b;

endmodule
```

```
// behavior
module halfadder (a,b,sum,cout);
input a, b;
output reg sum; cout;

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

endmodule
```

8-b Adder

- declare signal as bit-vector
  - e.g., **wire** [7:0] s; // 8-bit signal s
    - moar significant bit (MSV) is s[7]
    - least significant bit (LSB) is s[0]

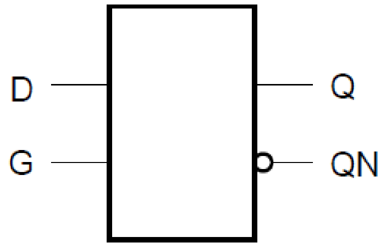
```
module add8 (A, B, S, Co);
input [7:0] A;  // declare a signal as a multi-bit vector
input [7:0] B;
output [7:0] S;
output Co;

assign {Co, S}= A + B;

endmodule
```

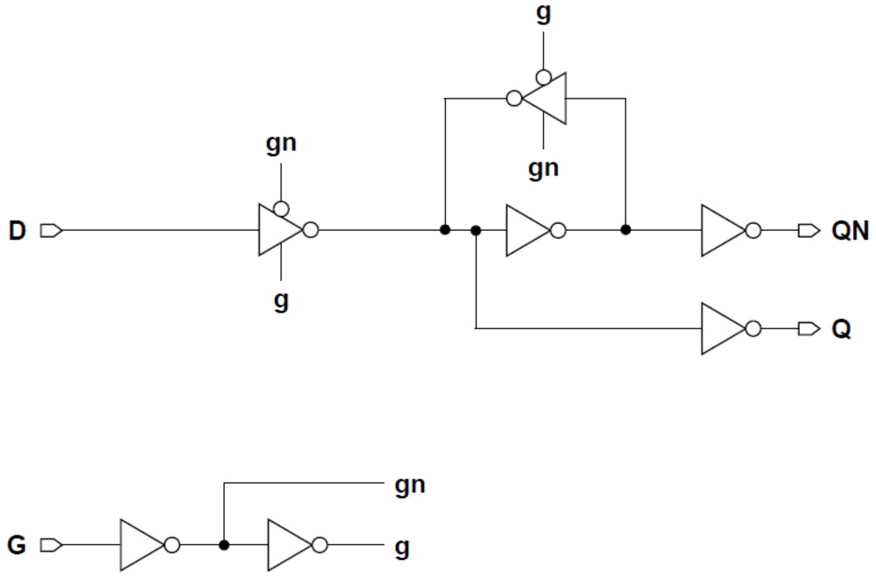
Latch

- Use "**always**" block to describe latch
  - pass input to output when control signal is active
  - keep the previous output value when control signal is in-active

```
module latch (D, G, Q, QN);
  input   wire  D, G;
  output  reg Q, QN;

  always @ ( D or G )
  begin
     if (G)
        begin
           Q <= D;
           QN <= ~D;
         end
  end
endmodule
```

G	D	Q[n+1]	QN[n+1]
1	0	0	1
1	1	1	0
0	x	Q[n]	QN[n]



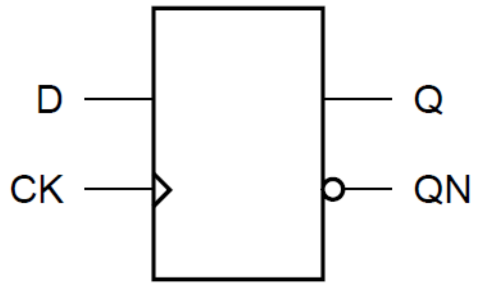


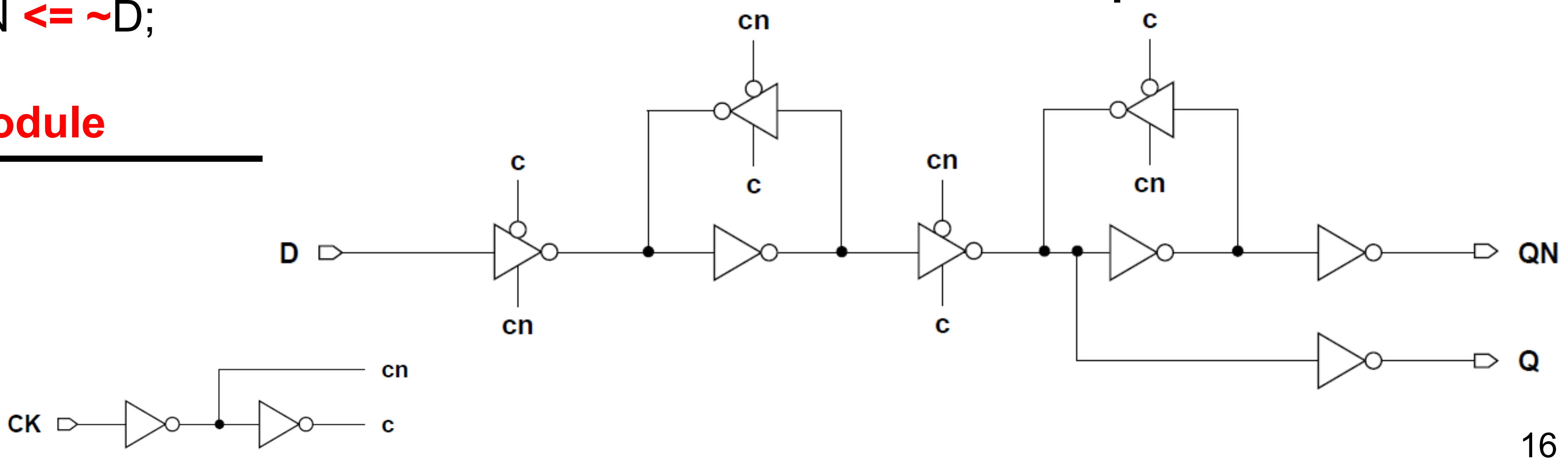
Flip-Flop (FF)

- Use “always” block to describe flip-flop (FF)
  - pass the input to output at the time instance of clock edge
  - keep the previous output values at other time
  - usually consists of two back-to-back latches

```
module flip_flop (D, CK, Q,QN);
  input   wire  D, CK;
  output  reg Q, QN;

  always @ ( posedge CK )
  begin
     Q   <= D;
     QN <= ~D;
  end
endmodule
```

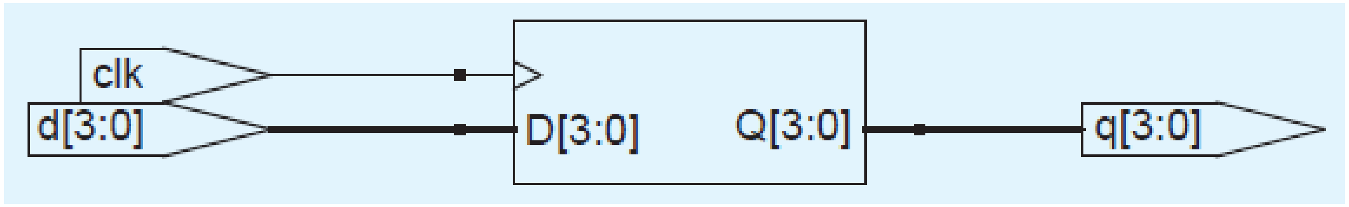
D	CK	Q[n+1]	QN[n+1]
0	↑	0	1
1	↑	1	0
x	↓	Q[n]	QN[n]





register (multiple-bit flip-flops)

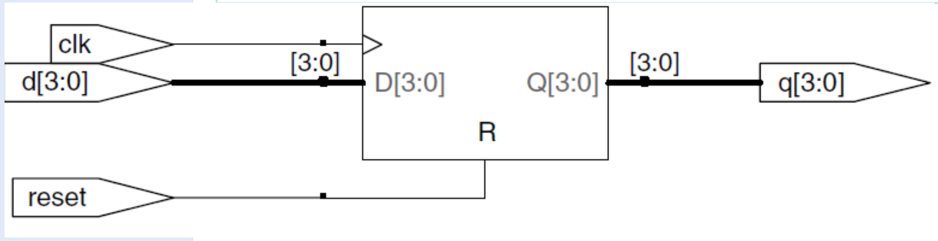
```
module flop(input              clk,
            input        [3:0] d,
             output reg  [3:0] q);

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



resettable register

```
module flopr(input              clk,
             input              reset,
             input        [3:0] d,
             output reg   [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if(reset) q <= 4'b0;
    else      q <= d;
endmodule

module flopr (input             clk,
              input             reset,
              input       [3:0] d,
              output reg  [3:0] q);

  // synchronous reset
  always @ (posedge clk)
    if (reset)q <= 4'b0;
    else      q <= d;
endmodule
```

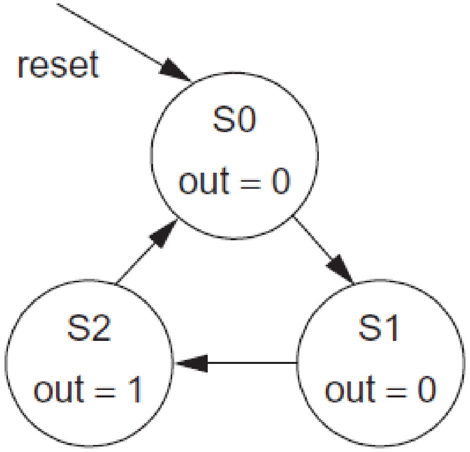
```
// asynchronous reset FF
…
always @ (posedge clk
or positive reset )
   if (reset) q <= 4’d0;
   else q <= d;
…
```



Finite State Machine (FSM): divide-by-3 counter (3 times period, or 1/3 frequency)

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  logic [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk)
    if (reset) state <= 2'b00;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      2'b00: nextstate = 2'b01;
      2'b01: nextstate = 2'b10;
      2'b10: nextstate = 2'b00;
      default: nextstate = 2'b00;
    endcase
```

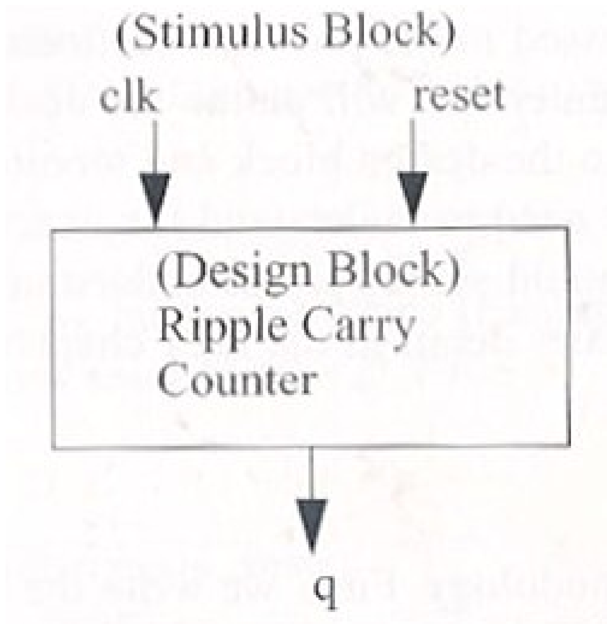


```
// Output Logic
  assign y = (state == 2'b00);
endmodule
```

Testbench

simulation and testbench

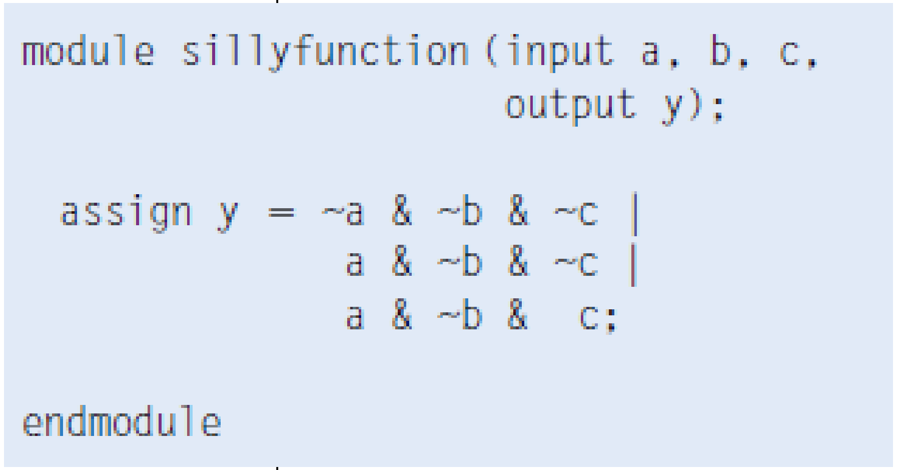
- verify the hardware design
- simulation block (also called **tes tbench**, or **test fixture)**
  - instantiates design block, i.e., the hardware under test
  - apply stimulus (test patterns or test vectors)
  - observe/compare the output with expected results
- design block must be in **synthesizable** HDL codes
- simulation block could be in **non-synthesizable** HDL codes
- testbench consists 4 parts
  - instantiate hardware design
  - prepare inputs and expected outputs
    - ✓ inside testbench, or read files from outside
  - assign inputs to the hardware
  - compare outputs with expected
    - ✓ inside testbench, or write files to outside



Testbench example (manually specified inputs)

- Not synthesizeable statements are OK

```
module testbench1();
  reg  a, b, c;
  wire y;
  // instantiate device under test
  sillyfunction dut (a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

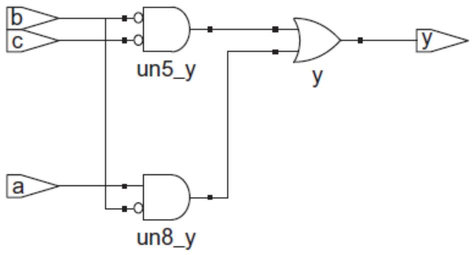


```
module sillyfunction(input a, b, c,
                     output y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

Testbench examples

Verilog

```
module sillyfunction (input a, b, c,
                      output y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;
```



Verilog

```
module testbench2 ();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut (a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1;               #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0;        #10;
    if (y !== 0) $display("010 failed.");
    c = 1;               #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1;               #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0;        #10;
    if (y !== 0) $display("110 failed.");
    c = 1;               #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Verilog

```
module testbench3 ();
  reg        clk, reset;
  reg        a, b, c, yexpected;
  wire       y;
  reg [31:0] vectornum, errors;
  reg [3:0]  testvectors [10000:0];

  // instantiate device under test
  sillyfunction dut (a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb ("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @ (posedge clk)
    begin
      #1; {a, b, c, yexpected} =
            testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @ (negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display ("Error: inputs = %b", {a, b, c});
        $display (" outputs = %b (%b expected)",
                  y, yexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display ("%d tests completed with %d errors",
                  vectornum, errors);
        $finish;
      end
    end
endmodule
```

Self-checking Testbench

need to know the expected correct outputs for comparison with the real hardware outputs

```
module testbench2();
  reg  a, b, c;
  wire y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

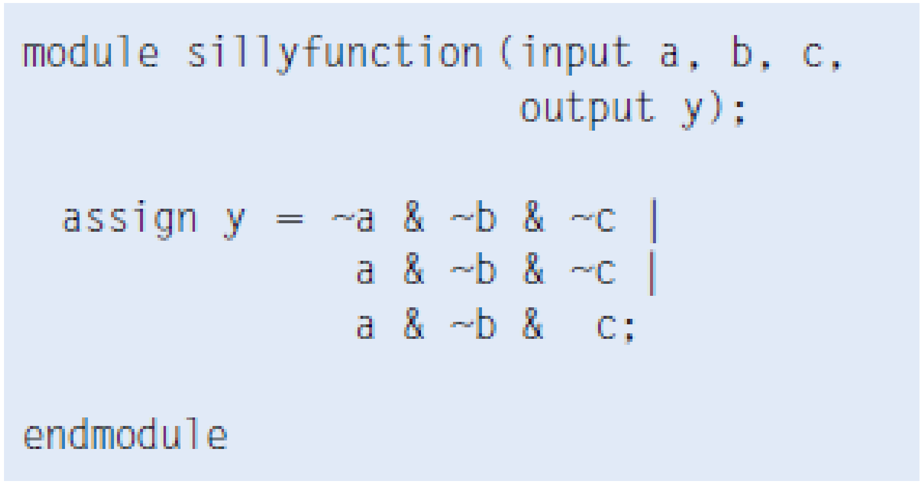
Testbench with Testvectors from files

- prepare testvectors file which could be from other Verilog codes or other high-level language like C

- Testbench:
  1. Instantiate Design (with specified inputs and outputs), Generate clock
  2. Read testvectors file (including inputs and expected outputs)
  3. Assign inputs
  4. Compare outputs with expected

Testbench: 1. Instantiate Design, Generate Clock

```
module testbench3();
  reg         clk, reset;
  reg         a, b, c, yexpected;
  wire        y;
  reg  [31:0] vectornum, errors;
  reg  [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut (a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end
```



```
module sillyfunction(input a, b, c,
                     output y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

2. Read Testvectors

```
 // at start of test, load vectors
// and pulse reset

initial
   begin
    // system task $readmemb reads binary data from file
     $readmemb ("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1; #27; reset = 0;
   end
```

```
// testvector
// fine namej
// example.tv

000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

3. Assign Inputs

```
// apply inputs from  test vectors on rising edge of clk
//  device under test is the instantiated      sillyfunction dut (a, b, c, y);

  always @(posedge clk)
   begin
     #1; {a, b, c, yexpected} = testvectors[vectornum];
         // assign inputs a, b, c;  get output y from sillyfunction dut (a, b, c, y);

    end
```

4. Compare Outputs with Expected

```
// check results on falling edge of clk
  always @(negedge clk) begin
   if (~reset) begin // skip during reset
    if (y !== yexpected) begin
      $display("Error: inputs = %b",
              {a, b, c});
      $display("  outputs = %b (%b expected)",
               y, yexpected);
      errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
              vectornum, errors);
      $finish;
    end
   end  // end of always @ (posedge clk)
endmodule
```

Test Patterns

- access files for inputs and expected outputs
  - $fscanf, $read_memb, $fdisplay, …
  - need other programs (e.g., C, Python) to generate the input files of test patterns
  - need other programs to compare the simulation results in the output file
  - the previous example read test patterns and expected results from a file, and make comparision in the testbench
- generate inputs in testbench module
  - randomly generate input test patterns (e.g., $random, …)
  - could use non-synthesizable codes to generate expected results (e.g., real x, y, z ‘ assign z=x+y; // floating-pt. add)
  - compare simulation results with expected ones

Another Example

- fixed-point signed adder
- instantiate hardware module
  - – connection-by -ame I/O pins
- testbench (next slide)
  - – read input test patterns from files, or randomly generate inputs in testbench
  - – generate expected results in testbench
  - – make comparison in testbench

