Chapter 3: Verilog HDL

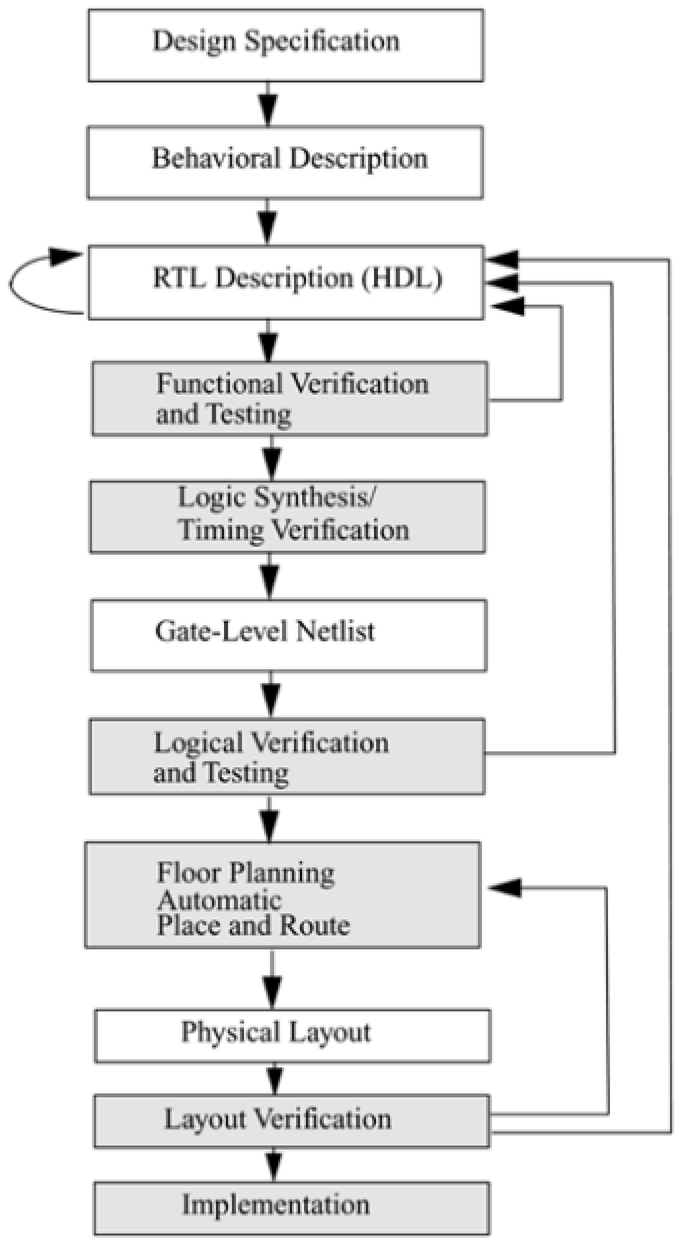
Trương Ngọc Sơn, Ph.D

What is Verilog

- Hardware Description Language (HDL)
- Hardware description languages such as Verilog HDL and VHDL became popular
- Developed in 1983
- Standard: IEEE 1364, Dec 1995
- Hardware Description Languages (HDLs)

Design flow

Figure 1-1. Typical Design Flow


Design Specification
Behavioral Description
RTL Description (HDL)
Functional Verification and Testing
Logic Synthesis/ Timing Verification
Gate-Level Netlist
Logical Verification and Testing
Floor Planning Automatic Place and Route
Physical Layout
Layout Verification
Implementation

Popularity of Verilog

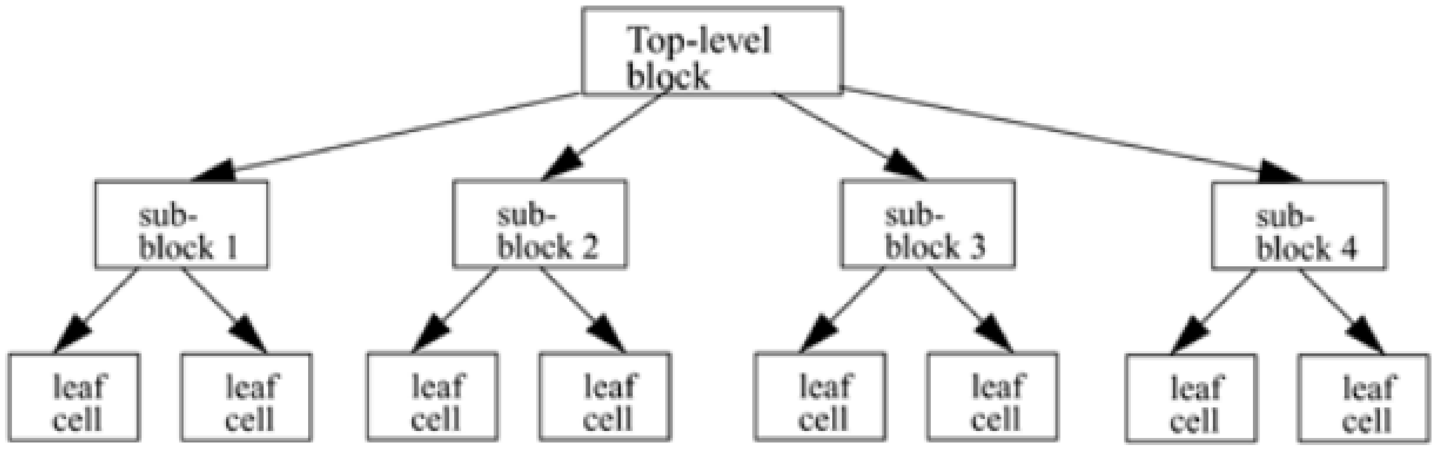
- Verilog HDL is a general-purpose hardware description language that is easy tolearn and easy to use
- Verilog HDL allows different levels of abstraction to be mixed in the same model: gates, RTL, or behavioral code
- Most popular logic synthesis tools support Verilog HDL
- All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation

Abstraction Levels in Verilog

- Behavioral
- RTL
- Gate

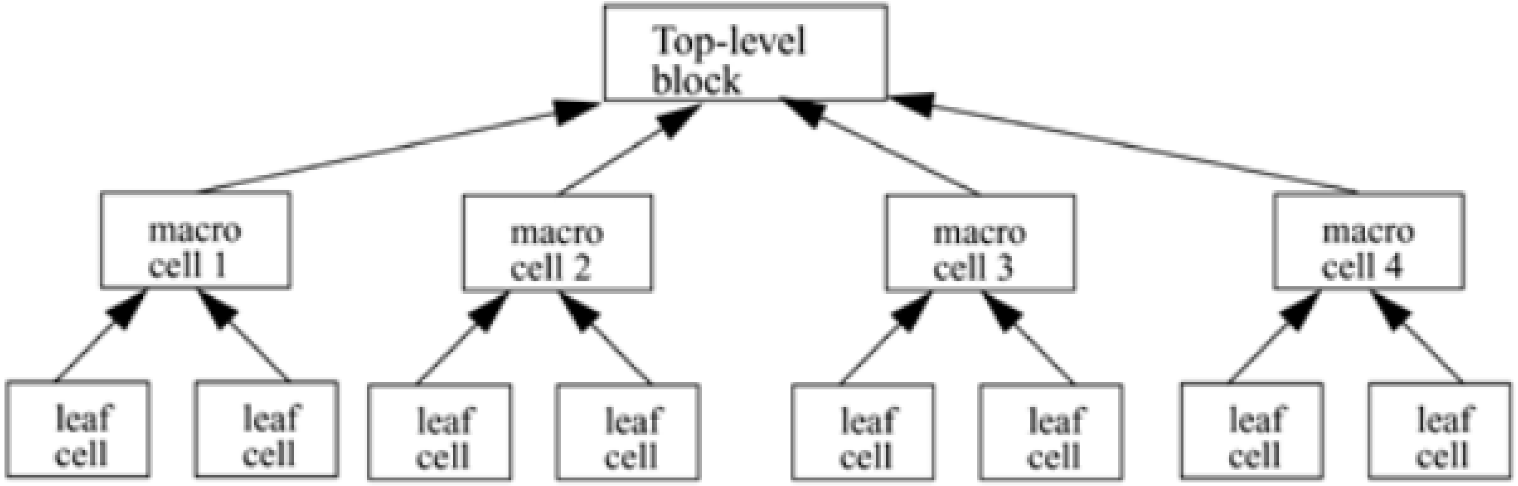
Design Methodologies

- Top-down design methodology: we define the top-level block and identify the sub-blocks necessary to build the top-level block



