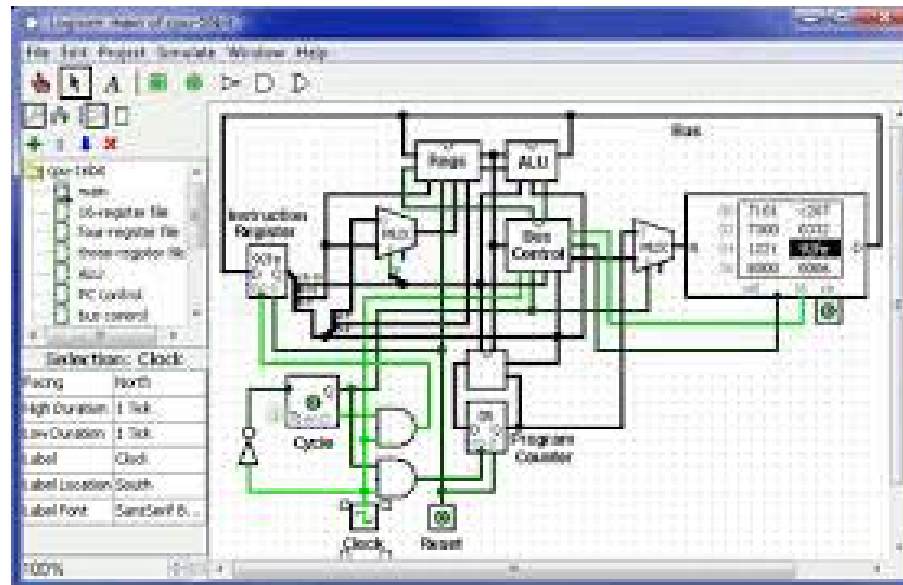
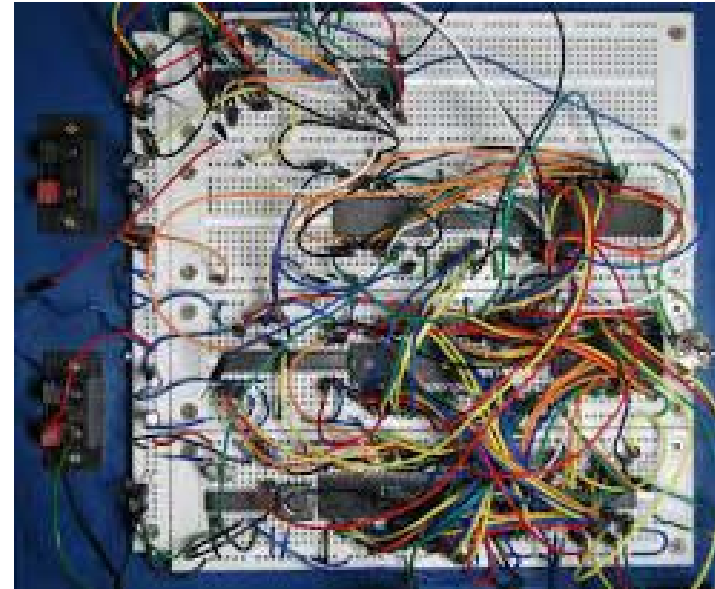
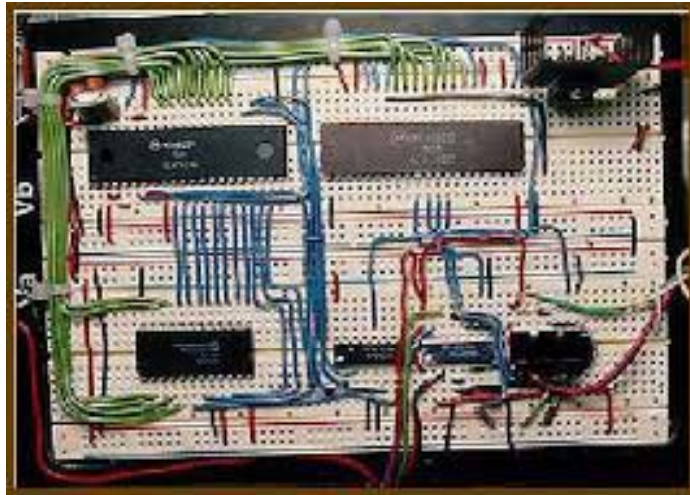


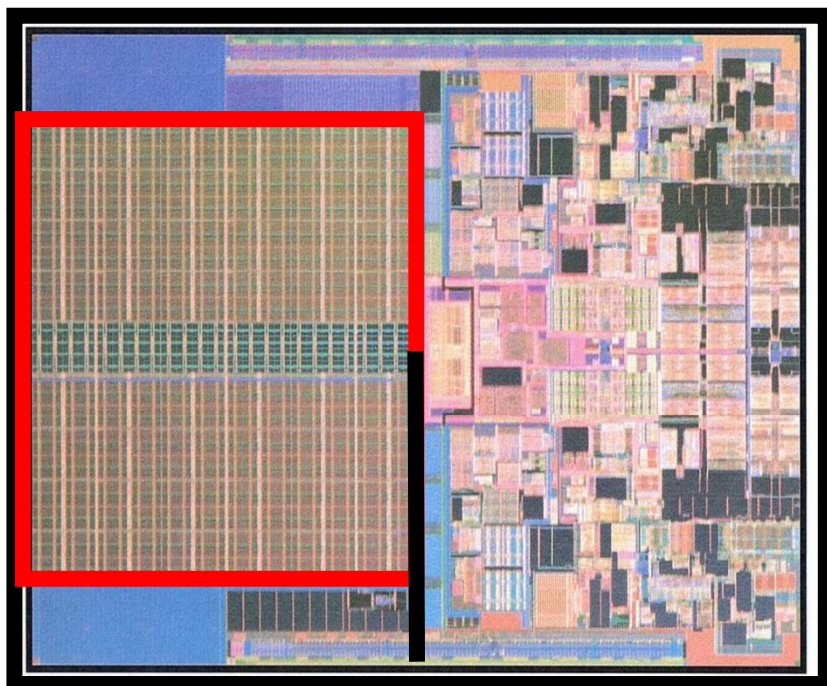
Verilog HDL Basics

So far



Modern System

L2 Cache
6MB

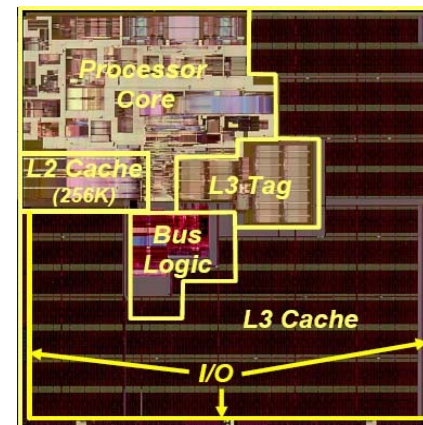


Penryn (dual-core)
45nm Technology

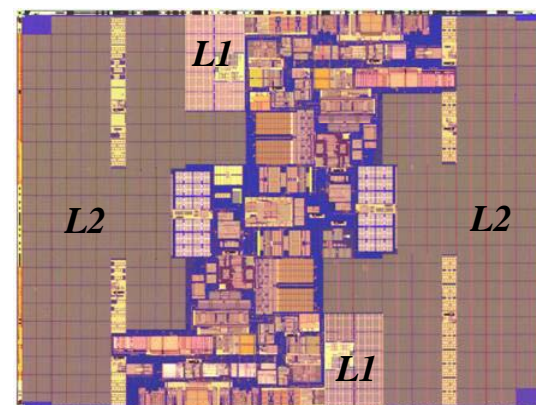
Core 0

Core 1

Montecito* (dual-core)
(L2-24MB, L1-2MB)
90nm Technology



Itanium 2* (L3-9MB)
130nm Technology



Different versions



Silicon Process Technology 1.5 μ 1.0 μ 0.8 μ 0.6 μ 0.35 μ 0.25 μ 0.18 μ 0.13 μ

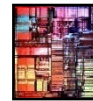
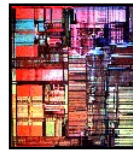
Intel386™ DX
Processor



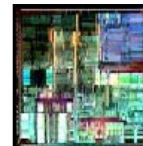
Intel486™ DX
Processor



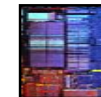
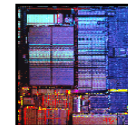
Pentium®
Processor



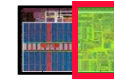
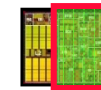
Pentium® Pro
Processor



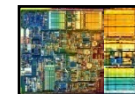
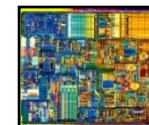
Pentium® II
Processor



