Verilog 簡介 (上)

陳泓烈(CIC)

Hotline : (03)5773693 ext.885

Hot mail : hotline@cic.narl.org.tw

Outline

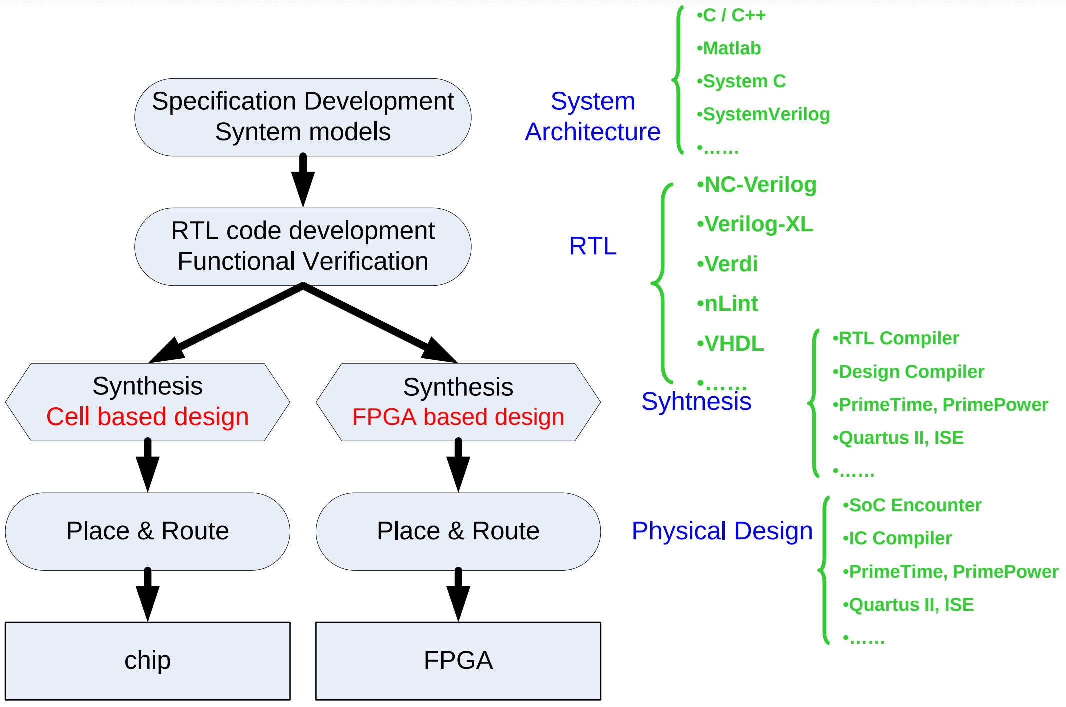
- ✓ Section 1. Introduction to design flow
- ✓ Section 2. Basic Description of Verilog
  - ➢ Module
  - ➢ Port
  - ➢ Datatype
  - ➢ Integer Value
  - ➢ Lexical Convention
- ✓ Operator
- ✓ Gate-Level Modeling
- ✓ Test Fixture

Introduction to Design Flow

- ✓ Cell-based IC
  - Use pre-designed logic cells (known as standard cells) and micro cells (e.g. microcontroller)
  - Designers save time, money, and reduce risk
  - Each standard cell can be optimized individually
  - Custom blocks can be embedded
  - Can use FPGA design flow for prototyp

- ✓ Full-Custom Design Flow
  - Designed for some special-case Ics
  - Required cells/Ips are not available
  - Existing cell libraries are not fast/small/low-power enough
  - Technology migration (mixed-mode design)
  - Demand long design cycle

HDL Design Flow



Specification Development
Syntem models

RTL code development
Functional Verification

Synthesis
Cell based design

Synthesis
FPGA based design

Place & Route

Place & Route

chip

FPGA

System Architecture
- •C / C++
- •Matlab
- •System C
- •SystemVerilog
- •……

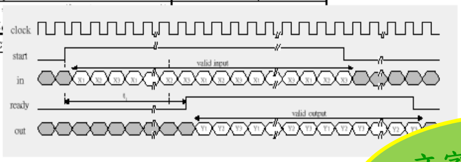
RTL
- •NC-Verilog
- •Verilog-XL
- •Verdi
- •nLint
- •VHDL
- •……

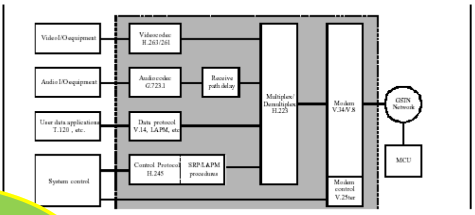
Syhtnesis
- •RTL Compiler
- •Design Compiler
- •PrimeTime, PrimePower
- •Quartus II, ISE
- •……

Physical Design
- •SoC Encounter
- •IC Compiler
- •PrimeTime, PrimePower
- •Quartus II, ISE
- •……

From RTL Design to Physical Layout

腳位名稱		描述	Drive Strength/Output Load
clk	輸入	系統時脈	assume infinite
reset	輸入	系統重置訊號，high active	1 ns/pf
din	輸入	每個clock cycle輸入一個16-bit 正整數	1 ns/pf
ready	輸出	reset為1時 出前的半個	
dout	輸出	每個clock c	

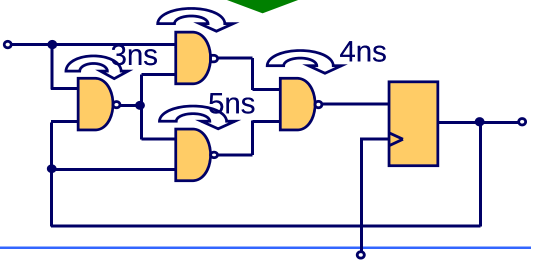


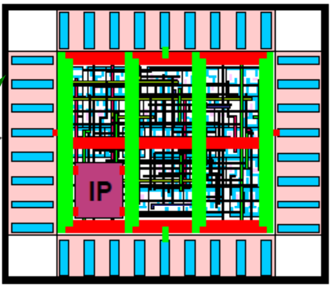


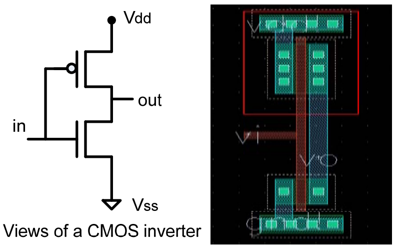
文字⇒電路架構

1. partition
2. wordlength

```
module XXX(clk, reset, din, ready, dout);
        ……
        Verilog Code
        ……
endmodule
```







Views of a CMOS inverter

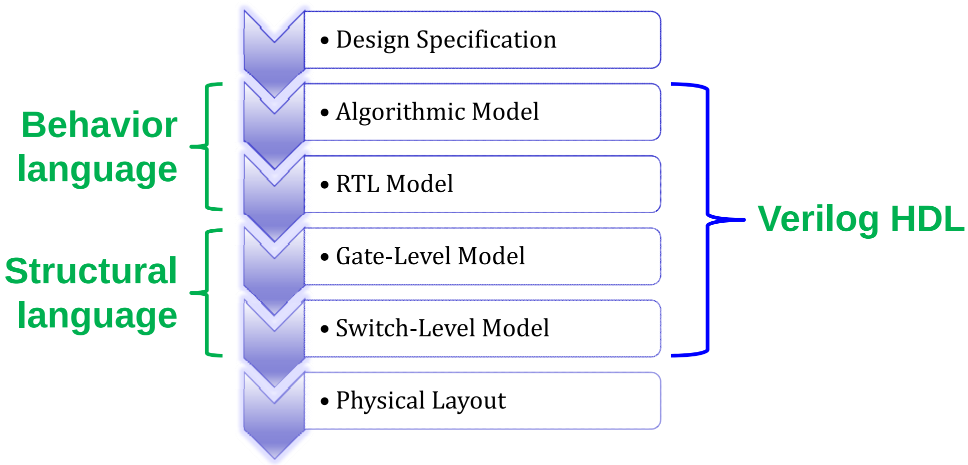
What Is a Hardware Description Language?

A Hardware Description Language (HDL) is a high-level programming language with special constructs used to model the function of hardware logic circuits.

- ✓ Express **concurrency**
- ✓ The language constructs describe circuits with two forms :
  - ➢ Structural descriptions :
    - Connections of components. Nearly one-to-one correspondence to with schematic diagram.
  - ➢ Behavioral descriptions :
    - Use high-level constructs (similar to conventional programming) to describe the circuit function.

Design Method

✓ Typical Design Flow



- Design Specification
- Algorithmic Model
- RTL Model
- Gate-Level Model
- Switch-Level Model
- Physical Layout

Behavior language

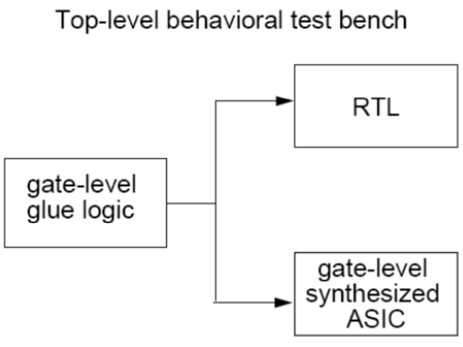
Structural language

Verilog HDL

✓ Cadence/Verilog-XL, Cadence/NC-Verilog, and Synopsys/VCS, (Mentor/ModelSim) are simulators that read Verilog HDL and simulate the description to reflect the behavior of real hardware.

