

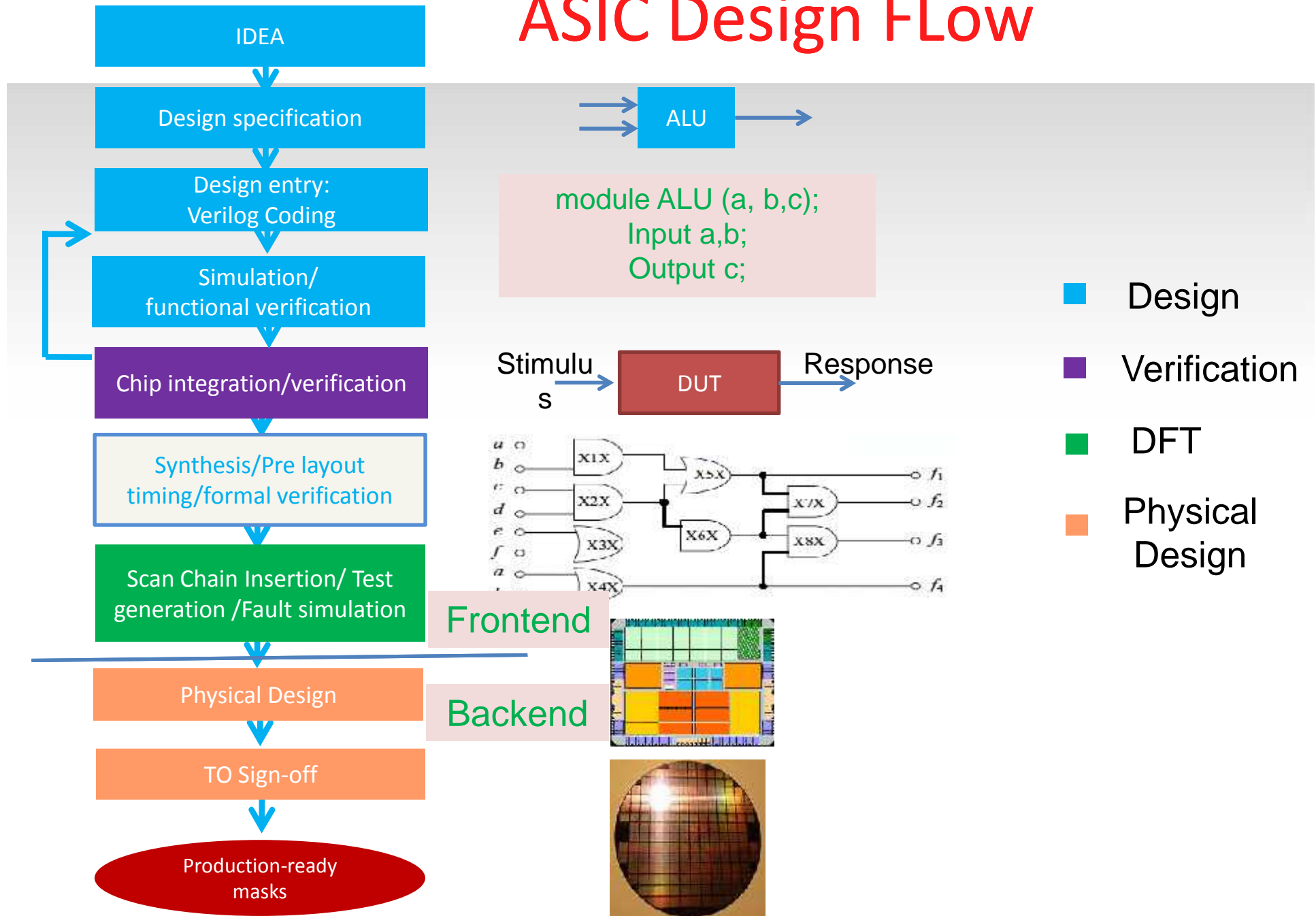
Verilog RTL 编程实践

klin

qixin_soc@163.com

启芯工作室

ASIC Design Flow



What is Verilog

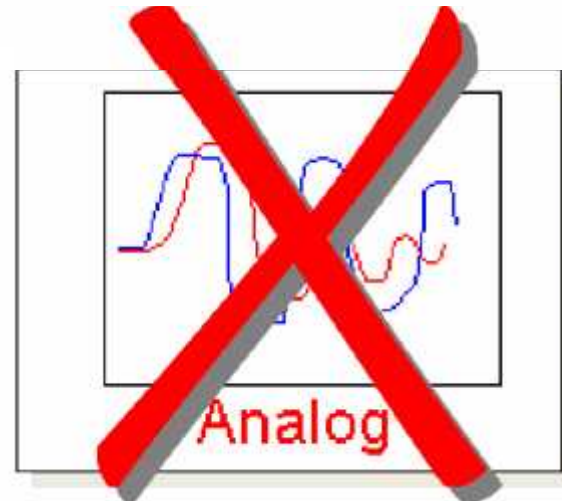
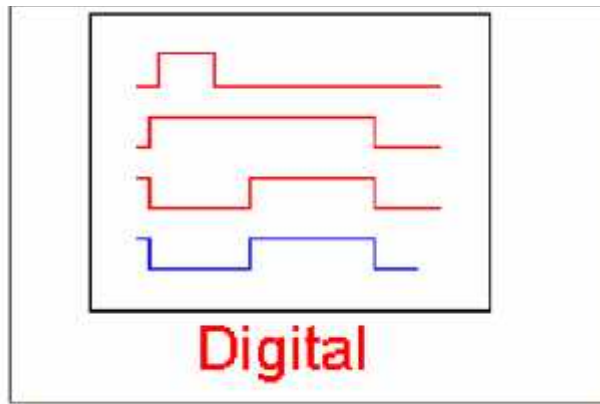
- Hardware Description Language (HDL)
- Developed in 1984
- Standard: IEEE 1364, Dec 1995
- Standard: IEEE 1364-2001, 2002
- In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005 .
- SystemVerilog is an extension of Verilog-2005; all features of that language are available in SystemVerilog.

References

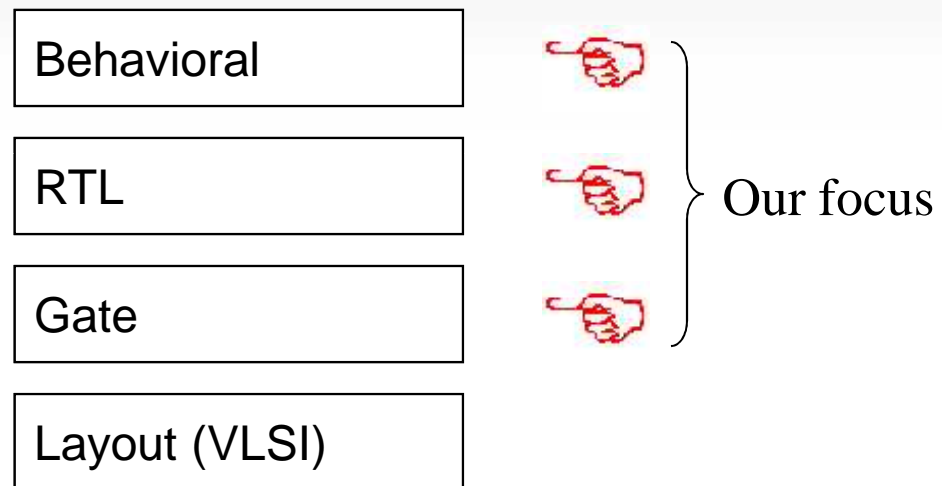
- IEEE 1364 standard
- VERILOG HDL硬件描述语言
- Verilog综合实用教程
- Verilog数字系统设计教程

Basic Limitation of Verilog

Description of digital systems only



Abstraction Levels in Verilog

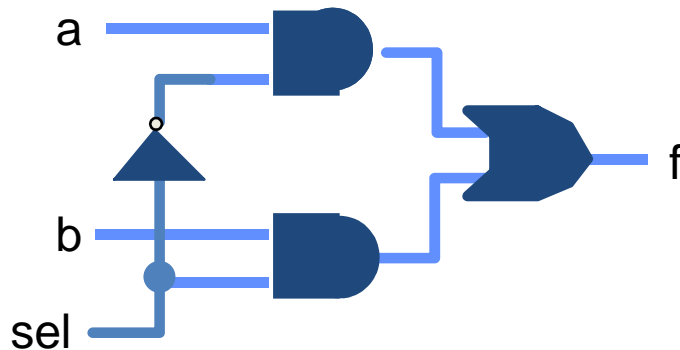


Verilog Example: behavioral model

```
module adder_4_RTL (a, b, c_in, sum, c_out);  
    output [3:0] sum;  
    output c_out;  
    input [3:0] a, b;  
    input c_in;  
  
    assign {c_out, sum} = a + b + c_in;  
  
endmodule
```

Verilog Example: Structural Models

- Structural models
 - Are built from gate primitives and/or other modules
 - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
 - You could describe your assignment1 circuit this way
- Identify
 - Gate instances, wire names, delay from a or b to f.

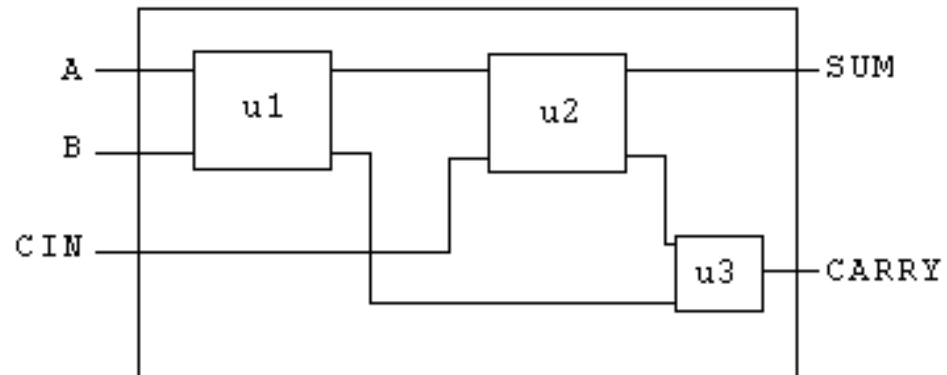
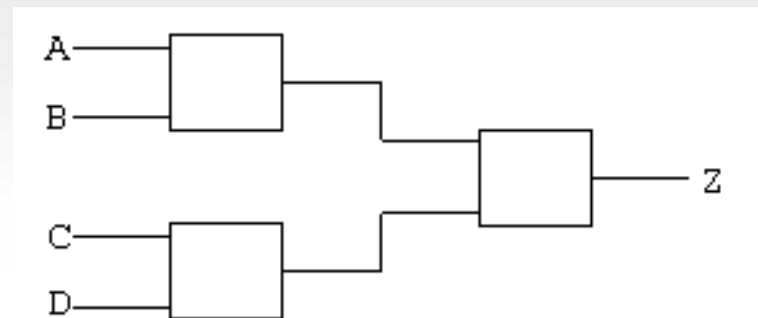


```
module mux (f, a, b, sel);
    output   f;
    input    a, b, sel;

    and #5   g1 (f1, a, nsel);
    and #5   g2 (f2, b, sel);
    or  #5   g3 (f, f1, f2);
    not     g4 (nsel, sel);
endmodule
```