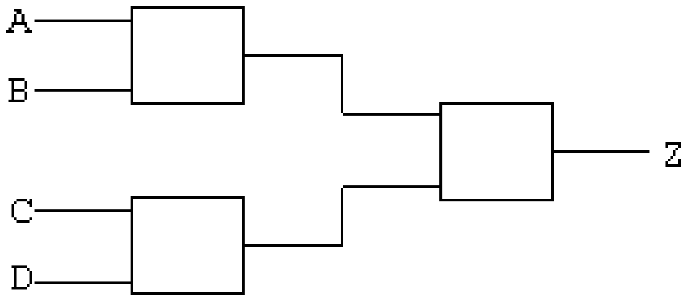
Design Methodologies

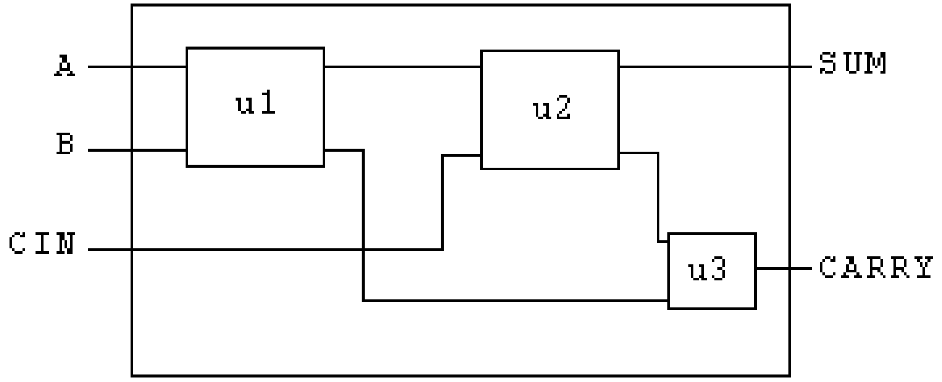
- Bottom-up design methodology: we first identify the building blocks that are available to us. We build bigger cells, using these building blocks



Main Language Concepts (i)

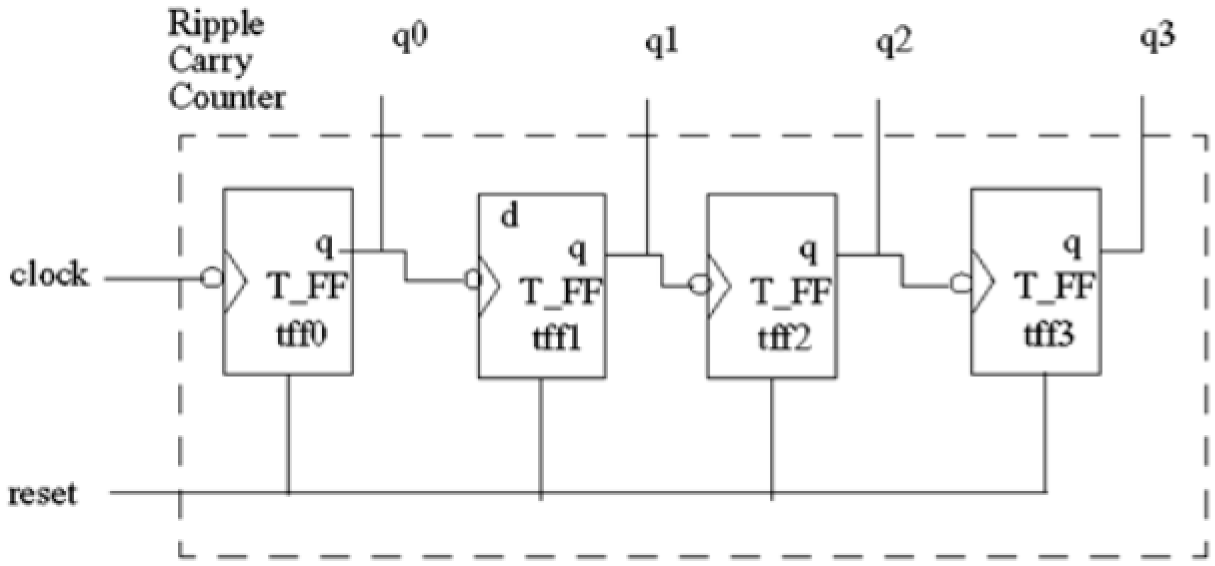
- Concurrency



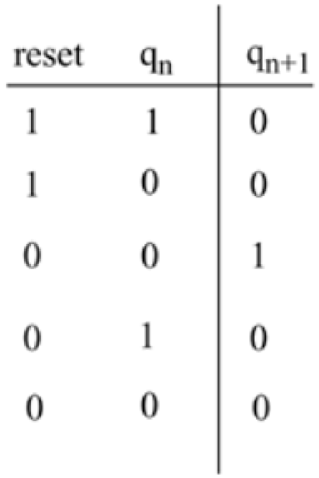


- Structure

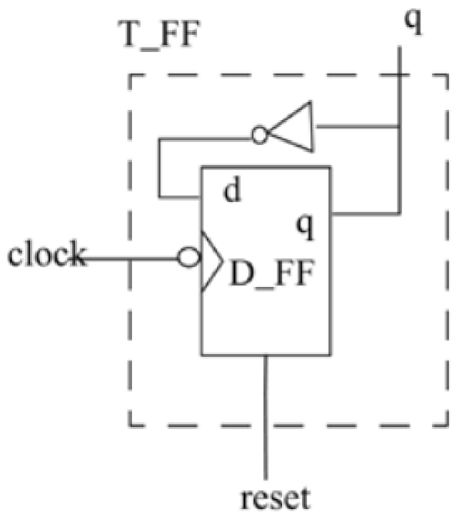
4 bit counter

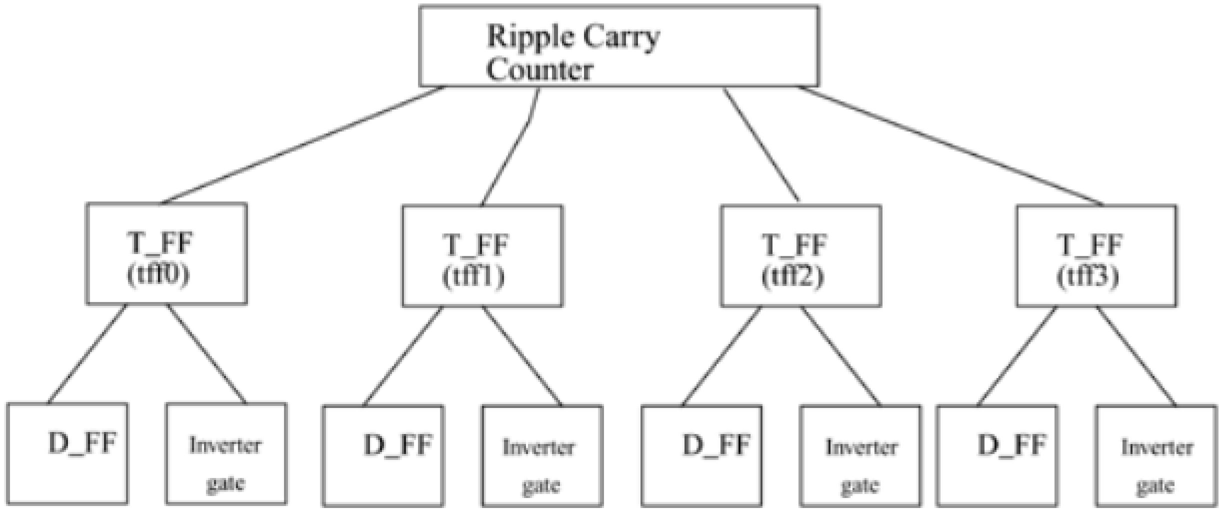


4 bit counter, top-down design



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





Module

- A module is the basic building block in Verilog.
- A module can be an element or a collection of lower-level design blocks
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs)

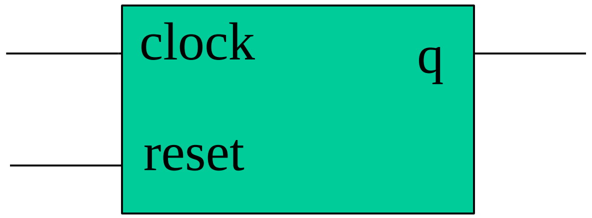
Module

```
module <module_name> (<module_terminal_list>);
...
<module internals>
...
endmodule
```

Module

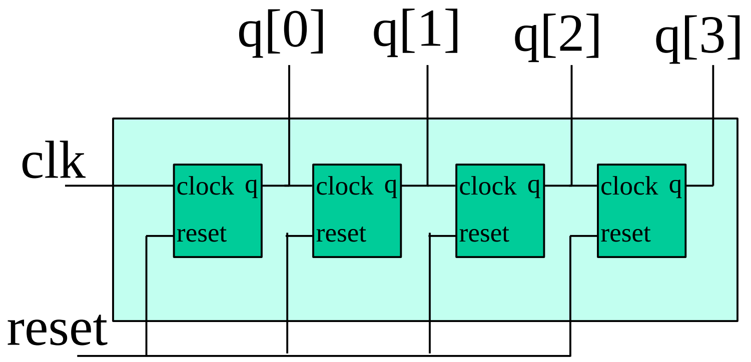
Specifically, the T-flipflop could be defined as a module as follows:

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



Instances

```
module ripple_carry_counter(q, clk, reset);
output [3:0] q;.
input clk, reset;
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);
endmodule
```