```
module FXP_adder (input signed [31:0] a, b, output signed [31:0] d);

assign d = a + b;

endmodule
```

```
… // instantiate hardware
FXP_adder test_module ( .a(input_a), .b(input_b), .d(outcome));
… // read test patterns from files
file_a = $fopen( “a.txt” , "r");
file_b = $fopen( “b.txt”  , "r");
file_d = $fopen(“c.txt” , "w");
for (i = 0; i < 100; i=i+1) begin …
  garbage = $fscanf(file_a, “%X”, data_a[i]);   // hexa-decimal format
  garbage = $fscanf(file_b, “%X”, data_b[i]); … end
… // compute expected answer, get hardware outputs and write to a file
for (i=0; i< 100; i=i+1) begin …
  input_a = data_a[i];   // apply test patterns to hardware inputs
  input_b = data_b[i];
  answer = $signed(input_a) + $signed(input_b); // expected results
  if (output != answer) ….    // error occurs
  $fwrite(file_d, "%X\n", outcome);  … end
… // an alternative of generating input test patterns randomly
 for (i=0; i< 100; i=i+1) begin …
  input_a = $random % (2**31);   // between -2^31 + 1 and 2^31 – 1
  input_b = $random % (2**31);
  answer = $signed(input_a) + $signed(input_b); … end
…
```

System Task $

- read from input
  - $readmemb, $readmemh
  - $fscanf, $fget. $fread
- write to output
  - $fopen, $fclose
  - $display, $strobe, $monitor, $write
  - $fdisplay, $fstrobe, $fmonitor, $fwrite
- random generator
  - $random
- suspend ($stop) or finish ($finish) simulation
- conversion functions
  - $signed, $unsigned

Compiler Directive `

- start with back-quote symbol `
  - cp. quote symbol, e.g., 8’b0101_0101
- `define
  - define textmacro
  - e.g., `define FILE_A “a.txt”
    - ✓ `FILE_A is equivalenet to “a.txt”
- `ifdef … `endif
  - if some textmacro is defined, statements inside the `ifdef block will be executed
  - e.g., back-annotate wire delay information for simulation

```
`ifdef SDF_FILE
    initial $sdf_annotate(`SDF_FILE,   test_module)
`endif
```

Verilog System Tasks

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 is 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

Some System Tasks (see *IEEE std. 1364 document* for details)

- file output
  - **$fopen, $fdisplay, $fmonitor, $fwrite, $fstrobe, $fclose**
- display module hierarchy
  - **%m** option in $display, $monitor
- display signal values after all other events of that step
  - **$strobe**
- random number generation
  - **$random**
- initializing memory from file
  - **$readmemb, $readmemh**
- value change dump (vcd) file
  - **$dumpvars, $dumpfile, $dumpon, $dumpoff, $dumpall**

$strobe vs. $display

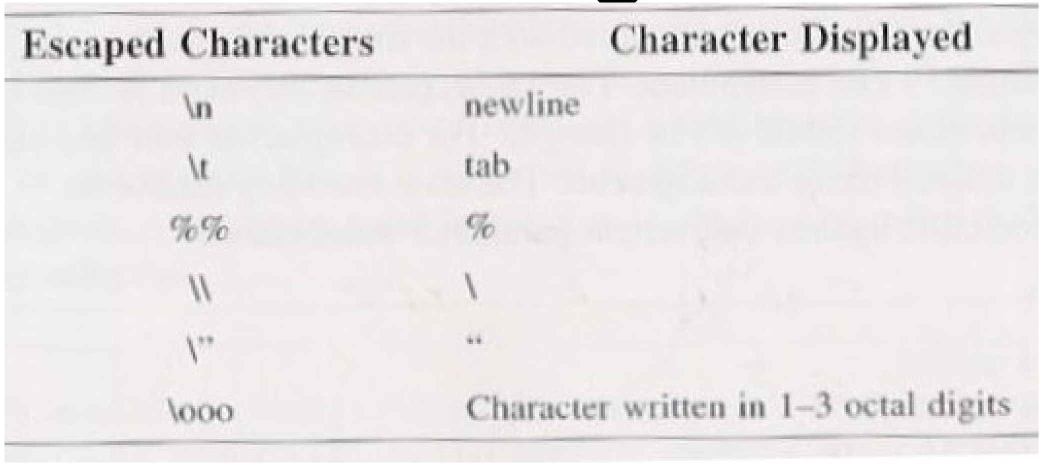
- **$display**
  - if many other statements are executed in the same time unit as the task **$display**, the execution order of the statements and **$displa**y task is non-deterministic (depending on all the scheduled events at that time)
- **$strobe**
  - task **$strobe** is always executed after all other assignment statements in the same time unit have executed
  - provide synchronization mechanism to display data only after all other assignments are executed

Input, Output, Random, Conversion

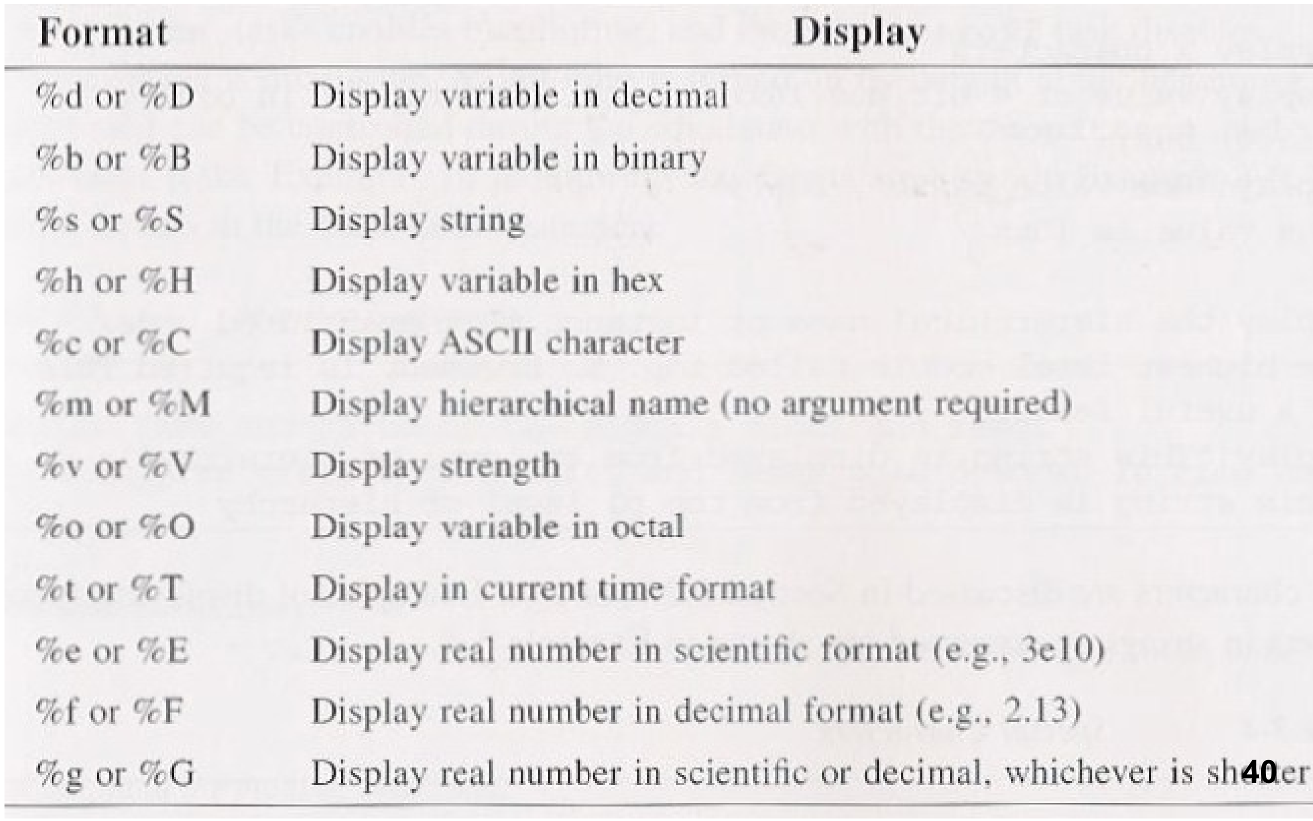
- read from input
  - $readmemb, $readmemh
  - $fscanf, $fget. $fread
- write to output
  - $fopen, $fclose
  - $display, $strobe, $monitor, $write
  - $fdisplay, $fstrobe, $fmonitor, $fwrite
- random generator
  - $random
- suspend ($stop) or finish ($finish) simulation
- conversion functions
  - $signed, $unsigned

Escape character and string format

- %d
  - decimal
- %b
  - binary
- %o
  - octal
- %h
  - hexa-decimal
- %m
  - hierarachical name



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



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 sh[illegible]ter

opening file

```
// multichannel descriptor
integer handle1, handle2, handle3; // integers are 32-bit values

// standard output: descriptor=32’h0000_0001 (bit 0 is set to 1)

initial
begin

// one-hot encoding for file handlers, i.e., only one of the bits is 1

handle1 = $fopen(“file1.out”); // handle1=32’h0000_0002 (bit 1 is set to 1)

handle2 = $fopen(“file2.out”); // handle2=32’h0000_0004 (bit 2 is set to 1)

handle3 = $fopen(“file3.out”); // handle3=32’h0000_0008 (bit 3 is set to 1)

end
```

writing to files, closing files

```
integer desc1, desc2, desc3;
initial
begin

desc1 = handle1 | 1; // desc1=32'h0000_0003
$fdisplay (desc1, "Display 1");    // write to files 'file1.out' & stdout

desc2 = handle2 | handle 1; // desc2=32'h0000_0006
$fdisplay (desc2, "Display 2");    // write to files 'file1.out' & 'file2.out'

desc3 = handle3;
$fdisplay (desc3, "Display 3";    // write to file 'file3.out' only

end

//closing files
$fclose (handle1);
```

Reading data from a file

```
// open a file for read type
// generate the 32-bit file descriptor fd
integer fd = $fopen(“filename”, “r”);

// read a line at a time
// read characters into reg str
// return code = 0 if error, otherwise, code = # of characters read
integer code = $fgets (str, fd);

// read formatted data and place the result in variable args
integer code = $fscanf (fd, format, args);
```

Displaying Hierarchy

```
module M;
…
initial
   $display (“Displaying in %m”);
endmodule

// instantiate module M
module top;
   M m1();   // module instance m1 in top module M
   M m2();  // module instance m2 in top module M
   M m3();   // module instance m3 in top module M
endmodule

// output looks like
// Displaying in top.m1
// Displaying in top.m2
// Displaying in top.m3
```

Random Number Generation ($random)

- *Seed* can be either **reg, integer,** or **time** variable
- **$random** returns a 32-bit *signed* integer

```
module test;
integer r_seed;
reg [31:0] addr;  // input to ROM
wire [31:0] data; // output from ROM
reg [23:0] rand1, rand2;
…
ROM rom1 (data, addr);
…
initial   r_seed = 2;  // arbitrarily define seed as 2, better use different seed values
always @ (posedge clock)
      addr  = $random (r_seed); // generate random numbers of signed integer
     rand1 = $random % 60; // generate random number between -59 and 59
     rand2 = {$random} % 60; // random number between 0 and 59
…
// check output of ROM against expected results
…
endmodule
```

Random Number Generation ($random)

- *Seed* can be either **reg, integer,** or **time** variable
- **$random** returns a 32-bit *signed* integer

```
module test;
integer r_seed;
reg [31:0] addr;  // input to ROM
wire [31:0] data; // output from ROM
reg [23:0] rand1, rand2;
…
ROM rom1 (data, addr);
…
initial   r_seed = 2;  // arbitrarily define seed as 2, better use different seed values
always @ (posedge clock)
      addr  = $random (r_seed); // generate random numbers of signed integer
      rand1 = $random % 60; // generate random number between -59 and 59
      rand2 = {$random} % 60; // random number between 0 and 59
…
// check output of ROM against expected results
…
endmodule
```

initializing memory from file

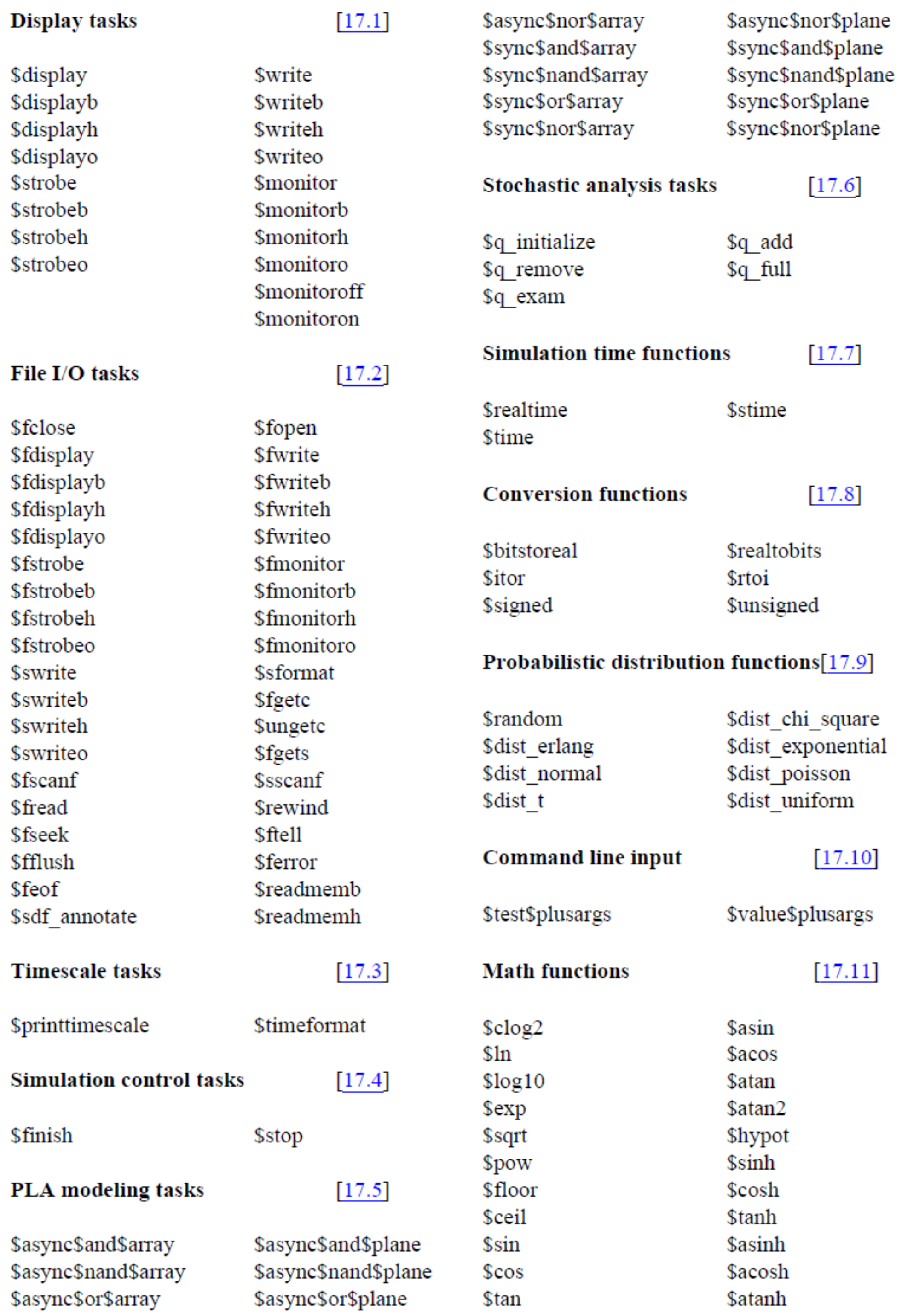
```
module test;
reg [7:0] memory [0:7];
integer i;

initial begin
// address in file init.dat is
// specified as @<address>
// and in hexadecimal numbers
$readmemb("init.dat", memory);

/* $readmemh is for reading in
     hexadecimal data */

for (i=0; i<8; i=i+1)
  $display("memory[%d]=%b", memory[i]);

end
endmodule
```

```
// content of file "init.dat"
// address in hexadecimal
@002
11111111  01010101
00000000  10101010
@006
1111zzzz  00001111
```

```
memory[0]=xxxxxxxx
memory[1]=xxxxxxxx
memory[2]=11111111
memory[3]=01010101
memory[4]=00000000
memory[5]=10101010
memory[6]=1111zzzz
memory[7]=00001111
```

System tasks

- Refer to Ch. 17 (System tasks and functions) in IEEE std. 1364-2005 document **IEEE Standard for Verilog® Hardware Description Language**
- system tasks are useful for writing testbench because most system tasks are non-synthesizable



Display tasks [17.1]

$display	$write
$displayb	$writeb
$displayh	$writeh
$displayo	$writeo
$strobe	$monitor
$strobeb	$monitorb
$strobeh	$monitorh
$strobeo	$monitoro
	$monitoroff
	$monitoron

File I/O tasks [17.2]

$fclose	$fopen
$fdisplay	$fwrite
$fdisplayb	$fwriteb
$fdisplayh	$fwriteh
$fdisplayo	$fwriteo
$fstrobe	$fmonitor
$fstrobeb	$fmonitorb
$fstrobeh	$fmonitorh
$fstrobeo	$fmonitoro
$swrite	$sformat
$swriteb	$fgetc
$swriteh	$ungetc
$swriteo	$fgets
$fscanf	$sscanf
$fread	$rewind
$fseek	$ftell
$fflush	$ferror
$feof	$readmemb
$sdf_annotate	$readmemh

Timescale tasks [17.3]

$printtimescale	$timeformat

Simulation control tasks [17.4]

$finish	$stop

PLA modeling tasks [17.5]

$async$and$array	$async$and$plane
$async$nand$array	$async$nand$plane
$async$or$array	$async$or$plane
$async$nor$array	$async$nor$plane
$sync$and$array	$sync$and$plane
$sync$nand$array	$sync$nand$plane
$sync$or$array	$sync$or$plane
$sync$nor$array	$sync$nor$plane

Stochastic analysis tasks [17.6]

$q_initialize	$q_add
$q_remove	$q_full
$q_exam	

Simulation time functions [17.7]

$realtime	$stime
$time	

Conversion functions [17.8]

$bitstoreal	$realtobits
$itor	$rtoi
$signed	$unsigned

Probabilistic distribution functions[17.9]

$random	$dist_chi_square
$dist_erlang	$dist_exponential
$dist_normal	$dist_poisson
$dist_t	$dist_uniform

Command line input [17.10]

$test$plusargs	$value$plusargs

Math functions [17.11]

$clog2	$asin
$ln	$acos
$log10	$atan
$exp	$atan2
$sqrt	$hypot
$pow	$sinh
$floor	$cosh
$ceil	$tanh
$sin	$asinh
$cos	$acosh
$tan	$atanh

conditional execution $test$plusargs

- All codes are compiled but executed conditionally
- flag set at *run time* by the option +DISPLAY_VAR when running the simulator tool