Main Language Concepts (i)

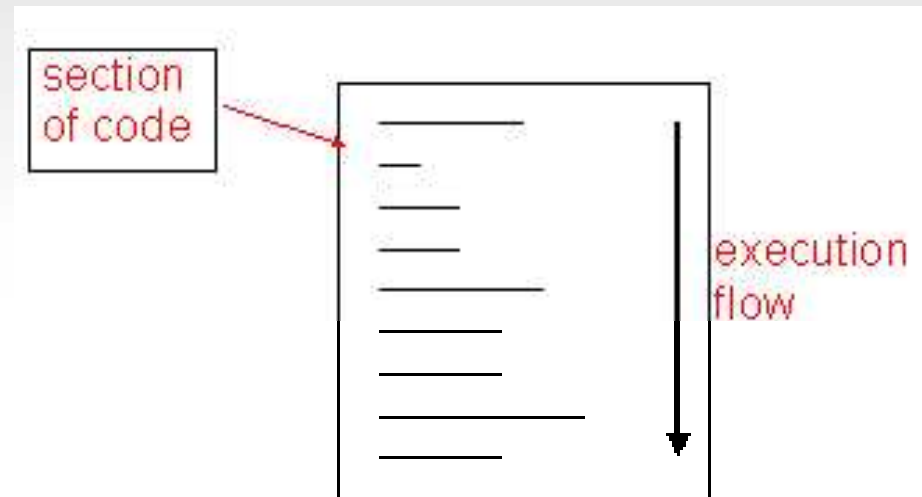
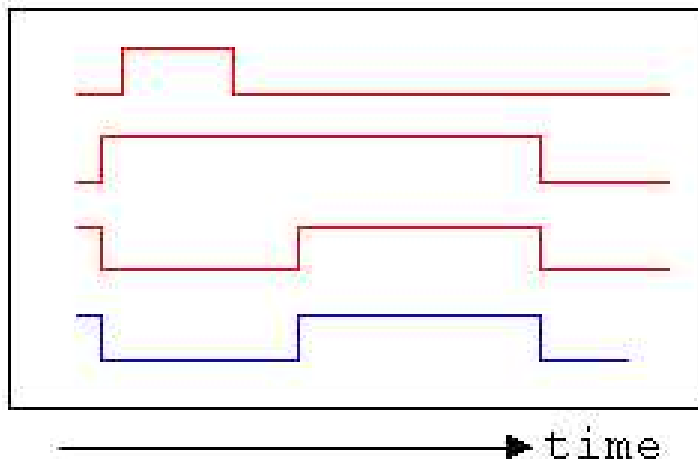
- Concurrency



- Structure

Main Language Concepts (ii)






- Procedural Statements



- Time

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\$ 

- Case sensitivity

- myid \neq Myid

Comments

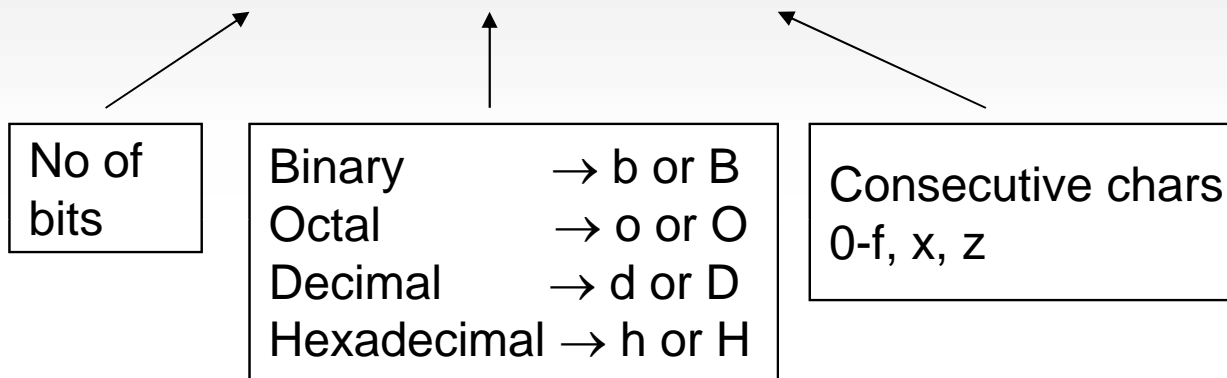
- `// The rest of the line is a comment`
- `/* Multiple line
comment */`
- `/* Nesting /* comments */ do NOT work */`

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>



- 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 \Rightarrow extend this
 - 4'b x1 = 4'b xx_x1
 - MS bit = 1 \Rightarrow zero extension
 - 4'b 1x = 4'b 00_1x

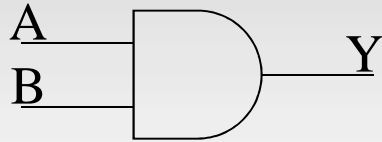
Numbers in Verilog (iii)

- If *size* is omitted 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 omitted too .. decimal is assumed
 - 15 = <size>'d 15

Nets (i)

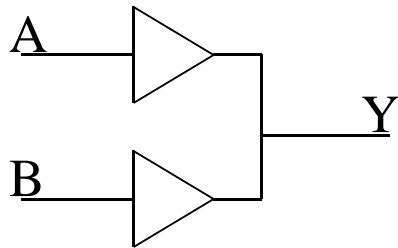
- Can be thought as hardware wires driven by logic
- Equal z when unconnected
- Various types of nets
 - `wire`
 - `wand` (wired-AND)
 - `wor` (wired-OR)
 - `tri` (tri-state)
- In following examples: Y is evaluated, ***automatically***, every time A or B changes

Nets (ii)



```

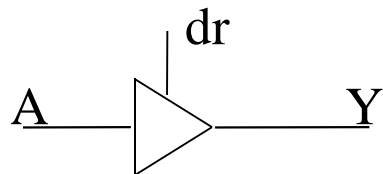
wire Y; // declaration
assign Y = A & B;
  
```



```

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

		A	
Y		0	1
		0	1
B	0	0	0
	1	0	1



```

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

		A	
Y		0	1
		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

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

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

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

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-dimensional 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:

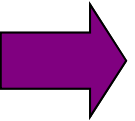
```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = "Hello Verilog";
string_val = "hello"; // MS Bytes are filled with 0
string_val = "I am overflowed"; // "I " is truncated
```

- Escaped chars:

- \n newline
- \t tab
- %% %
- \\ \
- \" "

Logical Operators

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

<code>A = 6;</code>		<code>A && B → 1 && 0 → 0</code>
<code>B = 0;</code>		<code>A !B → 1 1 → 1</code>
<code>C = x;</code>		<code>C B → x 0 → x</code>

but `C&&B=0`

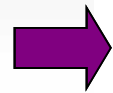
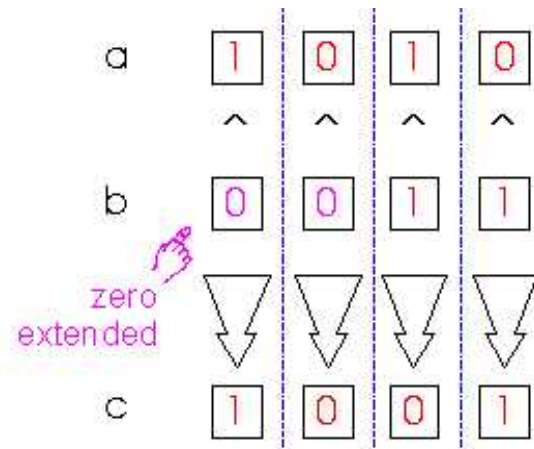
Bitwise Operators (i)

- $\&$ → bitwise AND
- $|$ → bitwise OR
- \sim → bitwise NOT
- \wedge → bitwise XOR
- $\sim \wedge$ or $\wedge \sim$ → bitwise XNOR
- Operation on bit by bit basis

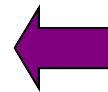
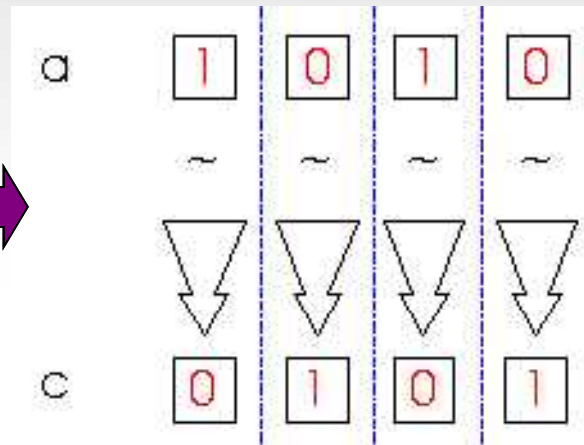
Bitwise Operators (ii)

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

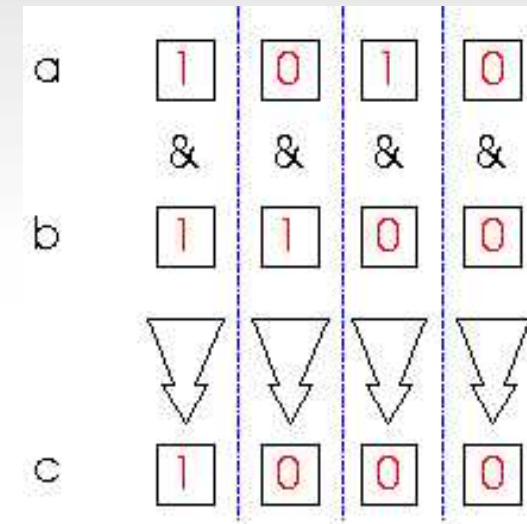
$c = a \wedge b;$



$c = \sim a;$



$c = a \& b;$



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

Reduction Operators

- $\&$ \rightarrow AND
- $|$ \rightarrow OR
- \wedge \rightarrow XOR
- $\sim\&$ \rightarrow NAND
- $\sim|$ \rightarrow NOR
- $\sim\wedge$ or $\wedge\sim$ \rightarrow XNOR
- One multi-bit operand \rightarrow 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

- `==` → logical equality
 - `!=` → logical inequality
 - `===` → case equality
 - `!==` → case inequality
- } Return *0*, *1* or *x*
- } Return *0* or *1*

— `4'b 1z0x == 4'b 1z0x` → *x*

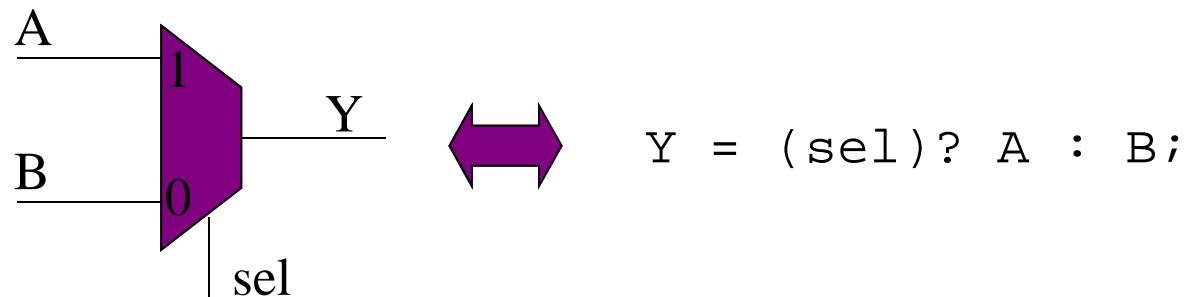
— `4'b 1z0x != 4'b 1z0x` → *x*

— `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 ..