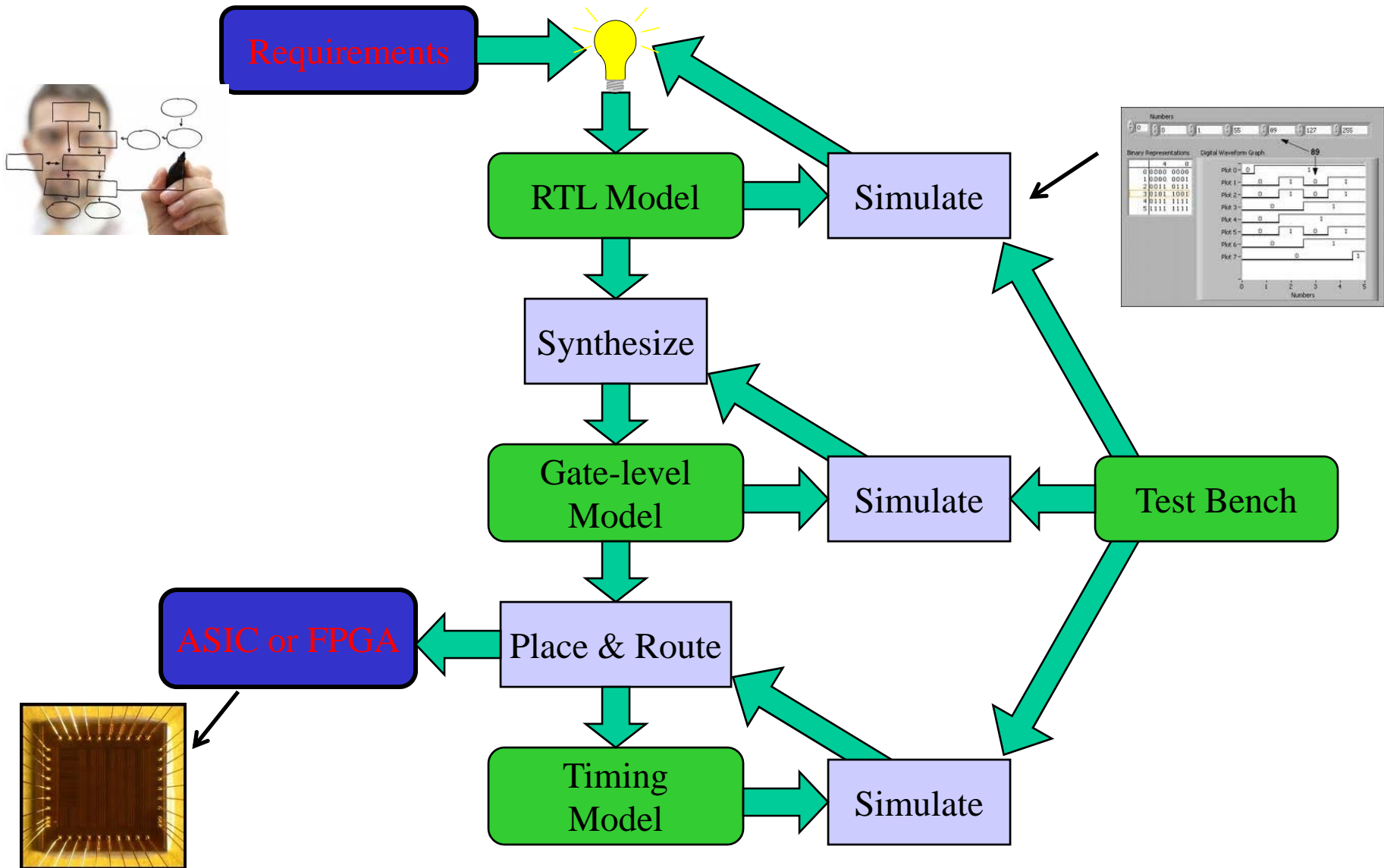
Pentium® III
Processor



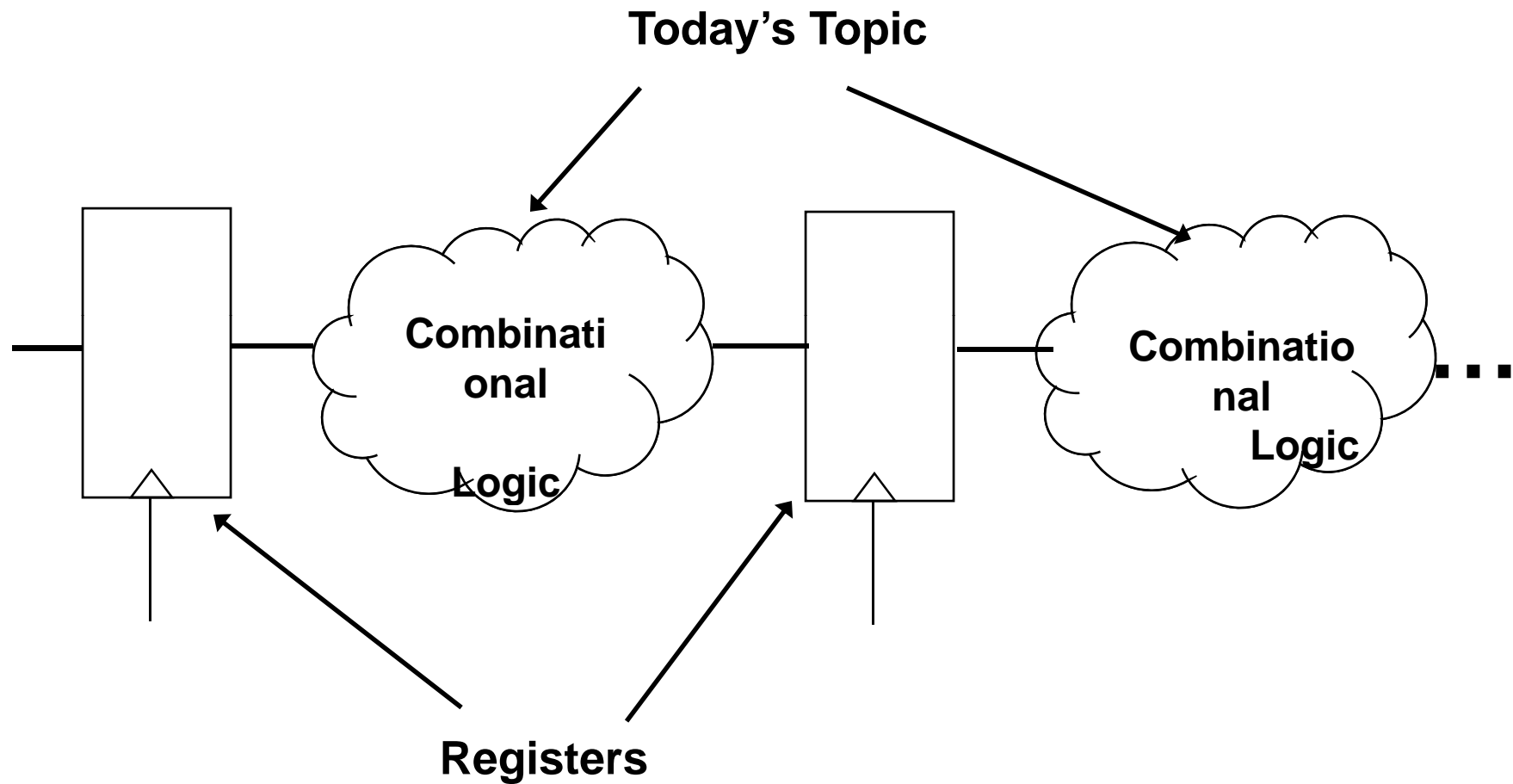
Pentium® 4
Processor



Basic Design Methodology



Register Transfer Level (RTL) Design Description

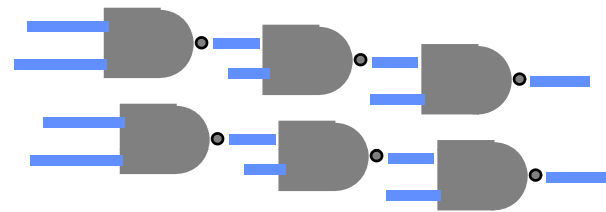


Basic Design Methodology

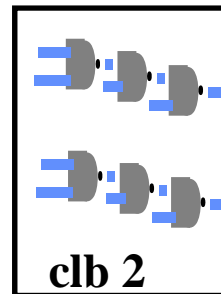
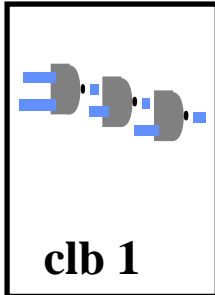
Verilog



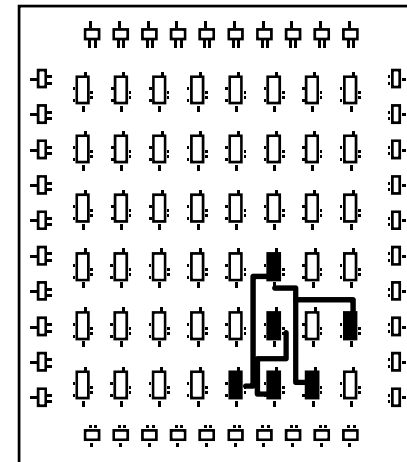
Synthesis



mapping



Place and Route

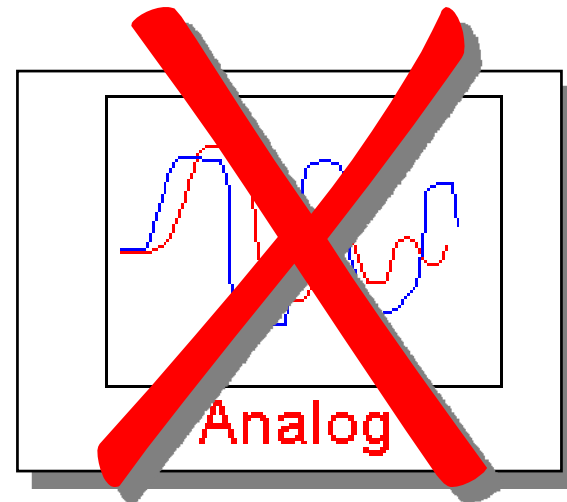
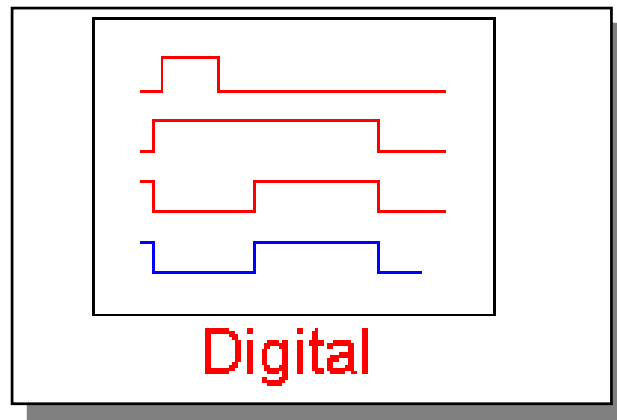


What is Verilog

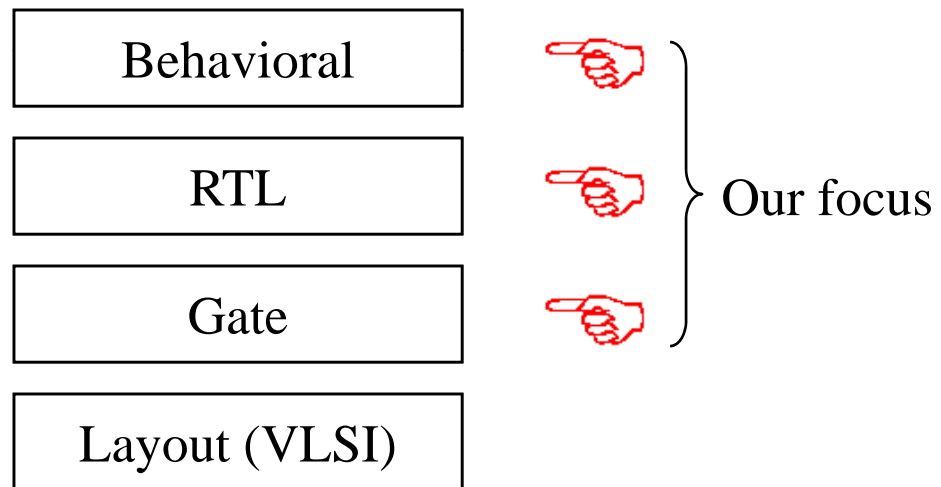
- Hardware Description Language (HDL)
- Developed in 1984
- Standard: IEEE 1364, Dec 1995