```
module test;

initial
begin
      if ( $test$plusargs("DISPLAY_VAR")  )
                $display("display=%b", {a,b,c});
      else
                $display("no display")
end
// the variables are displayed only if flag DISPLAY_VAR is set at run time

endmodule
```

invoke simulator with option +DISPLAY_VAR in the tool command line

conditional execution $value$plusargs

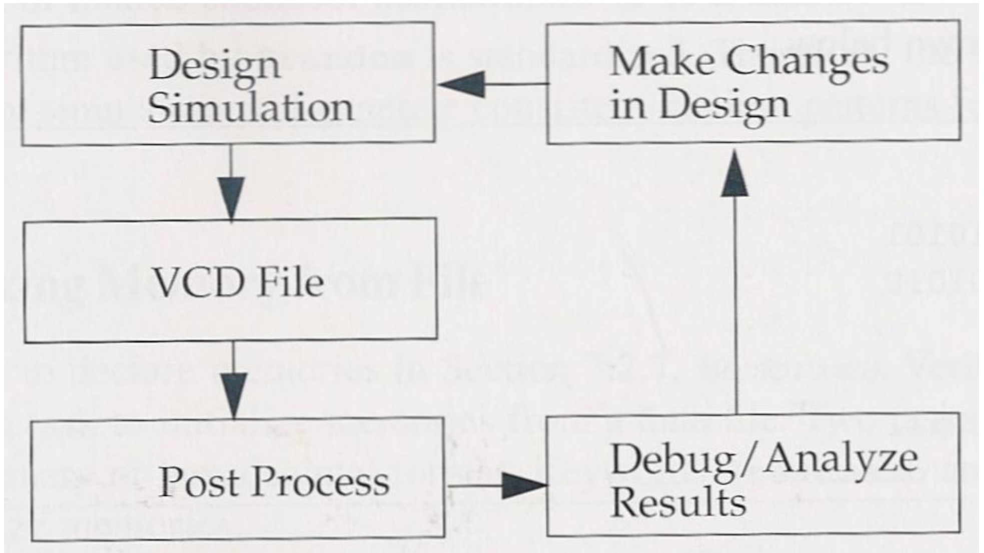
- **$value$plusargs** return 0 if a matching invocation is not found and nonzero if a matching option is found

```
module test;
reg [8*128-1:0] test_string;
integer clk_period;
initial
begin
      if ( $value$plusargs(“testname=%s”, test_string) )
          $readmemh (test_string, vectors);
      else
          $display(“test name not specified”);

      if ( $value$plusargs(“clk_t=%d”, clk_period) )
          forever #(clk_period/2) clk=~clk;
      else
          $display(“clk period not specified”);
endmodule
```

- invoke simulator with option +testname=test1.vec +clk_t=10 in the tool command line

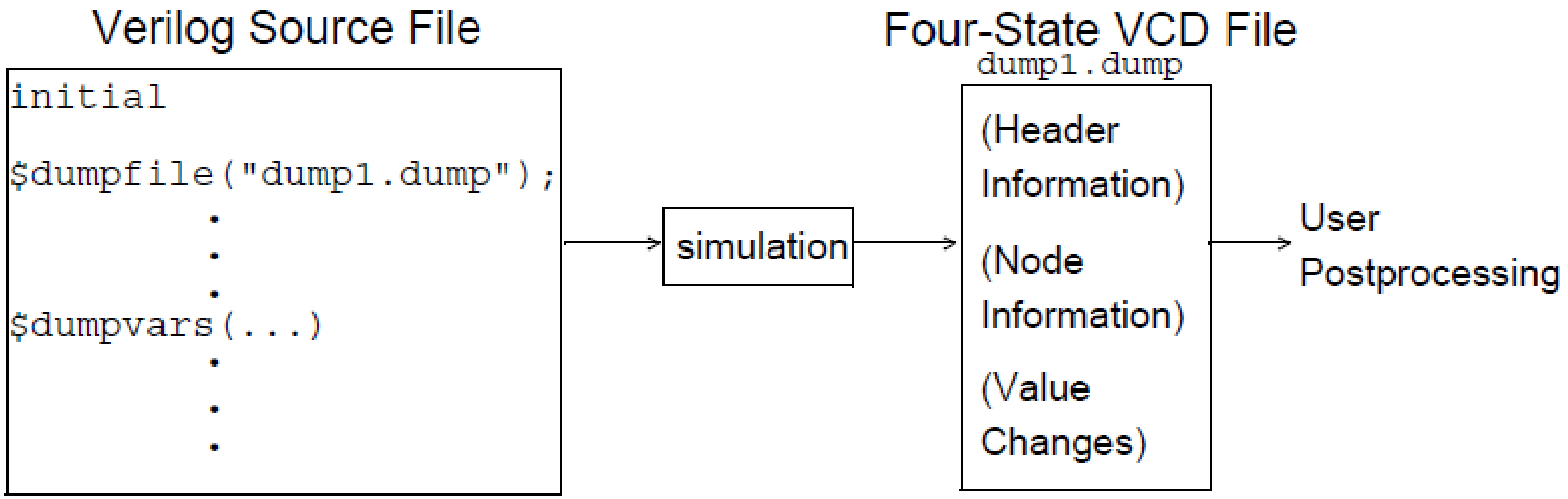
Value Change Dump (VCD) file

- A *VCD file* contains information about value changes on selected variables
- post-processing tools (such as Synopsys PrimeTime) can take the VCD file as input and visually display waveforms for easy debugging and analyzing
  - Synopsys PrimeTime is *dynamic timing analysis* tool that requires test input patterns to analyze the delay and power
  - c.p. Synopsys Design Compiler is static timing analysis (STA) tool which does not require any input test pattern
- VCD could be very large
  - only dump signals that need to be examined



creating 4-state VCD file

- insert VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped
- run the simulation
- 4-state VCD file
  - variable changes in 0, 1, x, z, with no strength information



```
initial
$dumpfile("dump1.dump");
        .
        .
        .
$dumpvars(...)
        .
        .
        .
```

system tasks for VCD

```
// dump simulation information into myfile.dmp
   inital   $dumpfile ("myfile.dmp");

// select module instances or module instance signals to dump,
// cause dump to start at the end of current simulation time
   Initial  $dumpvars; // no argument, dump all signals

// dump variables in module instance top, dump one hierarchy level below top
   initial   $dumpvars (1, top);

// dump up to 2 levels of hierarchy below top.m1
   initial   $dumpvars (2, top.m1);

// number 0 means dump the entire hierarchy below top.m1
   initial  $dumpvars (0, top.m1);

initial
   begin
   $dumpoff; // suspend the dump
   #10000; $dumpon; // resume the dump
   end
initial  // create a checkpoint. dump current values of all VCD variables
   $dumpall;
```

Compiler Derivatives

Compiler Directives ` (back quote)

- **`define**
  - define text macro for later text macro substitution

  `define WORD_SIZE 32
  // used as **`WORD_SIZE** in the code such as
  // **wire** [**`WORD_SIZE**-1:0] a, b, c;
- **`include**
  - include another Verilog file, e.g.,
  - **`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

`define `ifdef `else `endif

```
`define TEXT_MACRO_NAME // comment out this line if do not want this definition
…
// do not confuse with if … else in always procedure block which synthesize MUX
// compiler directive `ifdef … `else … does not synthesize MUX hardware
`ifdef TEXT_MACRO_NAME   // conditional compilation with `else
   // compile the following statements here if TEXT_MACRO_NAME is defined
   …
`else     // `else compiler directive is optional
   // compile the following statements here if TEXT_MACRO_NAME is not defined
   …
`endif

`ifdef TEXT_MACRO_NAME  // conditional  compilation without `else
   // compile the following statements here if TEXT_MACRO_NAME is defined
   // the following statements are skipped if TEXT_MACRO_NAME is not defined
   …
`endif
```

conditional compilation

- the conditional compile flag can be set by using the **`define** statement inside the Verilog file

```
`ifdef TEST  // compile module test only if text macro TEST is defined using `define
     module test; …. endmodule
`else // compile module stimulus by default
    module stimulus; … endmodule

module top;

bus_master b1();   // instantiate module unconditionally
`ifdef ADD_B2    bus_master b2(); // b2 is instantied if text macro ADD_B2 is defined
`elseif ADD_B3   bus_master b3(); // b3 is instantied if text macro ADD_B3 is defined
`else            bus_master b4(); // b4 is instantiated by default
`endif

`ifndef IGNORE_B5   bus_master b5();
                    // b5 is instantiated only if text macro IGNORE_B5 is not defined
`endif

endmodule
```

Compiler Directives ` (back quote)

- **`define**
  - define text macro for later text macro substitution

  `define WORD_SIZE 32

  // used as **`WORD_SIZE** in the code such as

  // **wire** [**`WORD_SIZE**-1:0] a, b, c;
- **`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

\`define \`ifdef \`else \`endif

```
`define TEXT_MACRO_NAME
…
`ifdef TEXT_MACRO_NAME   // conditional compilation with `else
  statement_1;
  …
`else     // `else compiler directive is optional
  altenative_statement_1;
  …
`endif

`ifdef TEXT_MACRO_NAME  // conditional  compilation without `else
  statement_1;
  …
`endif
```

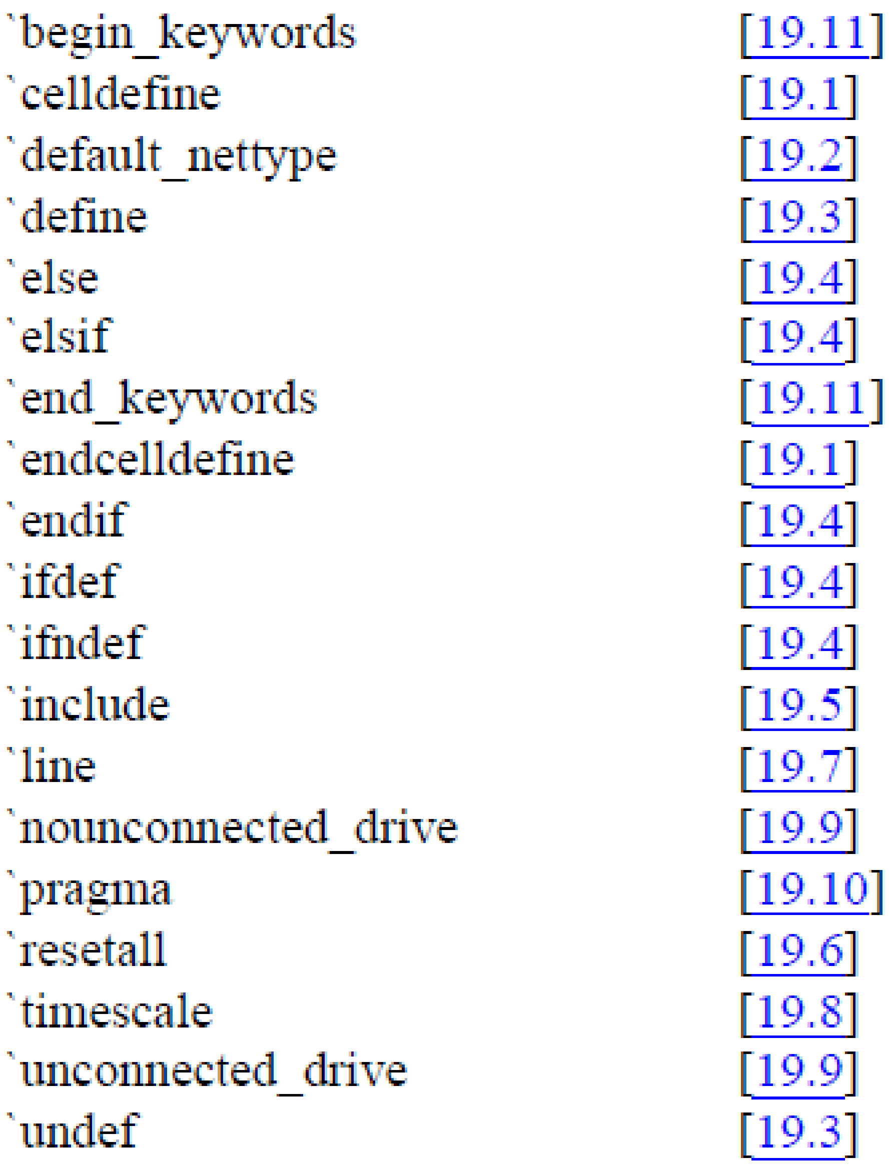
`timescale

- **`timescale** <time_unit> / <time_precision>
- Only 1, 10, 100 are valid integers

```
`timescale 100ns/1ns
module dummy;
reg toggle;
initial toggle=1'b0;
always #5
begin
   toggle = ~toggle;
   $display("%d, In %m toggle=%b", $time, toggle);
   // 5, in dummy toggle=1
   // 10, in dummy toggle=0
   //  15, in dummy toogle=1  ….
end
endmodule

`timescale 1us/10ns  // change time scale for module dummy2
module dummy2;
```

Compiler Directive

- Refer to Ch. 19 (Compiler directives) in IEEE std. 1364-2005 document **IEEE Standard for Verilog® Hardware Description Language**

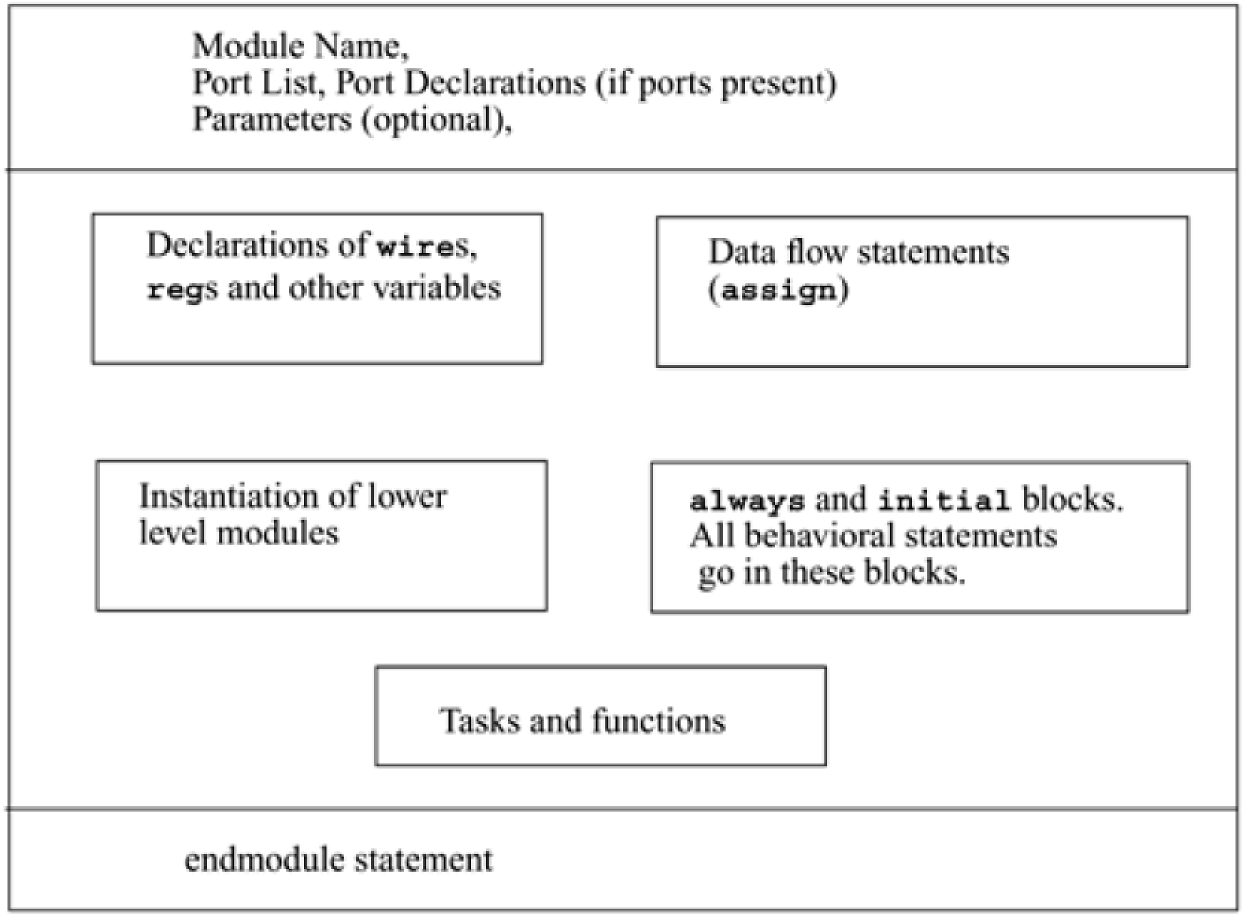


`begin_keywords	[19.11]
`celldefine	[19.1]
`default_nettype	[19.2]
`define	[19.3]
`else	[19.4]
`elsif	[19.4]
`end_keywords	[19.11]
`endcelldefine	[19.1]
`endif	[19.4]
`ifdef	[19.4]
`ifndef	[19.4]
`include	[19.5]
`line	[19.7]
`nounconnected_drive	[19.9]
`pragma	[19.10]
`resetall	[19.6]
`timescale	[19.8]
`unconnected_drive	[19.9]
`undef	[19.3]

Verilog Operators, Data Types

Components of a Verilog Module

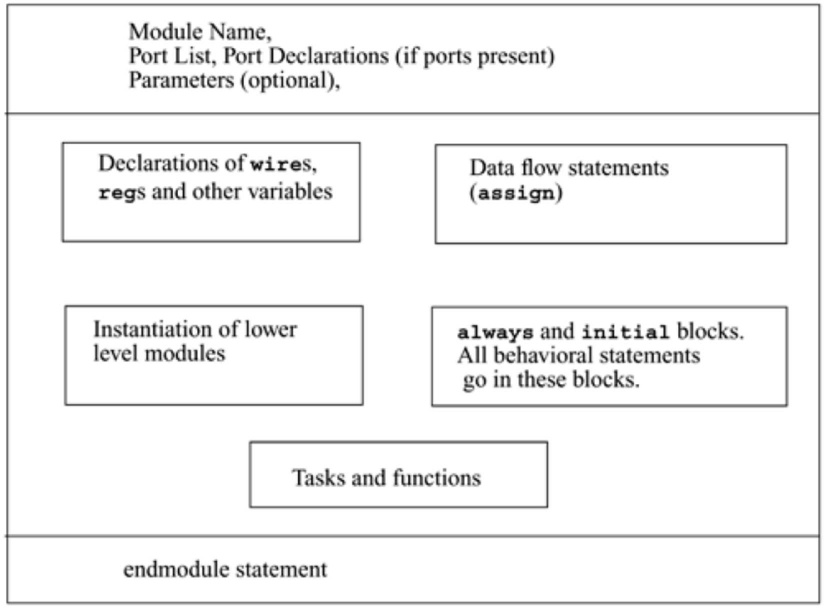
- declaration
  - inputs, outputs
  - parameters
  - data types
- structure model
- module instantiation
- dataflow model
  - continuous assignment
- behavior model
  - procedural assignment



Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