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)
```

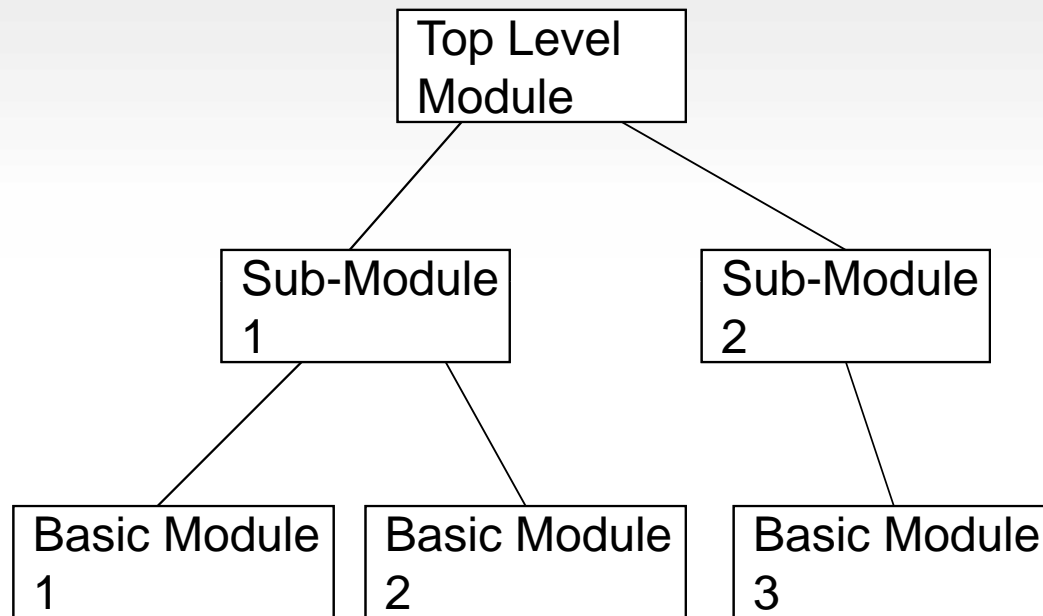
```
intA = -'d12/3;       // evaluates to 1431655761 (base spec)
```

Operator Precedence

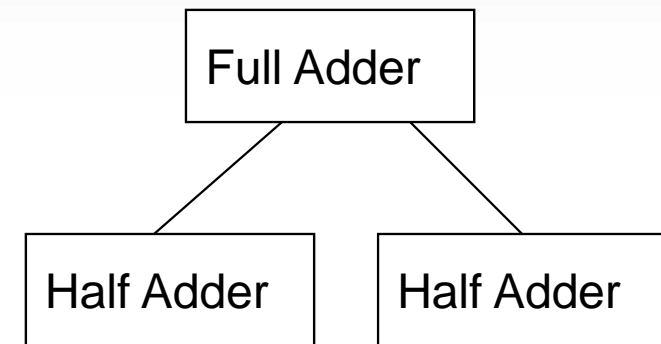
<code>+ - ! ~ unary</code>	highest precedence
<code>* / %</code>	
<code>+ - (binary)</code>	
<code>< < > ></code>	
<code>< <= == > ></code>	
<code>== != === !==</code>	
<code>& ~ &</code>	
<code>^ ^~ ~^</code>	
<code> ~ </code>	
<code>&&</code>	
<code> </code>	
<code>?: conditional</code>	lowest precedence

Use parentheses to
enforce your
priority

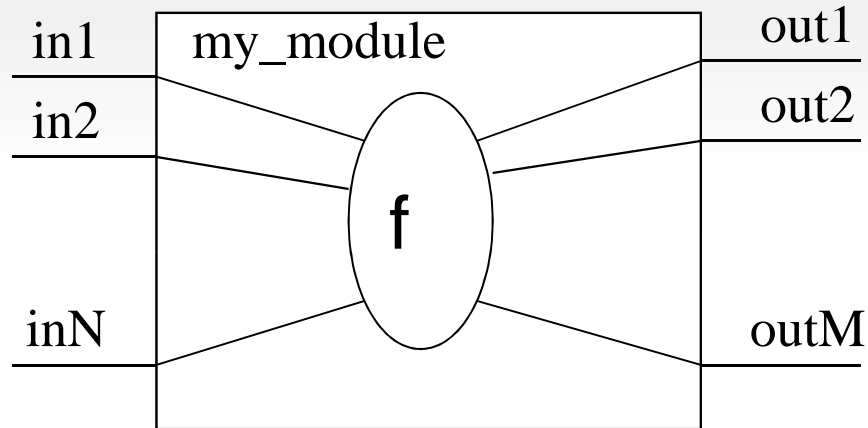
Hierarchical Design



E.g.



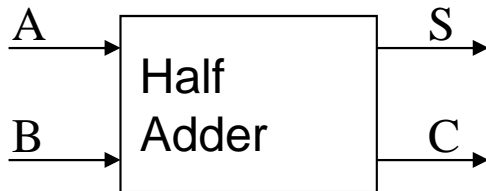
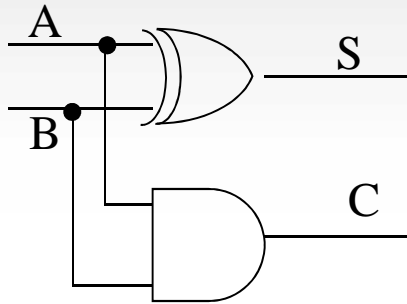
Module



```
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



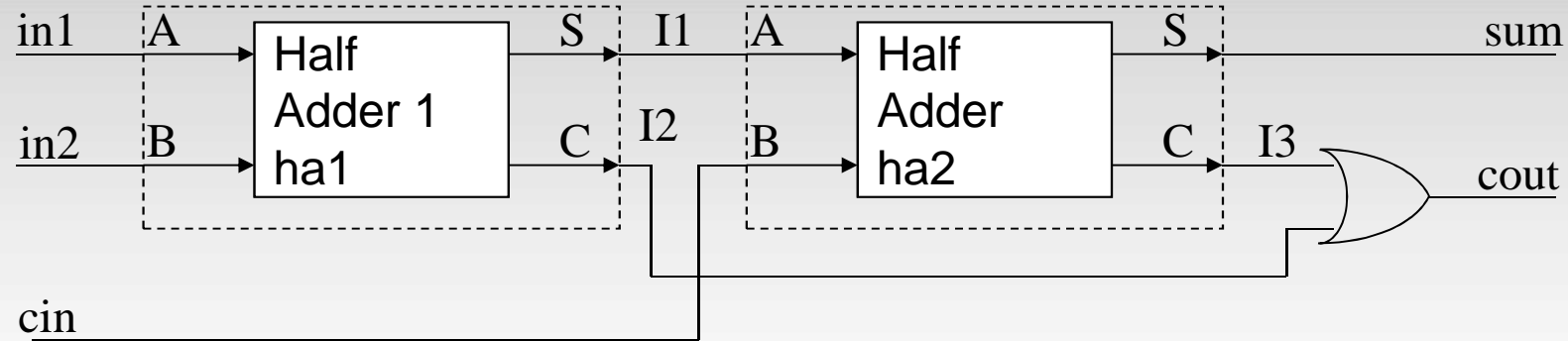
```
module half_adder(S, C, A, B);  
  output S, C;  
  input A, B;
```

```
  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;
```

Module
name

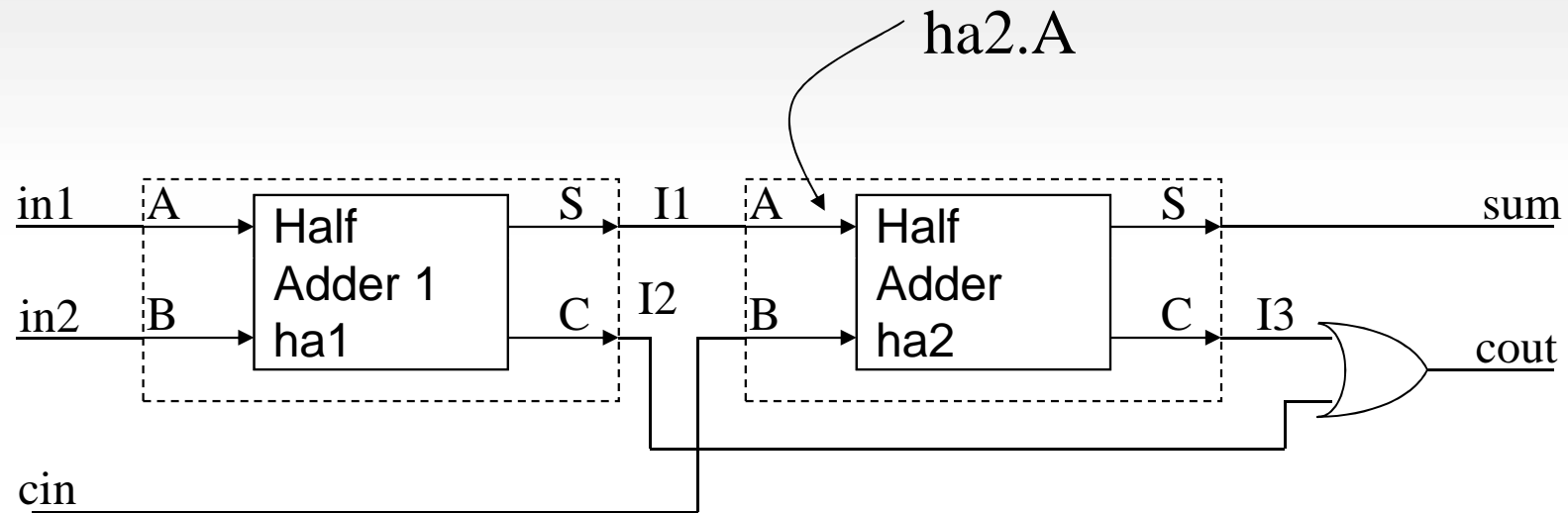
```
    half_adder ha1(I1, I2, in1, in2);  
    half_adder ha2(sum, I3, I1, cin);
```

