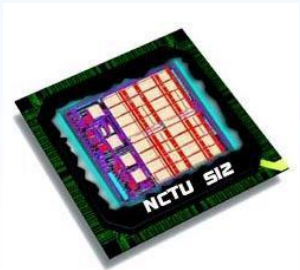


Combinational Circuits

Lecturer: Mu-Hua Yuan



Outline

- ✓ **Section 1 Introduction to design flow**
- ✓ **Section 2 Basic Description of Verilog**
- ✓ **Section 3 Behavior Models of Combinational circuit**
- ✓ **Section 4 Simulations**

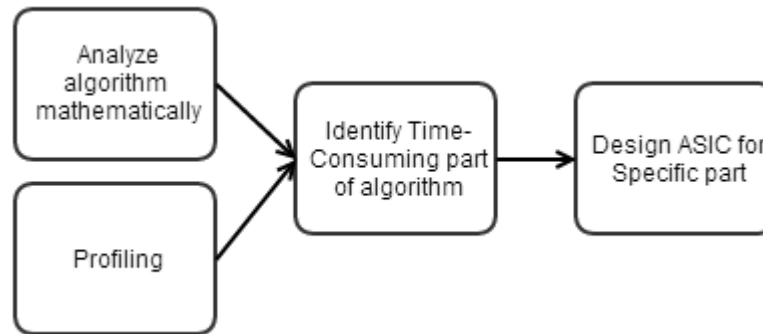


SECTION 1

INTRODUCTION TO DESIGN FLOW



How Does Hardware Accelerate System



✓ Profiling

- Profiling is a form of dynamic program analysis that measures the space/time complexity of a program to aid program optimization.
- by doing profiling we can find the most time-consuming part of the system
- designers can implement this part in hardware instead of software

✓ Application Specific IC (ASIC)

- Specially designed IC are much faster than general purpose CPU.
- we can design dedicated datapath and controller for the time-consuming part which requires less time

Example

✓ **An algorithm contains steps:**

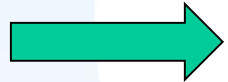
– (1) → (2) → (3) → (4)

✓ **Mathematical Analysis:**

- (1) : $O(C)$
- (2) : $O(n)$
- (3) : $O(n^2)$
- (4) : $O(n)$

✓ **Profiling**

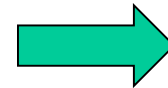
- Running 1000 times takes 100sec
- (1) : 5s
- (2) : 10s
- (3) : 70s
- (4) : 15s



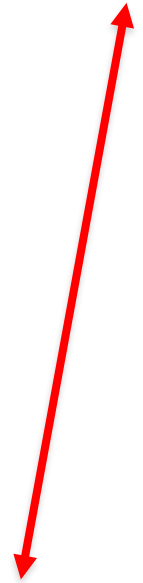
Make ASIC for (3), easily accelerated by 100x

✓ **Profiling with ASIC : Running 1000 times**

- (1) : 5s
- (2) : 10s
- (3) : 0.7s + 0.3s (communication time)
- (4) : 15s



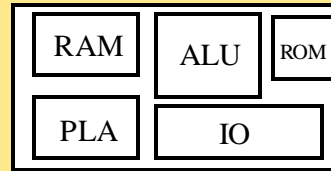
takes 31s



Introduction to Design Flow

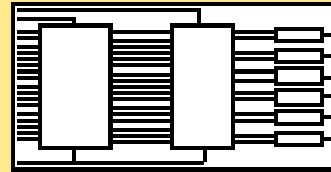
✓ Cell-based Design Flow

Architectural Level



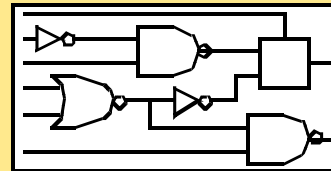
Description of circuits in large blocks and estimates of chip area
(Behavior; Area estimate)

Register Transfer Level



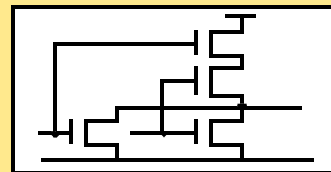
Partitioning of blocks into smaller functional modules
(Functions;)

Logic Level



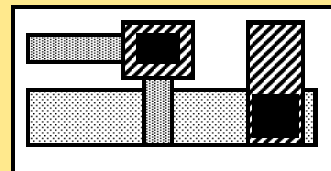
Description of blocs as logic gate and sequential elements
(Bits; Timing)

Circuit Level



Description of elements as a transistor and parasitic elements
(Voltages; Currents)

Physical Level



Description of elements as a transistor and parasitic elements
(Voltages; Currents)

Device Level Technology Level

Device modeling and electrical char. of transistors (I/V Char.)