One Language

✓ Mixing gate-level and RTL in a model description, and driving that composite model with a behavioral test bench, is common.



```
module FA_MIX(A, B, CIN, SUM, COUT);
    input A, B, CIN;
    output SUM, COUT;
    reg COUT;
    reg T1, T2, T3;

    wire S1;

    xor X1(S1, A, B); // gate instantiation

    always @ (A or B or CIN) // Always block
    begin
        T1 = A & CIN;
        T2 = B & CIN;
        T3 = A & B;
        COUT = (T1 | T2 | T3);
    end
    assign SUM = S1 ^ CIN; // Continuous assignment
endmodule
```

Structural level

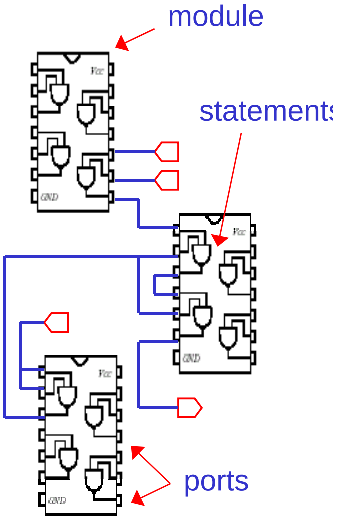
Behavioral level

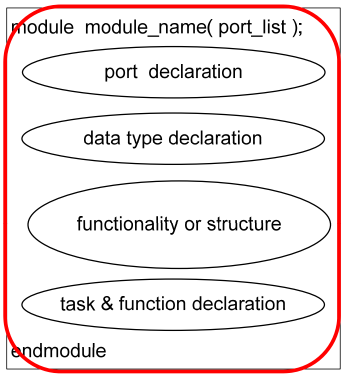
Module

The Verilog Module

- ✓ Modules are the basic building blocks in the design hierarchy.
- ✓ You place the descriptions of the logic being modeled inside modules.
- ✓ Modules can represent:
  - ➢ A physical block such as an IC or ASIC cell
  - ➢ A logical block such as the ALU portion of a CPU design
  - ➢ The complete system
- ✓ Every module description starts with the keyword ***module***, has a name (FIFO, DFF, ALU, etc.), and ends with the keyword ***endmodule***
- ✓ Example is shown as below :

```
module  ALU(....);
          ......
      ......
           ......
   endmodule
```





module module_name(port_list);

port declaration

data type declaration

functionality or structure

task & function declaration

endmodule

Identifiers

- ✓ An identifiers is used to given an object, such as a register or a module a **name** so that it can be referenced from other places in a description.
- ✓ An identifier shall be any sequence of letters (a-z, A-Z), digits (0-9), dollar signs '$', and underscore characters '_'.
- ✓ The first character of an identifier shall be a letter or an underscore character.
- ✓ Verilog is case sensitive, therefore, *sel* and *SEL* are different identifiers.
  - ➢ All Verilog keywords are in lowercase letters
  - ➢ Keywords are predefined non-identifiers that are used to define the language constructs. For example: module , endmodule , wire , reg

```
module MUX2_1 (out,a,b,sel);
output out;
input a,b,sel;
   not not1 (sel_,sel);
   and and1 (a1,a,sel_);
   and and2 (b1,b,sel);
   or or1 (out,a1,b1);
endmodule
```

Verilog Identifiers

□ Example of illegal identifiers :

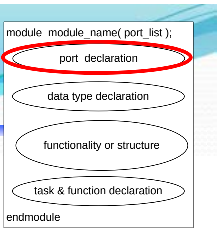
- ➢ 34net
- ➢ a*b_net
- ➢ n@238

Comments

- ✓ Single line : //
- ✓ Multiple line : /* ……. */
- ✓ Example :

```
module comments_eg (out, a, b, sel) ;
                    output out ;
                    input   a, b, sel; // This is a comments
                    not     not1(sel_,sel); /* this is a multiple-comments */
                    /*  and  #2  and1(a1,a,sel_);
                    and  #2     and2(b1,b,sel); */
                    or      or1(out,a1,b1);
        endmodule
```

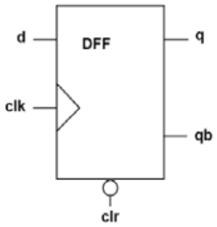
Module Ports

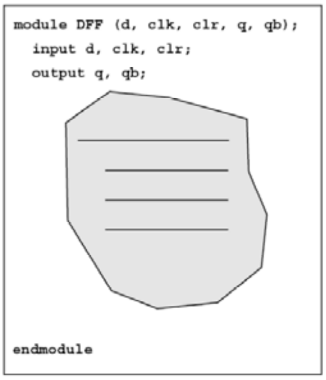


module module_name(port_list);
port declaration
data type declaration
functionality or structure
task & function declaration
endmodule

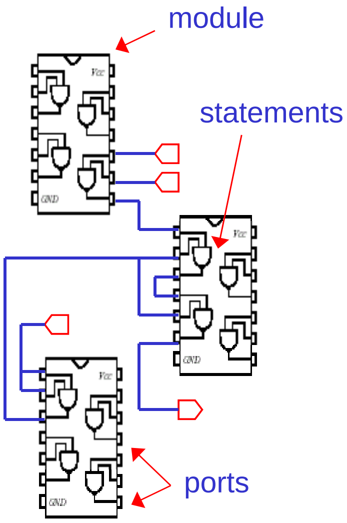
Module Ports

- ✓ Modules communicate with the outside world through ports.
- ✓ You list a module's ports in parentheses " ( ) " after the module name.
- ✓ <port_type> **[**range_begin **:** range_end**]** <port_name>[**,** port_name]* **;**
- ✓ Port declaration
  - ➢ input : input port
  - ➢ output : output port
  - ➢ inout : bidirectional port
- ✓ Example is shown as below : dff.v

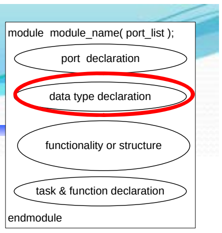




```
module DFF (d, clk, clr, q, qb);
  input d, clk, clr;
  output q, qb;

endmodule
```



Data Type



module module_name(port_list);

port declaration

data type declaration

functionality or structure

task & function declaration

endmodule

✓ Declaration Syntax

<data_type>[<MSB> : <LSB>]<list_of_identifier>

- **Nets** : represent physical connections between devices (default=z)
- **Register** : represent abstract data storage element (default=x)

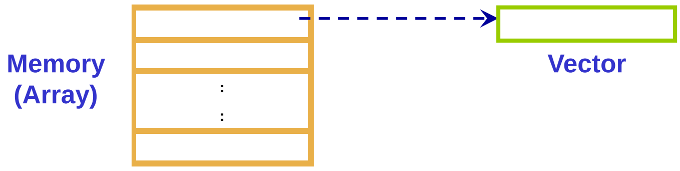
Net Types	Functionality
wire, tri	For standard interconnection wires (default)
supply1, supply0	For power or ground rails
wor, trior	For multiple drivers that are Wire-ORed
wand, triand	For multiple drivers that are Wire-ANDed
trireg	For nets with capacitive storage
tri1, tri0	For nets that pull up or down when not driven

Register type	Attribute
reg	Varying width
integer	32-bit signed (2's complement)
time	64-bit unsigned
real	Real number

Data Type *(cont.)*

- Vectors : the wire and register can be represented as a vector
  - wire [7:0] mem; ➔ 8-bit bus
  - reg [0:7] mem; ➔ mem[0] is the MSB
  - reg [0:0] mem; it's not well for the backend verifications.
- Arrays : <array_name>[<subscript>]

  →It isn't well for the backend verifications
  - integer mem[7:0] ➔ (8x32)-bit mem
  - reg [7:0] mem[0:1023] ➔ Memories!! (1k - 1byte)
- What's difference between Vector and Array?
  - **Vector** : multiple-bit and single-element
  - **Array** : multiple-bit and multiple-element

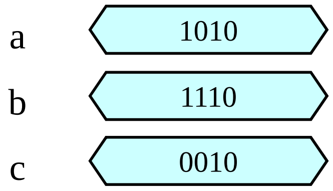


Data Type (cont.)

✓ **Signed & Un-signed**

– Verilog-1995
  - Signed : integer
  - Unsigned : reg, time, and all net data type.

– Verilog-2001
  - allows reg variables and all net data types to be declared using the reserved keyword **signed**



```
module verilog2001(a, b, c);

output signed [3:0] b;
output [3:0] c;
input  signed [3:0] a;

assign b = a >>> 2;
assign c = a >> 2;

endmodule
```

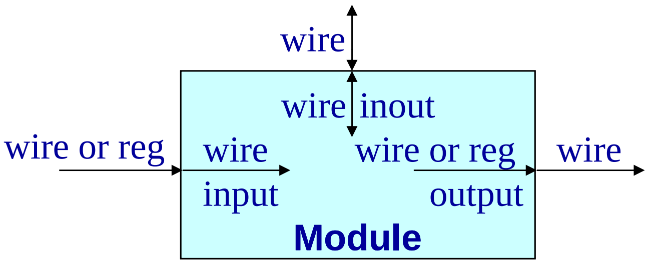
Port

✓ **Port connection**

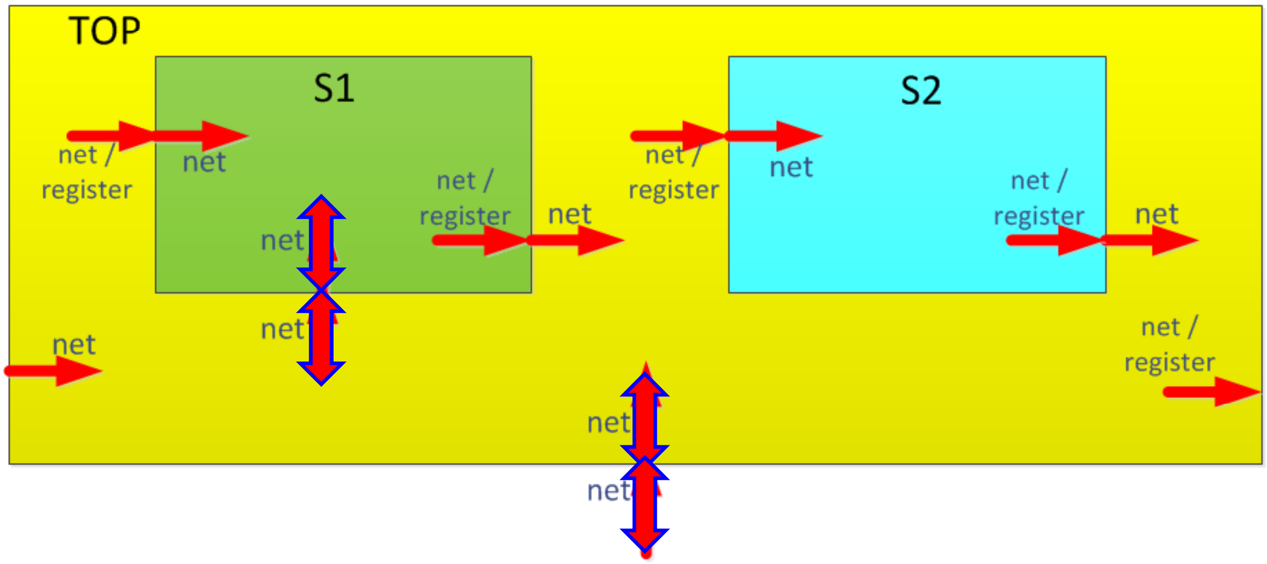
– input : only wire can be assigned to represent this port in the module

– output : only wire can be assigned to represent this port out of module

– inout : register assignment is forbidden neither in module nor out of module



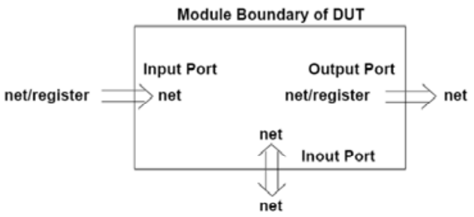
Port



Common Mistakes in Choosing Data Types

Listed here are common user errors, and the way they are typically referred to in error messages:

- ✓ You make a *procedural assignment* to a signal that you either declared as a net or you forgot to declare. This is an illegal assignment.
- ✓ You connect an output from an instance to a signal declared as a register. This is an illegal output port specification.
- ✓ You declare a signal as a module input and as a register. These are incompatible declarations.