Instance
name

```
    assign cout = I2 || I3;
```

```
endmodule
```

Hierarchical Names

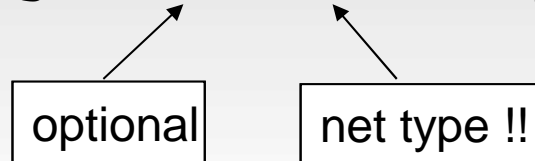


Remember to use instance names,
not module names

Continuous Assignments

- Syntax:

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



- 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:

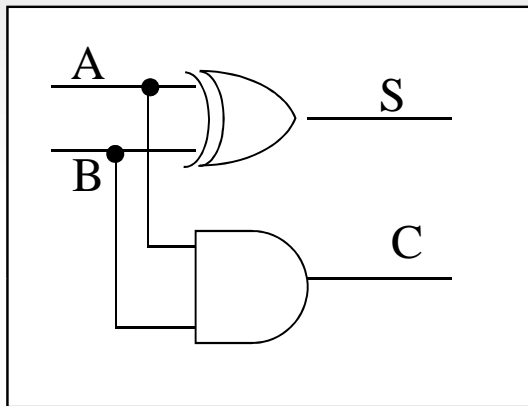
```
and, nand, nor, or, xor, xnor, buf, not, bufif0,  
bufif1, notif0, notif1
```

- 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;
```

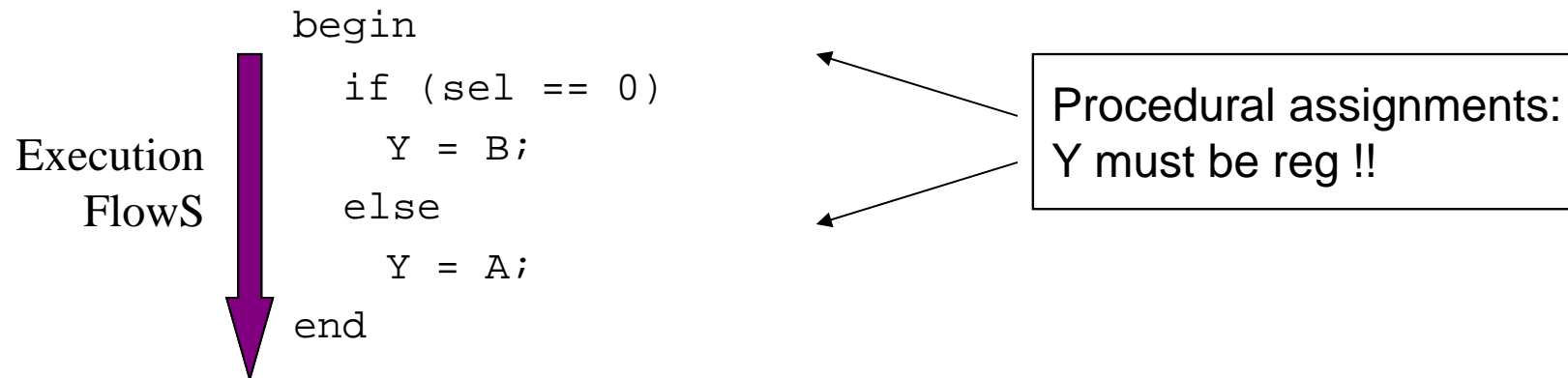
```
  wire S, C, A, B;
```

```
  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:



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)

Block statement (i)

- The Verilog HDL contains two types of blocks:
 - Sequential (begin-end blocks).
 - Parallel (fork-join blocks).
- These blocks can be used if more than one statement should be executed.
- All statements within sequential blocks (Example 1) are executed in the order in which they are given. If a timing control statement appears within a block, then the next statement will be executed after that delay.
- All statements within parallel blocks (Example 2) are executed at the same time. This means that the execution of the next statement will not be delayed even if the previous statement contains a timing control statement.

Block statement (ii)

Example 1:

begin

a = 1;

#10 a = 0;

#5 a = 4;

end

During the simulation, this block will be executed in 15 time units. At time 0, the 'a' variable will be 1, at time 10 the 'a' variable will be 0, and at time 15 (#10 + #5) the 'a' variable will be 4.

Example 2:

fork

a = 1;

#10 a = 0;

#5 a = 4;

join

During the simulation this block will be executed in 10 time units. At time 0 the 'a' variable will be 1, at time 5 the 'a' variable will be 4, and at time 10 the 'a' variable will be 0.

Block statement(iii)

always	c
begin	
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
end	

always	c
fork	
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
join	

initial	c
begin	
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
end	

initial	c
fork	
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
c	_____
join	

“Initial” Blocks

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

```
module nothing;
```

```
  initial
```

```
    $display( "I'm first" ); ←
```

Will be displayed
at sim time 0

```
  initial begin
```

```
    #50;
```

```
    $display( "Really?" ); ←
```

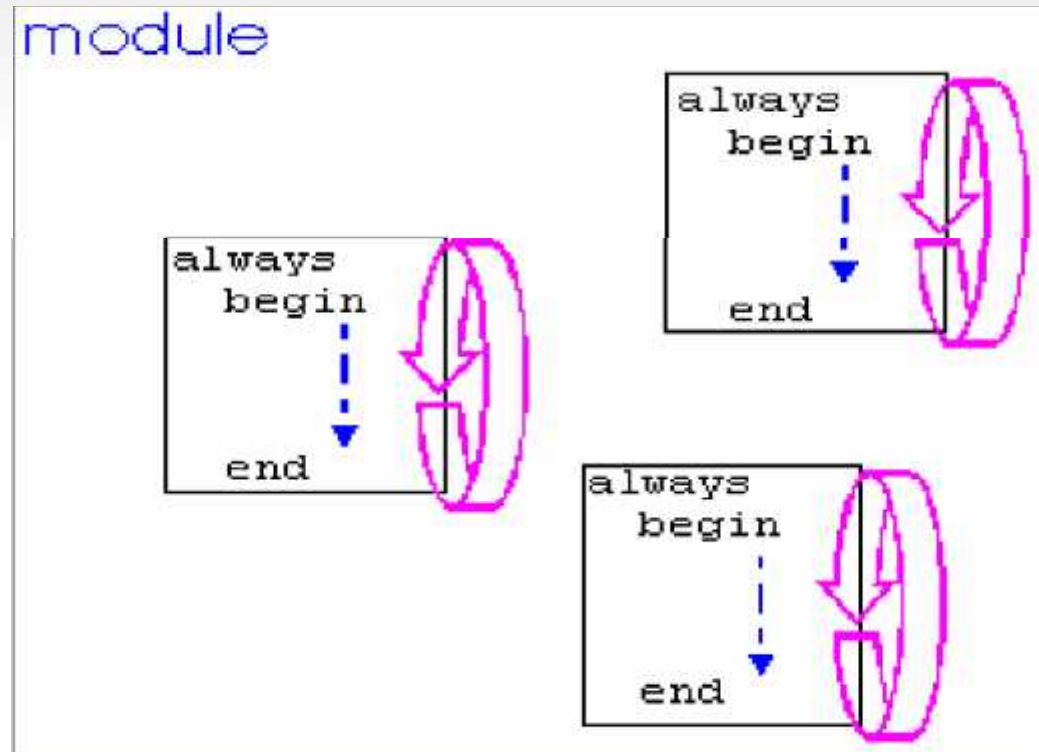
Will be displayed
at sim time 50

```
  end
```

```
endmodule
```

“Always” Blocks

- Start execution at sim time zero and continue until simulation 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 implementation

```
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)

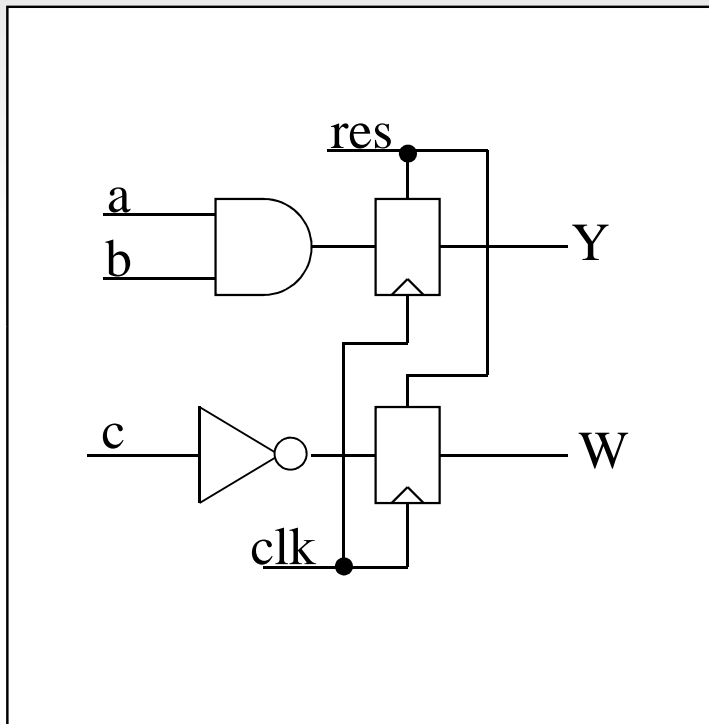
```
always begin
    wait (ctrl)
    #10 cnt = cnt + 1;
    #10 cnt2 = cnt2 + 2;

end
```

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

- e.g. Level triggered DFF ?

Example

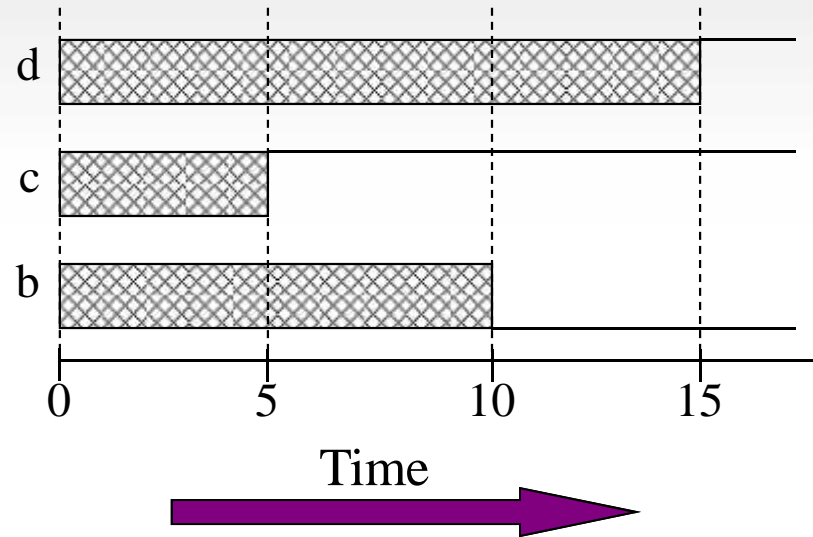
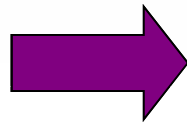