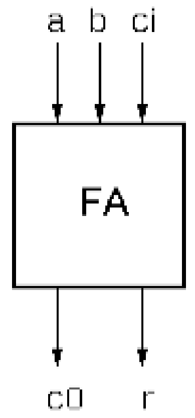


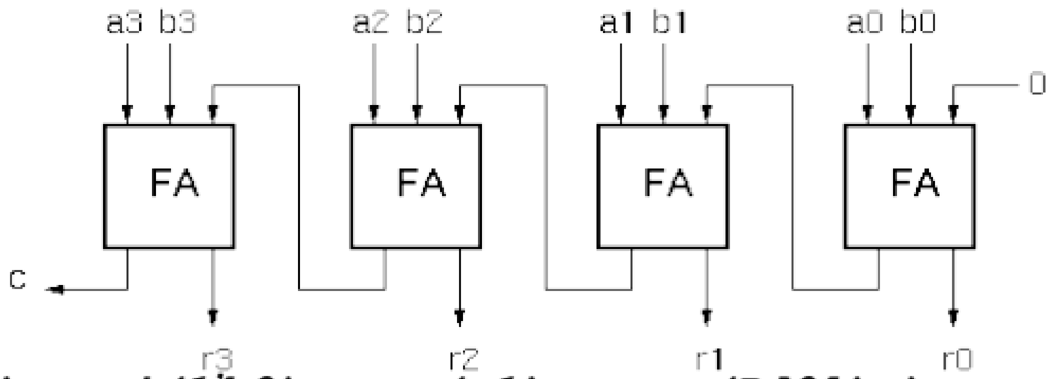
Instances

```
module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q);
endmodule
```

Review - Ripple Adder Example

```
module FullAdder(a, b, ci, r, co);
   input a, b, ci;
   output r, co;

   assign r = a ^ b ^ ci;
  assign co = a&ci + a&b + b&cin;

endmodule
```





```
module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;
  FullAdder
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),   .r(R[0]) ),
  add1(.a(A[1]), .b(B[1]), .ci(c1),    .co(c2),   .r(R[1]) ),
  add2(.a(A[2]), .b(B[2]), .ci(c2),    .co(c3),   .r(R[2]) ),
  add3(.a(A[3]), .b(B[3]), .ci(c3),    .co(R[4]), .r(R[3]) );
endmodule
```

User Identifiers

- Formed from {[A-Z], [a-z], [0-9], _, $}, but ..
- .. can’t begin with $ or [0-9]
  - `myidentifier` ☐
  - `m_y_identifier` ☐
  - `3my_identifier` ☐
  - `$my_identifier` ☐
  - `_myidentifier$` ☐
- Identifiers are case sensitive
  - `myid` ≠ `Myid`

Comments

- `// The rest of the line is a comment`

-
  ```
  /* Multiple line
     comment   */
  ```

- `/* Nesting /* comments */ do` **`NOT`** `work  */`

Verilog Value Set

- *0* represents low logic level or false condition
- *1* represents high logic level or true condition
- *x* represents unknown logic level
- *z* represents high impedance logic level

Numbers in Verilog (i)

<size>'<radix> <value>

No of bits

Binary	→ b or B
Octal	→ o or O
Decimal	→ d or D
Hexadecimal	→ h or H

Consecutive chars 0-f, x, z

– 8'h ax = 1010xxxx

– 12'o 3zx7 = 011zzzxxx111

Numbers in Verilog (ii)

- You can insert “_” for readability
  - 12’b 000_111_010_100
  - 12’b 000111010100
  - 12’o 07_24

  $\}$ Represent the same number
- Bit extension
  - MS bit = 0, x or z ⇒ extend this
    - 4’b x1 = 4’b xx_x1
  - MS bit = 1 ⇒ zero extension
    - 4’b 1x = 4’b 00_1x

Numbers in Verilog (iii)

- If *size* is ommitted it
  - is inferred from the *value* or
  - takes the simulation specific number of bits or
  - takes the machine specific number of bits
- If *radix* is ommitted <u>too</u> .. decimal is assumed
  - 15 = <size>'d 15

Parameters in Verilog (iii)

- A parameter associates an identifier name with a constant. Let the Verilog code include the following declarations:

parameter n = 4;
parameter S0 = 2’b00, S1 = 2’b01, S2 = 2’b10, S3 = 2’b11;

Signal in Verilog code

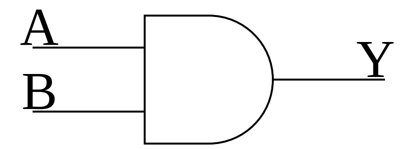
- In Verilog, a signal in a circuit is represented as a *net* or a *variable* with a specific type. The term *net* is derived from the electrical jargon, where it refers to the interconnection of two or more points in a circuit. A net or variable declaration has the form

type [range] signal_name{, signal_name};

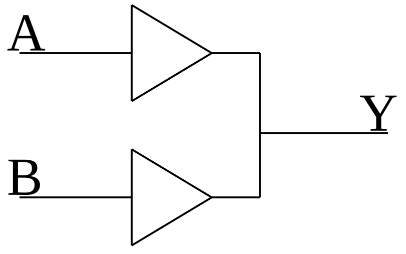
Nets (i)

- Can be thought as hardware wires driven by logic
- Equal *z* when unconnected
- Various types of nets
  - `wire`
  - `wand` (wired-AND)
  - `wor` (wired-OR)
  - `tri` (tri-state)
- In following examples: Y is evaluated, ***automatically***, every time A or B changes

Nets (ii)



```
wire Y;  // declaration
assign Y = A & B;
```

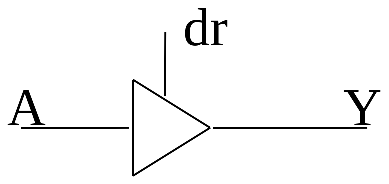


```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

Y		A	
		0	1
B	0	0	0
	1	0	1

```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

Y		A	
		0	1
B	0	0	1
	1	1	1



```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

Registers

- Variables that store values
- Do not represent real hardware but ..
- .. real hardware can be implemented with registers
- Only one type: `reg`

```
reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0
```

- Register values are updated explicitly!!

Vectors

- Represent buses

```
wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;
```

- Left number is MS bit
- Slice management

$$\texttt{busC = busA[2:1];} \quad \Leftrightarrow \quad \begin{cases} \texttt{busC[1] = busA[2];} \\ \texttt{busC[0] = busA[1];} \end{cases}$$

- Vector assignment (***by position!!***)

$$\texttt{busB = busA;} \quad \Leftrightarrow \quad \begin{cases} \texttt{busB[1] = busA[3];} \\ \texttt{busB[2] = busA[2];} \\ \texttt{busB[3] = busA[1];} \\ \texttt{busB[4] = busA[0];} \end{cases}$$

Integer & Real Data Types

- Declaration

```
integer i, k;
real r;
```

- Use as registers (inside procedures)

```
i = 1; // assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
```

- Integers are not initialized!!
- Reals are initialized to *0.0*

Time Data Type

- Special data type for simulation time measuring
- Declaration

```
    time my_time;
```

- Use inside procedure

```
    my_time = $time; // get current sim time
```

- Simulation runs at simulation time, not real time

Arrays (i)

- Syntax

```
integer count[1:5]; // 5 integers
reg var[-15:16]; // 32 1-bit regs
reg [7:0] mem[0:1023]; // 1024 8-bit regs
```

- Accessing array elements
  - Entire element: `mem[10] = 8’b 10101010;`
  - Element subfield (needs temp storage):

```
reg [7:0] temp;
..
temp = mem[10];
var[6] = temp[2];
```

Arrays (ii)

- Limitation: Cannot access array subfield or entire array at once

```
var[2:9] = ???; // WRONG!!
var = ???; // WRONG!!
```

- No multi-dimentional arrays

```
reg var[1:10] [1:100]; // WRONG!!
```

- Arrays don’t work for the Real data type

```
real r[1:10]; // WRONG !!
```

Strings

- Implemented with regs:

```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = “Hello Verilog”;
string_val = “hello”; // MS Bytes are filled with 0
string_val = “I am overflowed”; // “I ” is truncated
```

- Escaped chars:
  - `\n` newline
  - `\t` tab
  - `%%` `%`
  - `\\` `\`
  - `\"` `"`

Logical Operators

- `&&` $\rightarrow$ logical AND
- `||` $\rightarrow$ logical OR
- `!` $\rightarrow$ logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 6;        A && B → 1 && 0 → 0
B = 0;        A || !B → 1 || 1 → 1
C = x;        C || B → x || 0 → x
```

but C&&B=0

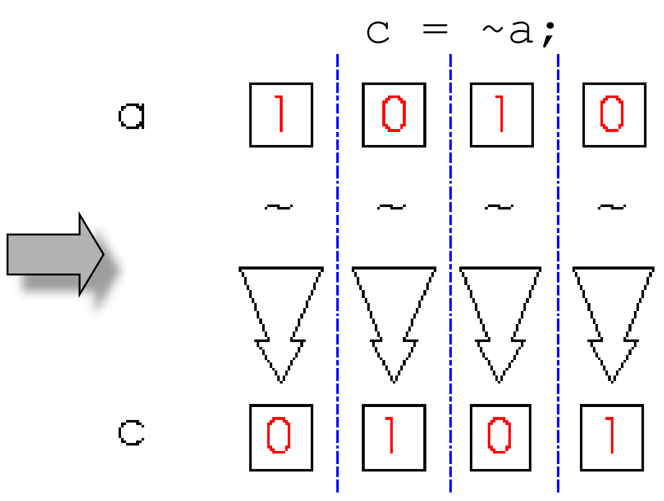
Bitwise Operators (i)