Basic Limitation of Verilog

Description of digital systems only

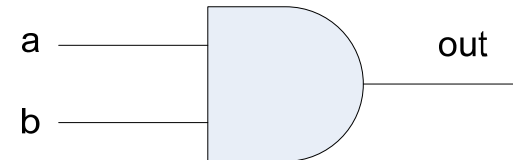


Abstraction Levels in Verilog



VERILOG HDL

- Basic Unit – A module
- Module
 - Describes the functionality of the design
 - States the input and output ports
- Example:



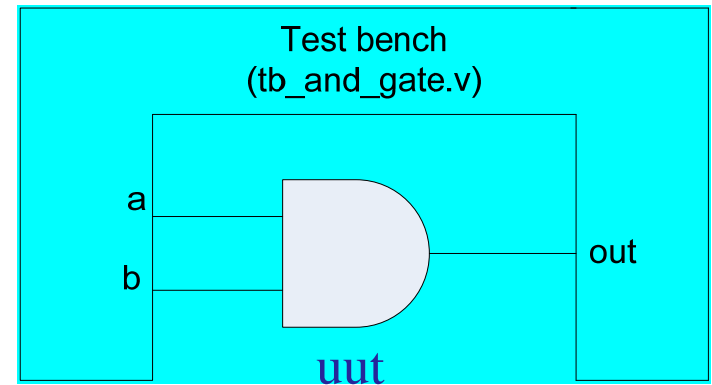
```
module and_gate(out, a,b);  
    output out ;  
    input a, b;  
    assign out = a & b;  
endmodule
```

Testing AND gate

- Instantiate a module

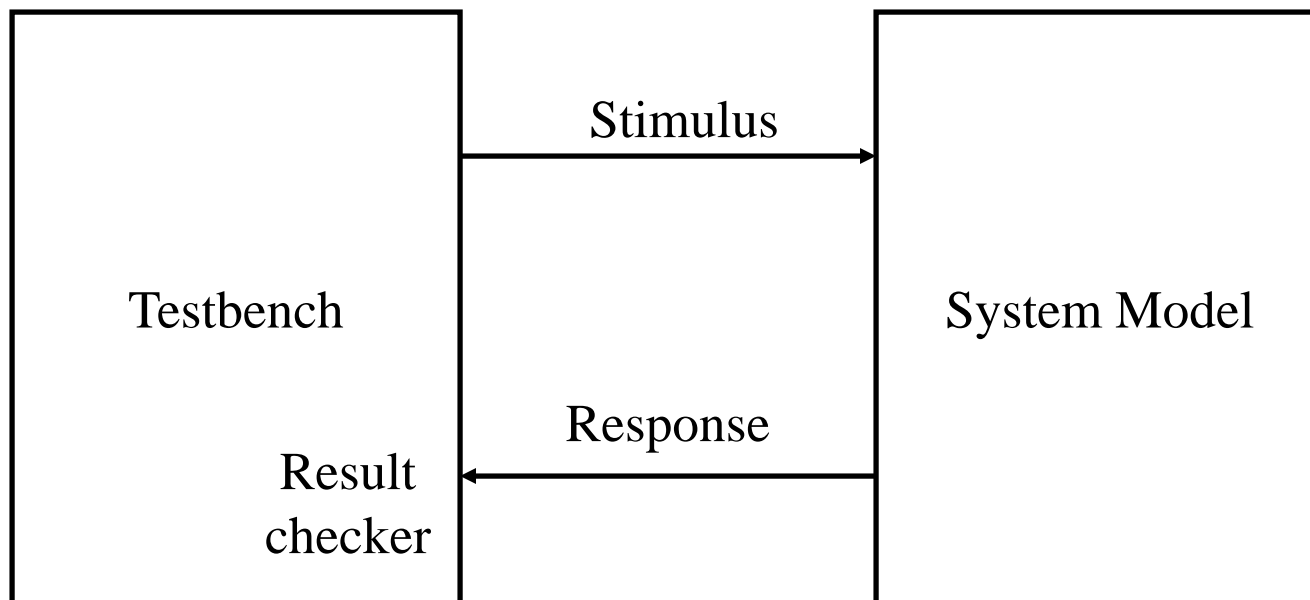
- apply test

```
module tb_and_gate();  
    reg a,b;  
    wire out;  
    and_gate uut (out,a,b); // this instantiates a and gate, uut is a label  
    initial  
    begin  
        a = 1'b0;           // here we apply inputs to the gate  
        b = 1'b0;  
        #10;  
        a = 1'b0;  
        b = 1'b1;  
        #10;  
        a = 1'b1;  
        b = 1'b1;  
        #10;  
        a = 1'b1;  
        b = 1'b0;  
        #10;  
    End  
endmodule
```



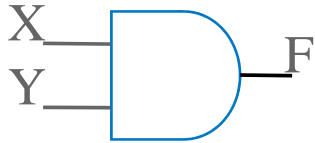
How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously

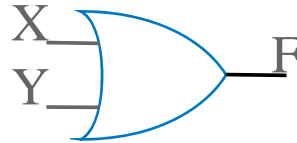


AND/OR/NOT Gates

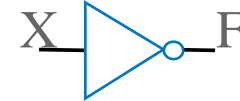
Verilog Modules and Ports



```
module And2(X, Y, F);  
  
    input X, Y;  
    output F;  
    ...
```



```
module Or2(X, Y, F);  
  
    input X, Y;  
    output F;  
    ...
```



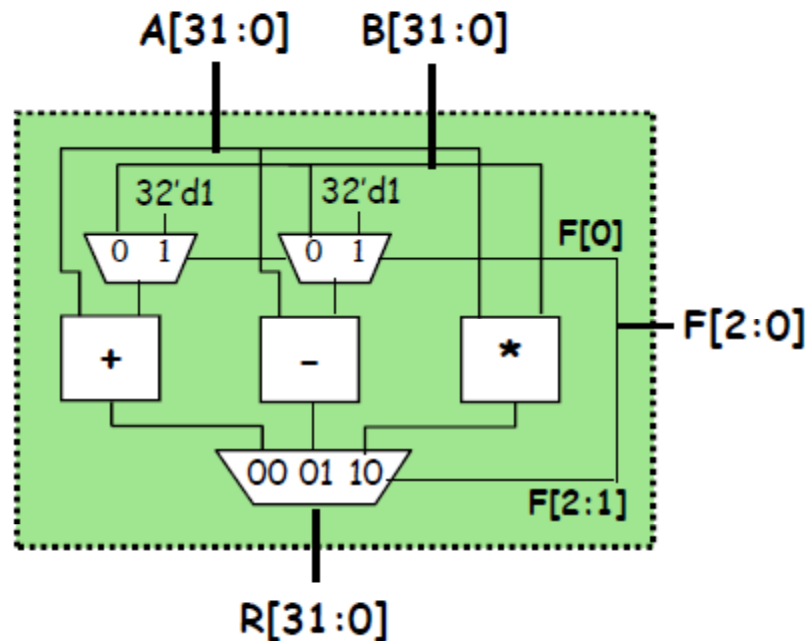
```
module Inv(X, F);  
  
    input X;  
    output F;  
    ...
```

- **module** – Declares a new type of component
 - Named “And2” in first example above
 - Includes list of ports (module's inputs and outputs)
- **input** – List indicating which ports are inputs
- **output** – List indicating which ports are outputs
- Each port is a bit – can have value of 0, 1, or x (unknown value)

Defining Processor ALU in 5 mins

- Modularity is essential to the success of large designs
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU



Function Table

F2	F1	F0	Function
0	0	0	$A + B$
0	0	1	$A + 1$
0	1	0	$A - B$
0	1	1	$A - 1$
1	0	X	$A * B$

ALU- Module Definitions

2-to-1 MUX

```
module mux32two
    (input [31:0] i0,i1,
     input sel,
     output [31:0] out);

    assign out = sel ? i1 : i0;
endmodule
```

32-bit Adder

```
module add32
    (input [31:0] i0,i1,
     output [31:0] sum);

    assign sum = i0 + i1;
endmodule
```

32-bit Subtractor

```
module sub32
    (input [31:0] i0,i1,
     output [31:0] diff);

    assign diff = i0 - i1;
endmodule
```

3-to-1 MUX

```
module mux32three
    (input [31:0] i0,i1,i2,
     input [1:0] sel,
     output reg [31:0] out);

    always @ (i0 or i1 or i2 or sel)
    begin
        case (sel)
            2'b00: out = i0;
            2'b01: out = i1;
            2'b10: out = i2;
            default: out = 32'bx;
        endcase
    end
endmodule
```

16-bit Multiplier

```
module mul16
    (input [15:0] i0,i1,
     output [31:0] prod);

    // this is a magnitude multiplier
    // signed arithmetic later
    assign prod = i0 * i1;

endmodule
```


Top-Level ALU Declaration

- Given submodules:

```

module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);

```

- Declaration of the ALU Module:

```

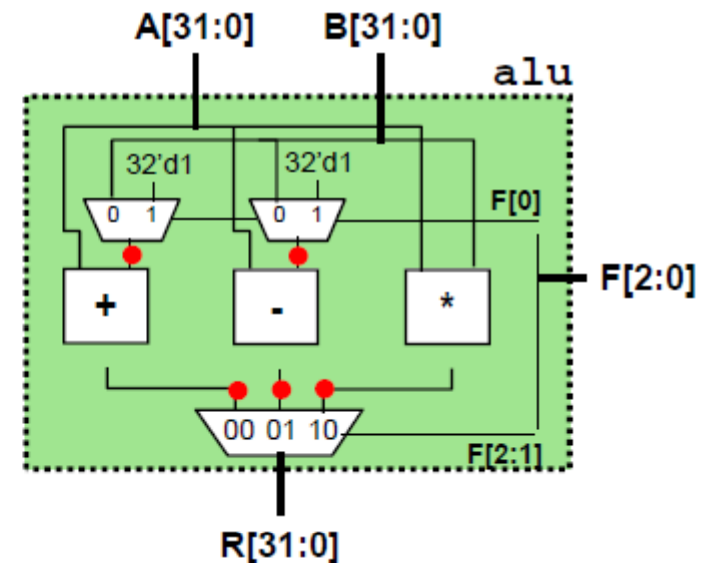
module alu
  (input [31:0] a, b,
   input [2:0] f,
   output [31:0] r);

  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;

  mux32two adder_mux(b, 32'd1, f[0], addmux_out);
  mux32two sub_mux(b, 32'd1, f[0], submux_out);
  add32 our_adder(a, addmux_out, add_out);
  sub32 our_subtractor(a, submux_out, sub_out);
  mul16 our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule

```



intermediate output nodes

module names

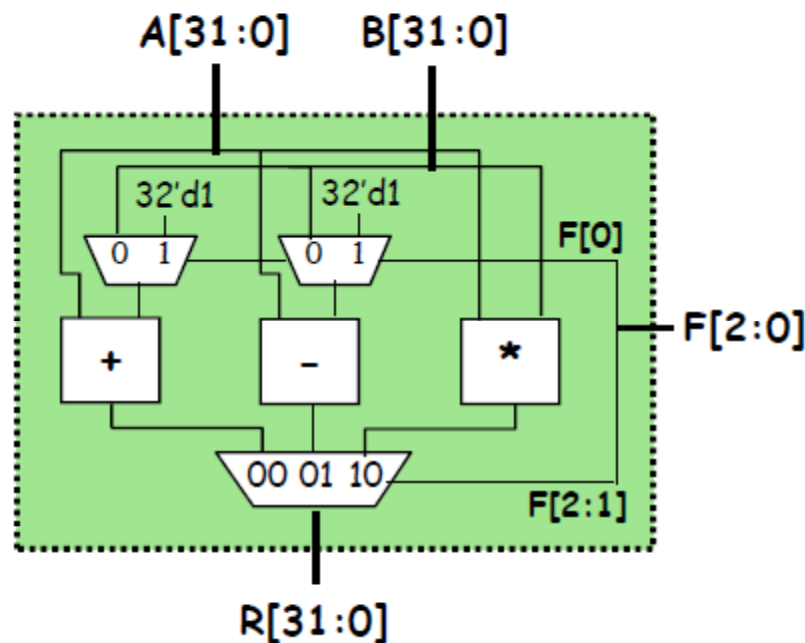
(unique) instance names

corresponding wires/regs in module alu

Defining Processor ALU in 5 mins

- Modularity is essential to the success of large designs
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU



Function Table

F2	F1	F0	Function
0	0	0	$A + B$
0	0	1	$A + 1$
0	1	0	$A - B$
0	1	1	$A - 1$
1	0	X	$A * B$

Adder

Full Adder (1-bit)

```
module full_adder
  (input a, b, cin,
   output reg sum, cout);

  always @(a or b or cin)
  begin
    sum = a ^ b ^ cin;
    cout = (a & b) | (a & cin) | (b & cin);
  end
endmodule
```

Full Adder (4-bit)

```
module full_adder_4bit
  ( input[3:0] a, b,
    input cin,
    output [3:0] sum,
    output cout),
  wire c1, c2, c3;

  // instantiate 1-bit adders
  full_adder FA0(a[0],b[0], cin, sum[0], c1);
  full_adder FA1(a[1],b[1], c1, sum[1], c2);
  full_adder FA2(a[2],b[2], c2, sum[2], c3);
  full_adder FA3(a[3],b[3], c3, sum[3], cout);
endmodule
```

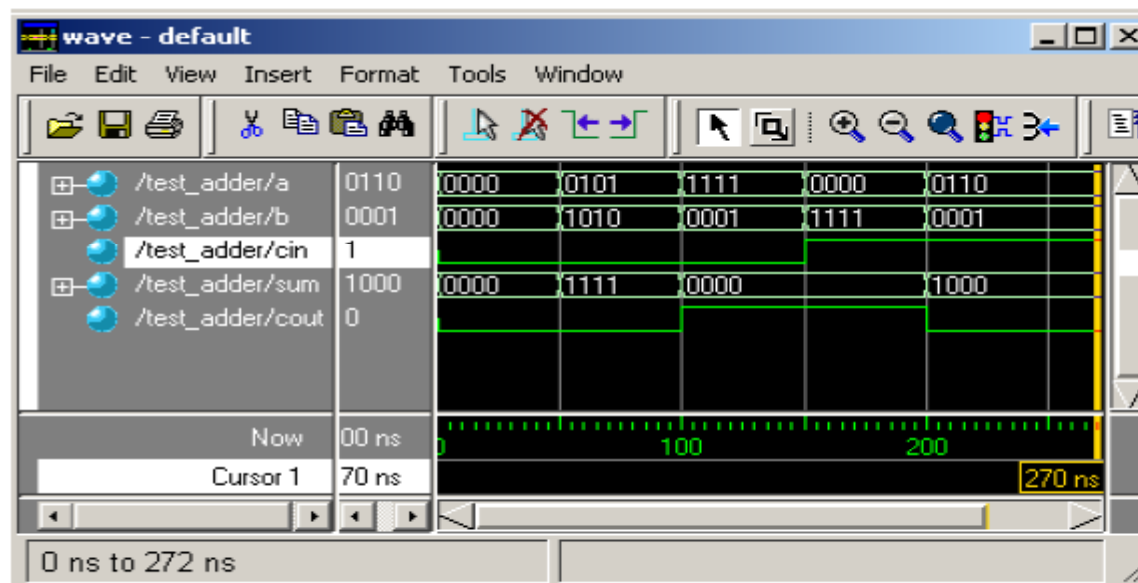
Testbench

```
module test_adder;
  reg [3:0] a, b;
  reg      cin;
  wire [3:0] sum;
  wire      cout;

  full_adder_4bit dut(a, b, cin,
                     sum, cout);

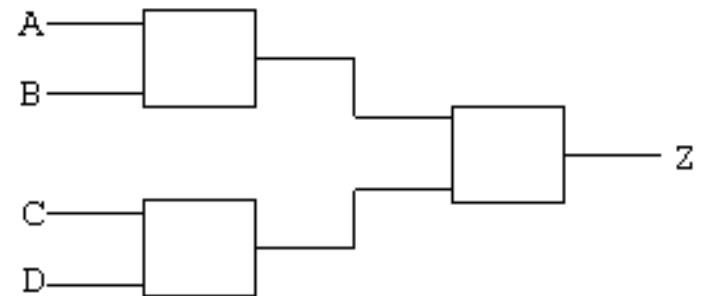
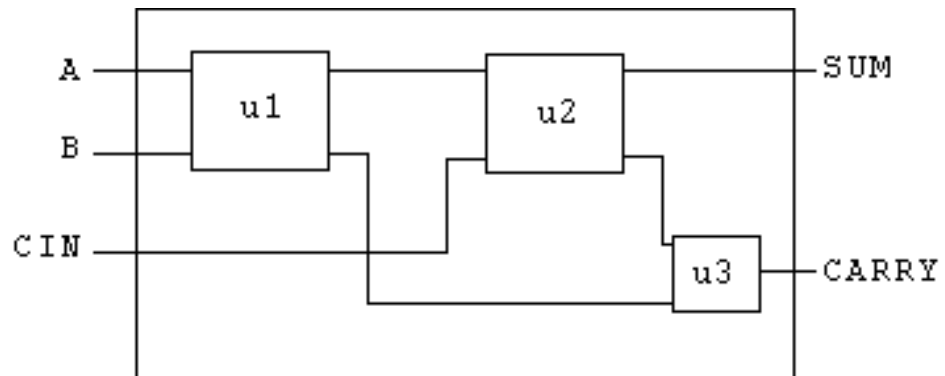
  initial
  begin
    a = 4'b0000;
    b = 4'b0000;
    cin = 1'b0;
    #50;
    a = 4'b0101;
    b = 4'b1010;
    // sum = 1111, cout = 0
    #50;
    a = 4'b1111;
    b = 4'b0001;
    // sum = 0000, cout = 1
    #50;
    a = 4'b0000;
    b = 4'b1111;
    cin = 1'b1;
    // sum = 0000, cout = 1
    #50;
    a = 4'b0110;
    b = 4'b0001;
    // sum = 1000, cout = 0
  end // initial begin
endmodule // test_adder
```

ModelSim Simulation



Main Language Concepts

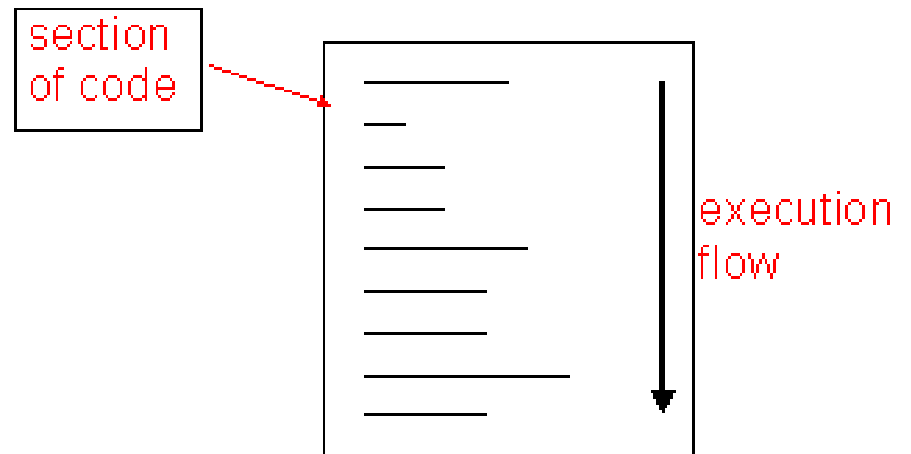
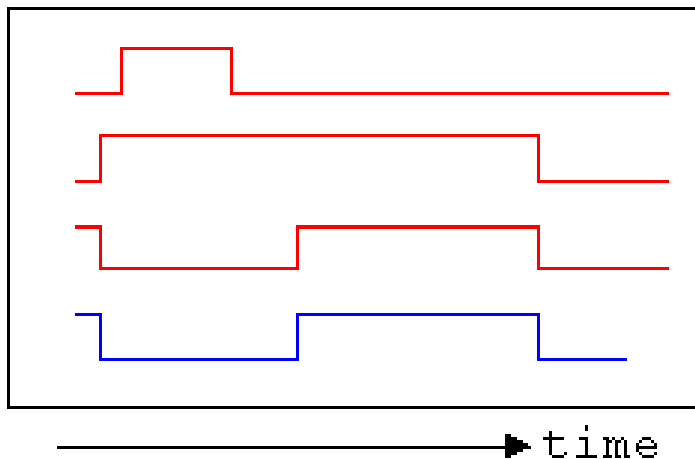
- Concurrency



- Structure




Main Language Concepts

- 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 X
 - \$my_identifier X
 - _myidentifier\$ 
- Case sensitivity
 - myid ≠ Myid