```
always @(posedge res or posedge clk) begin
    if (res) begin
        Y <= 0;
        W <= 0;
    end
    else begin
        Y <= a & b;
        W <= ~c;
    end
end
```


Timing (i)

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

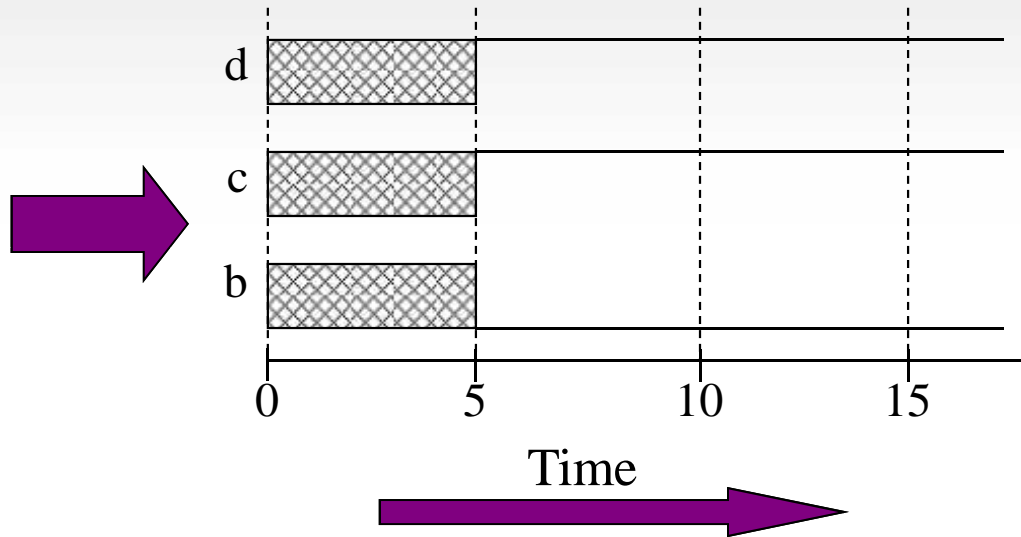
Each assignment is
blocked by its previous one



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

E.g.

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

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

`while (expr) stmt;`


```
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, ..);
```

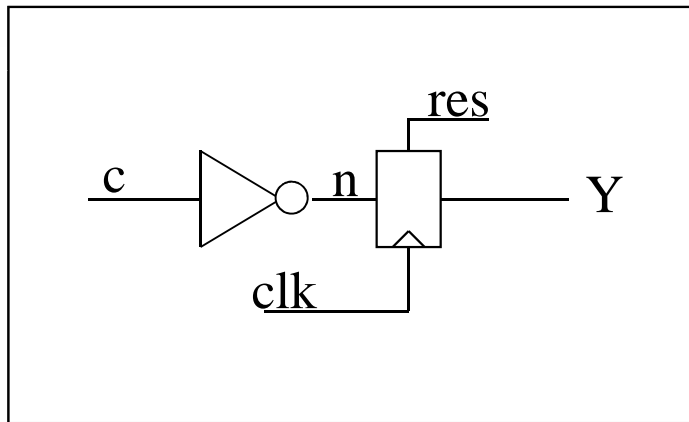
```
  other_module2 o2(.., clk, ..);
```

```
endmodule
```

$T_{\text{clk}} = 20$ 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 @(posedge res or posedge clk)  
    if (res)  
      Y <= 0;  
    else  
      Y <= n;  
endmodule
```

System Tasks

Always written inside procedures

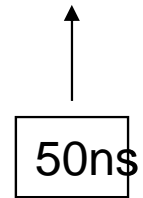
- `$display("..", arg2, arg3, ..);` → much like `printf()`, displays formatted string in std output when encountered
- `$monitor("..", arg2, arg3, ..);` → like `$display()`, but `..` displays string each time any of `arg2, arg3, ..` Changes
- `$stop;` → suspends sim when encountered
- `$finish;` → finishes sim when encountered
- `$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

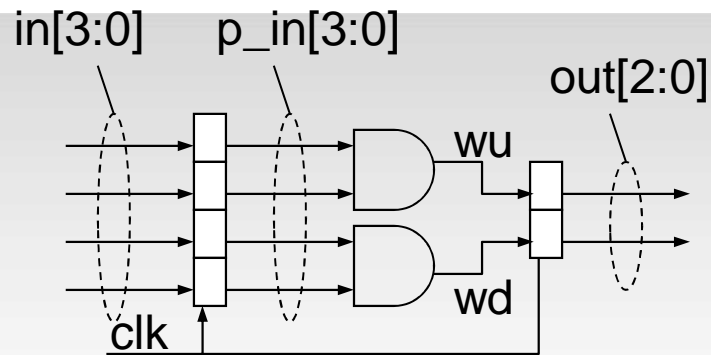
Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current 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;`



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
```

A. Implementation
without parameters

```
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. Implementation 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. Implementation 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

```
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
```


Today: Verilog and Sequential Logic

- Flip-flops
 - representation of clocks - timing of state changes
 - asynchronous vs. synchronous
- FSMs
 - structural view (FFs separate from combinational logic)
 - behavioral view (synthesis of sequencers)

Latch with Reset

```
module synLatchReset
(Q, g, d, reset);
Input g,d,reset;
output Q;
always @(d or reset or g)
if (~reset)
Q = 0;
else if (g)
Q = d;
endmodule
```

No edge sensitive
specification in sensitive
list
because there is no
specification for what
happens when reset is
one and g is zero, a latch
is needed to remember the
previous value of Q.
Thus latch inferred.

FF with async Resetn

```
module flipflop (D, Clock, Resetn, Q);  
    input D, Clock, Resetn;  
    output Q;  
    reg Q;  
    always @(negedge Resetn or posedge Clock)  
        if (!Resetn)  
            Q <= 0;  
        else  
            Q <= D;  
endmodule
```

Incorrect Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock to change

```
module dff (CLK, d, q);  
  
    input  CLK, d;  
    output q;  
    reg    q;  
  
    always @(CLK)  
        q = d;  
  
endmodule
```

Correct Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock edge

```
module dff (CLK, d, q);  
  
    input  CLK, d;  
    output q;  
    reg    q;  
  
    always @(posedge CLK)  
        q <= d;  
  
endmodule
```

More Flip-flops

- Synchronous reset/set

Synchronous

```
module dff (CLK, s, r, d, q);  
    input  CLK, s, r, d;  
    output q;  
    reg    q;  
  
    always @(posedge CLK)  
        if (r)      q = 1'b0;  
        else if (s) q = 1'b1;  
        else        q = d;  
  
endmodule
```

FF with Async&Sync Resetn

```
module flipflop (D, Clock, Resetn, Q, Syncrst);  
  input D, Clock, Resetn, Syncrst;  
  output Q;  
  reg Q;  
  always @(negedge Resetn or posedge Clock)  
    if (!Resetn)  
      Q <= 0;  
    elseif (Syncrst)  
      Q <= 1'b0;  
    else  
      Q <= D;  
endmodule
```

N-bit shift register

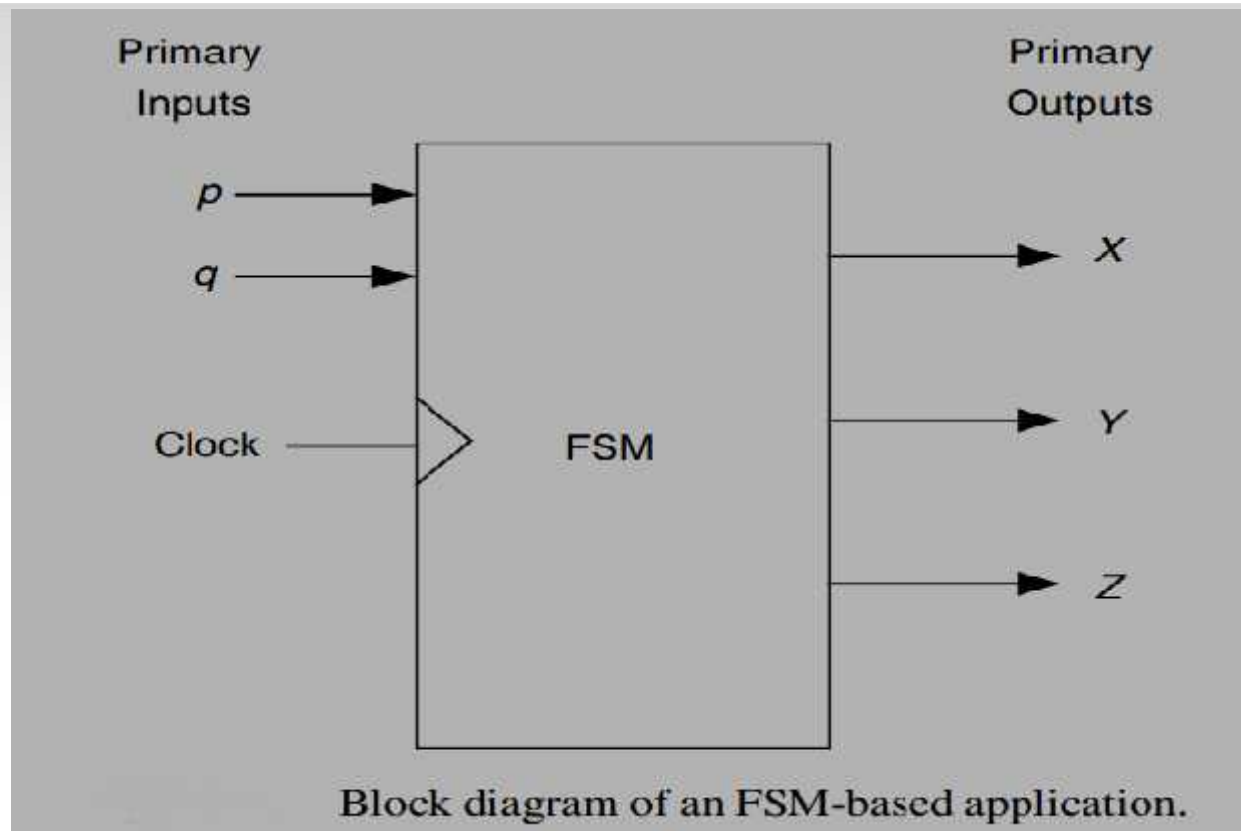
```
module shiftn (R, L, w, Clock, rst_n, Q);
    parameter n = 16;
    input [n-1:0] R;
    input L, w, Clock, rst_n;
    output [n-1:0] Q;
    reg [n-1:0] Q;
    integer k;

    always @(posedge Clock or negedge rst_n)
        if (rst_n)
            Q <= 0;
        else if (L)
            Q <= R;
        else
            begin
                for (k = 0; k < n-1; k = k+1)
                    Q[k] <= Q[k+1];
                Q[n-1] <= w;
            end
endmodule
```