- ✓ A Error Example :

```
module Top( in, out );
    input  in;
    output   out;
    ~~reg  in;~~
    reg [3:0] A,B;
    reg C_IN;
    reg [3:0] SUM;
    wire C_OUT;
    fulladd4 fa0(SUM,C_OUT,A,B,C_IN);
    ...
endmodule
```

Input要wire型態 !!

```
module fulladd4(s,co,a,b,cin);
…
…
endmodule
```

fulladd4的輸出不可以接到reg型態的sum

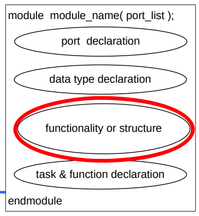
Module Connection

- ✓ Modules connected by port order (implicit)
  - ➢ Here order should match correctly. Normally it not a good idea to connect ports implicit. Could cause problem in debug, when any new port is added or deleted.
  - ➢ e.g. :

```
FA  U01( A, B, CIN, SUM, COUT );
```

- ✓ **Modules connect by name (explicit)**
  - ➢ Here name should match correctly.
  - ➢ e.g. :

```
FA U01(  .a(A), .b(B), .cin(CIN), .sum(SUM), .cout(COUT) );
```


module module_name(port_list);
port declaration
data type declaration
functionality or structure
task & function declaration
endmodule

Example

```
module MUX2_1(out,a,b,sel,clk,reset);
input       sel,clk,reset;
input       a,b;
output      out;
wire        c;
reg         a,b;        //incorrect define
reg         out;

//Continuous assignment
assign  c = (sel==1'b0)?a:b;

//Procedural assignment
always@(posedge reset or posedge clk)
begin
            if(reset==1'b1) out <= 0;
            else out <= c;
end
endmodule
```

【 sub module】

```
`include “mux.v”
module test;
reg         out;        //incorrect define
reg         a,b;
reg         clk,sel,reset;

// 1. connect port by ordering
MUX2_1 mux(out,a,b,sel,clk,reset);

// 2. connect port by name
MUX2_1 mux(.clk(clk), .reset(reset),
 .sel(sel), .a(a), .b(b), .out(out));

initial begin
………
end
endmodule
```

【test module】

Example

Module Instances

Examples are shown as below:

```
module DFF (d, clk, clr, q, qb);
....
endmodule
```

- ✓ Connecting ports by **ordered list**

```
module REG4(d,clk,clr,q,qb);
output [3:0] q, qb;
          input [3:0] d;
          input clk, clr;
                          DFF i_d0 (d[0],clk,clr,q[0],qb[0]);
                          DFF i_d1 (d[1],clk,clr,q[1],qb[1]);
                          DFF i_d2 (d[2],clk,clr,q[2],qb[2]);
                          DFF i_d3 (d[3],clk,clr,q[3],qb[3]);
          endmodule
```

- ✓ Connecting ports by **name**

Named ports. Highly recommended !!

```
module REG4(d,clk,clr,q,qb);
output [3:0] q, qb;
input [3:0] d;
input clk, clr;
                DFF i_d0 ( .d(d[0]), .clk(clk), .clr(clr), .q(q[0]),  .qb(qb[0]));
                DFF i_d1 ( .d(d[1]), .clk(clk), .clr(clr), .q(q[1]), .qb(qb[1]));
                DFF i_d2 ( .d(d[2]), .clk(clk), .clr(clr), .q(q[2]), .qb(qb[2]));
                DFF i_d3 ( .d(d[3]), .clk(clk), .clr(clr), .q(q[3]), .qb(qb[3]));
endmodule
```

Parameters

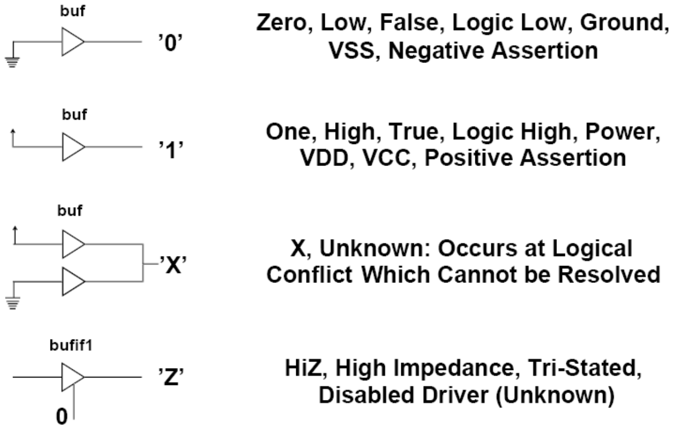
- ✓ Use parameters to declare run-time constants.
- ✓ Parameters are **local**, known **only to the module** in which they are defined.
- ✓ Parameter definition syntax is : ***parameter <list_of_assignments>***
  - ➢ list_of_assignments is a comma-separated list of parameters and their values.

- ✓ Example :

```
module mod1(out,in1,in2);
. . .
parameter  cycle = 20, prop_del = 3, setup = cycle/2 - prop_del,
                p1 = 8, x_word = 16'bx,
   file = "/usr1/jdough/design/mem_file.dat";
. . .
wire [p1:0] w1; // A wire declaration using parameter
. . .
endmodule
```

4-Value Logic System in Verilog

✓ The Verilog HDL value set consists of four basic values :



Zero, Low, False, Logic Low, Ground, VSS, Negative Assertion

One, High, True, Logic High, Power, VDD, VCC, Positive Assertion

X, Unknown: Occurs at Logical Conflict Which Cannot be Resolved

HiZ, High Impedance, Tri-Stated, Disabled Driver (Unknown)

✓ The "unknown" logic value in Verilog is not the same as "don't care." It represents a situation where the value of a node cannot be predicted. In real hardware, this node will most be at either 1 or 0.

✓ When the Z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value.

Integer value

✓ Number Specification

- **<size>'<base><value>**
  - <size> is the length of desired value in bits.
  - <base> can be b(binary), o(octal), d(decimal) or h(hexadecimal).
  - <value> is any legal number in the selected base.
  - When <size> is *smaller than <value>, then left-most bits of <value>*are truncated
  - When <size> is *larger than <value>, then left-most bits are filled,*
  - based on the value of the left-most bit in <value>.
    - □ Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'
  - Default size is 32-bits decimal number
  - e.g. 4'd10 ➔ 4-bit, 10, decimal
  - e.g. 6'hca ➔ 6-bit, store as 6'b001010 (truncated, not 11001010!)
  - e.g. 6'ha ➔ 6-bit, store as 6'b001010 (filled with 2-bit '0' on left!)
- **Negative : -<size>'<base><value>**
  - e.g. –8'd3 ➔ legal 8'd–3 ➔ illegal

Integer value(cont.)

- **Extension:**
  - 12’hz → zzzz zzzz zzzz
  - 6’bx → xx xxxx
  - 8’b0 → 0000 0000
  - 8’b1 → 0000 0001
- **Underline:**
  - 12’b1111_0010_1011 → 1111 0010 1011

Integer value(cont.)

- ✓ Verilog truncates the most significant bits when you specify the size to be Smaller than the number entered.
  - Ex: 3’b10_1101->3’b101
- ✓ When <size> is greater than <value>, and the MSB of <value> is x or z, it will extended to <size> bits.
  - 7’hx // 7-bit x (x extended)
  - 4’hz // 4-bit z (z extended)
  - 10’bx10 // x extended to fill the MSB,*10’bxxxxxxxx10*
  - 10’bz10 // z extended to fill the MSB,*10’bzzzzzzzz10*
  - 10’b010 // 0 extended to fill the MSB,*10’b0000000010*
  - 10’b110 // 0 extended to fill the MSB,*10’b1111111110*

Operator Types

- ✓ The following table shows the operators in Verilog, in order of precedence.
- ✓ An explanation of each with an example is shown in this chapter.

Type of Operators	Symbols
Concatenate & replicate	{ } {{ }}
Unary	! ~ & | ^
Arithmetic	* / % + -
Logical shift	<< >>
Relational	> < >= <=
Equality	== === != !==
Binary bit-wise	& | ^ ~^
Binary logical	&& ||
conditional	? :

Highest ↑ Precedence Lowest

Highest ← Precedence Lowest

➢ a < b-1 && c != d || e == f ⟶ worse

(a < b-1) && (c != d) || (e == f) ⟶ better

Operators

✓ Operators

➢ Arithmetic Description

- A = B + C;
- A = B – C;
- A = B * C;
- A = B / C;
- A = B % C; ➔ Most synthesis tools don't support

➢ Shift Operator (**bit-wise**)

- A = B >> 2; ➔ shift right 'B' by 2-bit
- A = B << 2; ➔ shift left 'B' by 2-bit

➢ Shift Operator (**arithmetic**)

- A = B >>> 2;
- A = B <<< 2;
- e.g. B = 4'b1000;
  A = B >>>2;

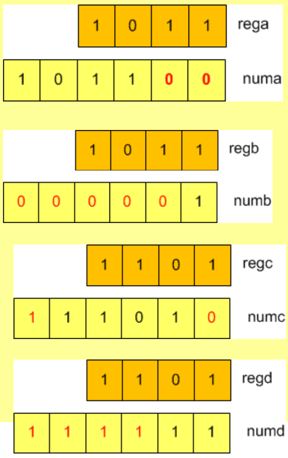
">>>", "<<<" are used only for '**signed**' data type in Verilog 2001

(A = 4'b1110 ,which is 1000 shifted to the right two positions and sign-filled.)

About Shift Operators

- ✓ a << n
  - a shift left n bits, and fills in zero bits
- ✓ b >> n
  - b shift right n bits, and fills in zero bits
- ✓ c <<< n
  - c shift left n bits, and fills in zero bits
- ✓ d >>> n
  - d shift right n bits, and fills in signed bits

```
numa = rega << 2 ;

numb = rega >> 3;

numc = regb <<< 1;

numd = regb >>> 2;
```



		1	0	1	1	rega
1	0	1	1	0	0	numa
		1	0	1	1	regb
0	0	0	0	0	1	numb
		1	1	0	1	regc
1	1	1	0	1	0	numc
		1	1	0	1	regd
1	1	1	1	1	1	numd

Operators(cont.)