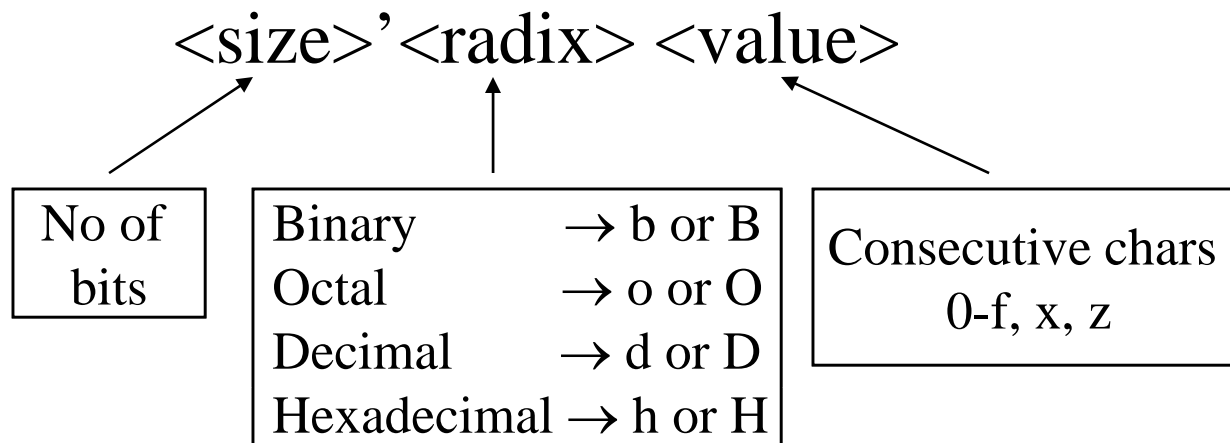
Comments

- `// 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)



- `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 -examples

Constant values can be specified with a specific width and radix:

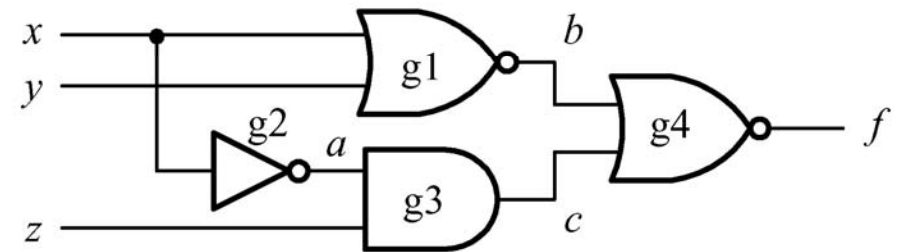
```
123          // default: decimal radix, unspecified width
'd123        // 'd = decimal radix
'h7B         // 'h = hex radix
'o173        // 'o = octal radix
'b111_1011   // 'b = binary radix, "_" are ignored
'hxx         // can include X, Z or ? in non-decimal constants
16'd5        // 16-bit constant 'b0000_0000_0000_0101
11'h1X?      // 11-bit constant 'b001_XXXX_ZZZZ
```

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too). You can specify a signed constant as follows:

```
8'shFF       // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent **it's usually best to explicitly specify the width and radix.**

Module basic_gates



```
module basic_gates (x, y, z, f) ;  
    input  x, y, z;  
    output f ;  
    wire a, b, c;  
    // Structural modeling  
    nor g1 (b, x, y);  
    not g2 (a, x);  
    and g3 (c, a, z);  
    nor g4 (f, b, c);  
endmodule
```

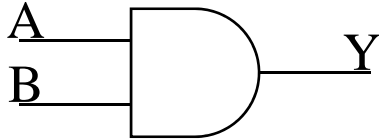
Numbers in Verilog (iii)

- 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
 - $15 = \langle \text{size} \rangle' d 15$

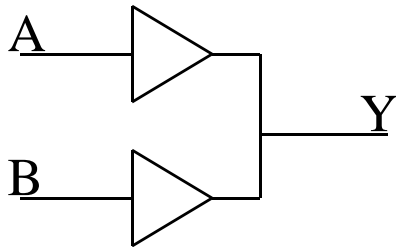
Nets (i)

- 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;
```

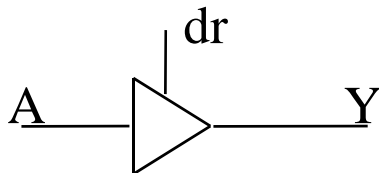


```
wand Y; // declaration
assign Y = A;
assign Y = B;
```

		A	
Y		0	1
		<hr/>	
B	0	0	0
	1	0	1

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

		A	
Y		0	1
		<hr/>	
B	0	0	1
	1	1	1



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

Two Main Components of Verilog

- Concurrent, event-triggered processes (behavioral)
 - *Initial* and *Always* blocks
 - can perform standard data manipulation tasks (assignment, if-then, case)
 - Processes run until they delay for a period of time or wait for a triggering event
- Structure (Plumbing)
 - Verilog program build from modules with I/O interfaces
 - Modules may contain instances of other modules
 - Modules contain local signals, etc.
 - Module configuration is static and all run concurrently

Two Main Data Types

- Nets represent connections between things
 - Do not hold their value
 - Take their value from a driver such as a gate or other module
 - Cannot be assigned in an *initial* or *always* block
- Regs represent data storage
 - Behave exactly like memory in a computer
 - Hold their value until explicitly assigned in an *initial* or *always* block
 - Can be used to model latches, flip-flops, etc., but do not correspond exactly

		always
initial		begin
begin		statements ...
statements ...		end
end		

end

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

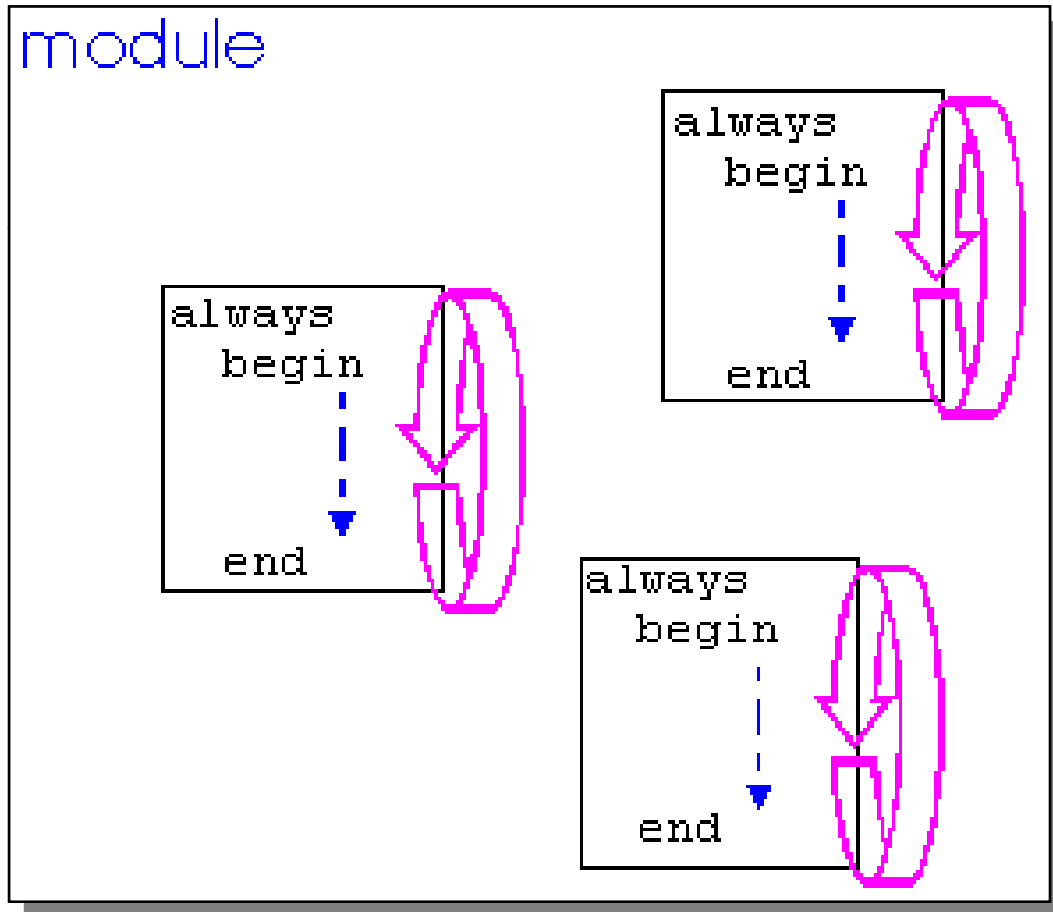
Runs when simulation starts
Restarts when control reaches the end
Good for modeling/specifying hardware

initial	c
c	statement
c	...
c	...
c	...
c	...
c	...

always	c
c	statement
c	...
c	...
c	...
c	...
c	...

“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
```

Initial and Always

- Run until they encounter a delay

```
initial begin
```

```
    #10 a = 1; b = 0;
```

```
    #10 a = 0; b = 1;
```

```
end
```

- or a wait for an event

```
always @(posedge clk)
```

```
    q = d;
```

```
always
```

```
begin wait(i); a = 0;
```

```
    wait(~i); a = 1;
```

```
end
```

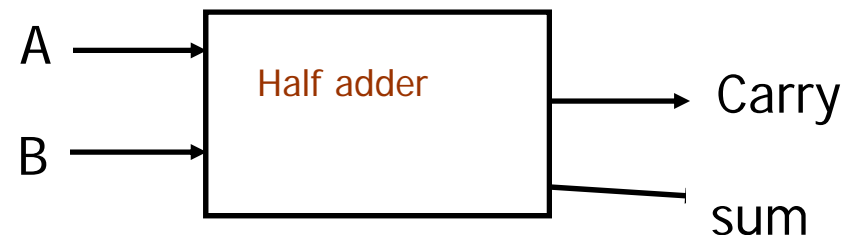
Description Styles

- General definition

```
module module_name ( port_list );  
    port declarations;  
    ...  
    variable declaration;  
    ...  
    description of behavior  
endmodule
```

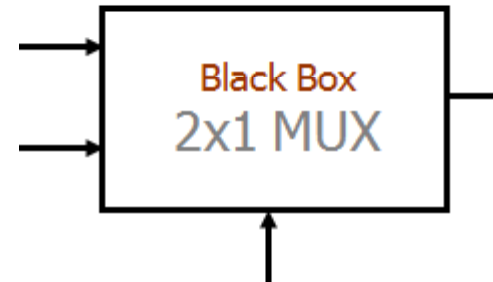
□ Example

```
module HalfAdder (A, B, Sum Carry);  
    input A, B;  
    output Sum, Carry;  
    assign Sum = A ^ B;  
    // ^ denotes XOR  
    assign Carry = A & B;  
    // & denotes AND  
endmodule
```



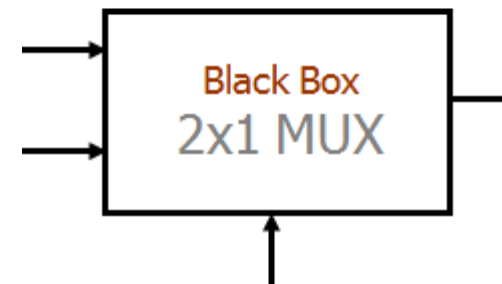
Mux: Description Styles (cont.)