A upcounter with load value

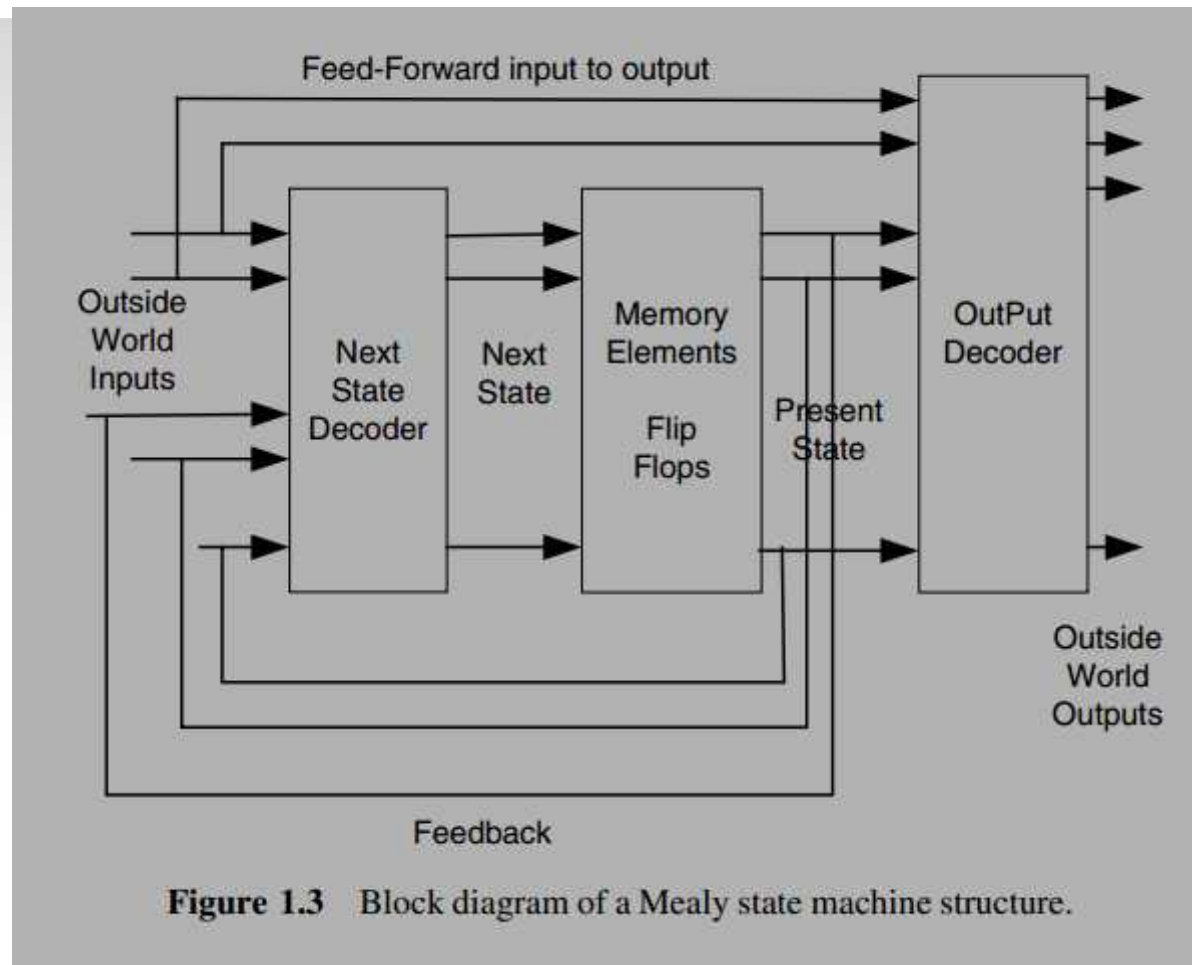
```
module upcount (R, Resetn, Clock, E, L, Q);  
    input [3:0] R;  
    input Resetn, Clock, E, L;  
    output [3:0] Q;  
    reg [3:0] Q;  
    always @(negedge Resetn or posedge Clock)  
        if (!Resetn)  
            Q <= 0;  
        else if (L)  
            Q <= R;  
        else if (E)  
            Q <= Q + 1;  
endmodule
```

What's an FSM



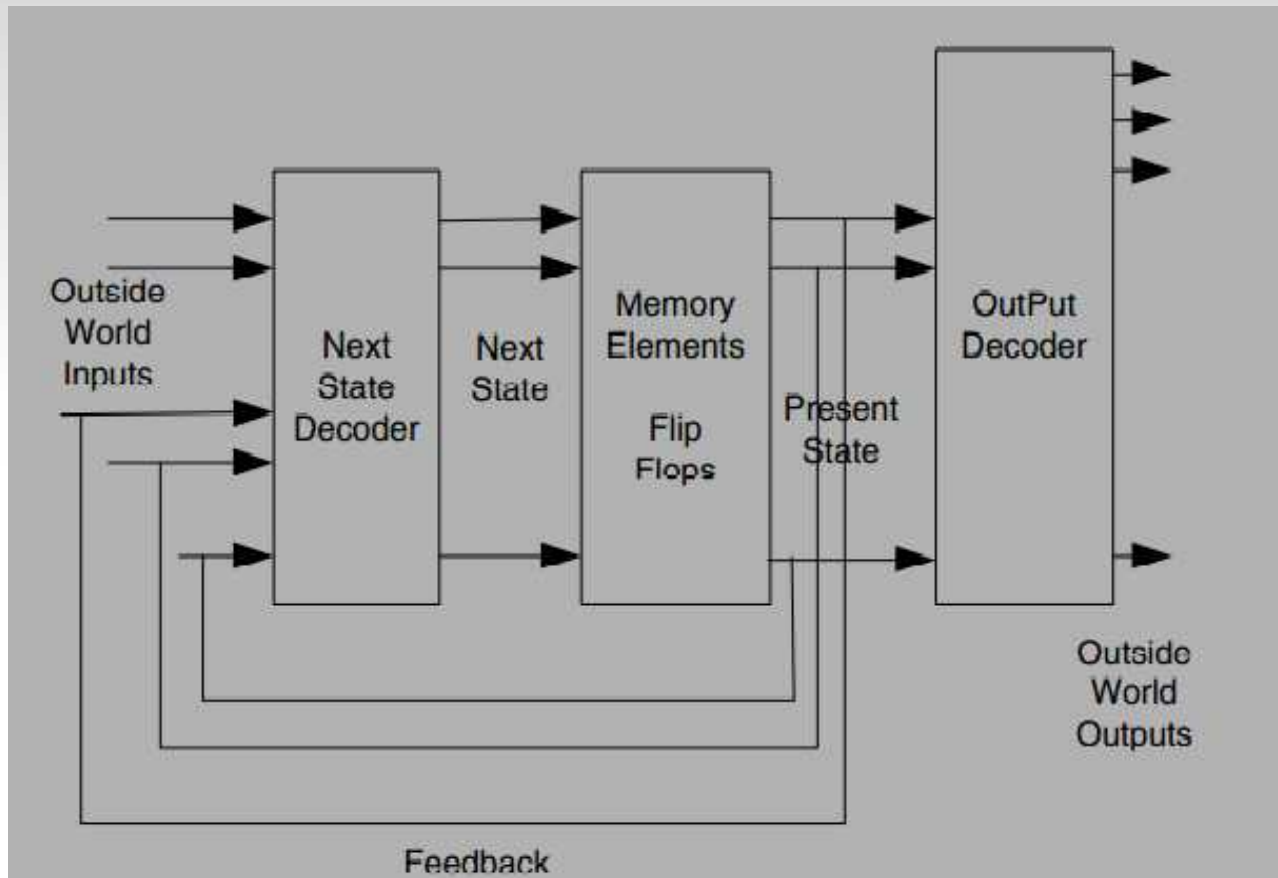
An FSM is a digital sequential circuit that can follow a number of predefined states under the control of one or more inputs. Each state is a stable entity that the machine can occupy. It can move from this state to another state under the control of an outside-world input.

Mealy state machine



Mealy: outputs may depend on current state and current inputs

Moore FSM



Block diagram of a Moore state machine structure.

Moore: outputs depend on current state only

Verilog Structural View of a FSM

- General view of a finite state machine in verilog

```
module FSM (CLK, in, out);  
    input        CLK;  
    input        in;  
    output       out;  
    reg          out;  
  
    // state variable  
    reg [1:0]    state;  
  
    // local variable  
    reg [1:0]    next_state;  
  
    always @(posedge CLK) // registers  
        state = next_state;  
  
    always @(state or in)  
        // Compute next-state and output logic whenever state or inputs change.  
        // (i.e. put equations here for next_state[1:0])  
        // Make sure every local variable has an assignment in this block!  
endmodule
```

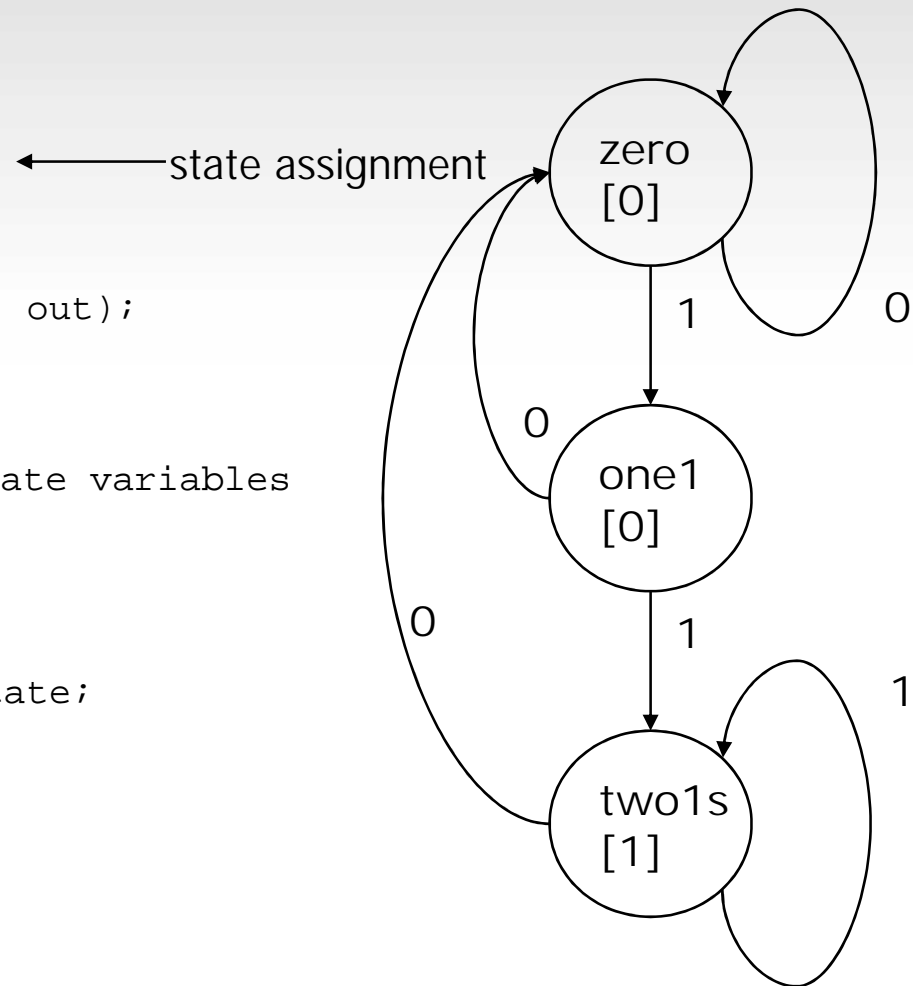
Moore Verilog FSM

- Reduce 1's example

```
`define zero 0  
`define one1 1  
`define twols 2
```

```
module reduce (CLK, reset, in, out);  
  input CLK, reset, in;  
  output out;  
  reg out;  
  reg [1:0] state;      // state variables  
  reg [1:0] next_state;
```

```
  always @(posedge CLK)  
    if (reset) state = `zero;  
    else      state = next_state;
```