```
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;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end
```

```
// dataflow: continuous assignment
assign W1 = expr ;  // use operators

// module instances: structural modeling
COMP U1 (.PP2(W2), .PP1(W1));
COMP U2 (W3, W4);

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction

endmodule
```


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Verilog Operators

a = ~b // **not** b (unary operator); only one operand: b

// output could be multiple bits, the same bit-width as the input

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

// two operands: b, c

// output a is either TRUE (logic 1) or FALSE (logic 0)

// cp. a = b **&** c; // bitwise (bit-by-bit) operator

// output of bit-wise operator could be multi-bit

// with the same bit-width as that of inputs

a = b**?** c **:** d; // **MUX2** (ternary operator)

// a=c if b=1,

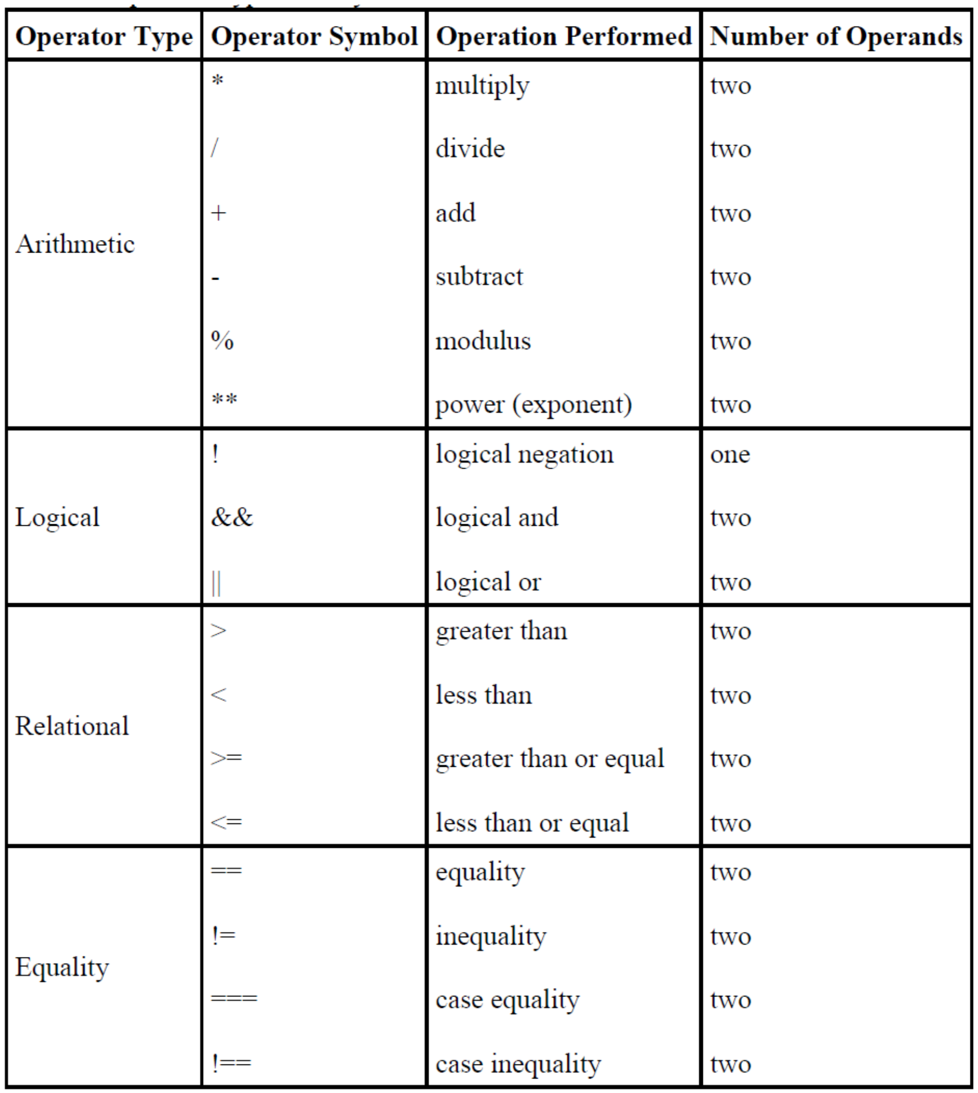
// a=d if b=0

// three operands: b, c, d

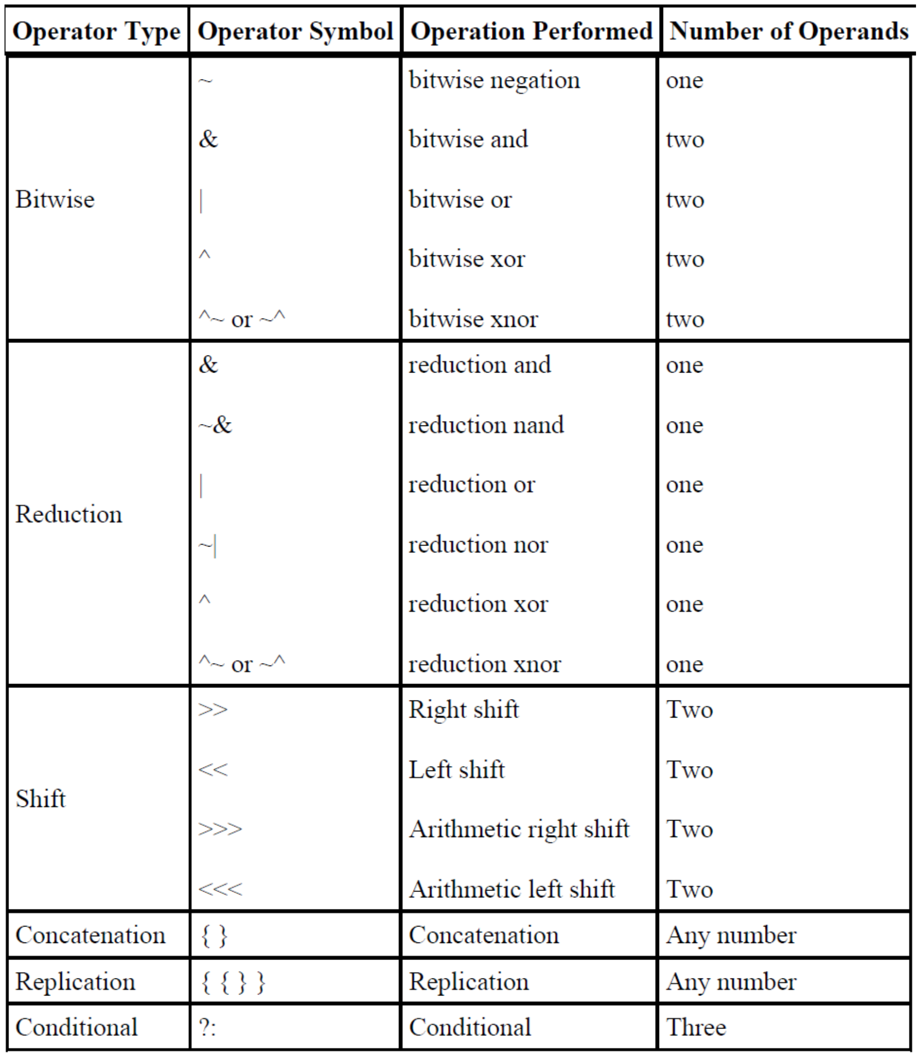
Other operators

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

Operator Types and Symbols



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



Operator Type	Operator Symbol	Operation Performed	Number of Operands
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

Equality Operator

- $A=4_{10}=0100_2$, $B=3_{10}=0011_2$,

  A == B // results in FALSE, (logical 0)

  X != Y   // result in TRUE (logical 1)

  X == Z   // result in x

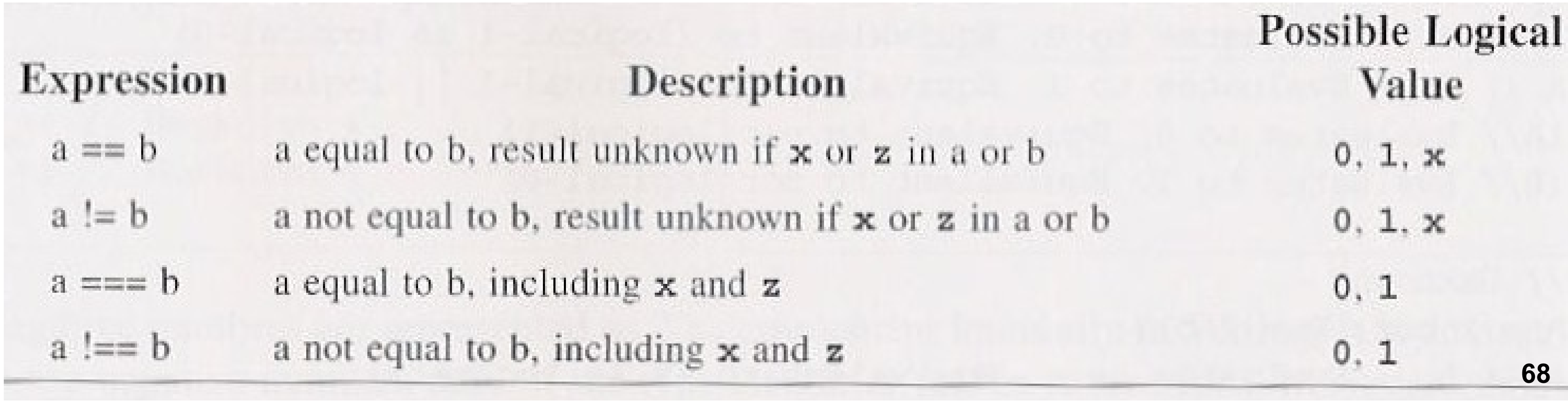
  Z === M   // result in logical 1 TRUE (all bits match, including x and z)

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

  M !== N   // result in logical 1 TRUE

  ===, !== are in general non-synthesizable

X=4’b1010
Y=4’b1101
Z=4’b1xxz
M=4’b1xxz
N=4’b1xxx



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

Equality Operator: Logical vs. Case

- logical equality operator (==)
  - 0, 1, x (unknown), z (floating)

==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

- case equality operator (===)

===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

reduction operator

- perform bitwise operator on all bits of a single vector and yield a 1-bit result
- e.g.

```
X = 4’b1010
&X   // = 1&0&1&0 = 1’b0
|X    // = 1|0|1|0 = 1’b1
^X   // = 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’b1111_0101

Number

unsized numbers (decimal by default), e.g., b=a+1;

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 such as `include

e.g.,

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

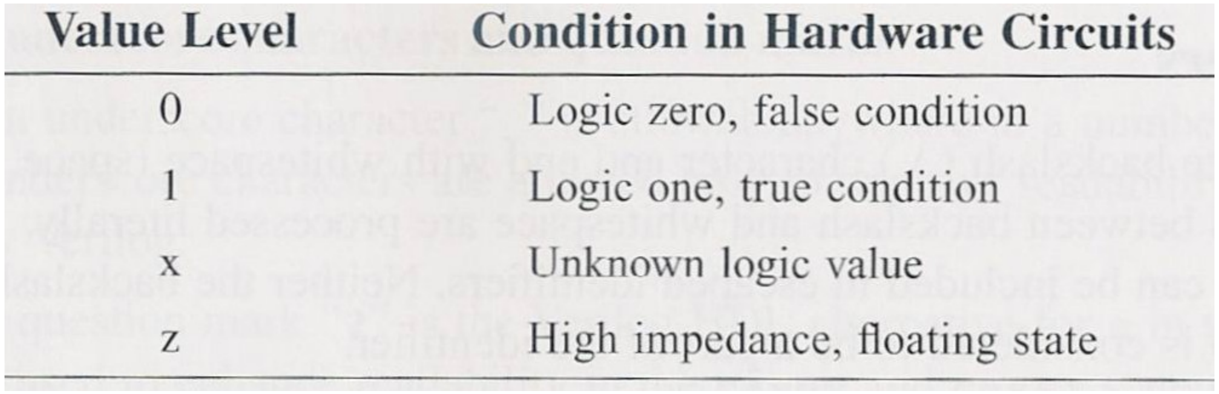
12'**h**abc // 12-bit hexadecimal number = $1010\_1011\_1100_2$

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

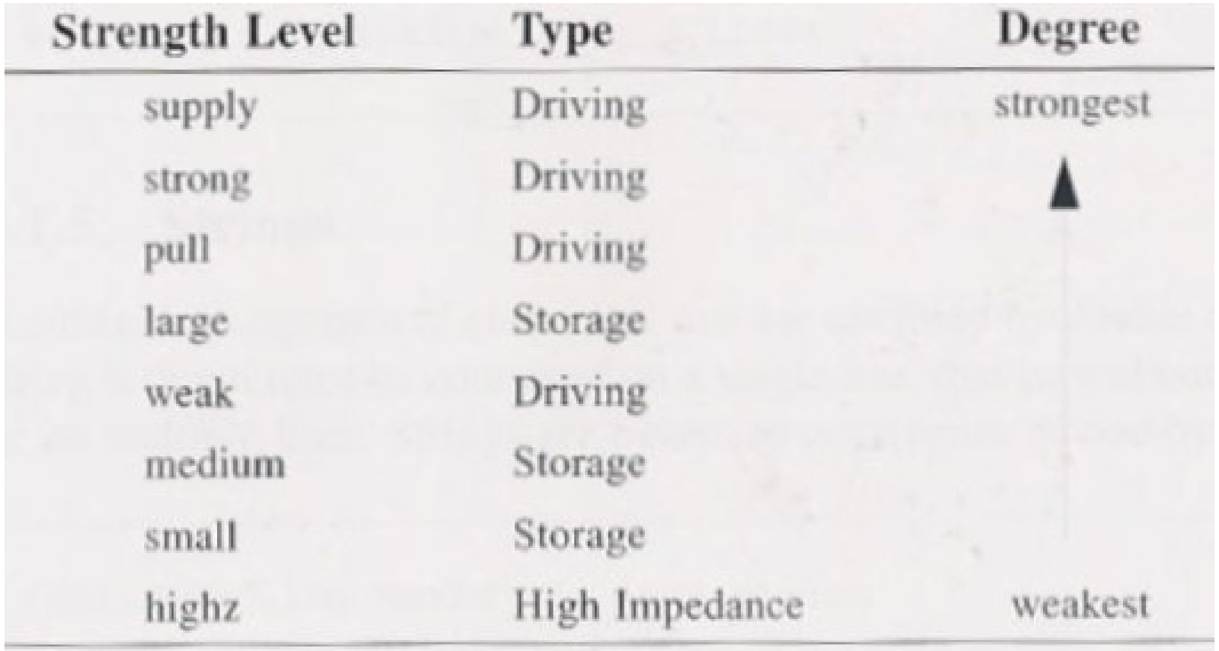
'**b**1111 // $1111_2=15_{10}$ extended to required bit-width

Data types: value set

- 4 values (0, 1, z, x), 8 strengths
  - strength is used when two data signals are conflicting



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	
highz	High Impedance	weakest

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, i.e., floating node

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

- Nets (**wire**): connections between hardware elements
  - nets have values continuously driven by outputs of devices they are connected to, using continuous assignment **assign**

  e.g.: **wire** a, b, c;

  wire a = b & c; // equivalent **to** **wire** a; **assign** a = b & c;
- Registers (**reg**): data storage element
  - a variable that can hold a value in procedural assignment blocks starting with **always** or **initial**
  - *not necessarily* mean a flip-flop (one-bit register) in real circuit

  e.g.: **reg** q;

  always @ (posedge clk) q <= d;
- SystemVerilog use **logic** to replace 4-value **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 signal, bus[7] is the MSB, bus[0] is the LSB
  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;   // MSB->LSB with decreasing index order
      reg [0:255] data2;   // MSB->LSB with increasing index order
      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]

e,g,.  reg [255:0] data1;
      for (j=0; j<=31; j=j+1) byte = data1[(j*8)+:8] ;
      // sequence is [7:0], [15:8], [23:16], …, [255:248]
```

register data types (integer, real, time)

- **integer**: register data type used for signed quantity
  - **reg** store a multi-bit signal which could mean a *unsigned* quantity, while **integer** store *signed* quantity

    e.g., **integer** counter;   counter = -1; // used as a counter
- **real**: real number constant or register data type

  e.g. **real** delta;

    delta = 4e10; delta = 2.14; // delta is a real variable
- **time**: register data type to store simulation time

  e.g. **time** save_sim_time;

    save_sim_time = **$time**; // define a time variable

      // system task **$time** is invoked to get current simulation time
- **real** and **time** are *non-synthesizable* codes, for simulation only

signed vs. unsigned

- unsigned (default Verilog data type)

```
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 (new data type added in 2001)

```
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 numer
```

Arithmetic (unsigned vs. signed)

- unsigned

```
wire [7:0] a, b, s;
wire [8:0] s1;
wire [15:0] p;
wire c;

assign s = a + b;   // no overflow detection (e.g., 255+1 =  1111_1111+0000_0001 = 0000_0000 = 0)
// assign s1 = {1'b0, a} + {1b'0, b};  // adding 1 bit of zero to prevent overflow
// assign s1 = a + b;  // implicit extension by Verilog
assign p = a * b;
```

- signed

```
wire signed [7:0] a, b, s;
wire signed [8:0] s1;
wire signed [15:0] p;

wire signed [8:0] s;

assign s = a + b;    // no overflow detection (e.g., (-1)+(-255) = 1111_1111+1000_000 = 0111_1111 = +127
// assign s1 = {a[7], a} + {b[7], b};  // sign extension by 1 bit to prevent overflow in 2's complement
// assign s1 = a + b;  // implicit sign extension by Verilog
assign p = a * b;
```

- both unsigned and signed generate the same results in case of normal operation (without overflow or bit-width mis-match

Same Bit-Width in Assignment

- In most normal cases, LSH and RHS of assignments have the same bit-width
  - unsigned and signed types represent same hardware

```
wire [7:0] a, b, c;
assign c = a+b;
```

```
wiresigned  [7:0] a, b, c;
assign c = a+b;
```

- avoid assignments with mis-matched bit-width
  - otherwise, synthesizer might expand bit-width which is not as you expect

```
wire [7:0] a, b;
wire [8:0] c;
assign c = a+b;
// if a, b=8'b1000_0000=+128
// c=9'b1_0000_0000=+256
```

```
wire [7:0] a, b;
wire [8:0] c;
assign c = a+b;
// if a, b=8'b1000_0000= -128
// c=9'b1_0000_0000= -256
```

data types (array and memory)