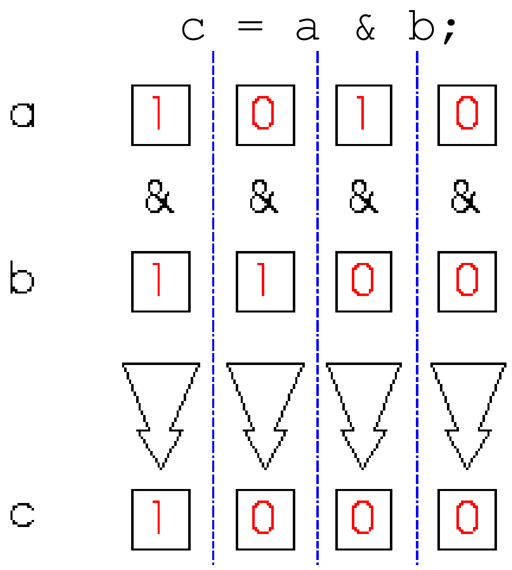
`&`	$\rightarrow$ bitwise AND
`|`	$\rightarrow$ bitwise OR
`~`	$\rightarrow$ bitwise NOT
`^`	$\rightarrow$ bitwise XOR
`~^` or `^~`	$\rightarrow$ bitwise XNOR

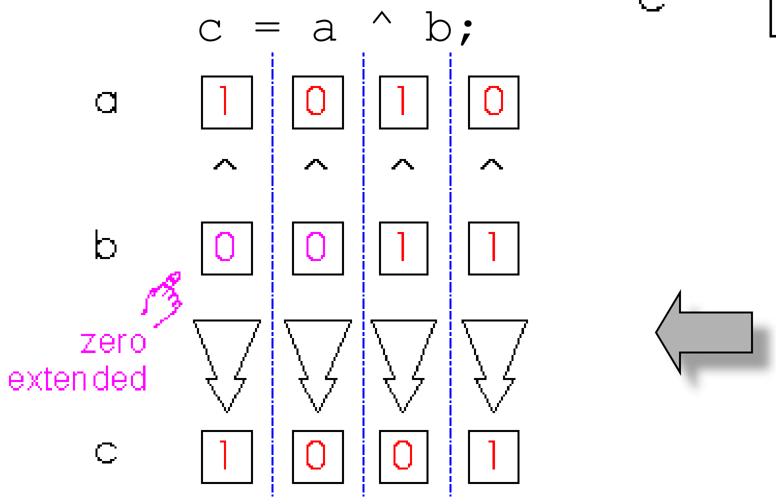
- Operation on bit by bit basis

Bitwise Operators (ii)

- `a = 4'b1010;`
  `b = 4'b1100;`







- `a = 4'b1010;`
  `b = 2'b11;`

Reduction Operators

• `&`	→ AND
• `|`	→ OR
• `^`	→ XOR
• `~&`	→ NAND
• `~|`	→ NOR
• `~^` or `^~`	→ XNOR

- One multi-bit operand → One single-bit result

```
a = 4'b1001;
..
c = |a; // c = 1|0|0|1 = 1
```

Shift Operators

- `>>` → shift right
- `<<` → shift left

- Result is same size as first operand, **always zero filled**

```
a = 4’b1010;
...
d = a >> 2;   // d = 0010
c = a << 1;   // c = 0100
```

Concatenation Operator

- `{op1, op2, ..}` → concatenates op1, op2, .. to single number
- Operands must be sized !!

```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};           // catx = 1_010_101
caty = {b, 2'b11, a};       // caty = 010_11_1
catz = {b, 1};              // WRONG !!
```

- Replication ..

```
catr = {4{a}, b, 2{c}};  // catr = 1111_010_101101
```

Relational Operators

- `>` $\rightarrow$ greater than
- `<` $\rightarrow$ less than
- `>=` $\rightarrow$ greater or equal than
- `<=` $\rightarrow$ less or equal than

- Result is one bit value: *0*, *1* or *x*

`1 > 0`	$\rightarrow$ *1*
`’b1x1 <= 0`	$\rightarrow$ *x*
`10 < z`	$\rightarrow$ *x*

Equality Operators

$$\left.\begin{array}{l}\bullet\ \texttt{==}\ \rightarrow \text{logical equality} \\ \bullet\ \texttt{!=}\ \rightarrow \text{logical inequality}\end{array}\right\}\ \text{Return } \mathit{0},\ \mathit{1} \text{ or } \mathit{x}$$

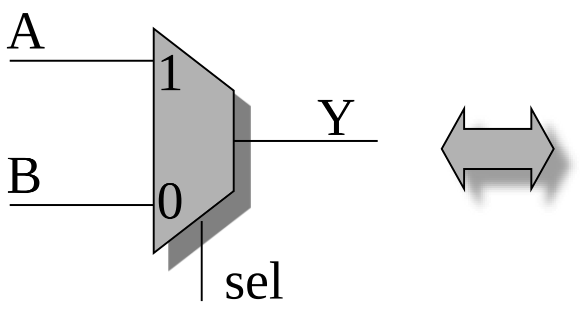
$$\left.\begin{array}{l}\bullet\ \texttt{===}\ \rightarrow \text{case equality} \\ \bullet\ \texttt{!==}\ \rightarrow \text{case inequality}\end{array}\right\}\ \text{Return } \mathit{0} \text{ or } \mathit{1}$$

- `4'b 1z0x == 4'b 1z0x` → *x*
- `4'b 1z0x != 4'b 1z0x` → *x*
- `4'b 1z0x === 4'b 1z0x` → *1*
- `4'b 1z0x !== 4'b 1z0x` → *0*

Conditional Operator

- `cond_expr ? true_expr : false_expr`

- Like a 2-to-1 mux ..



```
Y = (sel)? A : B;
```

Arithmetic Operators (i)

- `+, -, *, /, %`
- If any operand is *x* the result is *x*
- Negative registers:
  - regs can be assigned negative but are treated as unsigned

```
reg [15:0] regA;
..
regA = -4'd12;       // stored as 2^16-12 = 65524
```

`regA/3` evaluates to 21861

Arithmetic Operators (ii)

- Negative integers:
  - can be assigned negative values
  - different treatment depending on base specification or not

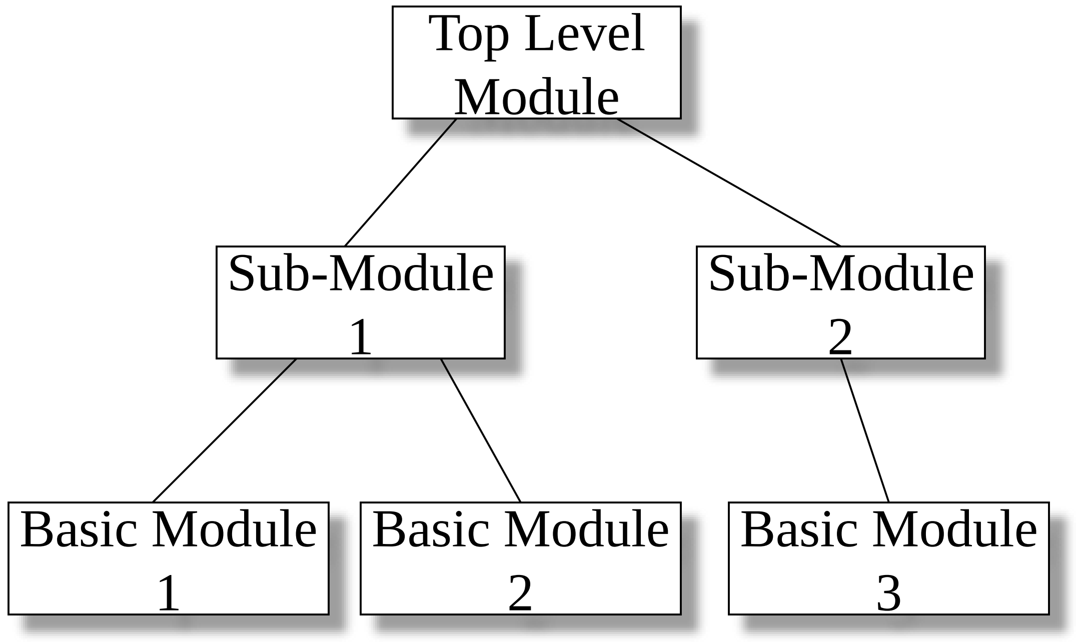
```
reg [15:0] regA;
integer intA;
..
intA = -12/3;     // evaluates to -4 (no base spec)
intA = -'d12/3;  // evaluates to 1431655761 (base spec)
```