Moore Verilog FSM (continued)

```
always @(in or state)
    case (state)
        `zero: // last input was a zero
        begin
            if (in) next_state = `one1;
            else    next_state = `zero;
        end

        `one1: // we've seen one 1
        begin
            if (in) next_state = `twols;
            else    next_state = `zero;
        end

        `twols: // we've seen at least 2 ones
        begin
            if (in) next_state = `twols;
            else    next_state = `zero;
        end
    endcase
```

crucial to include
all signals that are
input to state and
output equations

note that output only
depends on state

```
always @(state)
    case (state)
        `zero: out = 0;
        `one1: out = 0;
        `twols: out = 1;
    endcase
```

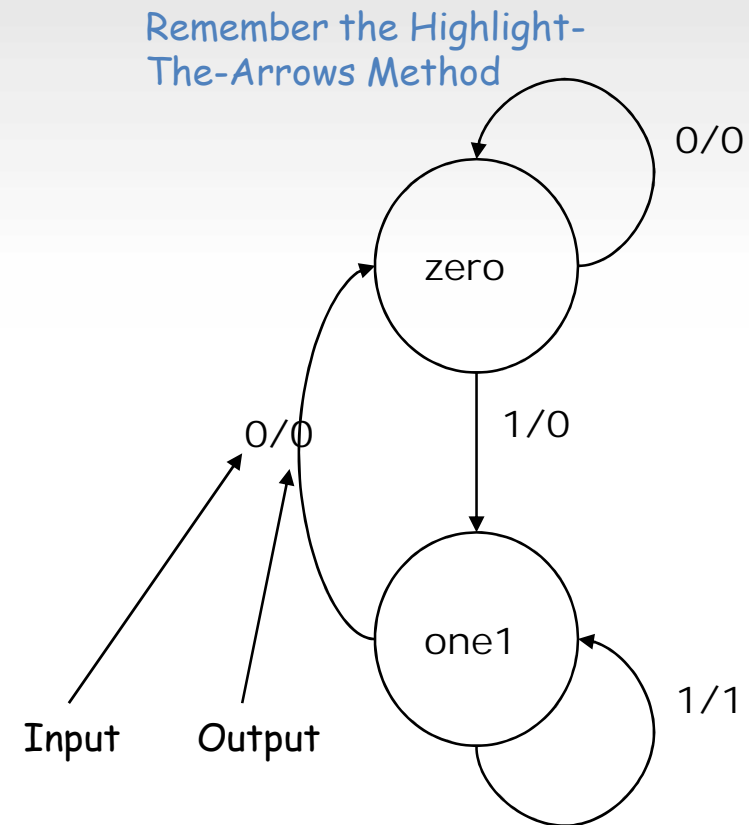
```
endmodule
```

Mealy Verilog FSM

```
module reduce (CLK, reset, in, out);
  input CLK, reset, in;
  output out;
  reg out;
  reg state;           // state variables
  reg next_state;

  always @(posedge CLK)
    if (reset) state = `zero;
    else      state = next_state;

  always @(in or state)
    case (state)
      `zero: // last input was a zero
        begin
          out = 0;
          if (in) next_state = `one;
          else    next_state = `zero;
        end
      `one: // we've seen one 1
        if (in) begin
          next_state = `one; out = 1;
        end else begin
          next_state = `zero; out = 0;
        end
    endcase
endmodule
```



Blocking and Non-Blocking Assignments

- Blocking assignments ($X=A$)
 - completes the assignment before continuing on to next statement
- Non-blocking assignments ($X<=A$)
 - completes in zero time and doesn't change the value of the target until a blocking point (delay/wait) is encountered

- Example: swap

```
always @(posedge CLK)
begin
    temp = B;
    B = A;
    A = temp;
end
```

```
always @(posedge CLK)
begin
    A <= B;
    B <= A;
end
```

RTL Assignment

- Non-blocking assignment is also known as an RTL assignment
 - if used in an always block triggered by a clock edge
 - mimic register-transfer-level semantics – all flip-flops change together

```
// B,C,D all get the value of A
always @(posedge clk)
begin
    B = A;
    C = B;
    D = C;
end
```

```
// this implements a shift register
always @(posedge clk)
begin
    {D, C, B} = {C, B, A};
end
```

```
// implements a shift register too
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```

How to build and test a module

Construct a “test bench” for your design

- Develop your hierarchical system within a module that has input and output ports (called “**design**” here)
- Develop a separate module to generate tests for the module (“**test**”)
- Connect these together within another module (“**testbench**”)

How to build and test a module

```
module testbench ();  
    wire l, m, n;  
  
    design d (l, m, n);  
    test t (l, m);  
  
    initial begin  
        //monitor and display  
        ...  
    end
```

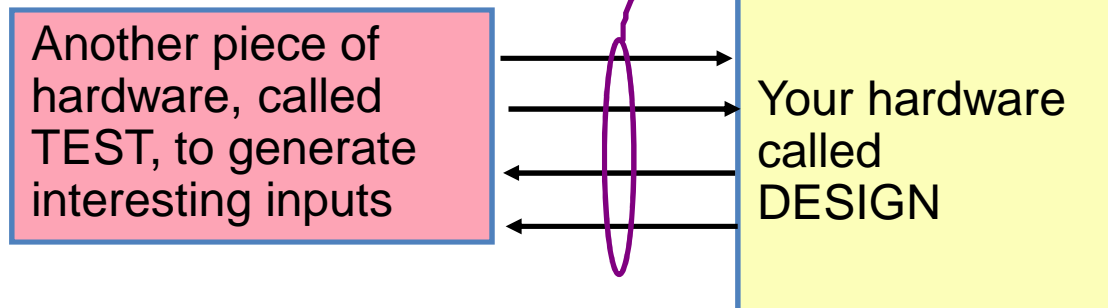
```
module design (a, b, c);  
    input  a, b;  
    output c;  
    ...  
end
```

```
module test (q, r);  
    output q, r;  
  
    initial begin  
        //drive the outputs with signals  
        ...  
    end
```

Another view of this

- 3 chunks of verilog, one for each of:

TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...



Verilog Examples

Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the design

```
module tBench;  
    wire    su, co, a, b;  
  
    halfAdd ad(su, co, a, b);  
    testAdd tb(a, b, su, co);  
endmodule
```

```
module halfAdd (sum, cOut, a, b);  
    output    sum, cOut;  
    input     a, b;  
  
    xor #2    (sum, a, b);  
    and #2    (cOut, a, b);  
endmodule
```

```
module testAdd(a, b, sum, cOut);  
    input     sum, cOut;  
    output    a, b;  
    reg a, b;  
  
    initial begin  
        $monitor ($time,,  
            "a=%b, b=%b, sum=%b, cOut=%b",  
            a, b, sum, cOut);  
        a = 0; b = 0;  
        #10 b = 1;  
        #10 a = 1;  
        #10 b = 0;  
        #10 $finish;  
    end  
endmodule
```

The test module

- It's the test generator
- \$monitor
 - prints its string when executed.
 - after that, the string is printed when one of the listed values changes.
 - only one monitor can be active at any time
 - prints at end of current simulation time

```
module testAdd(a, b, sum, cOut);  
    input    sum, cOut;  
    output   a, b;  
    reg a, b;  
  
    initial begin  
        $monitor ($time,,  
            "a=%b, b=%b, sum=%b, cOut=%b",  
            a, b, sum, cOut);  
        a = 0; b = 0;  
        #10 b = 1;  
        #10 a = 1;  
        #10 b = 0;  
        #10 $finish;  
    end  
endmodule
```

The test module (continued)

- Function of this tester
 - at time zero, print values and set a=b=0
 - after 10 time units, set b=1
 - after another 10, set a=1
 - after another 10 set b=0
 - then another 10 and finish

```
module testAdd(a, b, sum, cOut);  
    input    sum, cOut;  
    output   a, b;  
    reg a, b;  
  
    initial begin  
        $monitor ($time,,  
            "a=%b, b=%b, sum=%b, cOut=%b",  
            a, b, sum, cOut);  
        a = 0; b = 0;  
        #10 b = 1;  
        #10 a = 1;  
        #10 b = 0;  
        #10 $finish;  
    end  
endmodule
```


Other things you can do

- More than modeling hardware
 - \$monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time.
- e.g. ...

- `$monitor ($time,,, "a=%b, b=%b, sum=%b, cOut=%b",a, b, sum, cOut);`

extra commas
print a spaces

%b is binary (also, %h,
%d and others)

- The above will print:
2 a=0, b=0, sum=0, cOut=0<return>

newline
automatically
included

- \$display() — sort of like printf()
 - \$display ("Hello, world — %h", hexvalue)

display contents of data item called
"hexvalue" using hex digits (0-9,A-F)

Parameter

- Parameters are useful because they can be redefined on a module instance basis. That is, each different instance can have different parameter values. This is particularly useful for vector widths.
- For example, the following module implements a shifter:

```
module shift (shiftOut, dataIn, shiftCount);  
    parameter width = 4;  
    output [width-1:0] shiftOut;  
    input [width-1:0] dataIn;  
    input [31:0] shiftCount;  
    assign shiftOut = dataIn << shiftCount;  
endmodule
```
- This module can now be used for shifters of various sizes, simply by changing the width parameter.

Define Parameter Value

- There are **two** ways to change parameter values from their defaults, `defparam` statements and module instance parameter assignment.
 - The `defparam` statement allows you to change a module instance parameter directly from another module. This is usually used as follows:

```
shift sh1 (shiftedVal, inVal, 7); //instantiation
defparam sh1.width = 16; // parameter redefinition
```

- Parameter values can be specified in the module instantiation directly. This is done as follows:

```
shift #(16) sh1 (shiftedVal, inVal, 7);
//instance of 16-bit shift module
```

Task and Function

- Tasks and functions are declared within modules. The declaration may occur anywhere within the module, but it may not be nested within procedural blocks. The declaration does not have to precede the task or function invocation.
- Tasks may only be used in procedural blocks. A task invocation, or task enable as it is called in Verilog, is a statement by itself. It may not be used as an operand in an expression.
- Functions are used as operands in expressions. A function may be used in either a procedural block or a continuous assignment, or indeed, any place where an expression may appear.

Task

- Tasks may have zero or more arguments, and they may be input, output, or inout arguments.