- Arrays

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

  // array of 8 elements, with each element 5-bit wide

  port_id[6] = 5`b00000; // the array element with index=6

  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 (1024x8) memory

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

- Use tool-supported memory generator/compiler (instead of registers) to create area-efficient memory (instead of area-costly flip-flop-based array)

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 // parameter values cannot be changed

state1=4’b0001,

state2=4’b0010,

state3=4’b0100,

state4=4’b1000;

Overriding Parameters (defparam)

```
// module #(parameter id_num=0) hello_world;
module hello_world;
   parameter id_num = 0;
   initial $display(“hello _world id number=%d”, id_num);
endmodule

module top;
   // defparam is considered a bad coding style !!!
   // can use #() to overwrite the default parameter values
   defparam   w1.id_num=1, w2.id_num=2;
   hello_world w1();        // hello_world #(1) w1();
   hello_world w2();        // hello_world #(2) w2();
endmodule
```

Module instance parameter values (#)

```
module #(parameter delay1=2, delay2=3, delay3=7);
module bus_master;
parameter delay1=2;
parameter delay2=3;
parameter delay3=7;
endmodule

module top;
bus_master #(4,5,6) b1(); // delay1=4, delay2=5, delay3=6
bus_master #(9,4) b2();  // delay1=9, delay2=4,
bus_master #(.delay2(4), .delay3(7)) b3();
                                       // delay2=4, delay3=7
endmoudle
```

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 // parameter values cannot be changed

state1=4'b0001,

state2=4'b0010,

state3=4'b0100,

state4=4'b1000;

Notes on Dataflow Statements (1/2)

- dataflow statements with continuous assignment usually describe “combinational logic”
  - pure feed-forward datapath
  - LHS signal should be different from the RHS signals
  - e.g., assign c=a+b; // describe an adder
  - e.g., assign d=c+1; // describe an adder with one input=1
  - e.g., assign c=c+1; // Not OK!!! , c in LHS and RHS
    - ✓ a loop in the datapath
  - use register in behavioral modeling to describe a counter
  - e.g.,

```
// behavioral statements are inside always block
// draw the hardware of the following statements
reg [7:0] c;
always @ (posedge clk)
   c <= c+1;
```

Notes on Dataflow Statements (2/2)

- all dataflow statements are concurrent
  - different order of statements results in same hardware
  - e.g.,

```
// an adder, a subtractor,
// and a multiplier
wire [7:0] a, b, c, d
assign c=a+b;
assign d=a-b;
assign e=c*d;
```

```
// a subtractor, a multiplier,
// and a adder
wire [7:0] a, b, c, d
assign d=a-b;
assign e=c*d;
assign c=a+b;
```

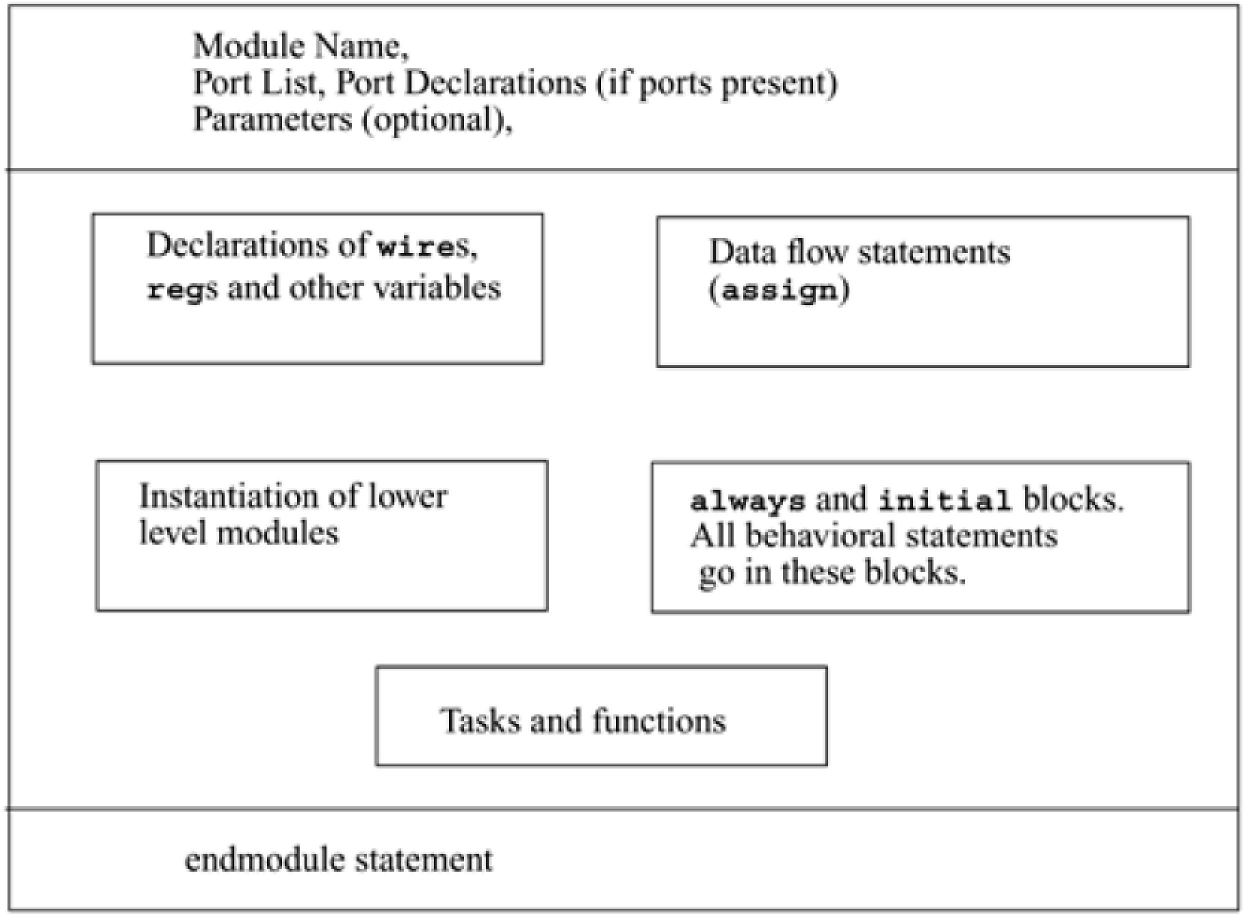
- similar behavioral statements could describe the same combinational logic
  - e.g., inside always block

```
reg [7:0] a, b, c, d;
always @(a, b, c, d)
begin
   c=a+b;
   d=a-b;
   e=c*d;
end
```

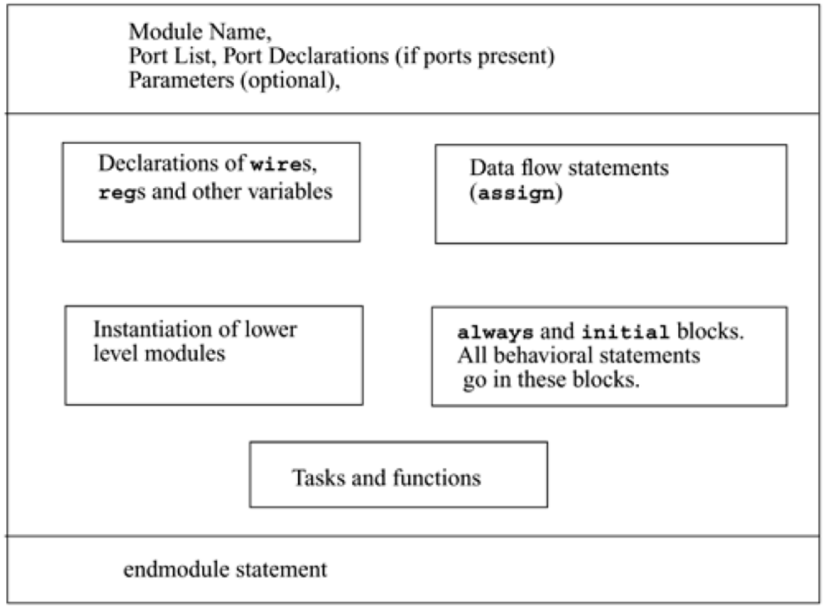
Modules and Ports

Components of a Verilog Module

- declaration
  - inputs, outputs
  - parameters
  - data types
- ***structure*** model
- module instantiation
- ***dataflow*** model
  - continuous assignment
- ***behavior*** model
  - procedural assignment


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

```
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;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end

// dataflow: continuous assignment
assign W1 = expr ;  // use operators

// module instances: structural modeling
COMP U1 (.PP2(W2), .PP1(W1));
COMP U2 (W3, W4);

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction

endmodule
```



Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Structural, Dataflow, Behavioral Modeling

- structural modeling
  - instantiation of other modules
  - e.g. FA instance_name (sum, cout, a, b, cin);
- dataflow modeling
  - continuous assignment **assign**
  - e.g., **assign** {cout, sum} = a + b + cin;
- behavioral modeling
  - procedural assignments inside **always @(…)** block
  - e.g.

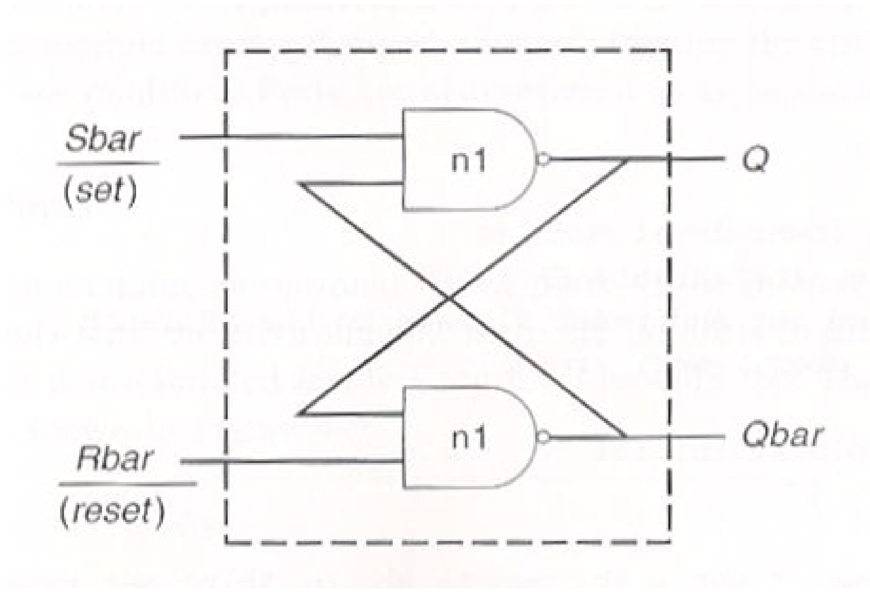
```
reg [31:0] sum;
reg cout;

always @ (a, b, cin)
   {cout, sum} = a + b + cin;
```

Statements inside Behavioral Modeling

```
#delay  // neglected during synthesis
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)  …
```

Example: SR latch

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



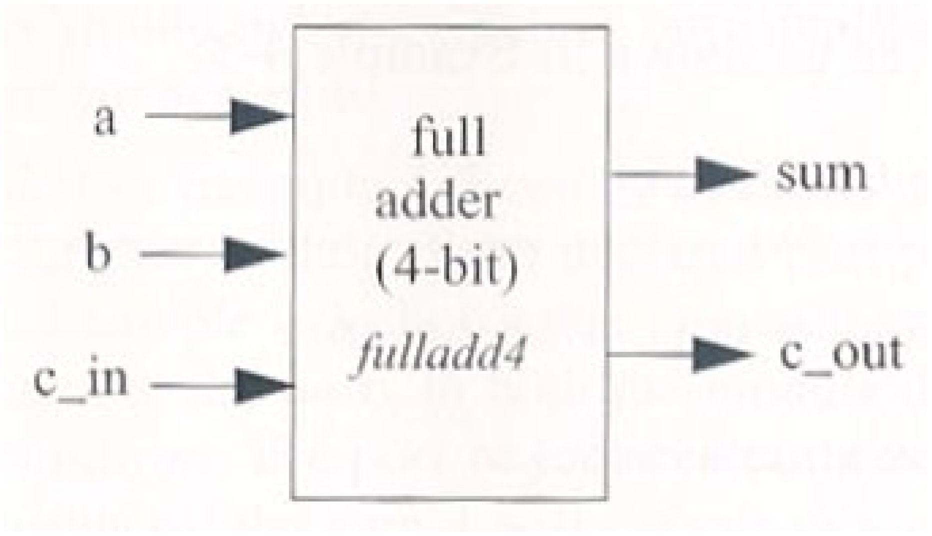
```
module top;  // testbench for the SR_latch
wire q, qbar;
reg set reset;

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 (1995 vs. 2001)

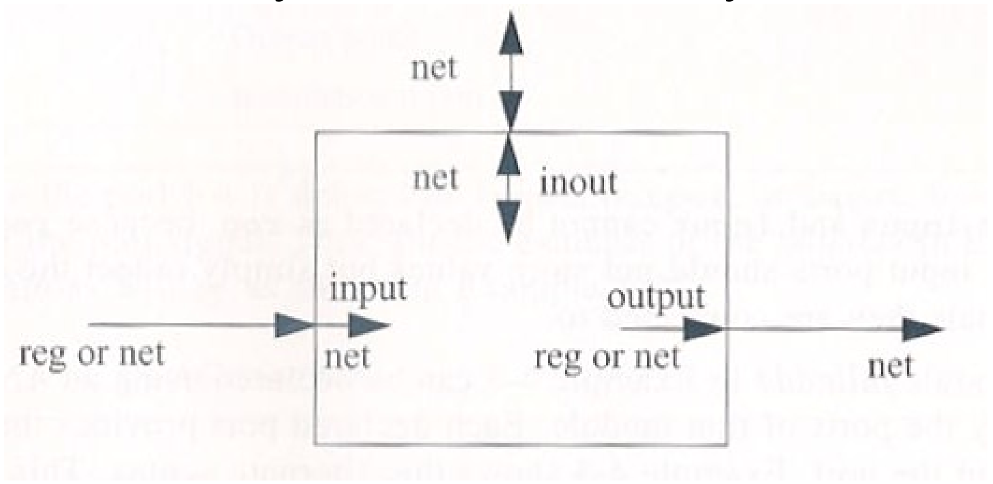
```
// IEEE 1364-1995 (old)
module fulladd4(sum, c_out, a, b, c_in);
parameter width = 4;
output [width-1:0] sum;
output c_out;
input [width-1:0] a, b;
input c_in;
reg [width-1:0] sum;
reg c_out;
…
endmodule
```

```
// 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
```

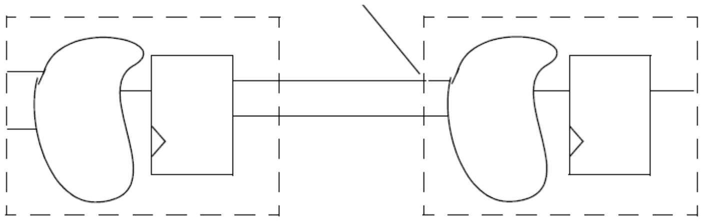


Port Connection Rules

- all port declarations are implicitly declared as net **wire**
- **input** and **inout** cannot be declared as **reg**
- **input**
  - internally (inside the module), always be of the type **wire**
  - externally (outside the module), 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



Latched module provides Enough driving strength, And break critical path



Common Errors in Data Types

- a procedural assignment (inside **always** block) is made to a net (wire)
  - error message: “illegal left-hand-side assignment”
- signal (in the externally upper module) connected to an output port of a module instance is a register
  - error message: “illegal output port specification”
- an input of a module is declared as a register
  - error message: “incompatible declaration”
- signal of a primitive gate output is a register
  - error message: “gate has illegal output specification”

Connecting Ports to External Signals

- Module instantiation

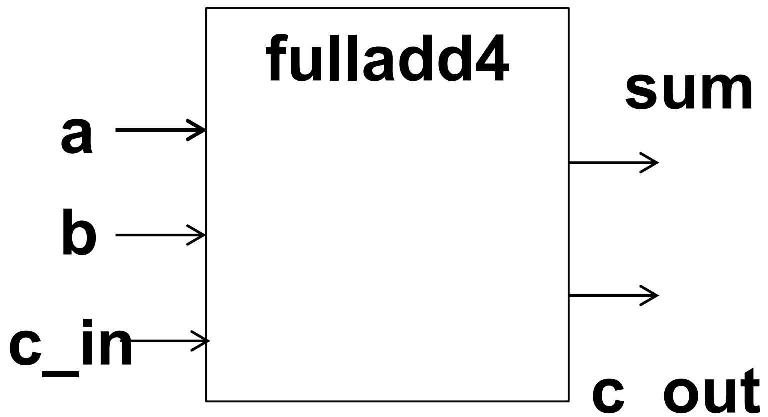
✓ connecting by order list

```
// instantiate   fulladd4 (output sum, c_out, input 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 user 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));
```

array NOT allowed in I module I/O Ports

- module input/output ports could be data type of bit-vector, but not array
  - e.g., MUX4 (input [3:0] in, [1:0] s, output [3:0] out); // OK
  - e.g., MUX4 (input in [3:0], [1:0] s, output out); // NOTOK
- use long bit-vector as input/outputs and make conversion inside the module
  - e.g. MUX4 with 16-bit inputs/output

```
module MUX4 (input [4*16-1:0] in_bv, [1:0] s, output [15:0] out);
reg [15:0] in[0:3];
always@ (*) begin
  for (i=0; i<4; i=i+1)  in[i]= in_bv[16*i +:16]; // long bit-vector -> array
…
endmodule
```

another example: DP4

- 4-D dot-product: a[0]*b[0]+ a[1]*b[1]+ a[2]*b[2]+ a[3]*b[3]

```
module DP4 (input[16*4-1:0]in_a, in_b, output [31:0]out);
wire [15:0] a[3:0],  b[3:0];