- Bit-wise Operator
  - NOT: A = ~ B;
  - AND: A = B & C;
  - OR: A = B | C;
  - XOR: A = B ^ C;
  - e.g. 4'b1001 | 4'b1100 ➔ 4'b1101
  - e.g. 4'b1001 | 4'b1010 ➔ 4'b1011
  - e.g. 4'b1010 & 4'b11x0 ➔ 4'b10x0
  - e.g. 4'b1010 | 4'b11x0 ➔ 4'b1110

- Logical Operators: return 1-bit true/false
  - Logical **binary** operators operate on logic values.
    - □ If an operand contains all zeroes, it is false (logic 0).
    - □ If it contains any ones, it is true (logic 1).
    - □ If it is unknown (contains only zeroes and/or unknown bits), its logical value is ambiguous.
  - NOT: A = ! B;
  - AND: A = B && C;
  - OR: A = B || C;
  - e.g. 4'b1001 || 4'b1100 ➔ true, 1'b1

Operators(cont.)

- Conditional Description
  - <LHS> = <condition> ? <true_expression> : <false_expression>;
  - ? : → c = sel ? a : b; // if (sel==1’b1)

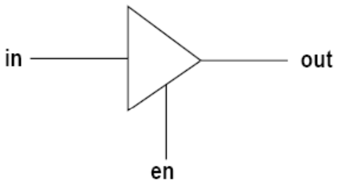
```
                    //     c = a;
                    // else
                    //     c = b;
```

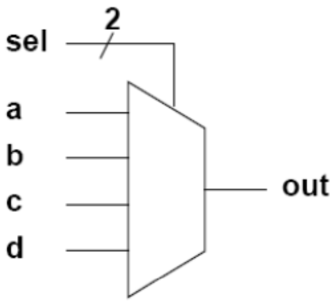
- Relational and equality(conditional)
  - <=, <, >, >=, ==, !=
  - i.e. if((a<=b) && (c==d) || (e>f))
  - 4'b0x10 > 4’b0011 // result = x
  - 4'b1010 < 4’b0011 // result = 0
  - 4'b1010 >= 4’b0011 // result = 1
  - 4'b1010 > 4'b0x10 // result = 1

Operators(cont.)

? : conditional

✓ An unknown value on the condition will result in an unknown value on out.



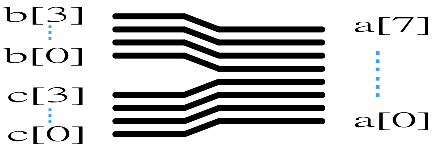


```
module likebufif (in,en,out);
input in;
input en;
output out;
     assign out = (en == 1) ? in : 'bz;
endmodule

module like4to1 (a,b,c,d,sel,out);
input a,b,c,d;
input [1:0] sel;
output out;
assign out = (sel == 2'b00) ? a :
                    (sel == 2'b01) ? b :
                 (sel == 2'b10) ? c : d;
endmodule
```

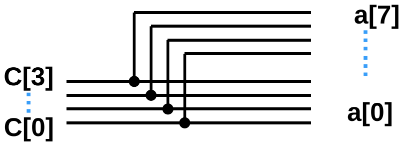
Operators(cont.)

- Concatenation

⅄ { } → a = {b, c};



⅄ {{}} → a = {2{c}};



⅄ a[4:0] = {b[3:0],1’b0}; ⇔ a = b << 1;

Verilog Primitives

✓ Verilog provides basic logical functions as predefined primitives.

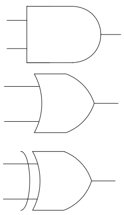
- You do not have to define this basic functionality.

Primitive Name	Functionality
and	Logical And
or	Logical Or
not	Inverter
buf	Buffer
xor	Logical Exclusive Or
nand	Logical And Inverted
nor	Logical Or Inverted
xnor	Logical Exclusive Or Inverted

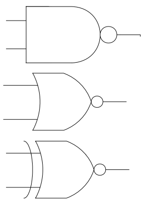
Gate-Level Modeling

✓ Primitive logic gate

- and
- or
- xor



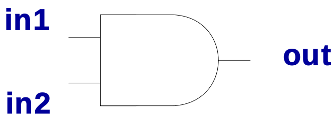
- nand
- nor
- xnor

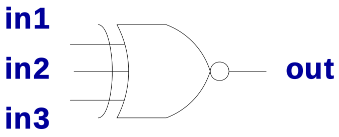


The gates have *one scalar output and multiple scalar inputs.* The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

-- can use without instance name ➔ e.g. and(out, in1, in2) ;

-- can use with multiple inputs ➔ e.g. xor(out, in1, in2, in3) ;

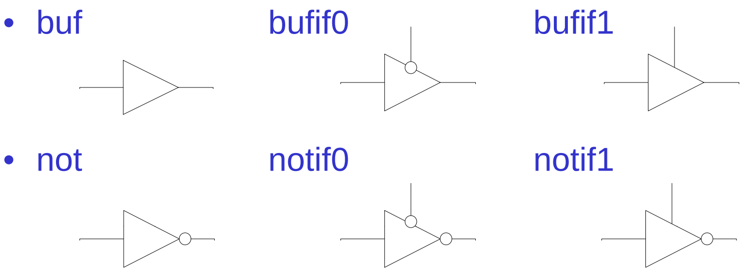




Gate-Level Modeling (cont.)

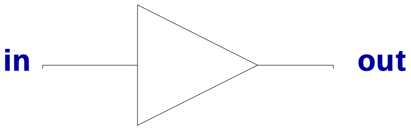
✓ Primitive logic gate

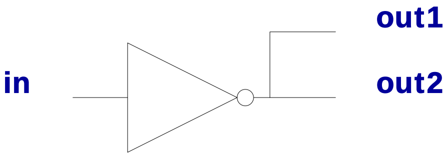
– buf/not gates



-- can use without instance name ➔ e.g. buf(out, in) ;

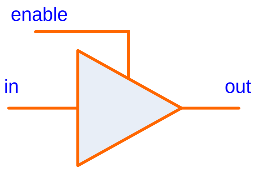
-- can use with multiple outputs ➔ e.g. not(out1, out2 ,in) ;





Gate-Level Modeling (cont.)

- ✓ Verilog has four different types of conditional primitives.
- ✓ They have only three pins: output, input, and ***enable***.
- ✓ They are enabled and disabled by the enable pin.
  - ➢ When conditional primitives are disabled, their outputs are at high impedance.
- ✓ Example： bufif1(out, in, enable);

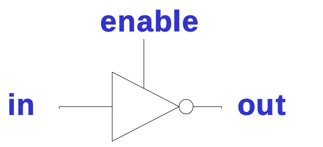


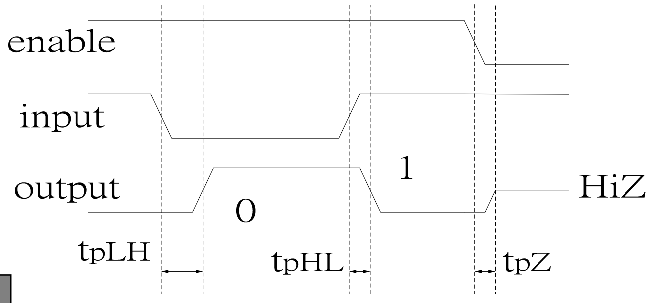
Primitive Name	Functionality
bufif1	Conditional buffer with logic 1 as enabling input
bufif0	Conditional buffer with logic 0 as enabling input
notif1	Conditional inverter with logic 1 as enabling input
notif0	Conditional inverter with logic 0 as enabling input

Gate-Level Modeling (cont.)

✓ Gate Delays

- Rise Delays
- Fall Delays
- Turn-off Delays





notif1 (out, in, enable) ;

gate #(each_delay) a1(out, i1, i2);

gate #(rise_delay, fall_delay) a2(out, i1, i2) ;

gate #(rise_delay, fall_delay, turnoff_delay) a3(out, in, enable) ;

– Only bufif0, bufif1, notif0, notif1 have turn-off delays

```
e.g. and #(3,4) ( out, i1, i2 );            //rise=3,fall=4
e.g. bufif1 #(3,4,5) ( out, in, enable );   //rise=3,fall=4,turnoff=5
```

Gate-Level Modeling (cont.)

✓ Gate Delays

- ➢ Min/Typ/Max delay time
  - ⅄ Min/Typ/Max : expectation minimum/typical/maximum delay

  gate #(mindelay:typdelay:maxdelay) b(out, i1, i2);

  e.g. nand #(1:2:3) (out , in1 , in2);

- ➢ Combine min/typ/max and rise/fall/turn-off delays

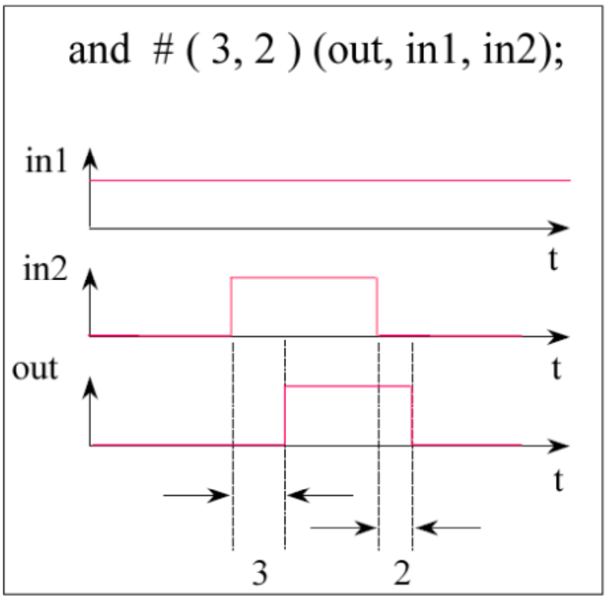
  e.g. notif1 #(3:4:5,6:7:8,1:2:3) (out, in, enable);

  /*minimum rise=3,fall=6,turn-off=1,typical rise=4,fall=7,turn-off=2,
  maximum rise=5,fall=8,turn-off=3*/

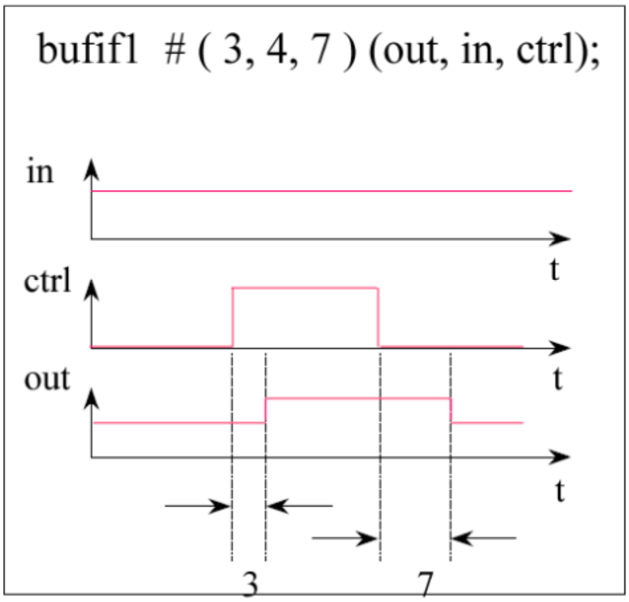
  - ⅄ Use the +maxdelays(typdelays/mindelays) to select maximum(typical/minimum) delays for simulation

  e.g. % ncverilog +maxdelays yourdesign.v

Gate-Level Modeling (cont.)