```
task do_read;
input [15:0] addr;
output [7:0] value;
begin
    adbus_reg = addr; // put address out
    adbus_en = 1; // drive address bus
    @(posedge clk) ; // wait for the next clock
    while (~ack)
        @(posedge clk); // wait for ack
    value = data_bus; // take returned value
    adbus_en = 0; // turn off address bus
    count = count + 1; // how many have we done
end
endtask
```

Function

- In contrast to tasks, no time or delay controls are allowed in a function. Function arguments are also *restricted to inputs only*. Output and inout arguments are not allowed. The output of a function is indicated by an assignment to the function name. For example,

```
function [15:0] relocate;  
    input [11:0] addr;  
    input [3:0] relocation_factor;  
begin  
    relocate = addr + (relocation_factor<<12);  
    count = count + 1; // how many have we done  
end  
endfunction
```

- The above function might be used like this:

```
assign absolute_address = relocate(relative_address, rf);
```

System Task

- System tasks are used just like tasks which have been defined with the *task ... endtask* construct. They are distinguished by their first character, which is always a "\$".
- There are many system tasks, but the most common are:
 - `$display`, `$write`, `$strobe`
 - `$monitor`
 - `$readmemh` and `$readmemb`
 - `$stop`
 - `$finish`

Example of System Task

- The `$write` system task is just like `$display`, except that it does not add a newline character to the output string.

- Example:

```
$write ($time, " array: " );  
for (i=0; i<4; i=i+1) write(" %h",  
    array[i]); $write("\n");
```

This would produce the following output:

```
110 array: 5a5114b3 0870261a 0678448d  
4e8a7776
```


System Function

- Likewise, system functions are used just like functions which have been defined with the *function ... endfunction* construct. Their first character is also always a "\$".
- There are many system functions, with the most common being:
 - `$time ($stime)`
 - `$random`
 - `$bitstoreal`

Example of System Function

- The `$time` system function simply returns the current simulation time. Simulation time is a 64-bit unsigned quantity, and that is what `$time` is assumed to be when it is used in an expression.
- `$stime` (short time) is just like `$time`, except that it returns a 32-bit value of time.
- Example:

```
$display ("The current time is %d", $time);
```

```
$display ($time, " now the value of x is %h", x);
```

Conversion Function

\$rtoi(real_value)

Returns a signed integer, truncating the real value.

\$itor(int_val)

Returns the integer converted to a real value.

\$realtobits(real_value)

Returns a 64-bit vector with the bit representation of the real number.

\$bitstoreal(bit_value)

Returns a real value obtained by interpreting the bit_value argument as an IEEE 754 floating point number.

```
module driver (net_r);  
    output net_r;  
    real r;  
    wire [64:1]  
    net_r = $realtobits(r);  
endmodule  
  
module receiver (net_r);  
    input net_r;  
    wire [64:1] net_r;  
    real r;  
    always @(net_r)  
        r = $bitstoreal(net_r);  
endmodule
```

XMR

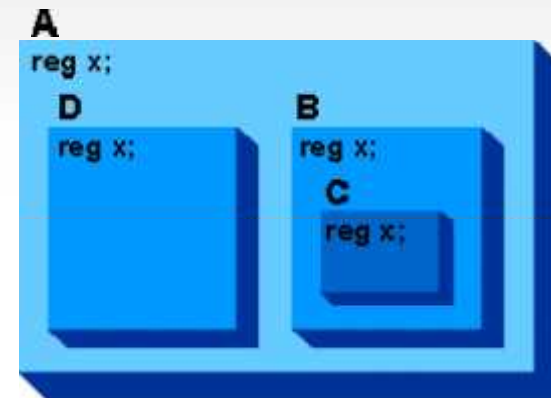
- Verilog has a mechanism for globally referencing nets, registers, events, tasks, and functions called the cross-module reference, or XMR. This is in marked contrast to VHDL, which rejected the concept.
- Cross-module references, or hierarchical references as they are sometimes called, can take several different forms:
 - References to a Different Scope within a Module
 - References between Modules
 - Downward Reference
 - Upward Reference

Hierarchical Module

- There is a static scope within each module definition with which one can locate any identifier. For example, in the following,

```
module A;  
  reg x; // 1  
  ...  
  task B;  
    reg x; // 2  
    begin  
      ...  
      begin: C  
        reg x; // 3  
        ...  
      end  
    end  
  endtask
```

```
  initial  
  begin: D  
    reg x; // 4  
    ...  
  end  
endmodule
```



Reference to Scopes within Module

- there is a module, a task, and two named blocks. There are four distinct registers, each named *x* within its local scope.

register	is contained in	is named
1	module A	A.x
2	module A task B	A.B.x
3	module A task B block C	A.B.C.x
4	module A block D	A.D.x

Verilog-2001 New features

ANSI-style port lists

```
module mux8 (y, a, b, en);  
  output [7:0] y;  
  reg [7:0] y;  
  input [7:0] a, b;  
  input en;
```

Verilog-95

```
module mux8 ( output  
  reg [7:0] y,  
  input wire [7:0] a,  
  input wire [7:0] b,  
  input wire en );  
  function [63:0] alu (  
    input [63:0] a,  
    input [63:0] b,  
    input [7:0] opcode );
```

Verilog-2001

Combined port / data type operator

```
module mux8 (y, a, b, en);  
output [7:0] y  
reg [7:0] y;  
input [7:0] a, b;  
input en;
```

Verilog-95

```
module mux8 (y, a, b, en);  
output reg [7:0] y;  
input wire [7:0] a, b;  
input wire en;
```

Verilog-2001

Module Port Parameter List

```
module adder (sum,  
co,a,b,ci);  
parameter MSB=31,  
           LSB=0;  
output [MSB:LSB] sum;
```

Verilog-95

```
module adder  
#(parameter MSB=31, LSB=0)  
(output reg [MSB:LSB] sum,  
output reg co ,  
input wire [MSB:LSB] a,b,  
Input wire ci);
```

Verilog-2001

Variable declaration with initial value

```
module test;  
reg clock;  
initial  
    clock=0;
```

Verilog-95

```
module test;  
reg clock=0;  
//Initial once at time 0
```

Verilog-2001

Constant Function

```
module ram;  
parameter SIZE=4096;  
parameter ADDR=12;  
input [ADDR-1:0]  
addr_bus;
```

Verilog-95

```
module ram;  
parameter SIZE=4096;  
input [log2(SIZE)-1:0] addr_bus;
```

Verilog-2001

Comma Separated Sensitivity list

```
always @ (a or b or ci)
```

```
.....
```

```
always @(posedge clk or  
negedge rst_n)
```

```
...
```

Verilog-95

```
always @ (a,b,c);
```

```
.....
```

```
always @(posedge clk, negedge rst_n)
```

Verilog-2001

Combinational Logic Sensitivity list

```
always @ (a or b or sel)
case(sel)
  1'b0: y=a;
  1'b1: y=b;
endcase
```

Verilog-95

```
always @ (*);
case(sel)
  1'b0: y=a;
  1'b1: y=b;
endcase
```

Verilog-2001

Vector Part Select

```
reg [63:0] vector1;  
reg [0:63] vecotr2;
```

```
byte = vector1 [31 -: 8]; //vector1[31:24]  
byte = vector1[24 +: 8]; //vector2[31:24]  
byte = vector2 [31 -: 8]; //vector2[24:31]  
byte = vector2[24 +: 8]; //vector2[24:31]
```

Multi-Dimensional Array

```
reg [31:0] array_1D[127:0]; //valid in both verilog 1995 and  
verilog 2001
```

```
reg [31:0] array_2D [127:0][127:0]; //only valid in verilog  
2001
```

```
reg [31:0] data;
```

```
data = array_2D[8][8];
```


Arrays of Net and Real

```
//in Verilog-1995,  
//only 1D arrays of reg, integer and time allowed
```

```
//in Verilog-2001  
wire [31:0] array_1D[127:0];  
real array_2D [127:0][127:0];
```

Arrays Bit and Part Select

```
reg [31:0] ram [0:255];  
reg [7:0] high_byte;  
reg [31:0] temp;  
  
temp = ram[5]  
high_byte=temp[31:24];
```

Verilog-95

```
reg [31:0] ram [0:255];  
reg [7:0] high_byte;  
  
high_byte=temp[5][31:24];
```

Verilog-2001

Power Operator **

```
module ram (...);  
parameter ADDR_WIDTH=12;  
...  
reg [(2**ADDR_WIDTH)-1:0] ram_data;
```

Sized and Typed Parameter

```
module fsm(...);  
parameter IDLE=3'd0,  
          S1  =3'd1;
```

Verilog-95

```
module fsm(...);  
parameter [2:0] IDLE=3'd0,  
          S1  =3'd1;
```

Verilog-2001

Explicit In-Line Parameter Passing

```
module ram (...);  
parameter SYNC=1;  
parameter WIDTH=10;  
parameter SIZE=1024;  
....  
ram mem1 #(1,12,4096)  
(...);
```

Verilog-95

```
module ram (...);  
parameter SYNC=1;  
parameter WIDTH=10;  
parameter SIZE=1024;  
....  
ram mem1  
#(.SIZE(1),.WIDTH(12),.SIZE(  
4096)) (...);  
;
```

Verilog-2001

Fixed Local Parameter

```
module multiplier (a, b, product);  
parameter a_width = 8, b_width = 8;  
localparam product_width = a_width + b_width;
```

parameter: can be redefined by defparam or in_line
parameter redefinition.

localparam: cannot be refined

Generate

- Use for loops to generate any number of instances of:
 - Modules primitives. Procedures, continuous assignment, tasks, functions, variables, nets
- Use if-else and case decision to control what instance are generated
- New reserved words
 - -generate, endgenerate, genvar

Generate Example (1/2)

```
generate
genvar i;
for (i=0; i<=7; i=i+1)
  begin : u
    adder8 add ( sum[(i*8)+:8] , co[i+1],
               a[(i*8)+:8], b[(i+8)+:8], ci[i])
  endgenerated

//u[0].add,u[1].add,...,u[7].add are generated
```


Generate Example (2/2)

```
module multiplier (a, b, product);  
  parameter a_width = 8, b_width = 8;  
  localparam product_width = a_width + b_width;  
  input [a_width-1:0] a;  
  input [b_width-1:0] b;  
  output [product_width-1:0] product;  
  
  generate  
    if ((a_width < 8) || (b_width < 8))  
      CLA_multiplier #(a_width, b_width) u1(a, b, product);  
    else  
      WALLACE_multiplier #(a_width, b_width) u1(a, b, product);  
  endgenerate  
endmodule
```