```
module mux1( select, d, q );  
  
    input[1:0]  select;  
    input[3:0]  d;  
    output      q;  
  
    wire        q;  
    wire[1:0]   select;  
    wire[3:0]   d;  
  
    assign q = d[select];  
  
endmodule
```



Description Styles (cont.)

```
module mux2( select, d, q );  
  
    input[1:0] select;  
    input[3:0] d;  
    output      q;  
  
    reg          q;  
    wire[1:0]    select;  
    wire[3:0]    d;  
  
    always @(d or select)  
        q = d[select];  
  
endmodule
```



Description Styles (cont.)

```
module mux3( select, d, q );

input[1:0] select;
input[3:0] d;
output q;

reg q;
wire[1:0] select;
wire[3:0] d;

always @( select or d )
begin
    if( select == 0)
        q = d[0];

    if( select == 1)
        q = d[1];

    if( select == 2)
        q = d[2];

    if( select == 3)
        q = d[3];
end

endmodule
```

Mux: Description Styles (cont.)

```
module mux4( select, d, q );

input[1:0] select;
input[3:0] d;
output      q;

reg        q;
wire[1:0] select;
wire[3:0] d;

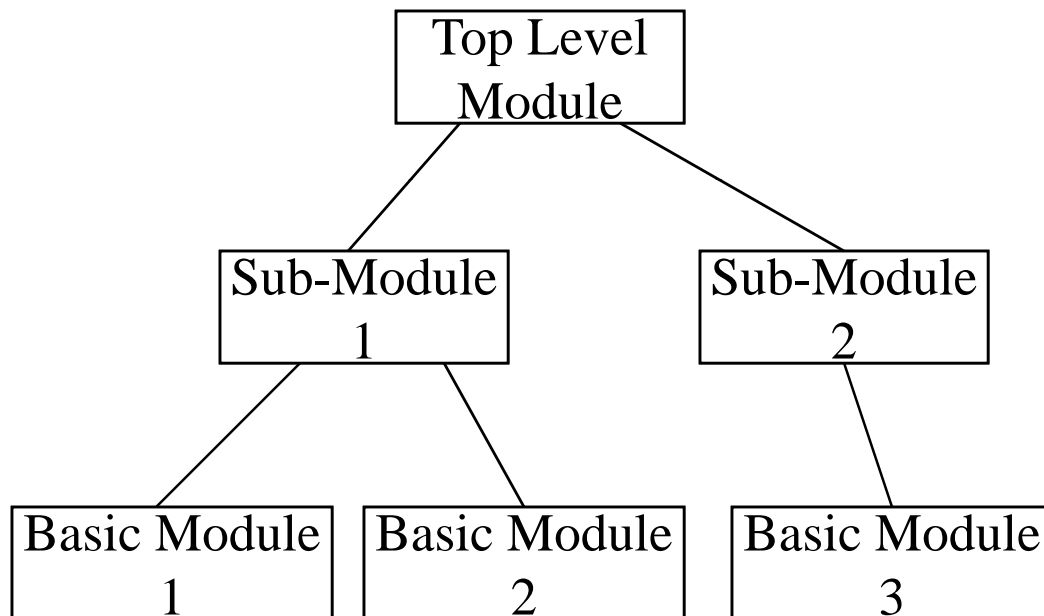
always @( select or d )
begin
    case( select )
        0 : q = d[0];
        1 : q = d[1];
        2 : q = d[2];
        3 : q = d[3];
    endcase
end

endmodule
```

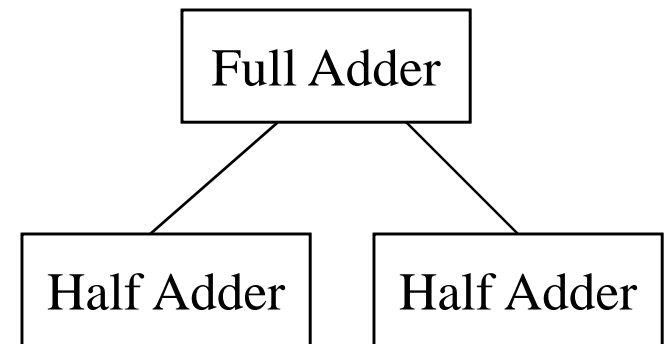
Mux: Description Styles (cont.)

```
module mux5( select, d, q );  
  
  input[1:0] select;  
  input[3:0] d;  
  output q;  
  
  wire q;  
  wire[1:0] select;  
  wire[3:0] d;  
  
  assign q = ( select == 0 )? d[0] : ( select == 1 )? d[1] : ( select == 2 )? d[2] : d[3];  
  
endmodule
```

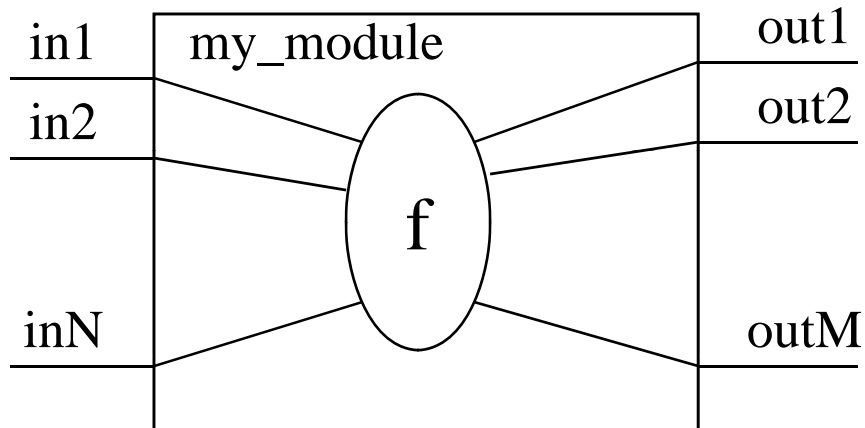
Hierarchical Design



E.g.



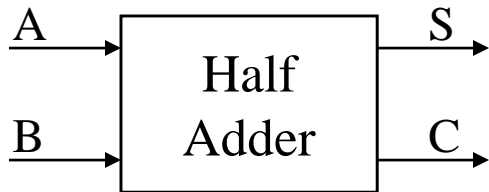
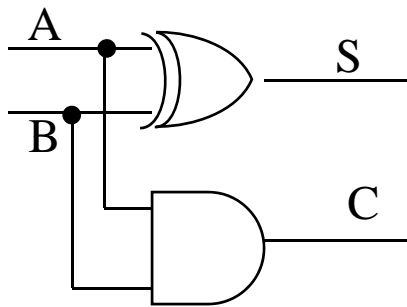
Module



```
module my_module(out1, ..., inN);  
    output out1, ..., outM;  
    input in1, ..., inN;  
  
    .. // declarations  
    .. // description of f (maybe  
    .. // sequential)  
  
endmodule
```

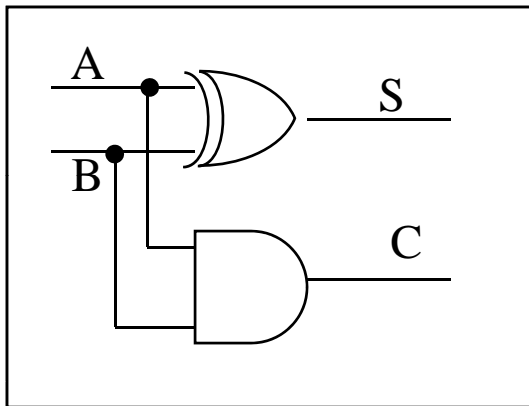
Everything you write in Verilog must be inside a module
exception: compiler directives

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: Half Adder, 2nd Implementation



Assuming:

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

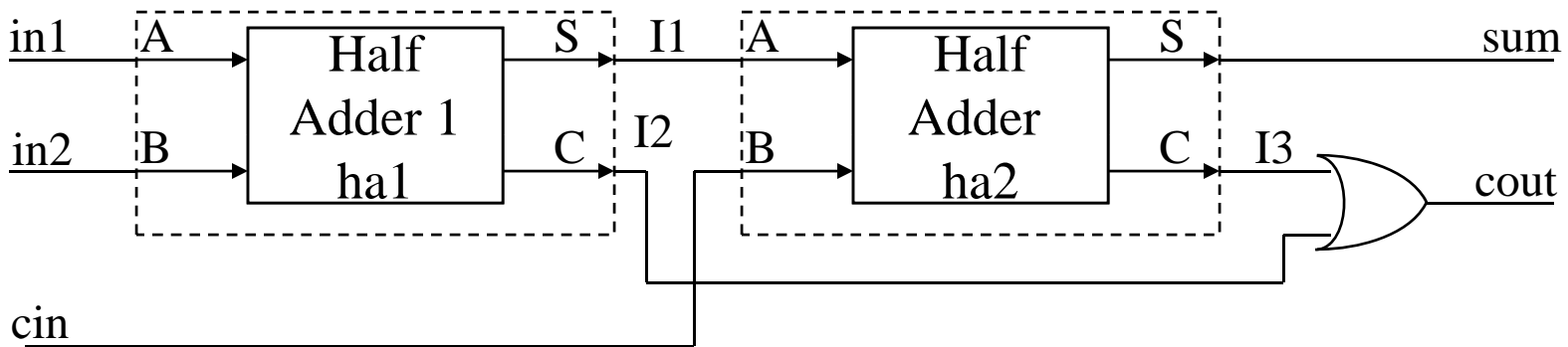
```
module half_adder(S, C, A, B);  
  output S, C;  
  input A, B;
```

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

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