Operator Precedence

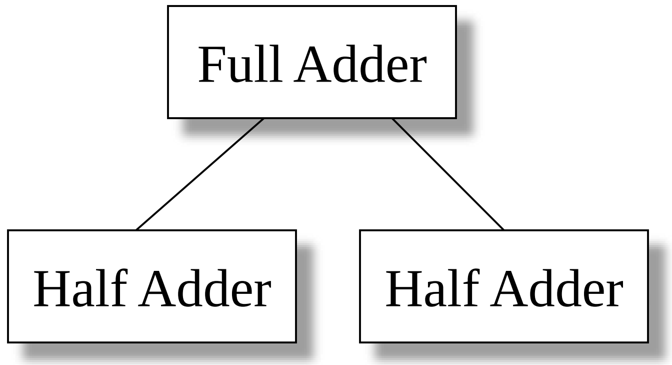
+-!~unary	highest precedence
*/%	
+-(binary)	
<< >>	
< <= => >	
== != === !==	
& ~&	
^ ^~ ~^	
| ~|	
&&	
||	
?: conditional	lowest precedence

Use parentheses to enforce your priority

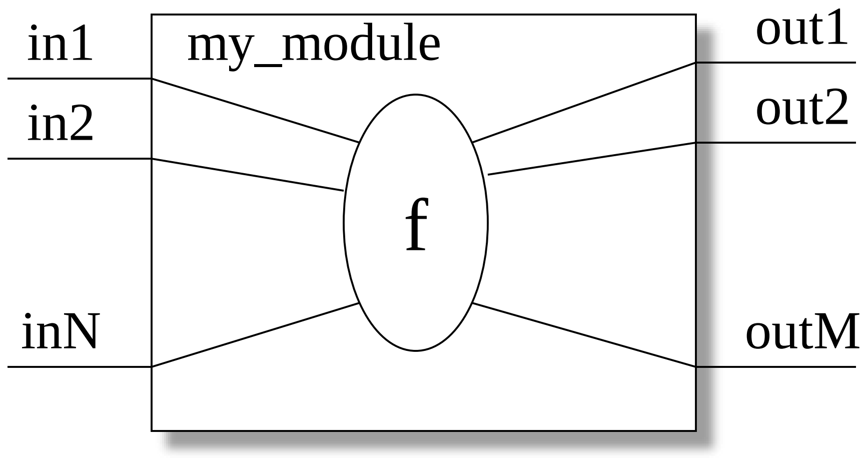
Hierarchical Design



E.g.



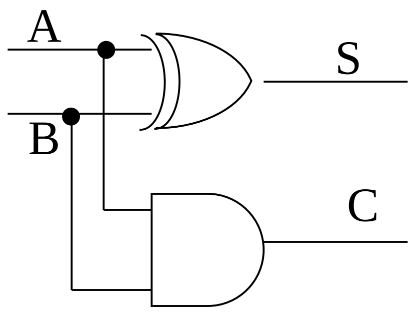
Module

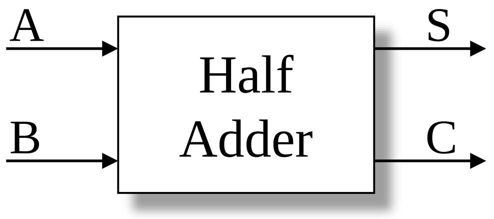


```
module my_module(out1, .., inN);
output out1, .., outM;
input in1, .., inN;

.. // declarations
.. // description of f (maybe
.. // sequential)

endmodule
```

Everything you write in Verilog must be inside a module
exception: compiler directives

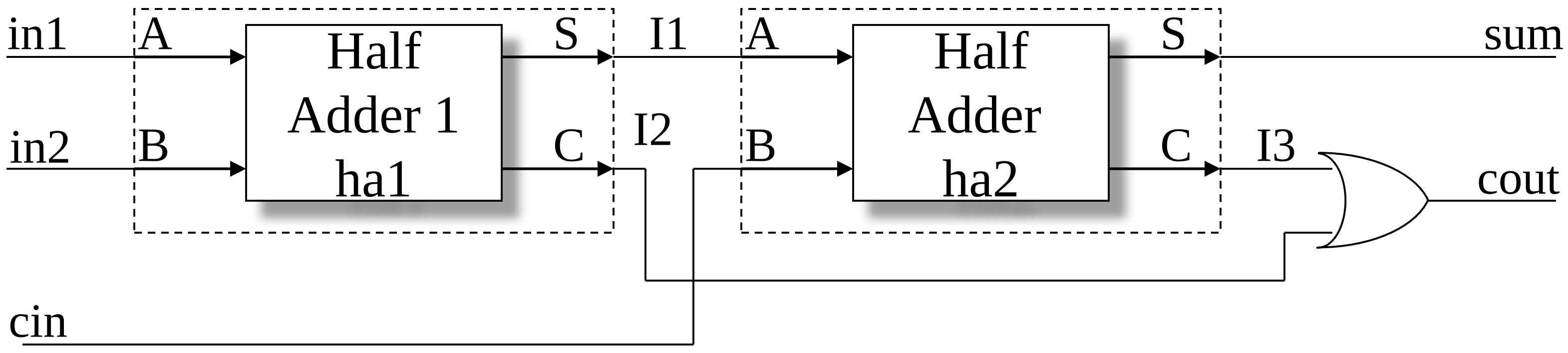
Example: Half Adder





```
module half_adder(S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

Example: Full Adder

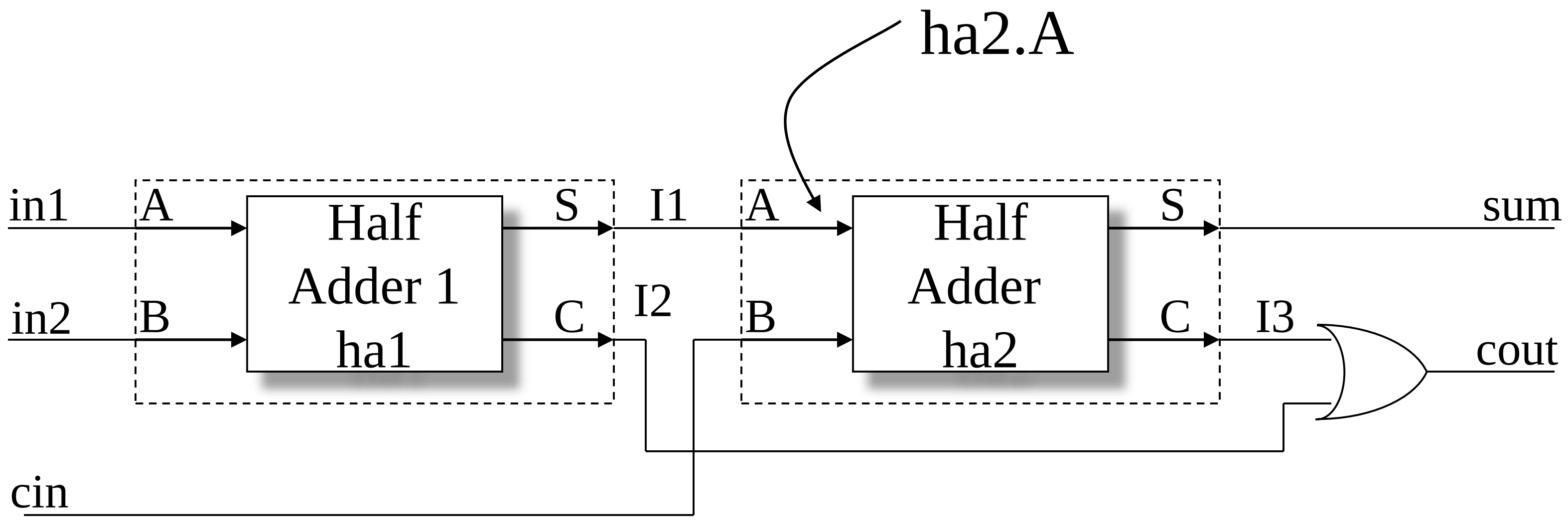


```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(I1, I2, in1, in2);
half_adder ha2(sum, I3, I1, cin);

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

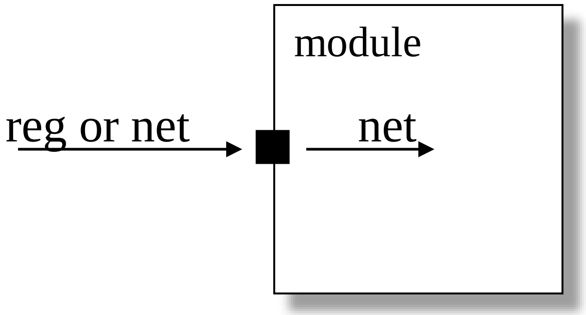
Hierarchical Names



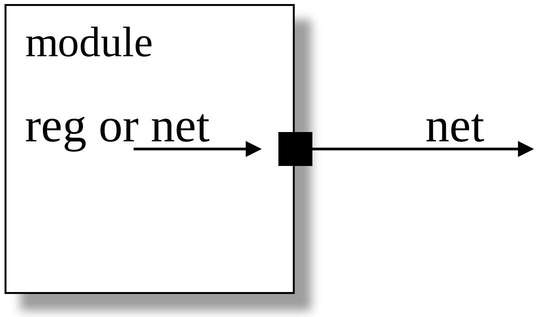
Remember to use instance names,
not module names