// convert two long bit-vectors in_a, in_b into two arrays a, b
assign {a[3],a[2],a[1],a[0]}=in_a;
assign {b[3],b[2],b[1],b[0]}=in_b;

integer i;
reg [31:0] tmp[0:3];
always @ (*) begin
   tmp[0] = a[0]*b[0];
   for (i=1; i<=3; i=i+1) tmp[i] = a[i]*b[i] + tmp[i-1];
end

assign out = tmp[3];

endmodule
```

another example: DPn

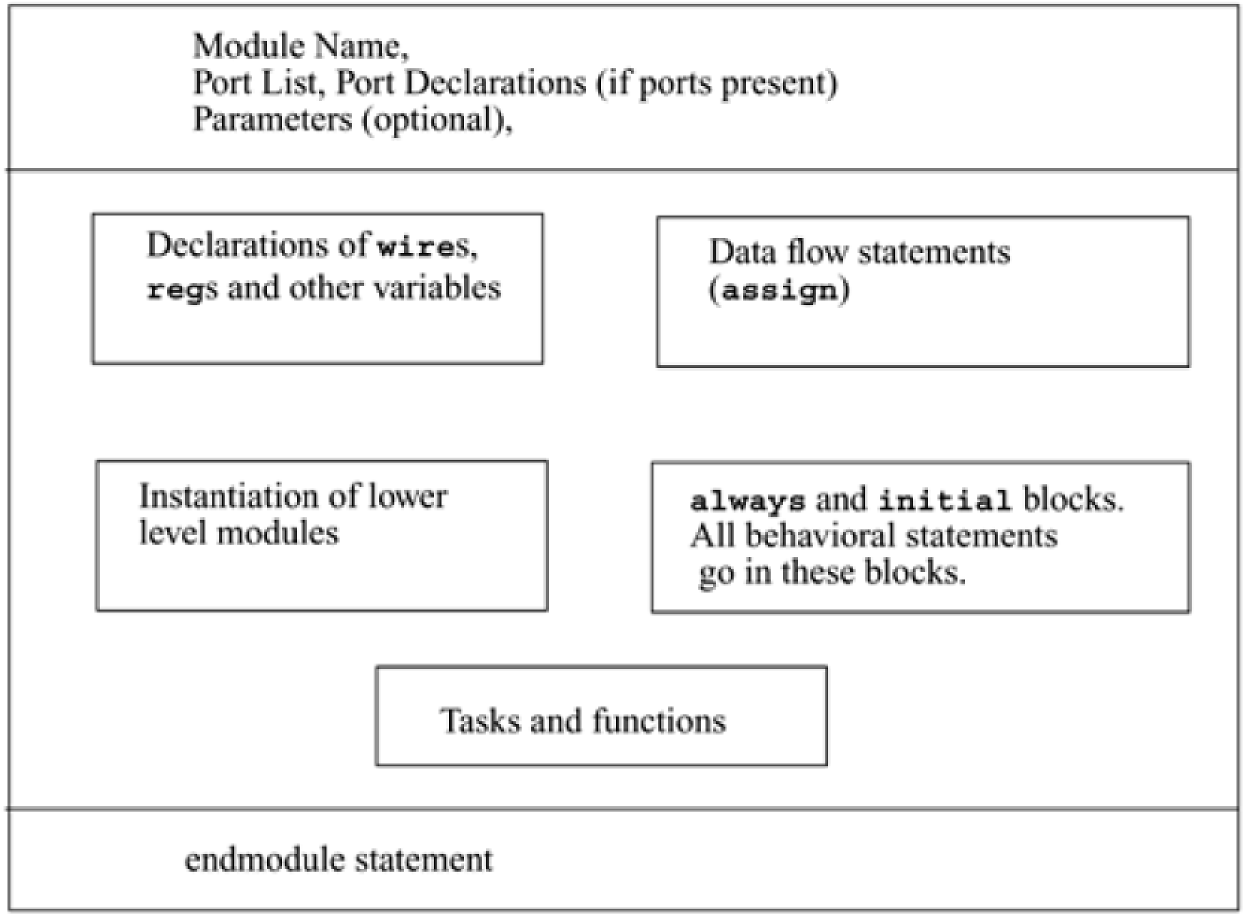
- n-D dot-product

```
module #parameter n=4 DPn (input[16*n-1:0]in_a, in_b, output [31:0]out);
reg [15:0] a[n-1:0];
reg [15:0] b[n-1:0];
integer i;
always@ (*) begin   // convert two long bit-vectors into two arrays
          for (i=0; i<n; i=i+1) begin
                    a[i]= in_a[16*i +:16];
                    b[i]= in_b[16*i +:16];
          end
end

reg [31:0] tmp[0:n];
always @ (*) begin
  tmp[0] = 0;
  for (i=0; i<=n-1; i=i+1) tmp[i+1] = a[i]*b[i] + tmp[i];
end
assign out = tmp[n];
endmodule
```

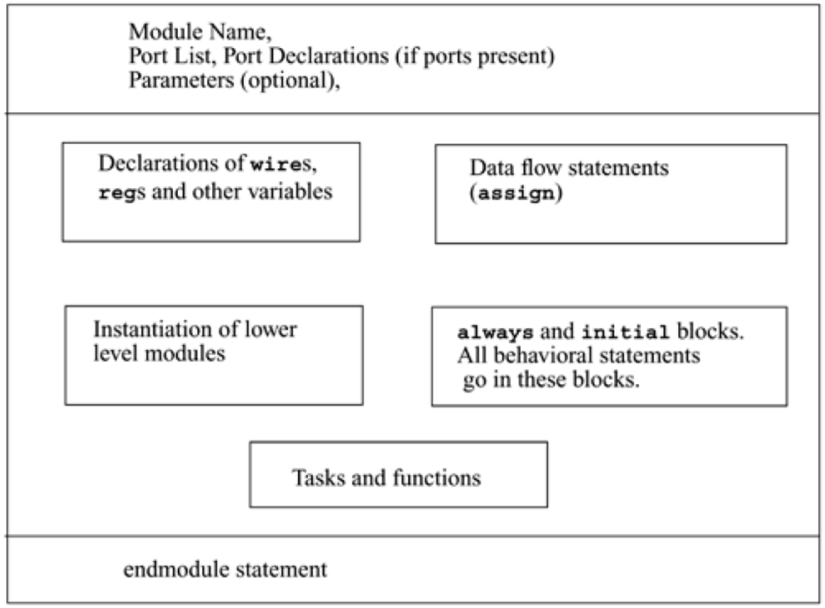
Structural-Level (Gate-Level) Modeling

Components of a Verilog Module

- declaration
  - inputs, outputs
  - parameters
  - data types
- ***structure*** model
- module instantiation
- ***dataflow*** model
  - continuous assignment
- ***behavior*** model
  - procedural assignment


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

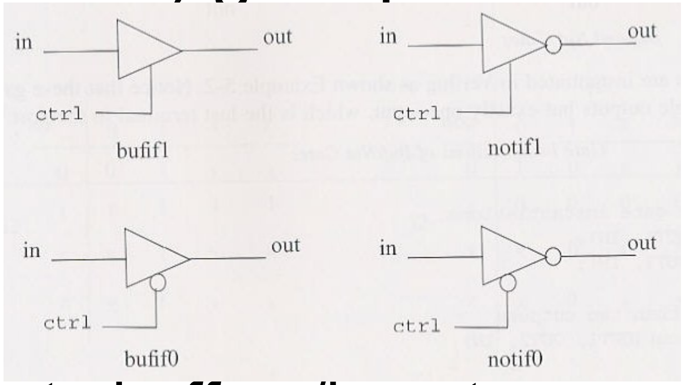
```
module M (P1, P2, P3, P4);
input P1, P2;
output [7:0] P3;
inout P4;

reg [7:0] R1, M1[0:1023];
wire W1, W2, W3, W4;
wire [3:0] W5;
parameter C1=const;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end

// dataflow: continuous assignment
assign W1 = expr ;  // use operators

// module instances: structural modeling
COMP U1 (.PP2(W2), .PP1(W1));
COMP U2 (W3, W4);

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction

endmodule
```



Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Verilog predefined (built-in) gates

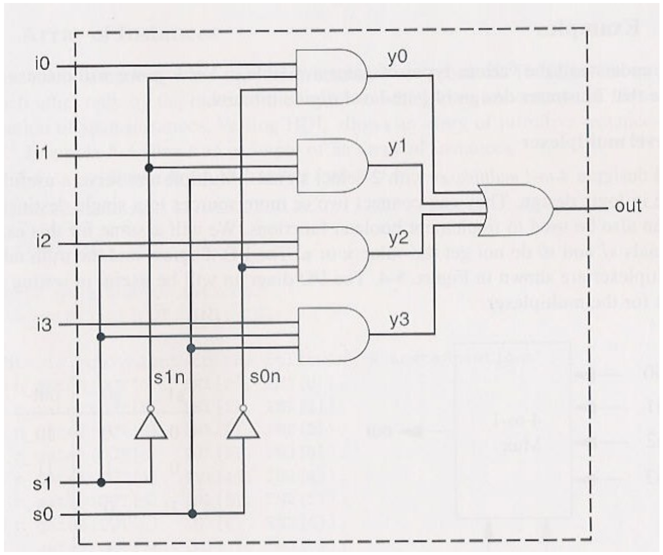
- basic logic gate as predefined (built-in) gate primitives
  - **and, or, nand, not, xor, xnor**
    - ✓eg., **and** a1(out, in1, in2);
  - **buff, not**
    - ✓eg., **buff** b1(out1, in1);
  - **bufif1, bufif0, notif1, notif0** // tristate buffers/inverters
    - ✓eg., bufif1 b1(out, in, ctrl);

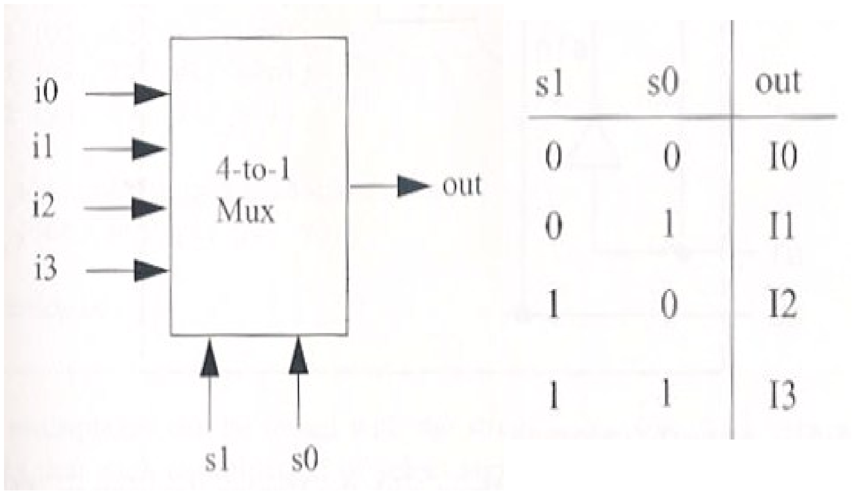


- array of instances

```
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];   */
```

4:1 Multiplexer (MUX4)





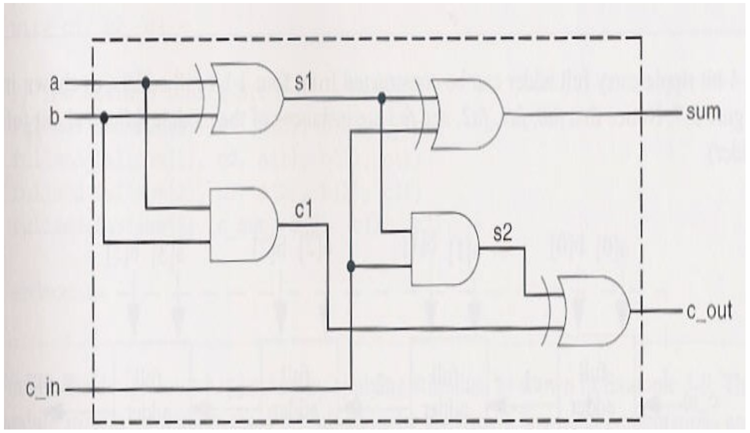
s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
// 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) // sensitivity list
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
```

1-bit Full Adder

```
// structural (gate)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); // assign s1=a^b;
        and (c1, a, b); // assign c1=a&b;
        xor (sum, s1, c_in);
        and (c2, s1, c_in)
        xor (c_out, c2, c1);

endmoudle
```

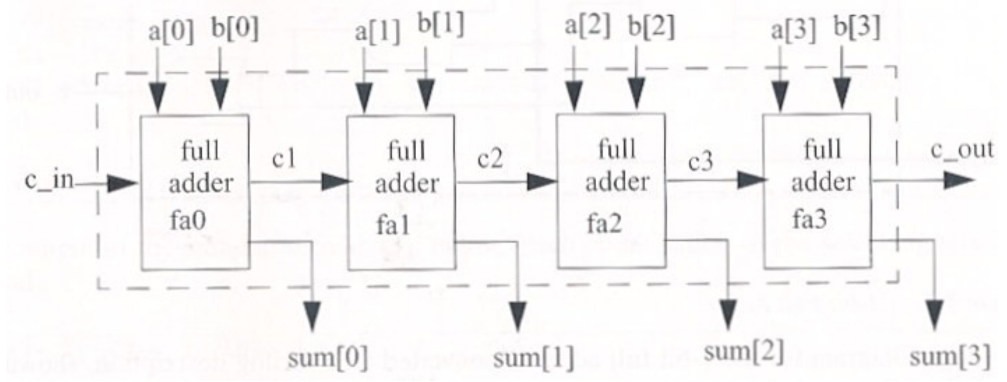


```
// dataflow modeling,  continuous assignment

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;
// synthesized gate netlist depends on
// area/delay constraints from users

endmoudle
```

```
// 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;

endmoudle
```

4-bit Ripple Carry Adder (RCA)

```
// 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;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

      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);

endmoudle
```



```
// dataflow level modeling
// using 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;
// synthesized gate netlist depends on
// area/delay constraints from users

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 of the hardware

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
endmoudle
```

RCA with generate loop

- use **generate** loop to duplicate module instances
  - e.g. in FA-cascaded RCA
- or simple
  - **array of instances**

```
// structural level modeling
module RCA (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;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

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);

endmoudle
```

```
module RCA_generate (sum, c_out, a, b, c_in);
parameter N=4;
output [N-1:0] sum;
output c_out
input [N-1:0] a, b;
input c_in;
wire [N:0] c;

assign c[0]=c_in;

genvar i;
generate
   for (i=0; i<=N-1; i=i+1)
      begin: FA_loop  // block named "FA_loop"
         fulladd  fa (sum[i], c[i+1], a[i], b[i], c[i]);
         // fulladd FA_loop[0].fa ( sum[0], c[1], a[0], b[0], c[0] );
         // fulladd FA_loop[1].fa ( sum[1], c[2], a[1], b[1], c[1] );
         // fulladd FA_loop[2].fa ( sum[2], c[3], a[2], b[2], c[2] );
         // fulladd FA_loop[3].fa ( sum[3], c[4], a[3], b[3], c[3] );
      end
endgenerate

// fulladd FA_loop[0:3] (sum[0:3], c[1:4], a[0:3], b[0:3], c[0:3]);

assign  c_out = c[N];

endmodule
```

generate blocks

- allow Verilog code to be generated *dynamically* at the elaboration time (before simulation)
  - HDL compiler usually contain two stages: compilation and elaboration
- help the creation of parameterized models
- applications
  - same operation or module instance is repeated
    - ✓ eg., 64-b ripple carry adder (cascaded of 64 FA cells)
  - certain Verilog code is conditionally included based on parameter definitions
    - ✓ e.g., Booth or non-Booth multiplier based on bit-width

generate loop

- allow multiple instantiation, or duplication of statements
- note that **generate** should be used outside of procedural blocks (**always**, **initial**)

```
module bitwise_xor(out, i0, i1);
parameter N=32;
output [N-1:0] out;
input [N-1:0] i0, i1;

genvar j; // var. j is local
generate
        for (j=0; j<N; j=j+1)
             begin: xor_loop
                 xor g(out[j], i0[j], i1[j]);
             end
// hierarchical name referencing:
// xor_loop[0].g, xor_loop[1].g,  ...,
endgenerate
// xor xor_loop[0].g(out[0], i0[0], i1[0]);
// xor xor_loop[1].g(out[1], i0[1], i1[1]);
…
// xor xor_loop[31].g(out[31], i0[31], i1[31]);
endmodule
```

```
// alternate style (behavioral level)
reg [N-1:0]  out;

genvar j;
generate
        for (j=0; j<N; j=j+1)
        begin: bit
           always @ (i0[j] or i1[j])   out[j] = i0[j] ^ i1[j];
        end
// always @(i0[0], i1[0])     out[0]=i0[0]^i1[0];
// always @(i0[1], i1[1])     out[1]=i0[1]^i1[1];
…
// always @(i0[31], i1[31]) out[31]=i0[31]^i1[31];
endgenerate
```

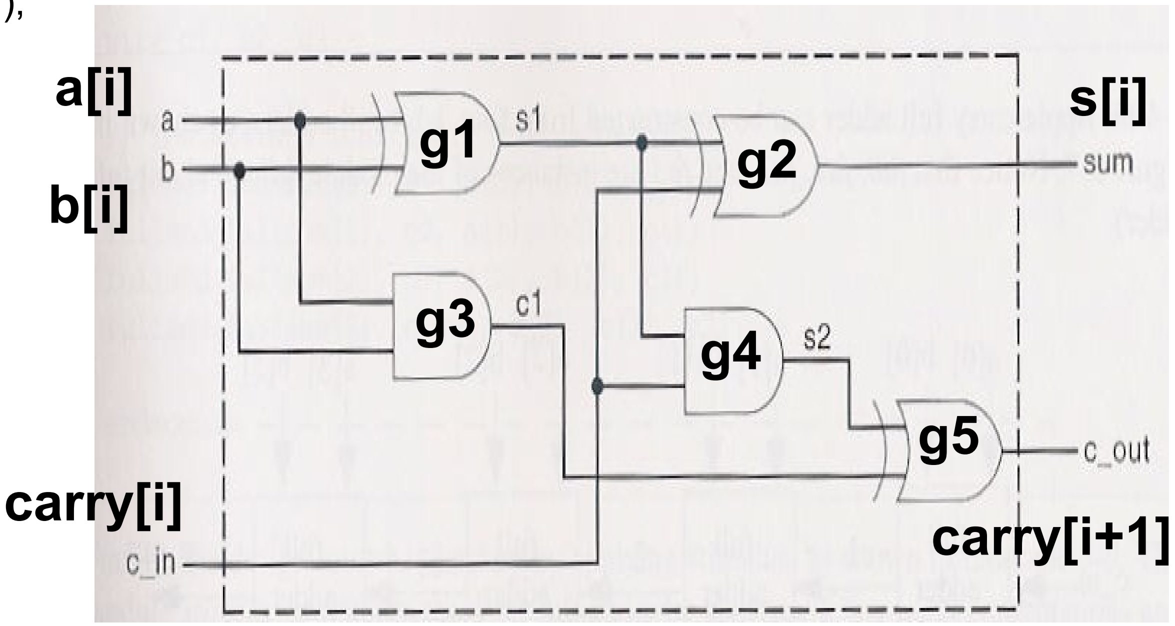
Discussion of generate loop block

- before simulation, simulator elaborates (unrolls) the code in the generate loop blocks
- then, the unrolled code is simulated
  - a convenient way of replacing multiple repetitive Verilog statements with a single statement inside a loop
- **genvar** is to declare local variables used in the generate block, do not exist in simulation
- hierarchical name referencing:

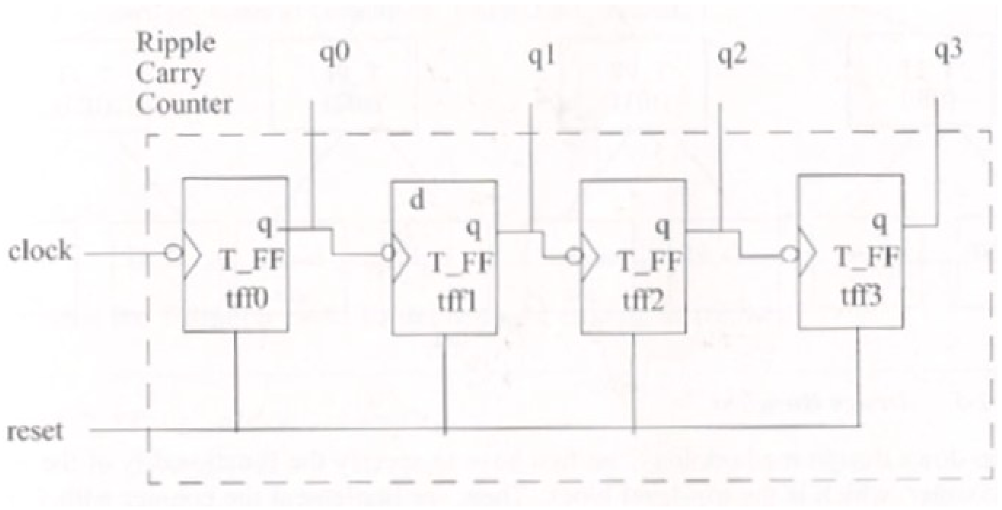
  xor_loop[0].g, xor_loop[1].g, ..., xor_loop[31].g

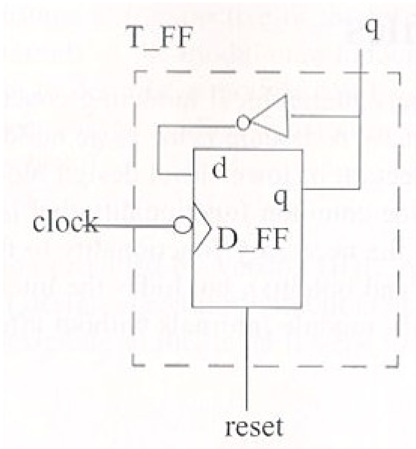
generated ripple carry adder

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N:0] carry;

assign carry[0]=ci;

genvar i;
generate
        for (i=0; i<N; i=i+1)
        begin: r_loop
                wire s1, c1, t3;              // the following hierarchical instance names are generated
                xor g1(s1, a0[i], a1[i]);        //  xor r_loop[0].g1(…);   xor r_loop[1].g1(…); …
                xor g2(sum[i], s1, carry[i]);    //  xor r_loop[0].g2(…);   xor r_loop[1].g2(…);  …
                and g3(c1, a0[i], a1[i]);        //  and r_loop[0].g3(…);   and r_loop[1].g3(…);  …
                and g4(s2, s1, carry[i]);        //  and r_loop[0].g4(…);   and r_loop[1].g4(…);  …
                xor g5(carry[i+1], c1, s2);      //  xor r_loop[0].g5(…);   xor r_loop[1].g5(…);  …
        end
endgenerate

assign co = carry[N]

endmodule
```

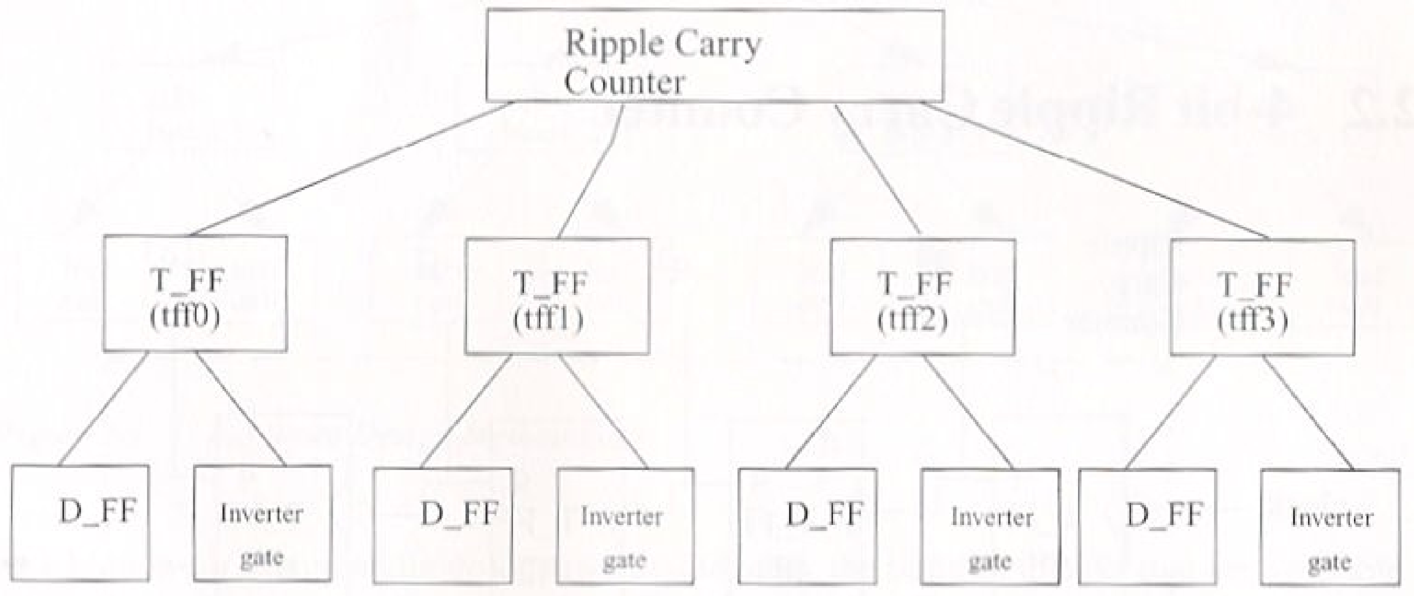


Example: 4-b Ripple Carry Counter

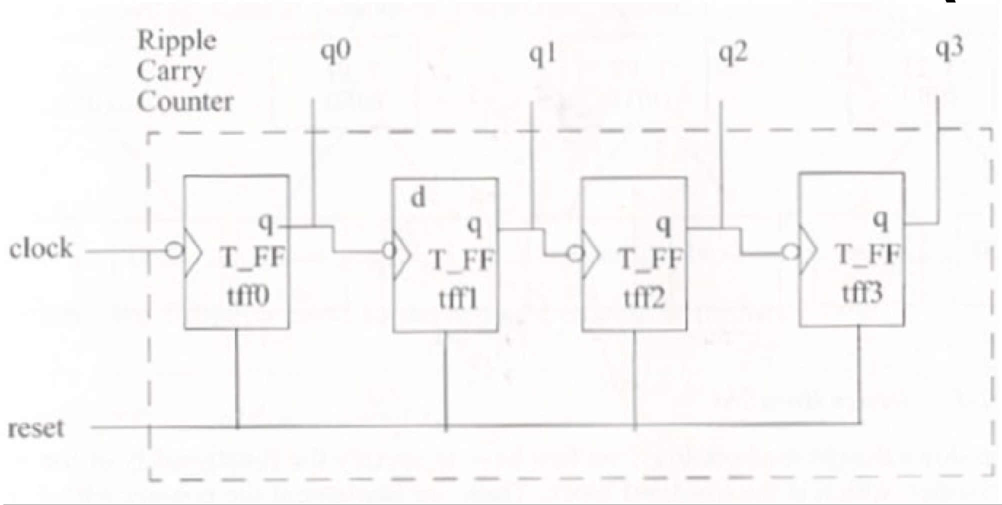


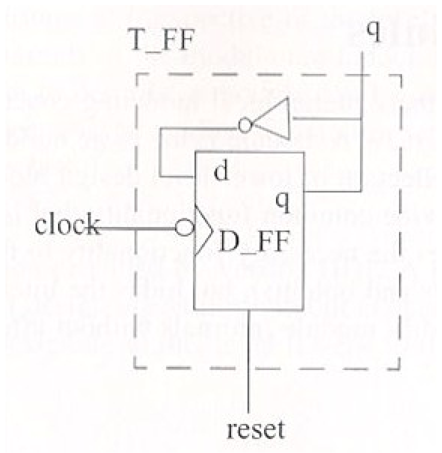


reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Module Instantiation (structural modeling)





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

// structural modeling
T_FF tff0 (q[0], clk, rst);
T_FF tff1 (q[1], q[0], rst);
T_FF tff2 (q[2], q[1], rst);
T_FF tff3 (q[3], q[2], 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. behavoral modeling

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

```
module D_FF (q, d, clk, reset);
output q;
input d, clk, reset;
reg q;

// behavioral modeling
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
```

Behavioral Modeling of Generic Counter

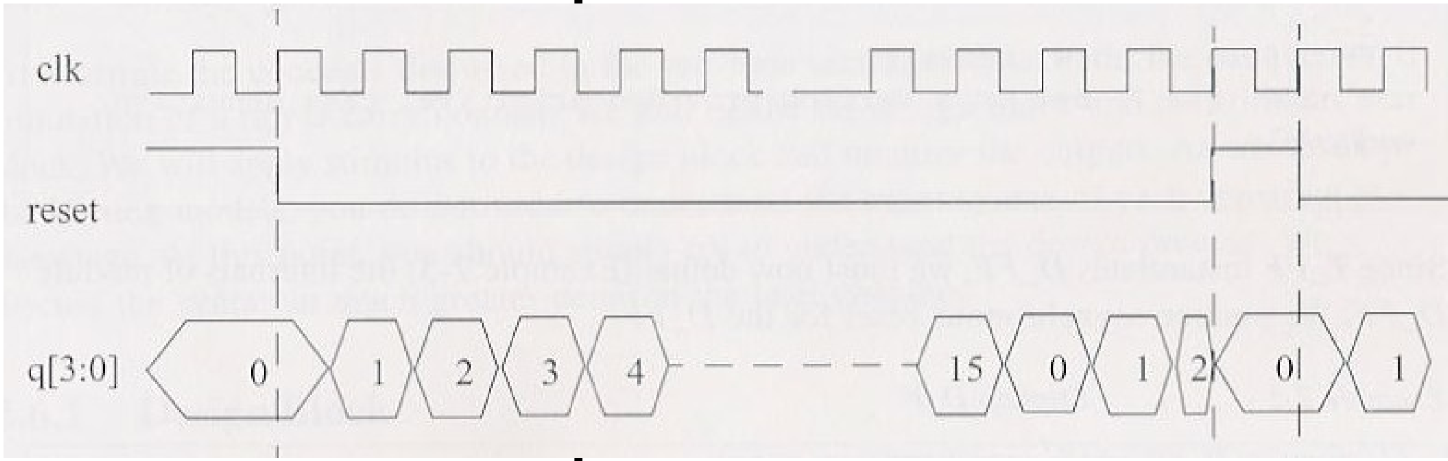
- Much simpler than previous structural modeling
  - but structural RCC -> generic counter without explicitly specified structure of the counter

```
module counter_beh (q, clk, rst);
Output reg [3:0] q;
input clk, rst;

// behavioral modeling
always @(negedge clk or posedge reset)
   if (reset )     q <= 4’b0;
   else            q <= q+1;

endmodule
```

Simulation Example

```
// non-synthesizable
// statements are OK
module testfixture;
// data type declaration
// instantiate design block
// apply stimulus
// display results
endmodule
```

```
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;  // clock period=10
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
```



generate conditional

```
module multiplier (product, a0, a1);
// 8-bit bus by default,
// could be changed during instantiation using defparam or #
parameter a0_width = 8;
parameter a1_width = 8;
// the following parameter cannot be modified using defparam
localparam product_width = a0_width + a1_width;

output [product_width-1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;

// instantiate different modules depending on input bitwidth
generate
  if (a0_width < 8) || (a1_width < 8)
    cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
  else
    tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate

endmodule
```

```
module top ( ….);
…
multiplier #(4,4) m1 (out1, a, b);
// cla_multiplier #(4,4) m1 (out1, a, b);
…
multiplier #(16, 16) m2 (out2, c, d);
// tree_multiplier #(16,16) m2 (out2, c, d);
…
multiplier #(4, 16) m3 (out3, e, f);
// cla_multiplier #(4,16) m3 (out3, e, f);
…
endmodule
```

generate case

```
module adder (co, sum, a0, a1, ci);

parameter N=4;

output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;

// instantiate different modules depending on input bitwidth
generate
      case (N)
              1 :  adder_1bit        adder1(co, sum, a0, a1, ci);
              2 :  adder_2bit        adder2(co, sum, a0, a1, ci);
        default :  adder_cla #(N) adder3(co, sum, a0, a1, ci);
      endcase
endgenerate

endmodule
```

```
module top ( ….);
…
adder  #(1) add1 (co1, s1, a1, b1);
// adder_1bit add1 (co1, s1,, a1, b1);
…
adder  #(2) add2 (co2, s2, a2, b2);
// adder_2bit  add2 (co2, s2, a2, b2);
...
adder  #(8) add3 (co3, s3, a3, b3);
// adder_cla  add3 (co3, s3, a3, b3);
…
adder  #(32) add4 (co4, s4, a4, b4);
// adder_cla  #(32) add4 (co4, s4, a4, b4);
……
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 after logic synthesis
  - real timing delay depends on the hardware components from real standard cell library of a specified process technology during synthesis

and #(5) a1(out, i1, i2);

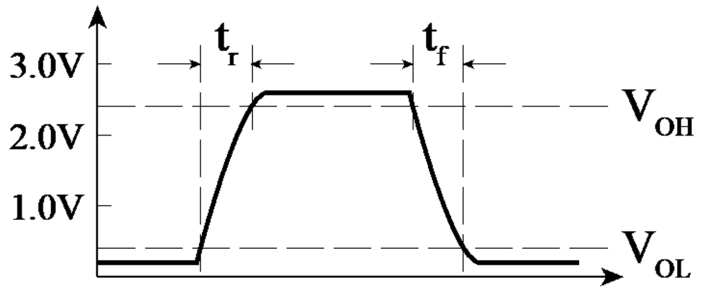
// delay of 5 time units for all transitions

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



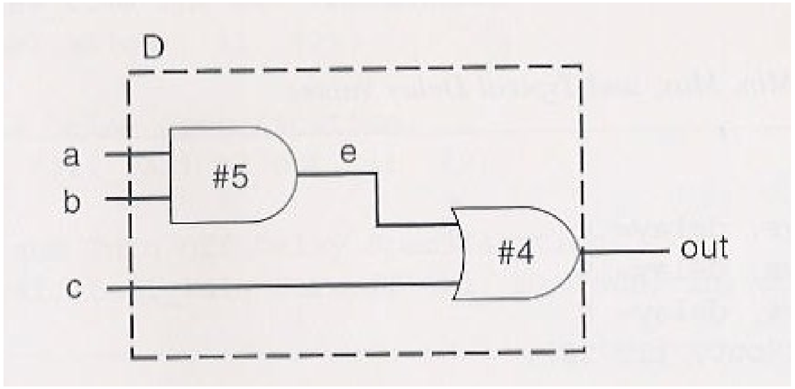
Rise time: shape of a single waveform
Rise delay (propagation L->H delay): Input to output delay

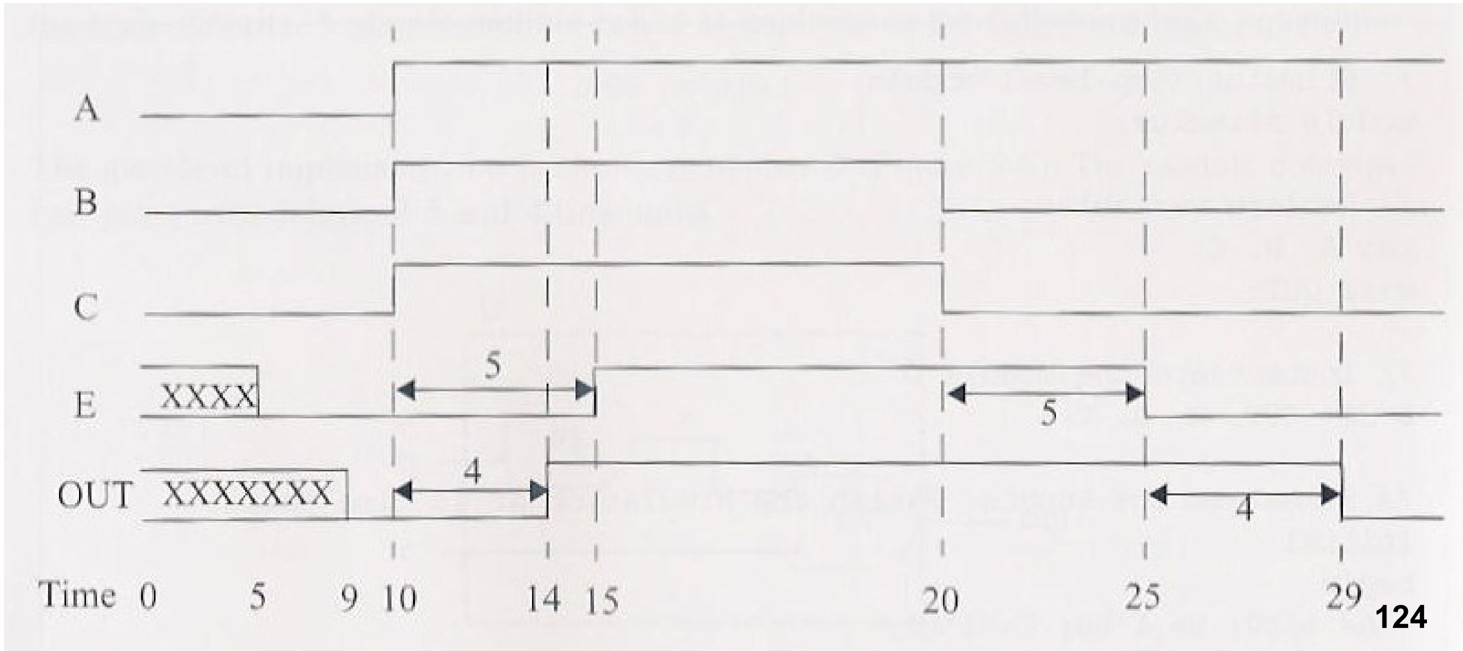
min/typy/max delays

```
// 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 originally defined
// 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
```