✓ Full-Custom Design Flow



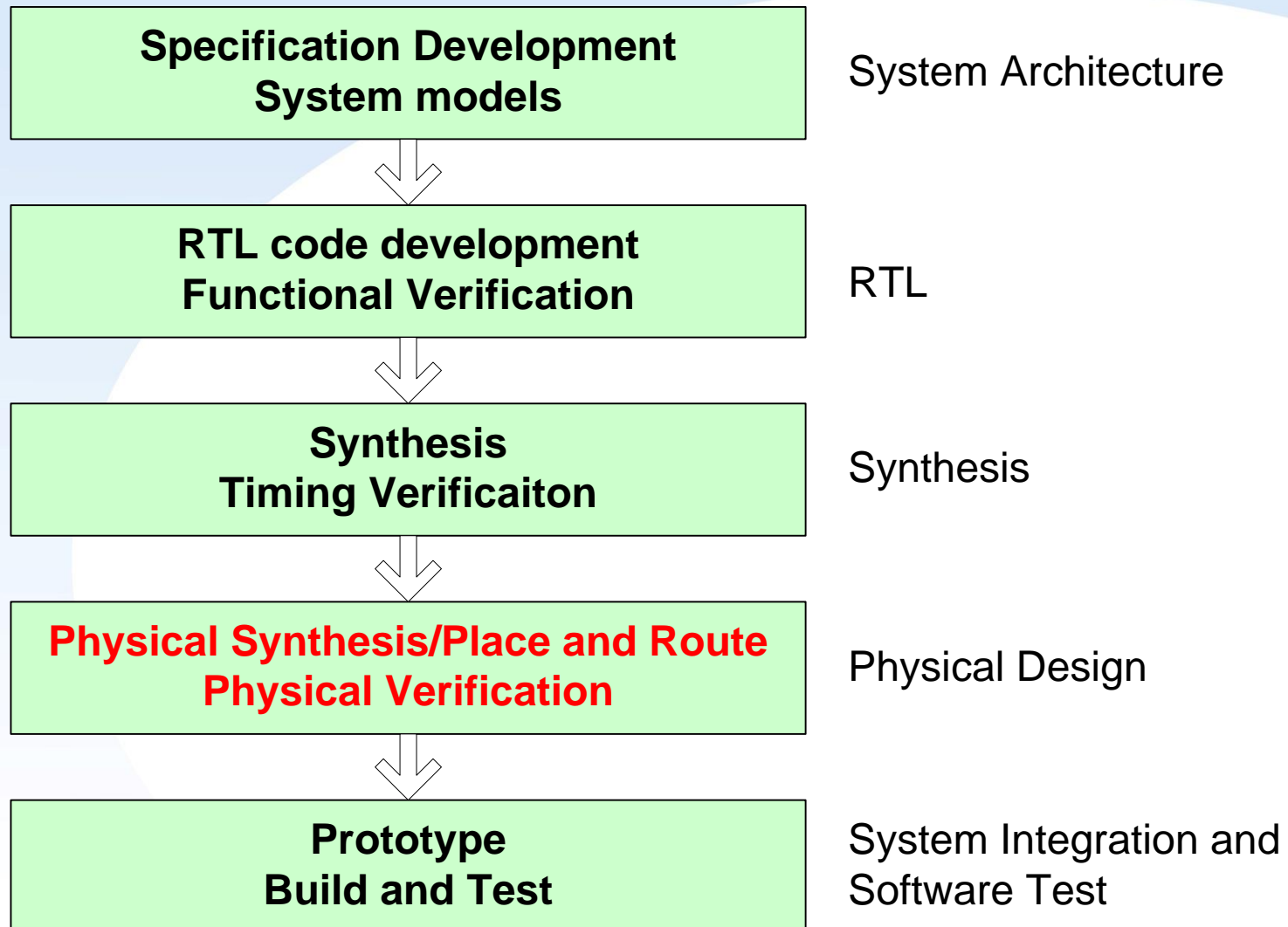
Cell-based ASIC

✓ Cell-based IC (CBIC)

- 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
- all mask layers are customized
- custom blocks can be embedded



Cell-based Design Flow



Cell-based Design Tools

✓ System and behavioral description (math. or building module)

- C/C++
- Matlab
- ...

✓ Hardware based description language

- System C
- SystemVerilog
- Verilog
- ...

✓ RTL simulation and debug

- NC-Verilog
- nLint, Verdi

✓ Synthesis and Verification

- Synopsys
 - RTL Compiler, Design Compiler
 - PrimeTime, SI and StarRC™.
- Cadence
 - BuildGates Extreme
 - Verplex (Formal Verification)
- ...

✓ Physical Design and post-layout simulation

- SoC Encounter
- IC compiler
- Calibre
- Nanosim, HSIM, UltraSim: a high-performance transistor-level FastSPICE circuit simulator ...