Port Assignments

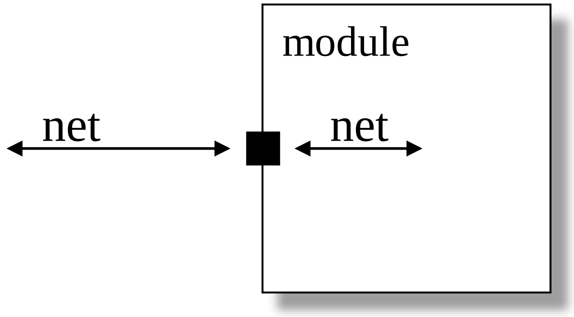
- Inputs



- Outputs



- Inouts



Continuous Assignements

a closer look

- Syntax:

```
assign #del <id> = <expr>;
```

optional

net type !!

- Where to write them:
  - inside a module
  - outside procedures
- Properties:
  - they all execute in parallel
  - are order independent
  - are continuously active

Structural Model (Gate Level)

- Built-in gate primitives:

  `and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1`

- Usage:

  `nand (out, in1, in2);` 2-input NAND without delay

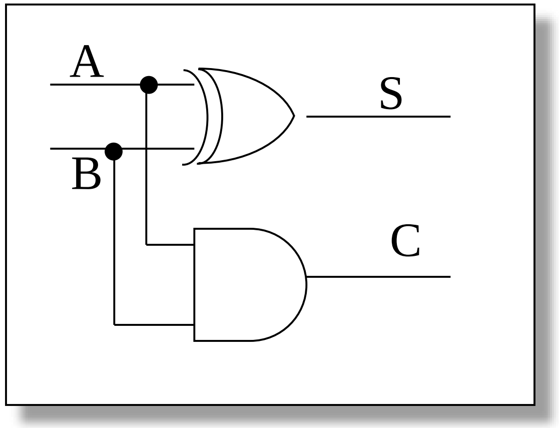
  `and #2 (out, in1, in2, in3);` 3-input AND with 2 t.u. delay

  `not #1 N1(out, in);` NOT with 1 t.u. delay and instance name

  `xor X1(out, in1, in2);` 2-input XOR with instance name

- Write them inside module, outside procedures

Example: Half Adder, 2nd Implementation



Assuming:
- XOR: 2 t.u. delay
- AND: 1 t.u. delay

```
module half_adder(S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

xor #2 (S, A, B);
and #1 (C, A, B);

endmodule
```

Behavioral Model - Procedures (i)

- Procedures = sections of code that we know they execute sequentially
- Procedural statements = statements inside a procedure (they execute sequentially)
- e.g. another 2-to-1 mux implem:

Execution Flow

```
begin
  if (sel == 0)
      Y = B;
  else
      Y = A;
end
```

Procedural assignments:
Y must be reg !!

Behavioral Model - Procedures (ii)

- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other) and ..
- .. can be expressed in two types of blocks:
  - initial → they execute only once
  - always → they execute for ever (until simulation finishes)

"Initial" Blocks

- Start execution at sim time zero and finish when their last statement executes

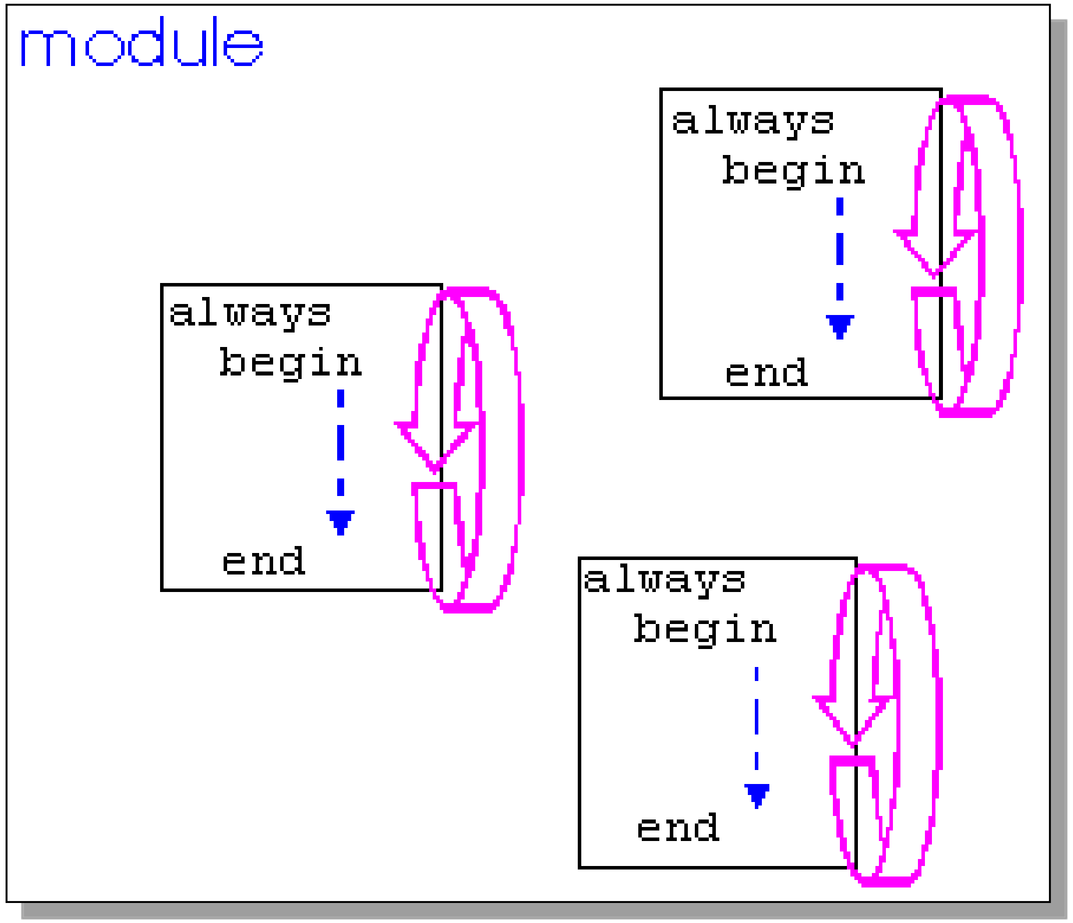
```
module nothing;

initial
  $display("I'm first");

initial begin
  #50;
  $display("Really?");
  end

endmodule
```

Will be displayed at sim time 0

Will be displayed at sim time 50

“Always” Blocks

- Start execution at sim time zero and continue until sim finishes



Events (i)

- *@*

```
always @(signal1 or signal2 or ..) begin
   ..
   end
```

execution triggers every time any signal changes

```
always @(posedge clk) begin
   ..
   end
```

execution triggers every time clk changes from *0* to *1*

```
always @(negedge clk) begin
   ..
   end
```

execution triggers every time clk changes from *1* to *0*

Examples

- 3rd half adder implem

```
module half_adder(S, C, A, B);
output S, C;
input A, B;

reg S,C;
wire A, B;

always @(A or B) begin
   S = A ^ B;
   C = A && B;
   end

endmodule
```

- Behavioral edge-triggered DFF implem

```
module dff(Q, D, Clk);
output Q;
input D, Clk;

reg Q;
wire D, Clk;

always @(posedge Clk)
   Q = D;

endmodule
```

Events (ii)

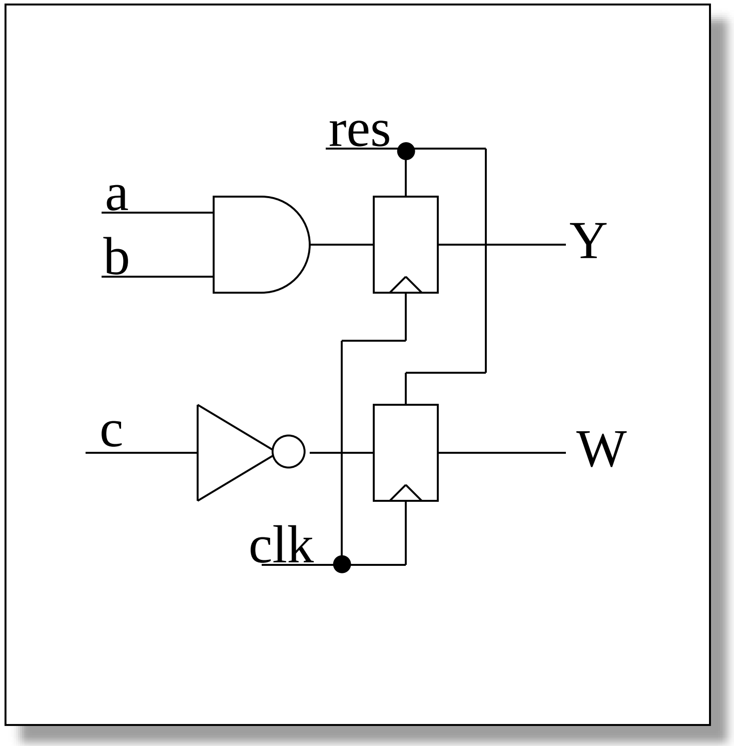
- wait (expr)

```
always begin
    wait (ctrl)
    #10 cnt = cnt + 1;
    #10 cnt2 = cnt2 + 2;

    end
```

execution loops every time ctrl = 1 (level sensitive timing control)

- e.g. Level triggered DFF ?