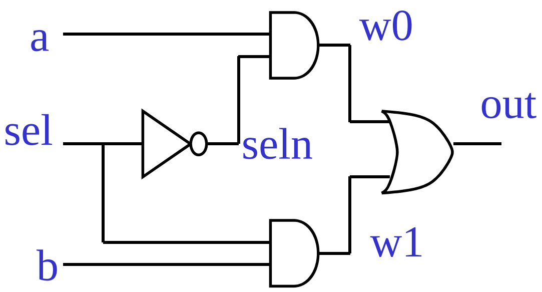
and # (3, 2) (out, in1, in2);



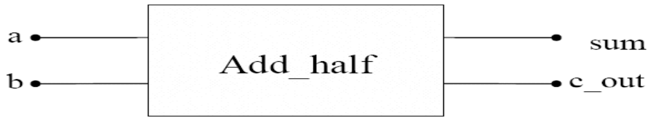
bufif1 # (3, 4, 7) (out, in, ctrl);

Gate-Level Modeling (cont.)

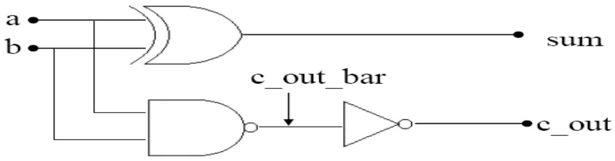
```
module MUX2_1(out,a,b,sel) ;
//declare ports
input           a, b, sel ;
output          out ;
//declare inside wire
wire            seln ;
wire            w0, w1 ;
//gate instances
not #(0.2:0.3:0.5, 0.2:0.3:0.5) n0( seln, sel ) ;

and #(0.4:0.6:0.9, 0.4:0.6:0.9) a0( w0, a, seln );
and #(0.4:0.6:0.9, 0.4:0.6:0.9) a1( w1, b, sel );

or #(0.4:0.6:0.9, 0.4:0.6:0.9) o0( out, w0, w1 ) ;

endmodule
```



Example



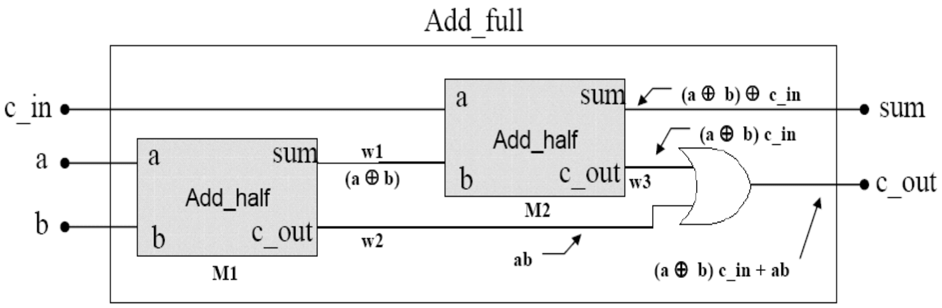
Block Diagram



Schematic

```
module ADDR_HALF(A, B, SUM, C_OUT);
input   A, B;
output  SUM, C_OUT;
wire    c_out_bar;
xor (SUM, A, B);
and (C_OUT, A, B);
endmodule
```

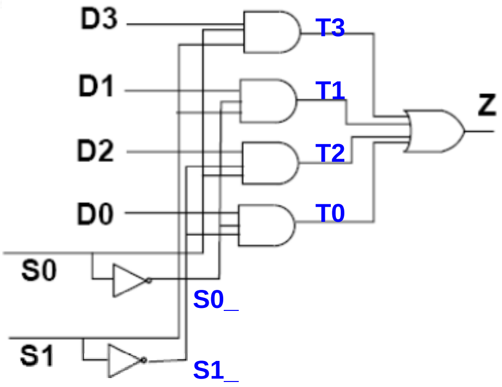
Example (cont.)



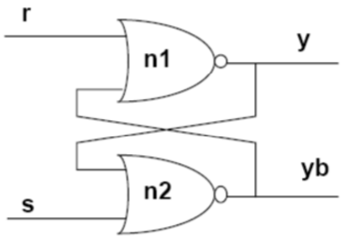
```
module ADDR_FULL(A, B, C_IN, SUM, C_OUT);
input   A, B, C_IN;
output  SUM, C_OUT;
wire    w1, w2, w3;
ADDR_HALF M1(.A(A), .B(B), .SUM(w1), .C_OUT(w2));
ADDR_HALF M2(.A(C_IN), .B(w1), .SUM(SUM), .C_OUT(w3));
or (C_OUT, w2, w3);
endmodule
```

Example (cont.)

```
module  MUX4x1(Z, D0, D1, D2, D3, S0, S1);
output  Z;
input    D0, D1, D2, D3, S0,S1;
    and         (T0, D0, S0_, S1_),
                (T1, D1, S0_, S1),
                (T2, D2, S0, S1_),
                (T3, D3, S0, S1);
    not         (S0_,S0),  (S1_,S1);
    or          (Z, T0, T1, T2, T3);
endmodule
```

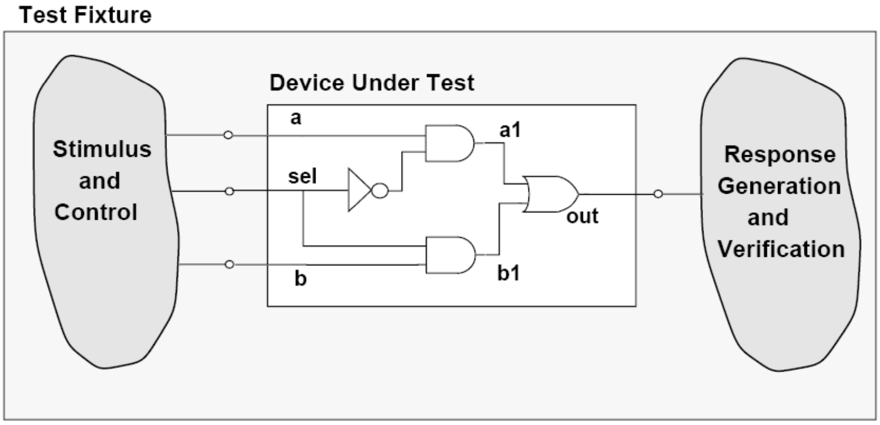


```
module     rs_latch (y, yb, r, s);
output     y, yb;
input      r, s;
    nor             n1(y, r, yb);
    nor             n2(yb, s, y);
endmodule
```



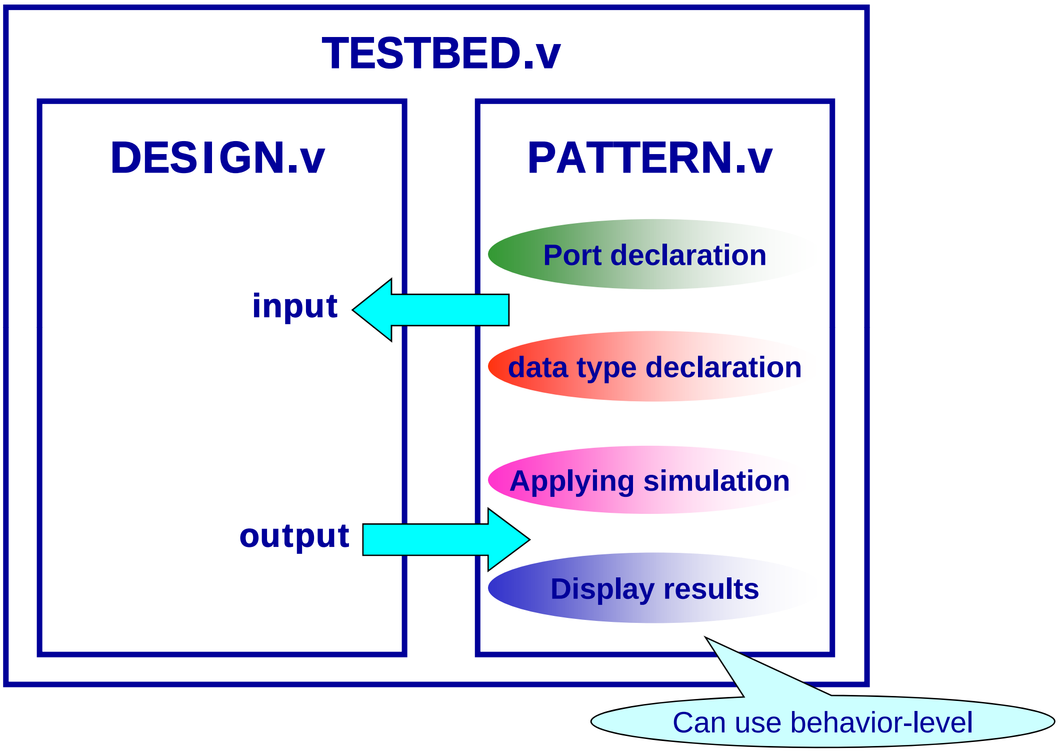
A Simple and Complete Example

- ✓ The system consists of a 2:1 multiplexer, which is the Device Under Test (DUT) , and a **test fixture** to provide a test stimulus and a verification mechanism.
- ✓ In the next few pages the test fixture will be described behaviorally and the DUT will be described at the gate level, to illustrate mixed-level simulation.



Test the 2:1 MUX’s Verilog model by applying test patterns and observing its output response

Simulation Environment



TESTBED.v

DESIGN.v

PATTERN.v

Port declaration

input

data type declaration

Applying simulation

output

Display results

Can use behavior-level

Simulation Environment (cont.)

TESTBED.v

```
`timescale 1ns/10ps
`include “MUX2_1.v”
`include “PATTERN.v”

module  TESTBED;

wire out,a,b,sel,clk,reset;

MUX2_1 mux(.out(out),.a(a),.b(b),.sel(sel));
PATTERN pat(.sel(sel),.out(out),.a(a),.b(b));

enmodule
```

Simulation Environment (cont.)