```
endmodule
```


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);
```

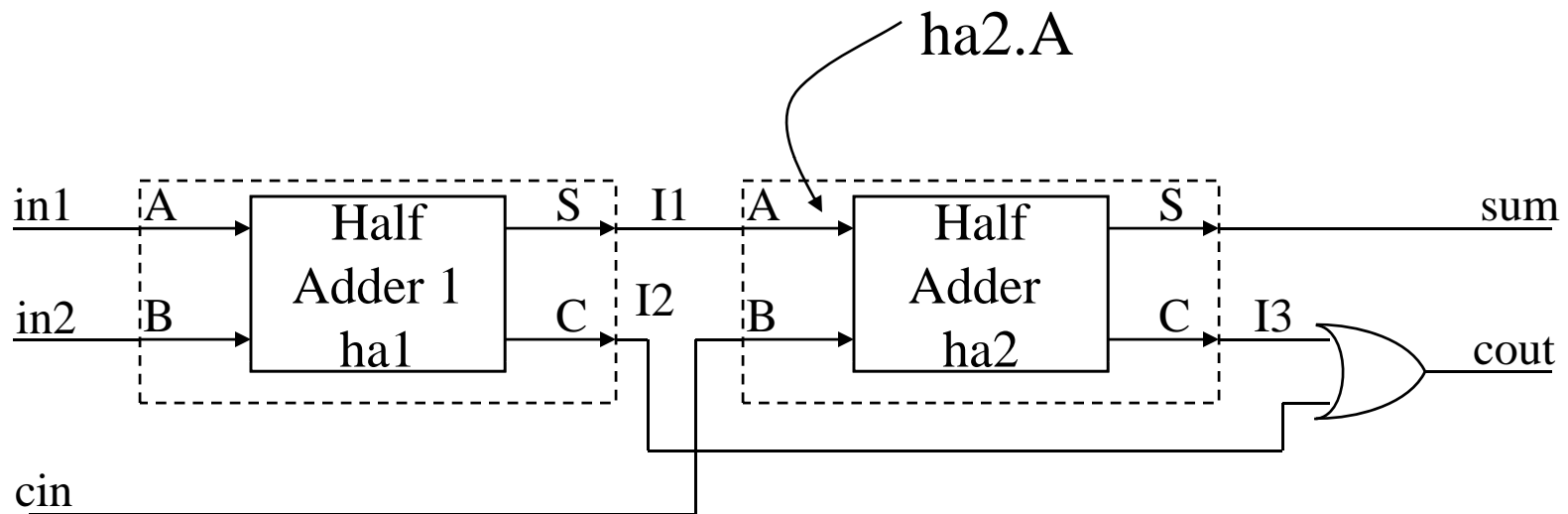
```
    assign cout = I2 || I3;
```

```
endmodule
```

Module
name

Instance
name

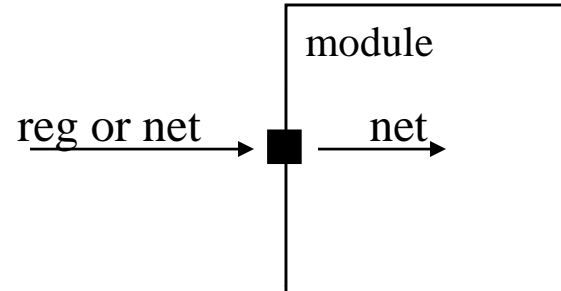
Hierarchical Names



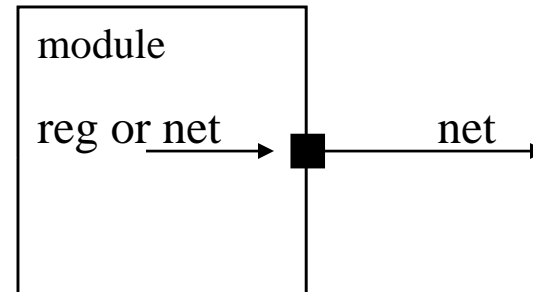
Remember to use instance names,
not module names

Port Assignments

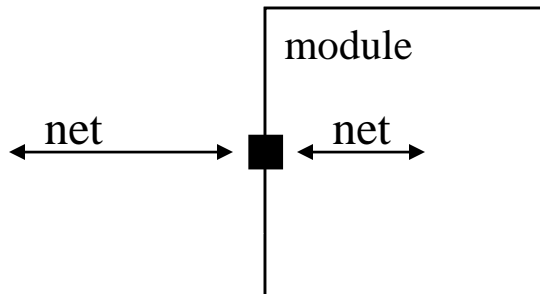
- Inputs



- Outputs



- Inouts



Integer Arithmetic

- Verilog's built-in arithmetic makes a 32-bit adder easy:

```
module add32
    (input[31:0] a, b,
     output[31:0] sum);


    assign sum = a + b;
endmodule
```

- A 32-bit adder with carry-in and carry-out:

```
module add32_carry
    (input[31:0] a,b,
     input cin,
     output[31:0] sum,
     output cout);

    assign {cout, sum} = a + b + cin;
endmodule
```

concatenation



Examples-Arithmetic

```
module Arithmetic (A, B, Y1, Y2, Y3, Y4, Y5);  
  
    input [2:0] A, B;  
    output [3:0] Y1;  
    output [4:0] Y3;  
    output [2:0] Y2, Y4, Y5;  
    reg [3:0] Y1;  
    reg [4:0] Y3;  
    reg [2:0] Y2, Y4, Y5;  
  
    always @(A or B)  
    begin  
        Y1=A+B;//addition  
        Y2=A-B;//subtraction  
        Y3=A*B;//multiplicaton  
        Y4=A/B;//division  
        Y5=A%B;//modulus of A divided by B  
    end  
endmodule
```



Examples- Relational

```
module Relational (A, B, Y1, Y2, Y3, Y4);  
    input [2:0] A, B;  
    output Y1, Y2, Y3, Y4;  
    reg Y1, Y2, Y3, Y4;  
  
    always @(A or B)  
    begin  
        Y1=A<B;//less than  
        Y2=A<=B;//less than or equal to  
        Y3=A>B;//greater than  
        if (A>B)  
            Y4=1;  
        else  
            Y4=0;  
    end  
endmodule
```

Examples-Equality

```
module Equality (A, B, Y1, Y2, Y3);  
    input [2:0] A, B;  
    output Y1, Y2;  
    output [2:0] Y3;  
    reg Y1, Y2;  
    reg [2:0] Y3;  
  
    always @(A or B)  
    begin  
        Y1=A==B;//Y1=1 if A equivalent to B  
        Y2=A!=B;//Y2=1 if A not equivalent to B  
        if (A==B)//parenthesis needed  
            Y3=A;  
        else  
            Y3=B;  
    end  
endmodule
```

Examples-Bitwise

```
module Bitwise (A, B, Y);  
  
    input [6:0] A;  
    input [5:0] B;  
    output [6:0] Y;  
    reg [6:0] Y;  
  
    always @(A or B)  
    begin  
        Y(0)=A(0)&B(0); //binary AND  
        Y(1)=A(1)|B(1); //binary OR  
        Y(2)=!(A(2)&B(2)); //negated AND  
        Y(3)=!(A(3)|B(3)); //negated OR  
        Y(4)=A(4)^B(4); //binary XOR  
        Y(5)=A(5)~^B(5); //binary XNOR  
        Y(6)=!A(6); //unary negation  
    end  
endmodule
```


Examples- Shift

```
module Shift (A, Y1, Y2);  
  
    input [7:0] A;  
    output [7:0] Y1, Y2;  
    parameter B=3; reg [7:0] Y1, Y2;  
  
    always @(A)  
  
begin  
  
        Y1=A<<B; //logical shift left  
        Y2=A>>B; //logical shift right  
  
end  
  
endmodule
```

Examples- Concatenation

```
module Concatenation (A, B, Y);  
  
    input [2:0] A, B;  
    output [14:0] Y;  
    parameter C=3'b011;  
    reg [14:0] Y;  
  
    always @(A or B)  
    begin  
        Y={A, B, (2{C}), 3'b110};  
    end  
endmodule
```

Examples- Reduction

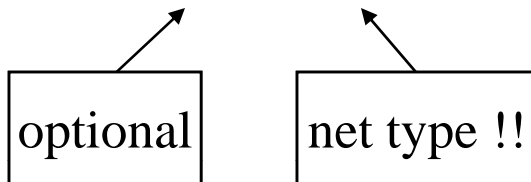
```
module Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);  
  
    input [3:0] A;  
    output Y1, Y2, Y3, Y4, Y5, Y6;  
    reg Y1, Y2, Y3, Y4, Y5, Y6;  
    always @(A)  
    begin  
        Y1=&A; //reduction AND  
        Y2=|A; //reduction OR  
        Y3=~&A; //reduction NAND  
        Y4=~|A; //reduction NOR  
        Y5 ^=A; //reduction XOR  
        Y6=~^A; //reduction XNOR  
    end  
endmodule
```

Continuous Assignments

a closer look

- 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

Four-valued Data

- Verilog's nets and registers hold four-valued data
- 0, 1
 - Obvious
- Z
 - Output of an undriven tri-state driver
- X
 - Models when the simulator can't decide the value
 - Initial state of registers
 - When a wire is being driven to 0 and 1 simultaneously

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*

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;
```

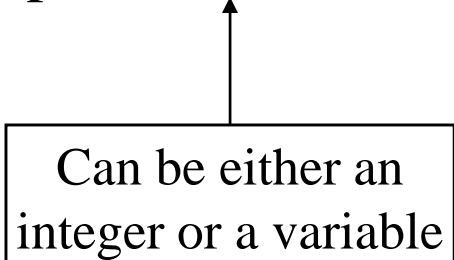
```
initial  
    Y = 0;
```

```
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;
```

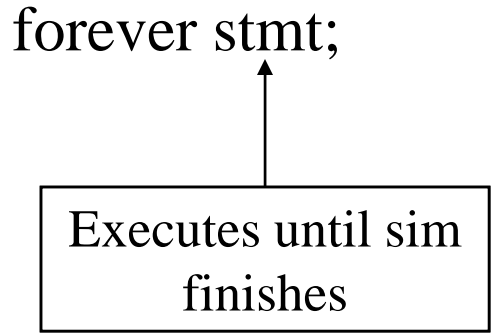
```
initial  
    Y = 0;
```

```
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;
```

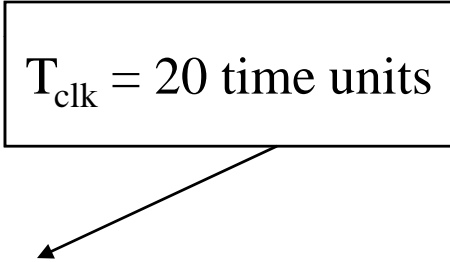
```
end
```

```
other_module1 o1(clk, ..);
```

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

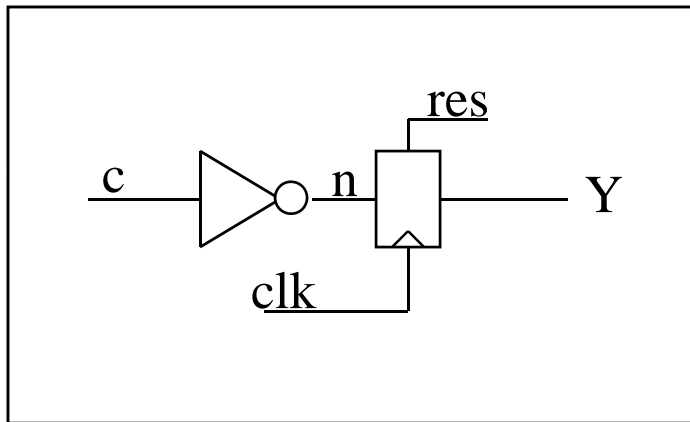
```
endmodule
```

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