Example

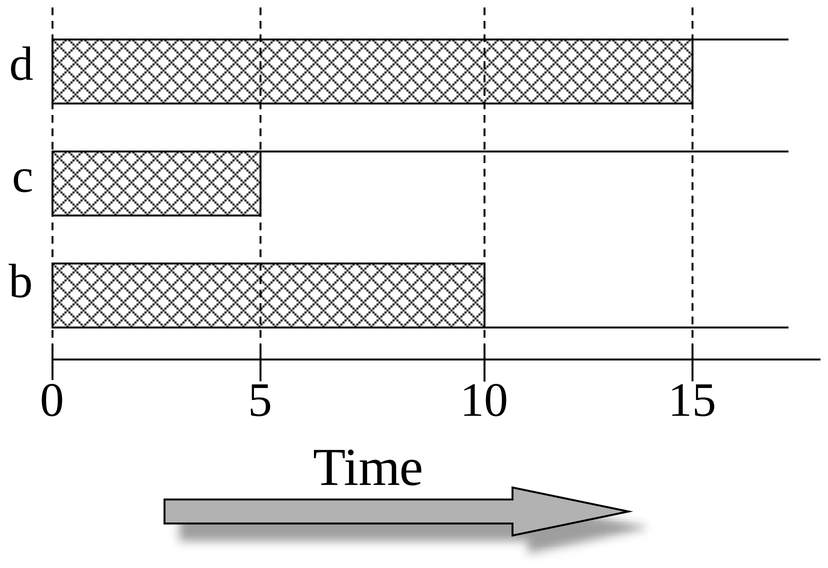


```
always @(res or posedge clk) begin
        if (res) begin
                Y = 0;
                W = 0;
                end
        else begin
                Y = a & b;
                W = ~c;
                end
        end
```

Timing (i)

```
initial begin
          #5 c = 1;
          #5 b = 0;
          #5 d = c;
          end
```

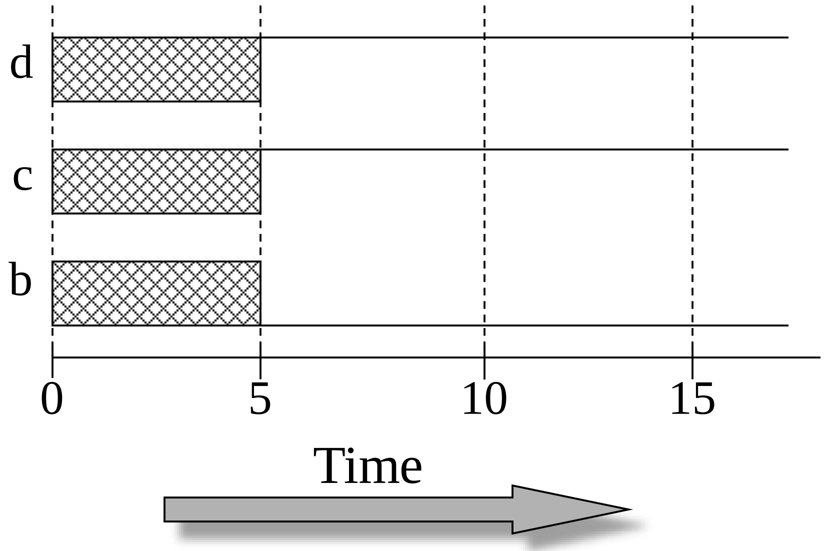
Each assignment is blocked by its previous one



Timing (ii)

```
initial begin
       fork
       #5 c = 1;
       #5 b = 0;
       #5 d = c;
       join
       end
```

Assignments are not blocked here



Procedural Statements: if

```
if (expr1)
        true_stmt1;

else if (expr2)
        true_stmt2;
..
else
        def_stmt;
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        if (sel == 0)
                out = in[0];
        else if (sel == 1)
                out = in[1];
        else if (sel == 2)
                out = in[2];
        else
                out = in[3];
endmodule
```

Procedural Statements: case

case (expr)

item_1, .., item_n: stmt1;
item_n+1, .., item_m: stmt2;
..
default: def_stmt;

endcase

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        case (sel)
        0: out = in[0];
        1: out = in[1];
        2: out = in[2];
        3: out = in[3];
        endcase
endmodule
```

Procedural Statements: for

for (init_assignment; cond; step_assignment)
        stmt;

E.g.
```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start)
        for (i = 0; i < 3; i = i + 1)
                #10 Y = Y + 1;
endmodule
```

Procedural Statements: while

while (expr) stmt;

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start) begin
        i = 0;
        while (i < 3) begin
                #10 Y = Y + 1;
                i = i + 1;
                end
        end
endmodule
```

Procedural Statements: repeat

repeat (times) stmt;

Can be either an integer or a variable

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;

initial
       Y = 0;

always @(posedge start)
       repeat (4) #10 Y = Y + 1;
endmodule
```

Procedural Statements: forever

forever stmt;

Executes until sim finishes

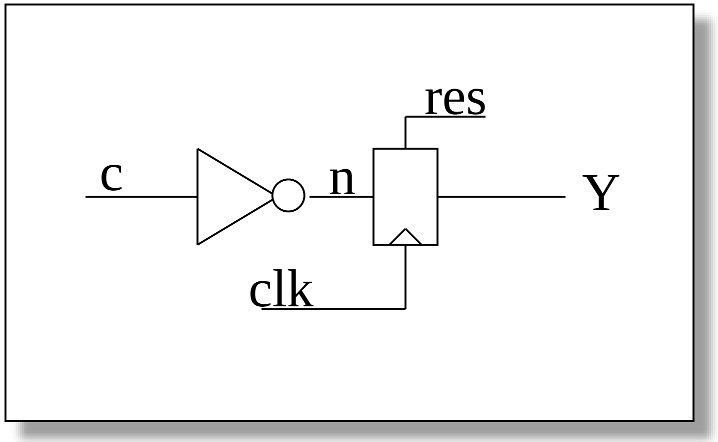
Typical example:
clock generation in test modules

```
module test;

reg clk;

initial begin
        clk = 0;
        forever #10 clk = ~clk;
        end

other_module1 o1(clk, ..);
other_module2 o2(.., clk, ..);

endmodule
```

T_{clk} = 20 time units

Mixed Model

Code that contains various both structure and behavioral styles



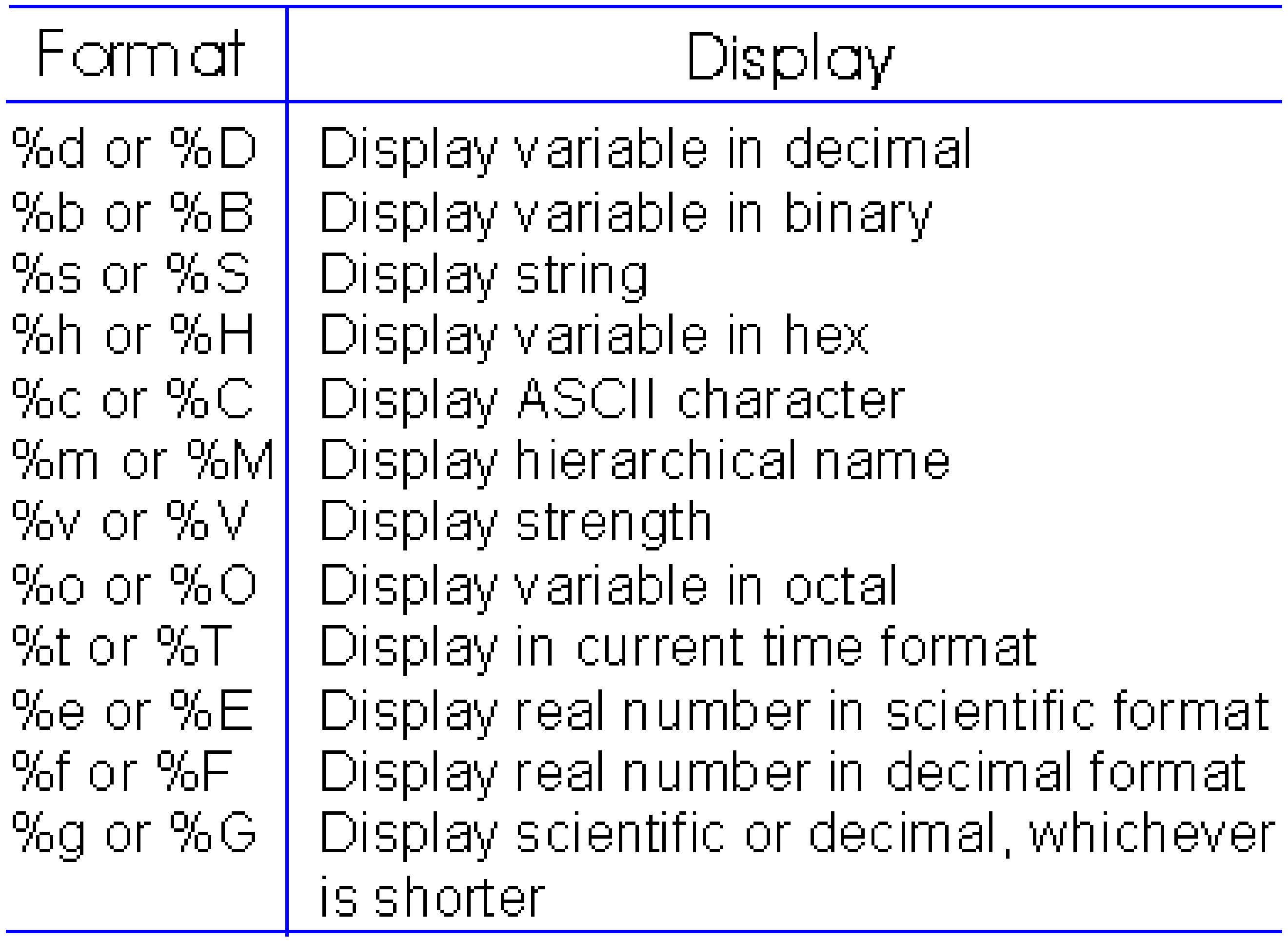
```
module simple(Y, c, clk, res);
output Y;
input c, clk, res;