PATTERN.v

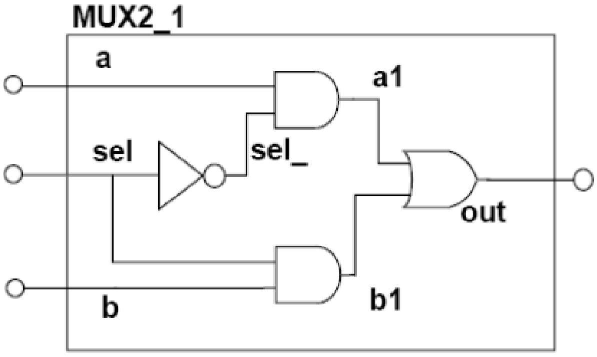
```
module PATTERN(sel,out,a,b);

input out;
output a,b,sel;

reg a,b,sel,clk,reset;
integer i;
parameter cycle=10;

always #(cycle/2) clk = ~clk;

initial begin
a=0;b=0;sel=0;reset=0;clk=0;
#3 reset = 1;
#10 reset = 0;
```

```
#cycle sel=1;
for(i=0;i<=3;i=i+1)
begin
#cycle {a,b}=i;
#cycle $display( "sel=%b, a=%b, b=%b,
                 out=%b" , sel, a, b, out);
end

#cycle sel=0;
for(i=0;i<=3;i=i+1)
begin
#cycle {a,b}=i;
#cycle $display( "sel=%b, a=%b, b=%b,
                 out=%b" , sel, a, b, out);
end

#cycle $finish;
end

endmodule
```

Device Under Test

- ✓ Example is shown as below : mux2_1.v

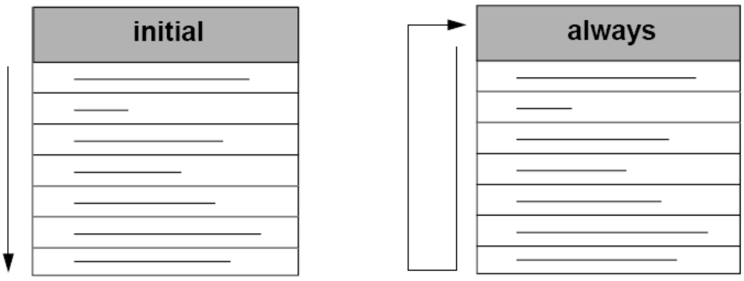
```
module MUX2_1 (out, a, b, sel);
        // Port declarations
        output out;
        input a, b, sel;
        wire out, a, b, sel;
        wire sel_, a1, b1;
        // The netlist (structural)
        not  u1(sel_, sel);
        and  u2(a1, a, sel_);
        and  u3(b1, b, sel);
        or  u4(out, a1, b1);
endmodule
```



Procedural Blocks

Procedural Blocks

- ✓ It is common to use procedural blocks during behavioral modeling. You must specify the stimulus part of the test fixture <u>inside a procedural block</u>.
- ✓ There are two types of procedural blocks:
  - ➢ *initial* procedural blocks, which execute <u>only once</u>
  - ➢ *always* procedural blocks, which execute <u>in a loop</u>
- ✓ There is a difference between the activation and the execution of procedural blocks:
  - ➢ All procedural blocks are activated at time 0, waiting to be executed based upon the user-specific conditions.
  - ➢ All procedural blocks execute concurrently, allowing you to model the inherent concurrence in hardware.



Data Assignment

- ✓ Continuous Assignment ➔ for wire assignment
  - ➢ Imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.
    - e.g.

```
wire [3:0] a;
assign     a = b + c; //continuous assignment
```

- ✓ Procedural Assignment ➔ for reg assignment
  - ➢ assignment to "**register**" data types may occur within *always*, *initial*, *task* and *function*. These expressions are controlled by triggers which cause the assignment to evaluate.
    - e.g.

```
reg a,clk;
always  #5 clk = ~clk;          //procedural assignment
```

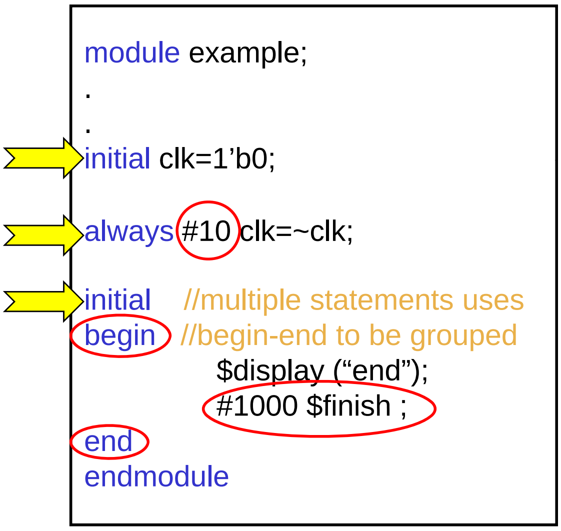
    - e.g.

```
always @ (b)        //procedural assignment with triggers
a = ~b;
```

Data Assignment *(cont.)*

In the behavior-level modeling, Verilog code is just like C language.
And in the behavior level, we use two essential statement.

- ✓ **initial**
  - An initial block *starts at 0*,
    and executes once in a simulation.
- ✓ **always**
  - An always block *starts at 0*,
    and executes repeatedly as a loop.



```
module example;
.
.
initial clk=1'b0;

always #10 clk=~clk;

initial    //multiple statements uses
begin      //begin-end to be grouped
        $display ("end");
        #1000 $finish ;
end
endmodule
```

Test Fixture *(cont.)*

Describing Stimulus

- ✓ In the example below, **a**, **b**, and **sel** are declared as data type *reg*. Signals of a register data type retain their value until reassigned.
- ✓ #5 is used to specify waiting for 5 time units.
- ✓ $finish is a system task that ends simulation.
- ✓ Example is shown as below :

```
module testfixture;
  //Data type declaration
        reg a, b, sel;
        wire out;
  //MUX instance
        MUX2_1 mux (out, a, b, sel);
  //Apply stimulus
        initial  begin
        a = 0;  b = 1;  sel = 0;
        #5 b = 0;
        #5 b = 1; sel = 1;
        #5 a = 1;
        #5 $finish;
        end
  //Display results
endmodule
```

Procedural Blocks

Time	Values		
	a	b	sel
0	0	1	0
5	0	0	0
10	0	1	1
15	1	1	1

Test Fixture *(cont.)*

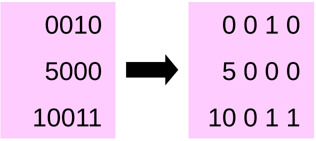
Response Generation

- ✓ Verilog provides a number of system tasks and system functions, including:
  - ➢ $time is a system function that returns the current simulation time.
  - ➢ $monitor is a system task that displays the values of the argument list at the end of any time unit in which any of the arguments change.
  - ➢ $monitor displays the values of the argument list at the end of any time unit in which any of the arguments change. It is not, however, triggered by the change in value caused by $time.

$monitor ([”format_specifiers”,] <arguments>);

- ✓ For example:
  - ➢ $monitor($time, o, in1, in2);
  - ➢ $monitor($time,, out,, a,, b,, sel);
  - ➢ $monitor($time, ”%b %h %d %o”, sig1, sig2, sig3, sig4);
  - ➢ $monitor($time, ”sig1=%b, sig2=%h, sig3=%d, sig4=%o”, sig1, sig2, sig3, sig4);

增加空格



```
0010        0 0 1 0
5000   ➡    5 0 0 0
10011       10 0 1 1
```

Test Fixture *(cont.)*

✓ Complete Test Fixture

```
module testfixture;
 //Data type declaration
     reg a, b, sel;
     wire out;
 //MUX instance
     MUX2_1 mux (out, a, b, sel);
 //Apply stimulus
     initial
     begin
     a = 0; b = 1; sel = 0;
     #5 b = 0;
     #5 b = 1; sel = 1;
     #10 a = 1;
     #10 $finish;
     end
 //Display results
     initial
     $monitor ($time,,"out=%b a=%b b=%b sel=%b", out, a, b, sel);
endmodule
```

- ◆ #10 and #5 used to specify for 10 time unit delay
- ◆ ***$finish*** is a system task that ends simulation

Time	Values		
	a	b	sel
0	0	1	0
5	0	0	0
10	0	1	1
20	1	1	1

Simulation Environment *(cont.)*

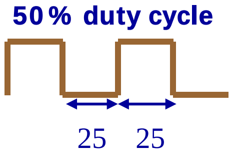
✓ Dump a FSDB file for debug

- ➢ General debussy waveform generator
  - $fsdbDumpfile(“file_name.fsdb”); ➔ Dump an fsdb file
  - $fsdbDumpvars; ➔ Dump all values
- ➢ Other debussy waveform generator
  - $fsdbSwitchDumpfile(“file_name.fsdb”);
    - ➔ close the previous fsdb file and create a new one and open it
  - $fsdbDumpflush (“file_name.fsdb”);
    - ➔ not wait the end of simulation and Dump an fsdb file
  - $fsdbDumpMem(memory_name, begin_cell, size);
    - ➔ the memory array is stored in an fsdb file
  - $fsdbDumpon; $fsdbDumpoff;
    - ➔ just Dump and not Dump

Simulation Environment (cont.)

✓ **Clock Generation**

```
initial clk = 0;
always #25 clk = ~clk;
```



✓ **Display simulation result**

– Texture format output

- $display(“%t, clk=%b in=%d out=%d \n”, $time clk, in, out);
- $monitor($time,”clk=%b out=%b\n”,clk,out);

format (display): %d (decimal), %b (binary), %h (hexadecimal),
%o (octal), %c(ASCII),%s (strings), %v
(strength), %m(hierarchical name), %t (time)

$time : current time

\n: new line, \t: tab, \\: backslash, \”: double quote

Simulation Environment (cont.)