```
endmodule
```


System Tasks

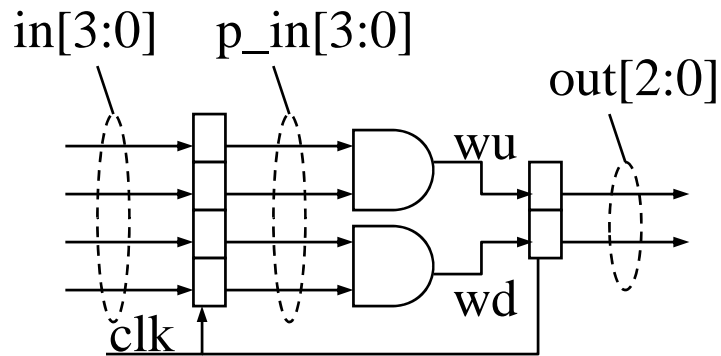
Always written inside procedures

- `$display("..", arg2, arg3, ..);` → much like `printf()`, displays formatted string in std output when encountered
- `$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

Parameters



A. Implementation 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. 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
```

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

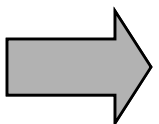
```
reg var[1:10] [1:100]; // WRONG!!
```

- Arrays don't work for the Real data type

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


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*

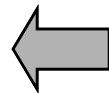
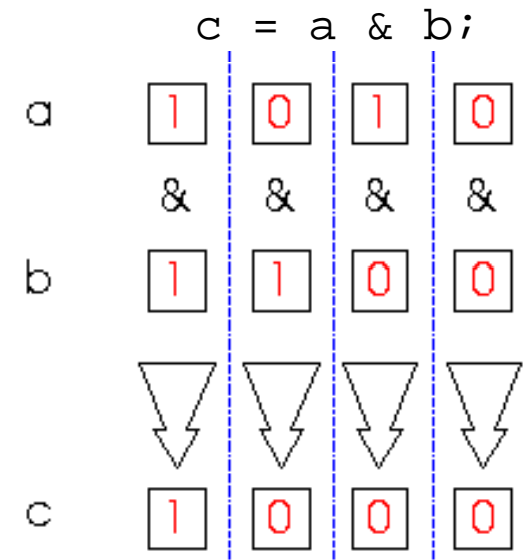
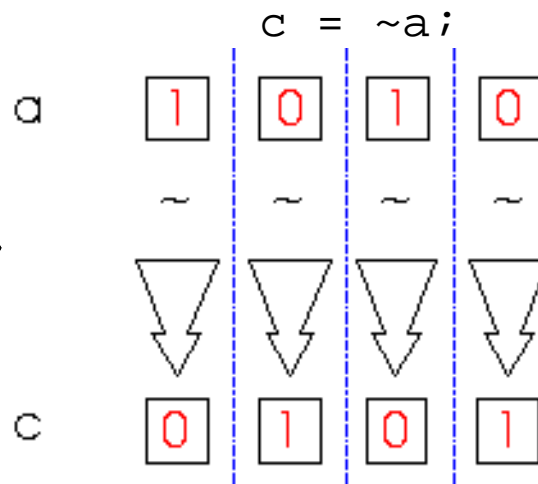
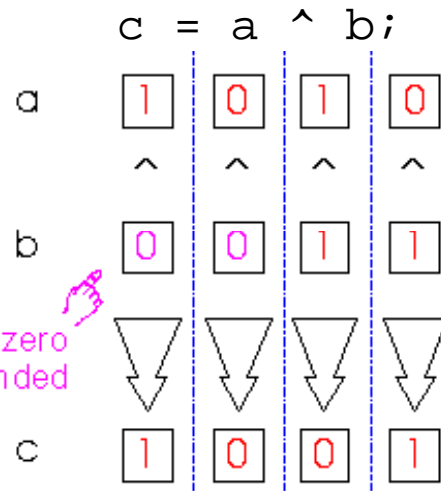
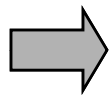
<code>A = 6;</code>		<code>A && B → 1 && 0 → 0</code>	
<code>B = 0;</code>		<code>A !B → 1 1 → 1</code>	
<code>C = x;</code>		<code>C B → x 0 → x</code>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">but <code>C&&B=0</code></div>

Bitwise Operators (i)

- $\&$ \rightarrow bitwise AND
- $|$ \rightarrow bitwise OR
- \sim \rightarrow bitwise NOT
- \wedge \rightarrow bitwise XOR
- $\sim \wedge$ or $\wedge \sim$ \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

- $\&$ \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

- $\{op1, op2, ..\} \rightarrow$ 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

$$1 > 0 \quad \rightarrow 1$$

$$'b1x1 <= 0 \quad \rightarrow x$$

$$10 < z \quad \rightarrow x$$

Equality Operators

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

– `4'b 1z0x == 4'b 1z0x \rightarrow x`

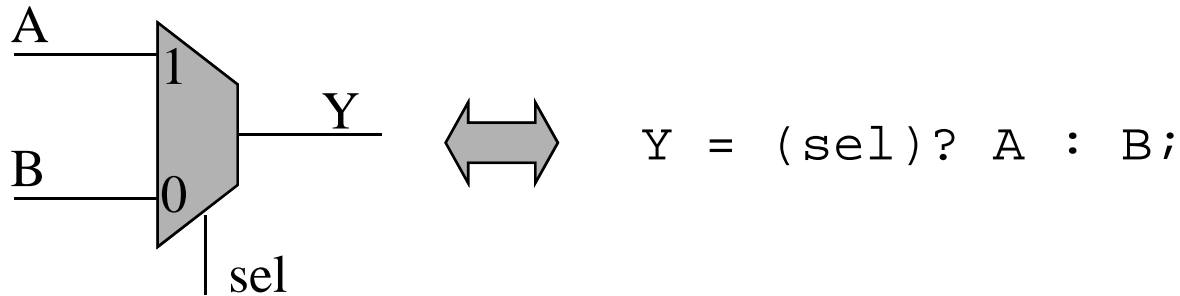
– `4'b 1z0x != 4'b 1z0x \rightarrow x`

– `4'b 1z0x === 4'b 1z0x \rightarrow 1`

– `4'b 1z0x !== 4'b 1z0x \rightarrow 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;
```

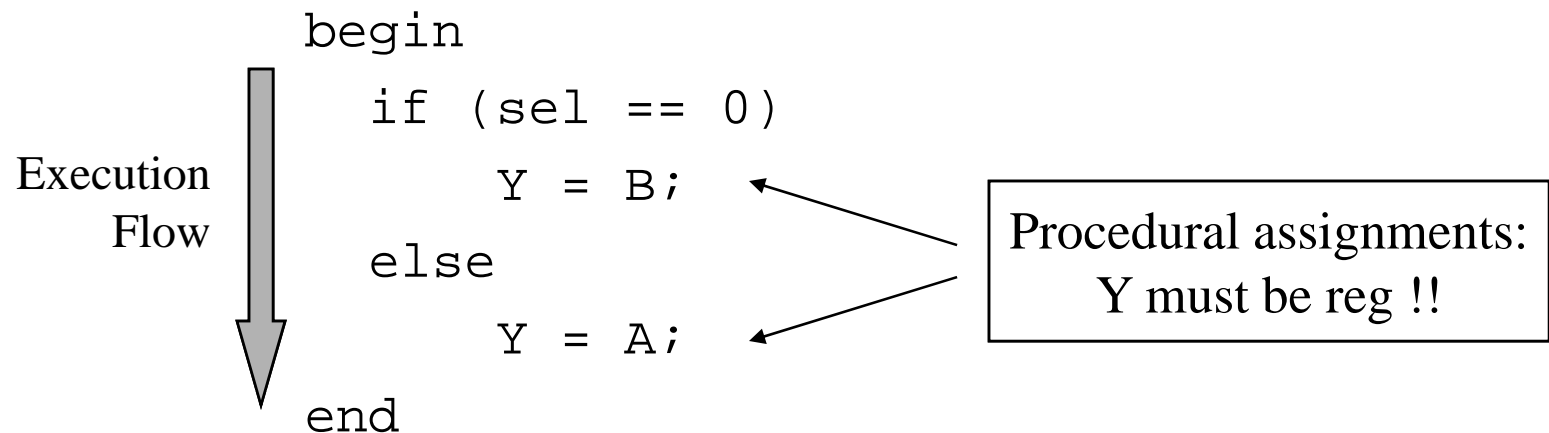
```
..
```

```
intA = -12/3;      // evaluates to -4 (no base spec)
```

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

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)

“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
```

Events (ii)

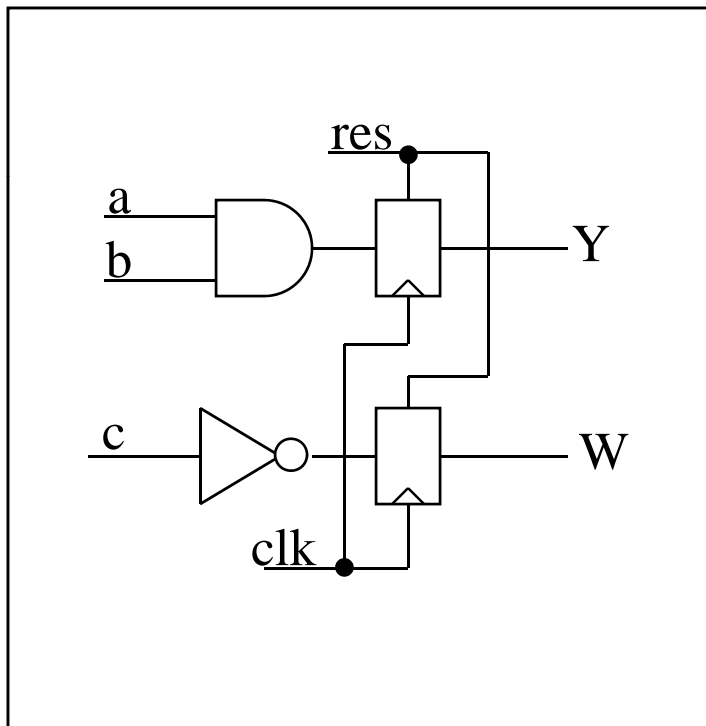
- wait (expr)

```
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