reg Y;
wire c, clk, res;
wire n;

not(n, c); // gate-level

always @(res or posedge clk)
        if (res)
                Y = 0;
        else
                Y = n;
endmodule
```

System Tasks

Always written inside procedures

- $display(“..”, arg2, arg3, ..); → much like printf(), displays formatted string in std output when encountered
- $monitor(“..”, arg2, arg3, ..); → like $display(), but .. displays string each time any of arg2, arg3, .. Changes
- $stop; → suspends sim when encountered
- $finish; → finishes sim when encountered
- $fopen(“filename”); → returns file descriptor (integer); then, you can use $fdisplay(fd, “..”, arg2, arg3, ..); or $fmonitor(fd, “..”, arg2, arg3, ..); to write to file
- $fclose(fd); → closes file
- $random(seed); → returns random integer; give her an integer as a seed

$display & $monitor string format



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
%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
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Compiler Directives

- `include "filename" → inserts contents of file into current file; write it anywhere in code ..

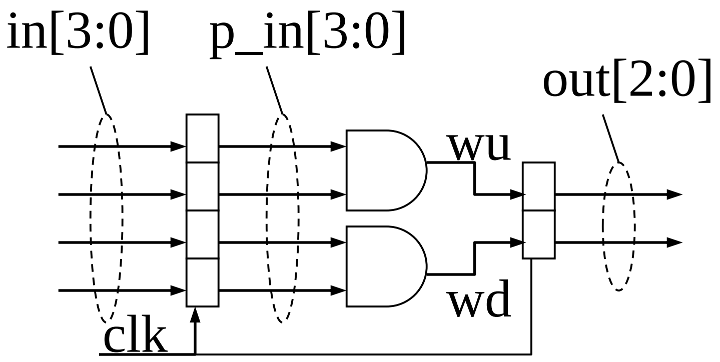
- `define <text1> <text2> → text1 substitutes text2;
  - e.g. `` `define BUS reg [31:0] `` in declaration part: `` `BUS data; ``

- `timescale <time unit>/<precision>
  - e.g. `` `timescale 10ns/1ns `` later: `#5 a = b;`

$\uparrow$

$\boxed{\text{50ns}}$

Parameters



A. Implelementation without parameters

```
module dff4bit(Q, D, clk);
output [3:0] Q;
input [3:0] D;
input clk;

reg [3:0] Q;
wire [3:0] D;
wire clk;

always @(posedge clk)
      Q = D;

endmodule
```

```
module dff2bit(Q, D, clk);
output [1:0] Q;
input [1:0] D;
input clk;

reg [1:0] Q;
wire [1:0] D;
wire clk;

always @(posedge clk)
      Q = D;

endmodule
```

Parameters (ii)

A. Implelementation without parameters (cont.)

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;    // internal nets
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff4bit instA(p_in, in, clk);
dff2bit instB(out, {wu, wd}, clk);
// notice the concatenation!!

endmodule
```

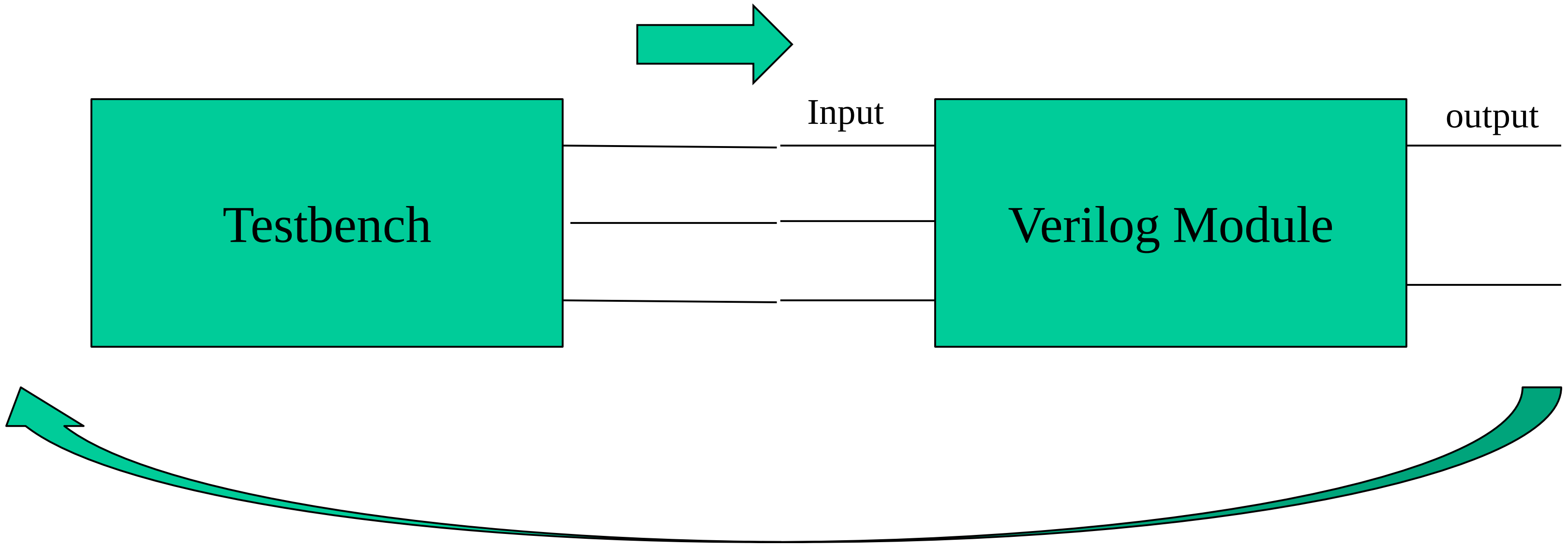
Parameters (iii)

B. Implelementation with parameters

```
module dff(Q, D, clk);
parameter WIDTH = 4;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @(posedge clk)
      Q = D;

endmodule
```

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff instA(p_in, in, clk);
// WIDTH = 4, from declaration

dff instB(out, {wu, wd}, clk);
      defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```

Testing Your Modules



Testing Your Modules

```
module top_test;
wire [1:0] t_out;   // Top’s signals
reg [3:0] t_in;
reg clk;

top inst(t_out, t_in, clk); // Top’s instance

initial begin       // Generate clock
       clk = 0;
       forever #10 clk = ~clk;
end

initial begin       // Generate remaining inputs
       $monitor($time, " %b -> %b", t_in, t_out);
       #5 t_in = 4'b0101;
       #20 t_in = 4'b1110;
       #20 t_in[0] = 1;
       #300 $finish;
end

endmodule
```

The Veriwell Simulator

- Assuming that modules `dff`, `top` and `top_test` reside in files `dff.v`, `top.v` and `top_test.v` respectively, run:

```
~hy225/veriwell/sparc_bin/veriwell dff.v top.v top_test.v
```

- result:

```
.. (initial messages)
0 xxxx -> xx
5 0101 -> xx
25 1110 -> xx
30 1110 -> 00
45 1111 -> 00
50 1111 -> 10
70 1111 -> 11
.. (final messages)
```