✓ **Random number generation**

- $random
- $random(seed)

e.g. in1=$random;
in2=$random(37);

✓ **Control command**

- $finish;

//finish the simulation

- $stop;

//stop the simulation

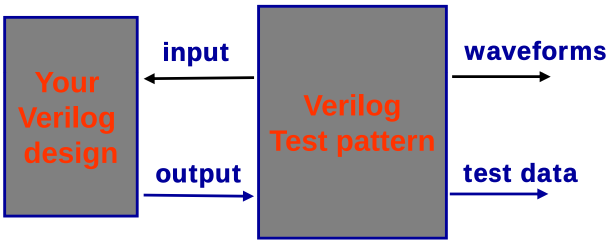
✓ **Simulation command**

– Verilog compile

- verilog test_file_name.v
- ncverilog test_file_name.v

– Debussy waveform generation

- nWave &

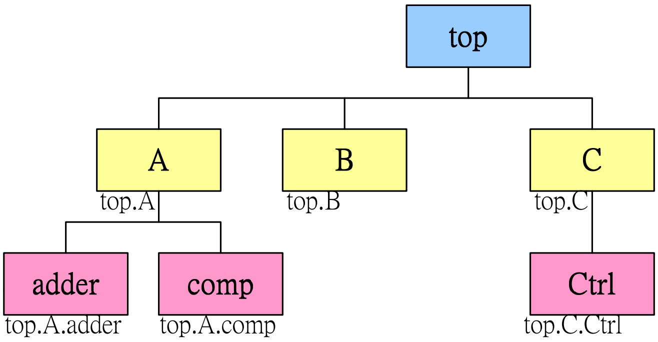


□ Modelsim simulation and debug tool

□ vsim &

Hierarchical Names

- ✓ Every identifier in a Verilog HDL description shall have a unique hierarchical path name.
- ✓ The hierarchy of names can be viewed as a tree structure.
- ✓ Hierarchical name referencing allows free data access to any object from any level in the hierarchy.



top

A top.A

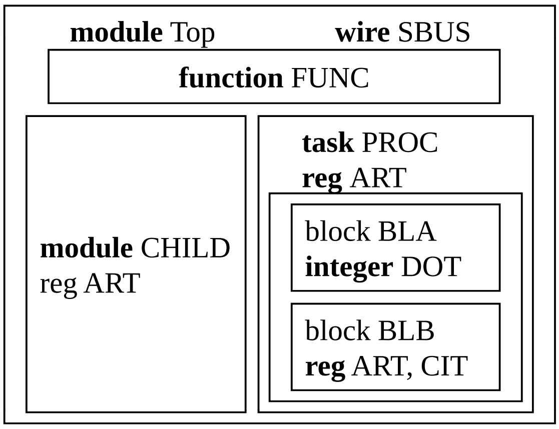
B top.B

C top.C

adder top.A.adder

comp top.A.comp

Ctrl top.C.Ctrl



module Top **wire** SBUS

function FUNC

module CHILD
reg ART

task PROC
reg ART

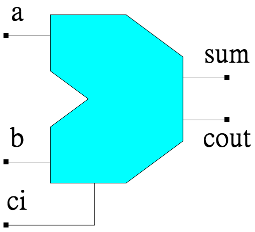
block BLA
integer DOT

block BLB
reg ART, CIT

TOP.SBUS
TOP.CHILD.ART
TOP.PROC.ART
TOP.PROC.BLB.CIT / TOP.PROC.BLA.DOT

A Full Adder Example (1/9) (OPTIONAL)

- ◆ Symbol of a full adder



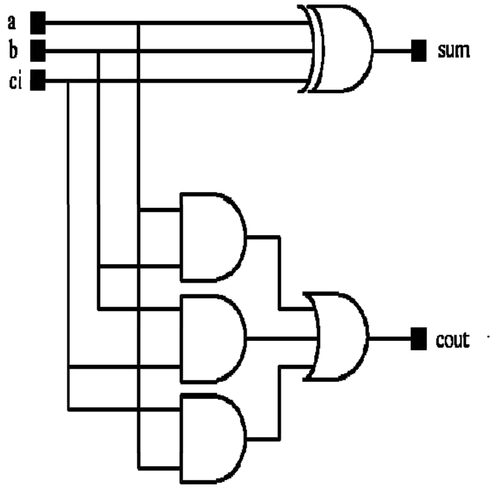
- ◆ Expected behavior of a full adder

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

A Full Adder Example (2/9) (OPTIONAL)

Gate level Verilog description of a full adder

```
module fadder (sum,cout,a,b,ci);
  //port declaration
  output sum, cout;
  input  a, b, ci;

  //netlist declaration
  xor u0 (sum, a, b, ci);
  and u1 (net1, a, b);
  and u2 (net2, b, ci);
  and u3 (net3, ci, a);
  or  u4 (cout, net1, net2, net3);
endmodule
```



A Full Adder Example (3/9) (OPTIONAL)

Behavior level Verilog description of a full adder

```
module fadder (sum,cout,a,b,ci);
  //port declaration
  output sum, cout;
  input  a, b, ci;

  reg sum,cout;
  //behavior declaration
  always @(a or b or ci)
    begin
      sum  = a ^ b ^ ci;
      cout = (a&b)|(b&ci)|(ci&a);
    end
endmodule
```

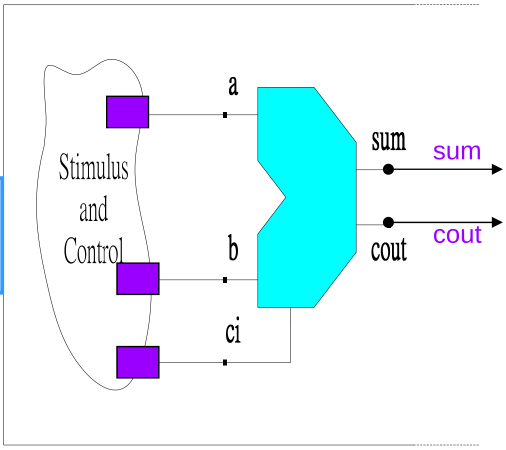
$\text{sum} = a \oplus b \oplus c_i$
$\text{cout} = ab + bc_i + c_ia$

A Full Adder Example (4/9) (OPTIONAL)

Test Fixture – Specifying an Instance & Data type Declaration

- Test patterns must be first stored in storage elements and then applied to DUT

```
module fadder_test;

  //Data type declaration
  reg a,b,ci;
  wire sum,cout;

  //Instantiate modules
  fadder fadd01(sum,cout,a,b,ci);

  //Apply stimulus

  //Display results

endmodule
```



A Full Adder Example (5/9) (OPTIONAL)

Test Fixture – Add the Stimulus

	a	b	ci
0	x	x	x
10	0	0	0
20	1	0	0
30	1	0	1
40	1	1	1
50	0	1	0

```
module fadder_test;
  //Data type declaration
  reg a,b,ci;
  wire sum,cout;
  //Instantiate modules
  fadder fadd01(sum,cout,a,b,ci);

  //Apply stimulus
  initial
    begin
      #10 a=0;b=0;ci=0;
      #10 a=1;b=0;ci=0;
      #10 a=1;b=0;ci=1;
      #10 a=1;b=1;ci=1;
      #10 a=0;b=1;ci=0;
      #10 $finish;
    end
  //Display results
endmodule
```

A Full Adder Example (6/9) (OPTIONAL)

Test Fixture – Observe the Response

```
module fadder_test;
  reg a,b,ci;
  wire sum,cout;
  fadder fadd01(sum,cout,a,b,ci);
  initial
    begin
      #10 a=0;b=0;ci=0;
      #10 a=1;b=0;ci=0;
      #10 a=1;b=0;ci=1;
      #10 a=1;b=1;ci=1;
      #10 a=0;b=1;ci=0;
      #10 $finish;
    end
  //Display results
  initial
  $monitor ($time,,”a=%b,b=%b,ci=%b, sum=%b,cout=%b”,
                    a,b,ci,sum,cout);
endmodule
```

A Full Adder Example (7/9) (OPTIONAL)

Simulation Result

```
% verilog testfixture.v fadder.v
. . . . . . . . .
Compiling source file "fadder.v"
Compiling source file "fadder_test.v"
Highest level modules:
testfadder
           0 a=x,b=x,ci=x, sum=x,cout=x
          10 a=0,b=0,ci=0, sum=0,cout=0
          20 a=1,b=0,ci=0, sum=1,cout=0
          30 a=1,b=0,ci=1, sum=0,cout=1
          40 a=1,b=1,ci=1, sum=1,cout=1
          50 a=0,b=1,ci=0, sum=1,cout=0

L13 "fadder_test.v": $finish at simulation time 60
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of VERILOG-XL 3.0.s005   Jan 21, 2003  12:39:26
```

Use the –f command line option to specify a file that contains command-line arguments.

verilog –f run.f

run.f

```
fadder.v
Testfixture.v
```

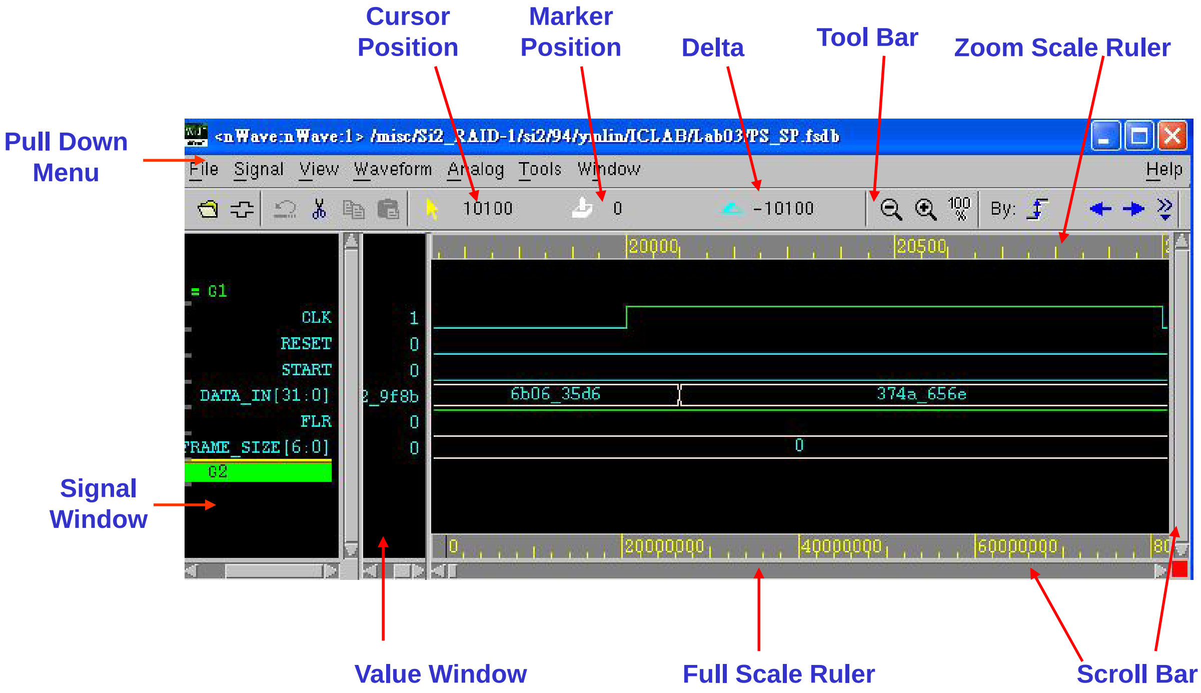
A Full Adder Example (8/9) (OPTIONAL)

Create the Simulation Waveform

```
module fadder_test;
  reg  a,b,ci;
  wire sum,cout;

      . . . . . . . .
      . . . . . . . .
      #10 a=0;b=1;ci=0;
      #10 $finish;
    end
  initial
  $monitor ($time,,”a=%b,b=%b,ci=%b, sum=%b,cout=%b”,
                  a,b,ci,sum,cout);
  initial
    begin
      $fsdbDumpfile(“fadder.vcd”);
      $fsdbDumpvars();
    end
endmodule
```