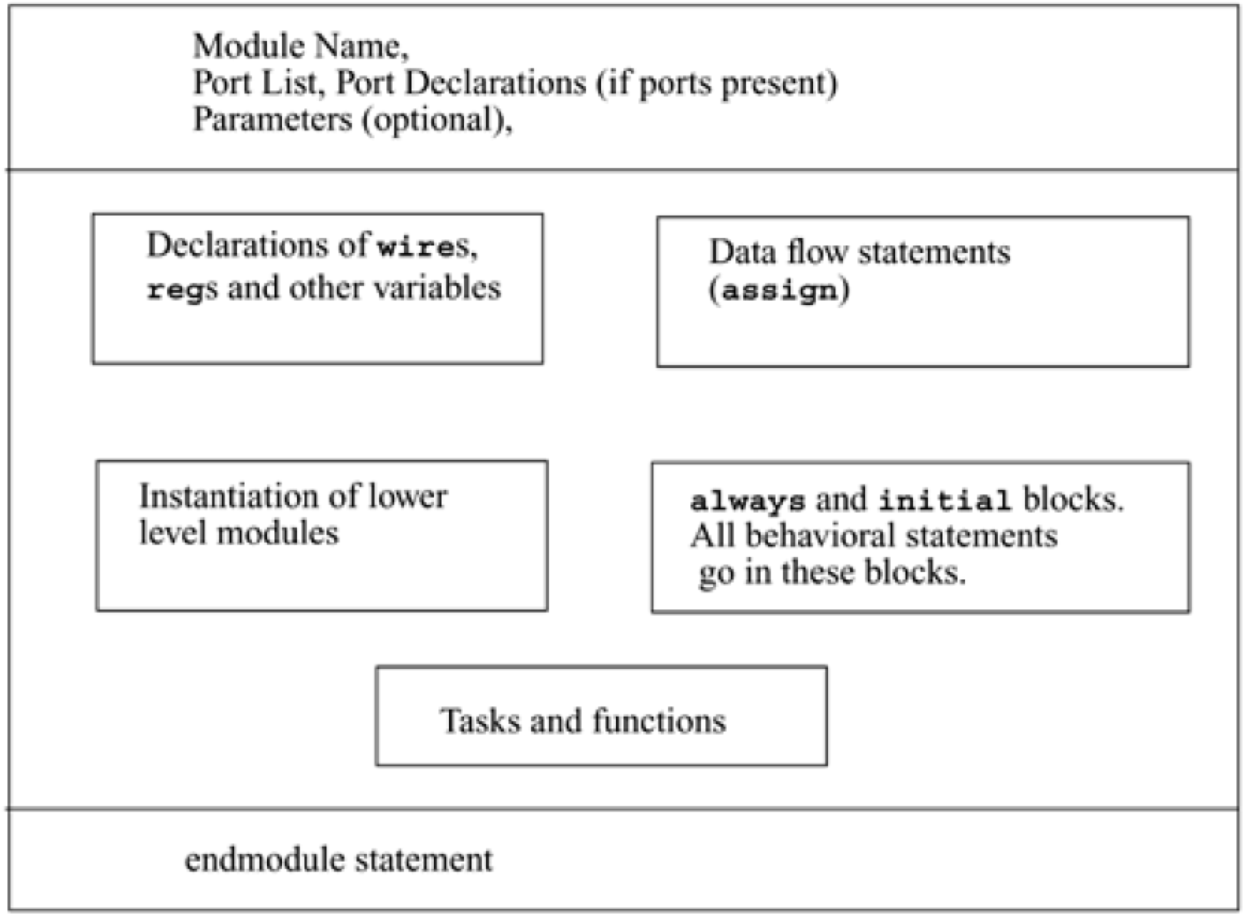
Timing Analysis Tools

- Static timing analsysi (STA)
  - Generate critical path delay without providing inputs
  - e.g., Synopsys Design
  - Might have false path delay which does not exist for any input combination
  - Can also generate power information, but not accurate for clock gating circuits
- Dynamic timing analysis
  - Generate the longest delay based on a sequence of user-specified inputs
  - e.g., Synopsys PrimeTime
  - Might not extract the crtical path delay if the provided input changes does not activate the critical path
  - Could also generate power information with clock gating

Dataflow Modeling
(continuous assignment: assign)

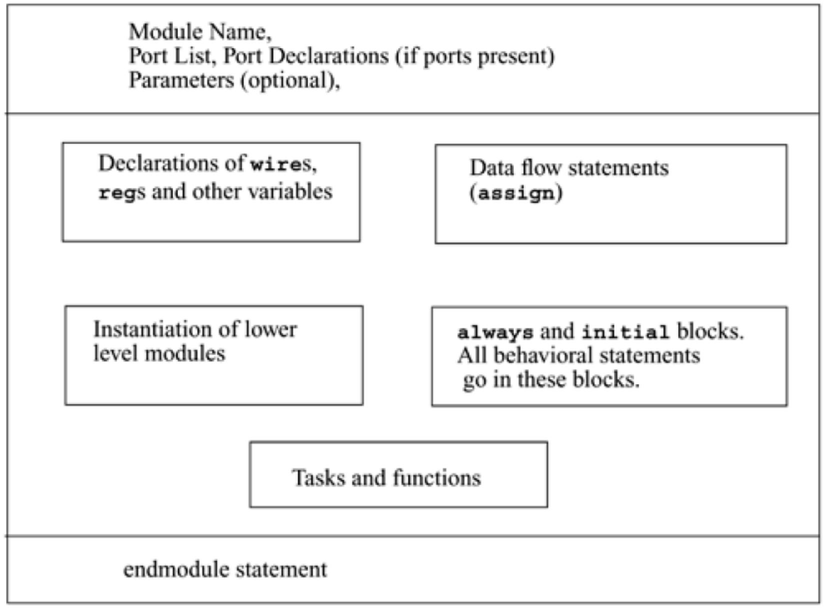
Components of a Verilog Module

- declaration
  - inputs, outputs
  - parameters
  - data types
- ***structure*** model
- module instantiation
- ***dataflow*** model
  - continuous assignment
- ***behavior*** model
  - procedural assignment


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

```
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;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end
```

```
// dataflow: continuous assignment
assign W1 = expr ;  // use operators

// module instances: structural modeling
COMP U1 (.PP2(W2), .PP1(W1);
COMP U2 (W3, W4);

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction

endmodule
```


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

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;    // LHS is wire data type
// or (out, i1, i2)

//  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 (LHS) 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 (LHS) must always be scalar or vector net (**wire**) (cannot be **reg**)*
- evaluated as soon as the right-hand-side (RHS) operands changes (unlike the behavioral procedural assignments where the statements in a procedure are executed depending on the sensitivity list)
- *expressions combine operators and operands*

Implicit Continuous Assignments

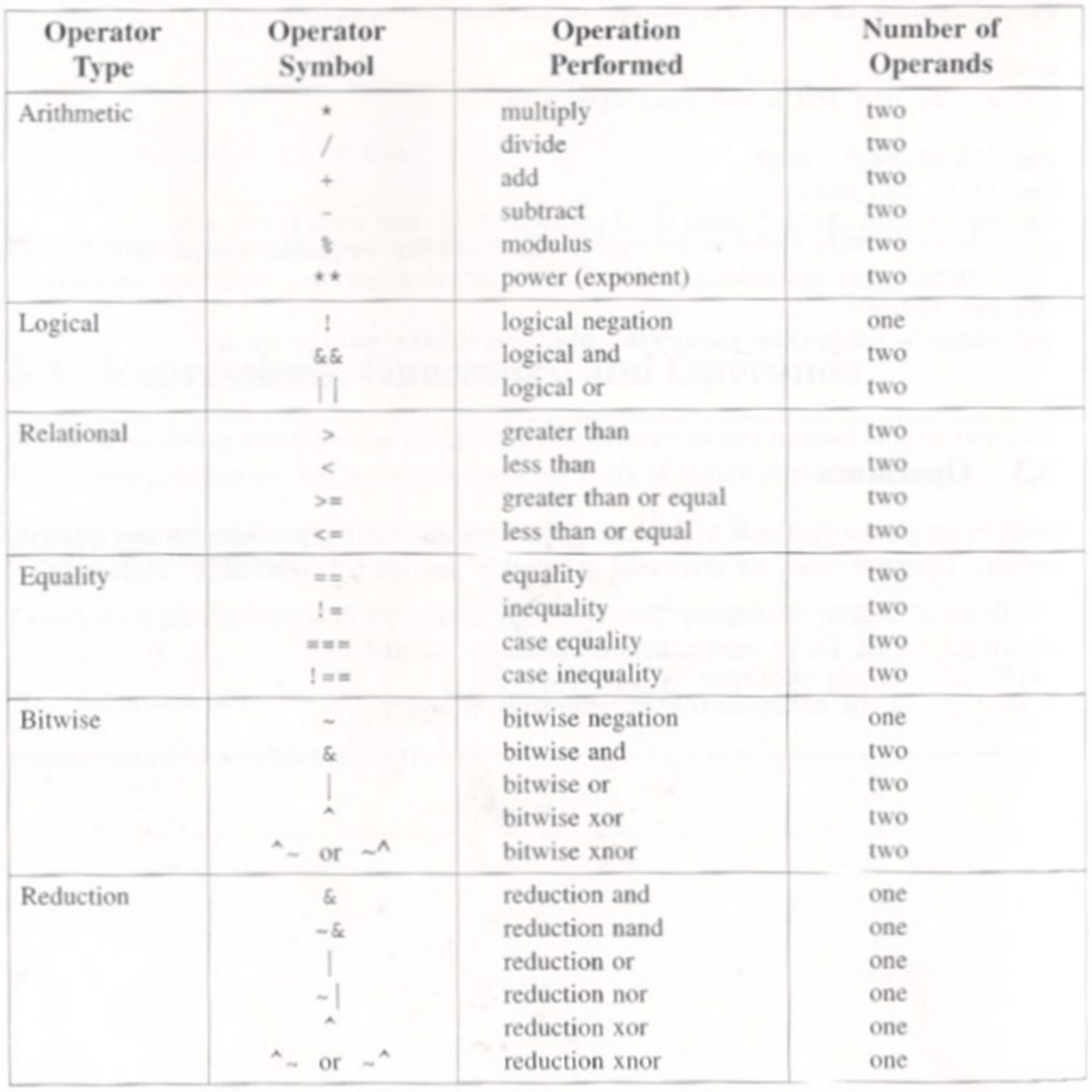
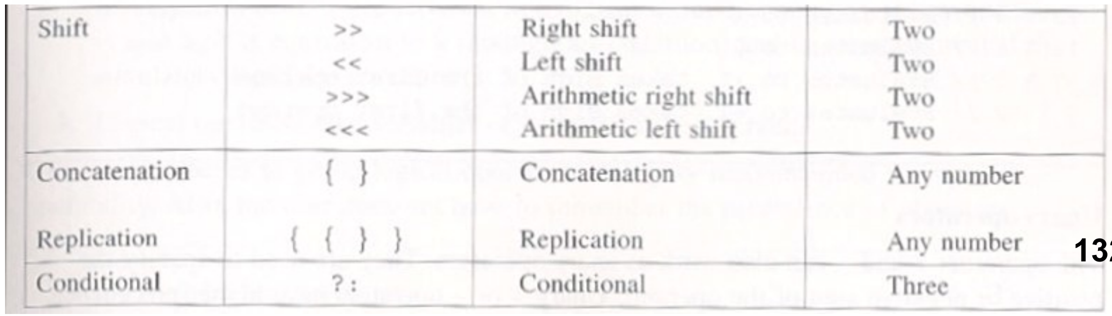
- // regular continuous assignment
  wire out;   // wire is default data type
  assign out **=** in1 **&** in2;\

- // 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
	**	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
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 of operators

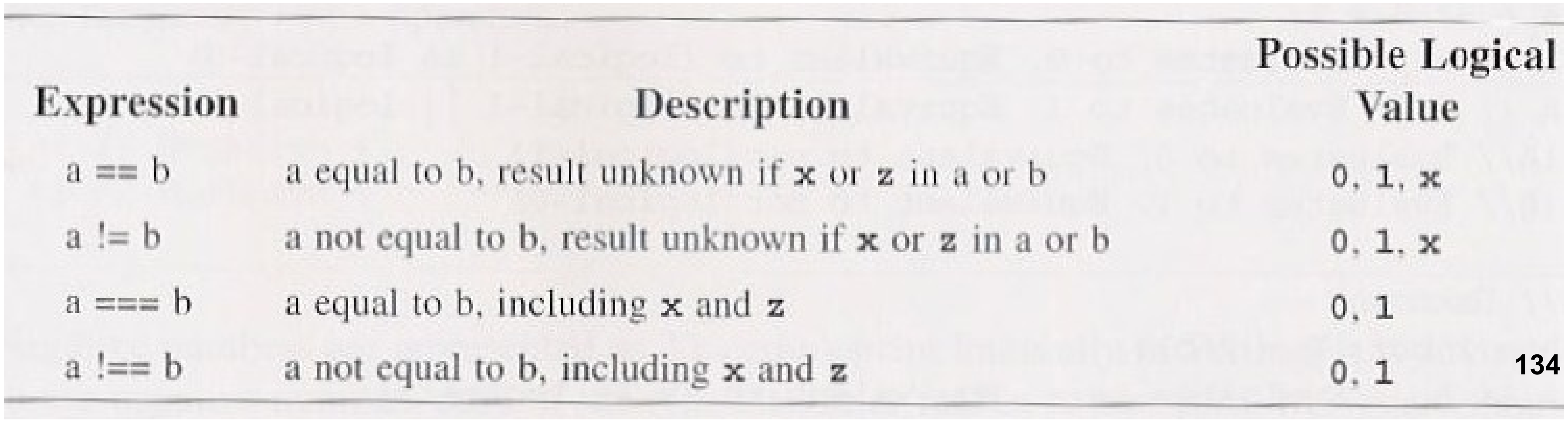
- if A=4’b0011; B=4’b0000

- A **+** B; // adding A and B equals 4’b1100
- A **&&** B; // evaluate to false (0) because A=3; B=0 (logical AND)
- A **||** B; // evaluate to true (1) because A=3; B=0;
- A **<=** B; // evaluate to false (a logical 0) because A=3; B=0
- A **==** B; // results in logical 0 because 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’b0011_0000
- C = {2{A}}; // replication. C=8’b0011_0011
- **assign** out = cntl **?** in1 **:** in2; // conditional operator

- **!** is logic operator, !A evaluates as false (or1’b0)
- **~** is bit-wise operator, ~A evaluates as 4’b1100

Case equality operators (===, !==) Non-synthesiable by Synopsy DC)

```
// 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

Equality Operator: Logical vs. Case

- logical equality operator (==)

==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

- case equality operator (===)

===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

Logical Shift (<<, >>) versus Arithmetic Shift (<<<, >>>)

```
// Verilog-2001

integer  data_value,
         data_value_1995,
         data_value_2001;
…
data_value = -9;                            // stored as 1111_…_1111_0111
…
data_value_1995 = data_value  >> 3;   // stored as  0001_..._1111_1110 > 0
…
data_value_2001 = data_value  >>> 3; // stored as  1111_...._1111_1110 = -2
                                      // repeat sign bit during arithmetic right shift
…
data_value_1995 = data_value << 3;    // stored as 1111_..._1011_1000
…
data_value_2001 = data_value <<< 3;  // stored as 1111_..._1011_1000 = -72
                                    // arithmetic left is equivalent to logical left  shift
                                    // but arithmetic left will trigger overflow,
                                    // while logical left shift will not trigger overflow
```

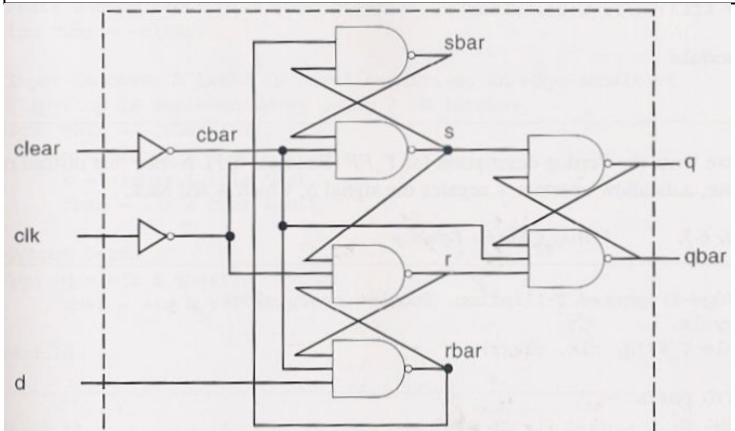
logical left shift vs. arithmetic left shift

- logical left shift is equivalent to arithmetic left shift if sign bit remains unchanged after ahift
  - multiplying positive integral power of 2
- but arithmetic left shift might trigger overflow
  - overflow occurs when the sign bit is changed
    - ✓ e.g., max positive number 0111…. <<< 1 => 1111…
    - ✓ e.g., mini negative number 1000…. <<< 1 => 0000….
- while logical left will NOT trigger overflow

behavioral (always) vs. dataflow (assign)

```
module D_FF (q, qbar, d, clk, clear);
output q, qbar;
input d, clk, clear;
reg q, qbar;

// DFF with asynchronous clear (reset)
always @ (posedge clear or negedge clk)
if (clear)
    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;

// same signals appear in LHS and RHS
// infer latch with  loop in datapath
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
```

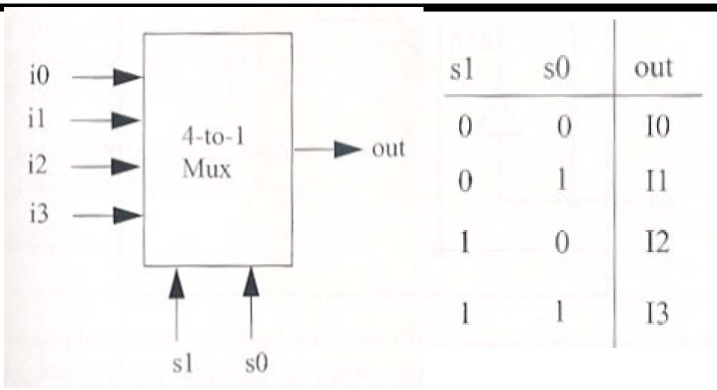


Both **always** and **assign** belong to RTL coding style

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
```



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

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
```

```
// explicit carry lookahead adder using p and g

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
```

$p_i = \quad \oplus \quad = \quad = \quad +$

$c_1 = \quad +$

$c_2 = \quad + \quad +$

$c_3 = \quad + \quad + \quad +$

$c_4 = \quad + \quad + \quad + \quad +$

n

$s_i = \quad \oplus$