A Full Adder Example (9/9) (OPTIONAL)

Springsoft ***Verdi (nWave)*** waveform viewer



Cursor Position

Marker Position

Delta

Tool Bar

Zoom Scale Ruler

Pull Down Menu

Signal Window

Value Window

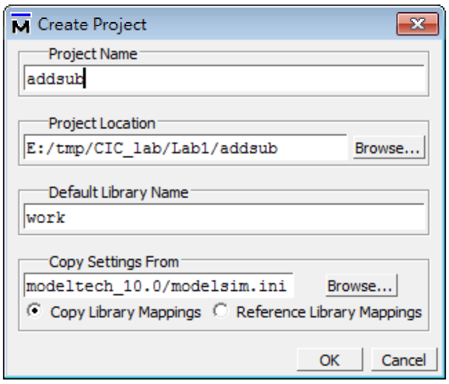
Full Scale Ruler

Scroll Bar

Modelsim Simulation Flow

Create Project

- ✓ File➔ New➔ Project...
- ✓ Add Existing File or Create New File



Create Project

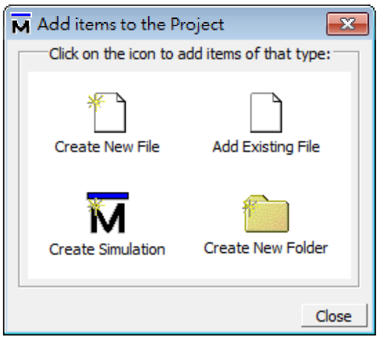
Project Name: addsub

Project Location: E:/tmp/CIC_lab/Lab1/addsub

Default Library Name: work

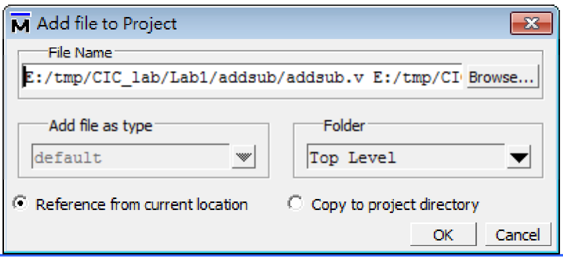
Copy Settings From: modeltech_10.0/modelsim.ini

● Copy Library Mappings ○ Reference Library Mappings



Add items to the Project

Click on the icon to add items of that type:



Add file to Project

File Name: E:/tmp/CIC_lab/Lab1/addsub/addsub.v E:/tmp/CI

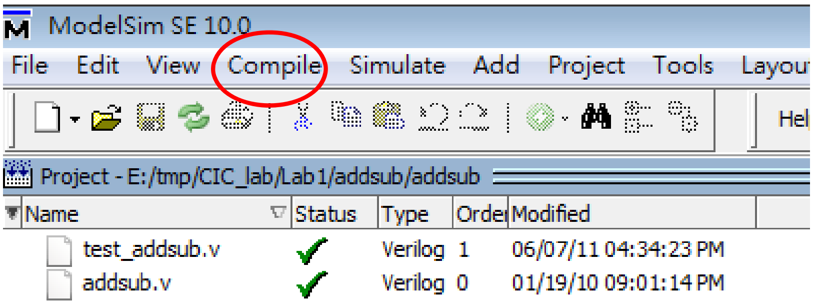
Add file as type: default

Folder: Top Level

● Reference from current location ○ Copy to project directory

Compile files

- ✓ Compile➔ Compile All…



ModelSim SE 10.0

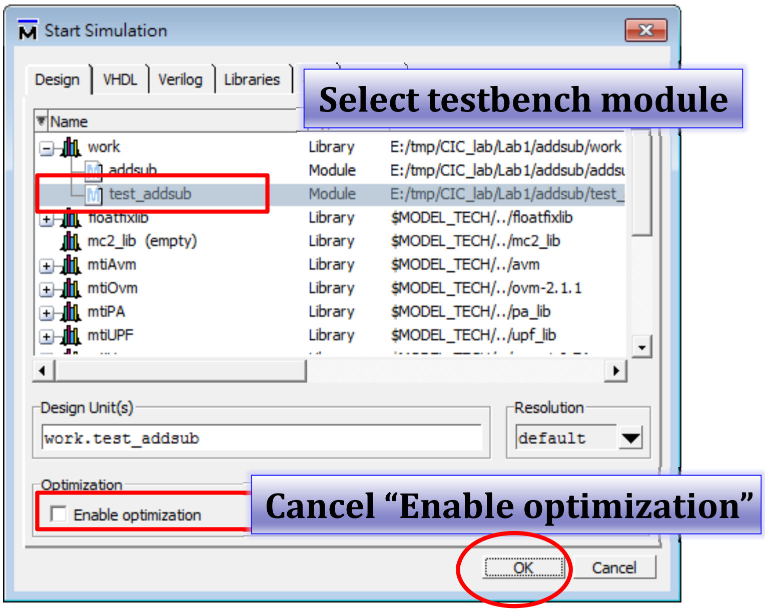
Project - E:/tmp/CIC_lab/Lab1/addsub/addsub

Name	Status	Type	Order	Modified
test_addsub.v	✓	Verilog	1	06/07/11 04:34:23 PM
addsub.v	✓	Verilog	0	01/19/10 09:01:14 PM

- ✓ Check the status

Simulation

- ✓ Simulate ➔ Start Simulation



Start Simulation

Name		
work	Library	E:/tmp/CIC_lab/Lab1/addsub/work
addsub	Module	E:/tmp/CIC_lab/Lab1/addsub/addsu
test_addsub	Module	E:/tmp/CIC_lab/Lab1/addsub/test_
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mc2_lib (empty)	Library	$MODEL_TECH/../mc2_lib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.1.1
mtiPA	Library	$MODEL_TECH/../pa_lib
mtiUPF	Library	$MODEL_TECH/../upf_lib

Design Unit(s): work.test_addsub

Resolution: default

Optimization

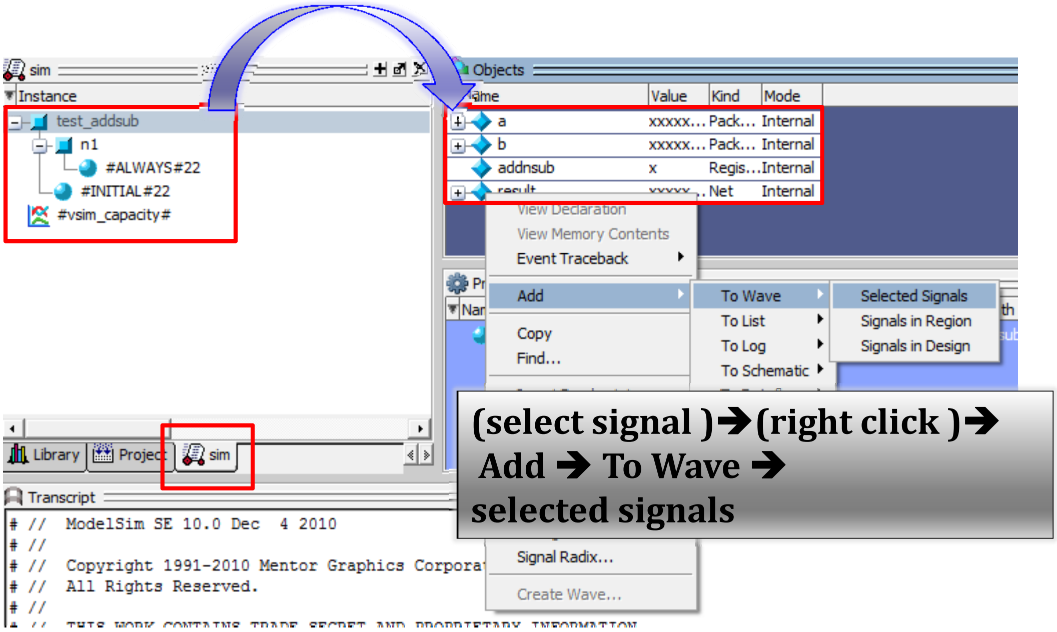
☐ Enable optimization

Select testbench module

Cancel “Enable optimization”

Add signal to Waveform window

✓ Select signal level

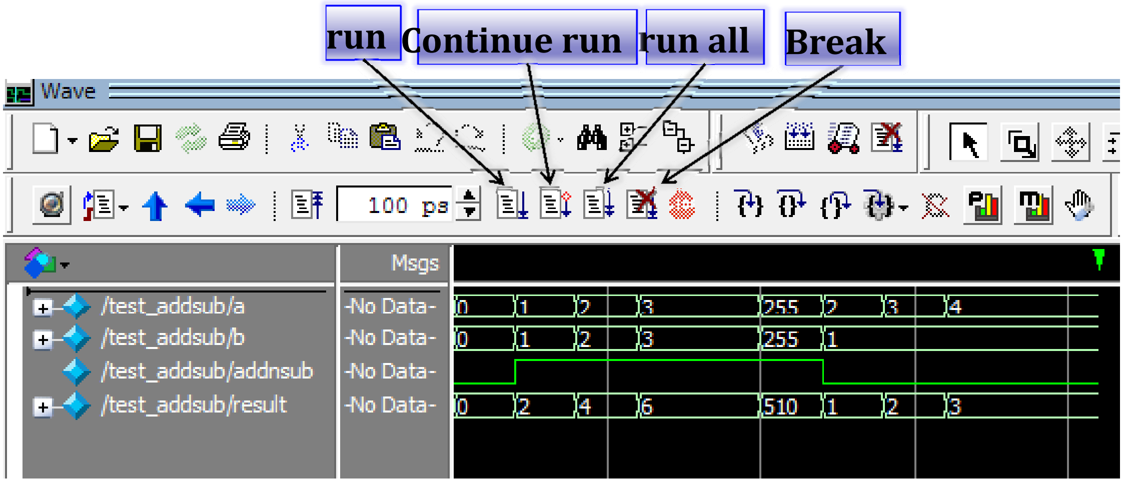


```
# //  ModelSim SE 10.0 Dec  4 2010
# //
# //  Copyright 1991-2010 Mentor Graphics Corporat
# //  All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
```

(select signal)→(right click)→ Add → To Wave → selected signals

Waveform window

✓ Waveform window


run
Continue run
run all
Break
Wave
100 ps
Msgs
/test_addsub/a
/test_addsub/b
/test_addsub/addnsub
/test_addsub/result
-No Data-
-No Data-
-No Data-
-No Data-
0 1 2 3 255 2 3 4
0 1 2 3 255 1
0 2 4 6 510 1 2 3