Full-Custom Design

- ✓ **An engineer designs some or all of the logic cells, circuits, layout specifically for one ASIC**
 - required cells/IPs are not available
 - existing cell libraries are not fast enough
 - logic cells are not small enough or consume too much power
 - technology migration (mixed-mode design)
 - ***demand long design cycle***

- ✓ **Not our focus**



SECTION 2

BASIC DESCRIPTION OF VERILOG



Basic Description of Verilog

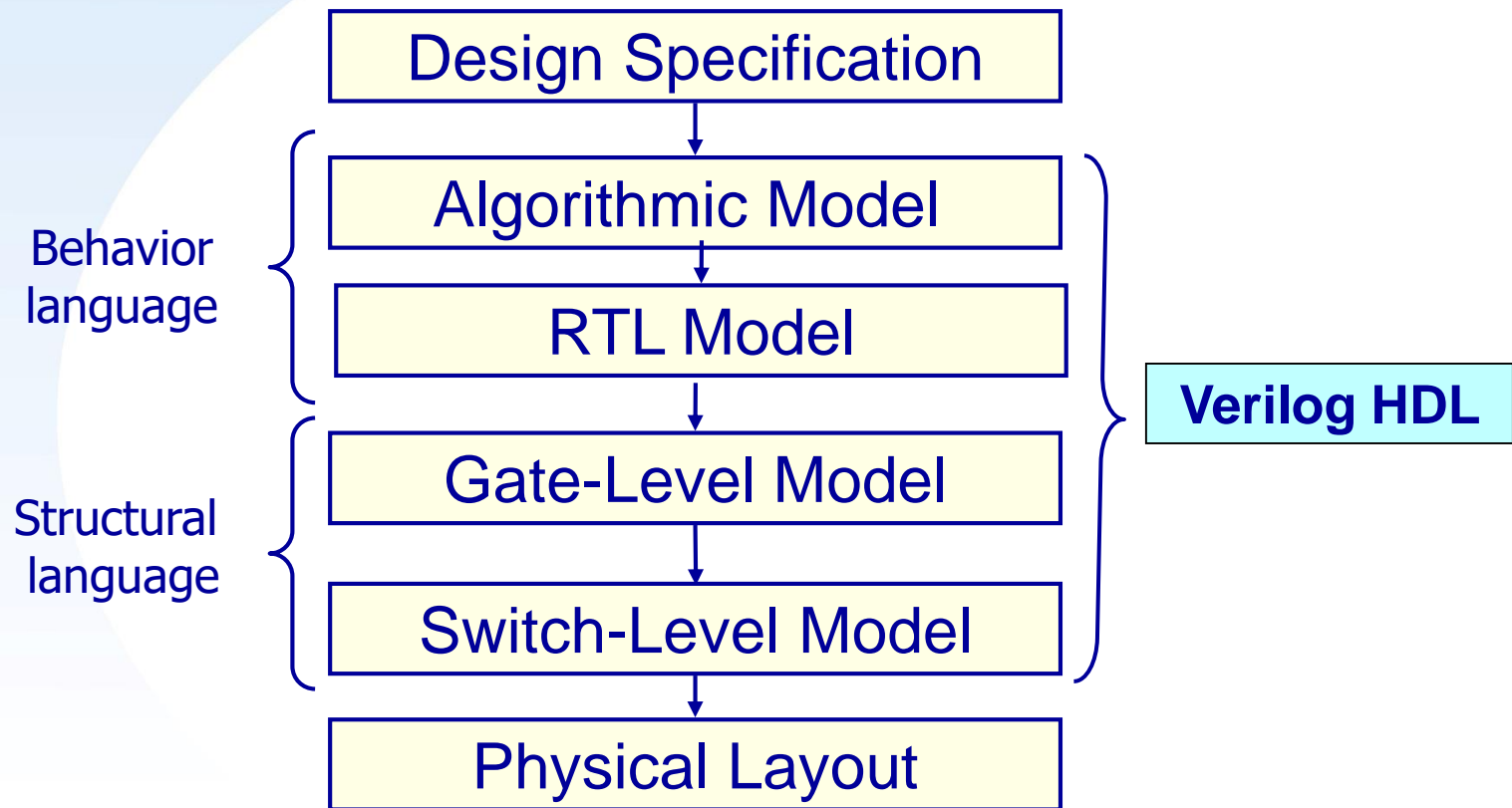
- ✓ **Design Scope of Verilog**
- ✓ **Lexical convention**
- ✓ **Data type & Port**
- ✓ **Gate level modeling**
- ✓ **Data assignment**
- ✓ **Simulation Environment**



Design Scope of Verilog

✓ We use Verilog HDL to design

- Typical design flow



What is Verilog?

✓ **Hardware** Description Language

✓ **Hardware** Description Language

✓ **Hardware** Description Language



Hardware Description Language

✓ Hardware Description Language

- HDL is a kind of language that can “describe” the hardware module we plan to design
- Verilog and VHDL are both widely using in the IC company
- ***The difference between HDL and other programming language is that we must put the “hardware circuit” in our brain during designing the modules***



A Module

- ✓ Encapsulate structural and functional details

module module_name(port_list);

port declaration

data type declaration

task & function declaration

module functionality or structure

endmodule

```
module test ( Q,S,clk );  
output reg Q;  
input  S,clk;  
  
always@(S or clk)  
    Q<=(S&clk) ;  
endmodule
```

- ✓ All modules run **concurrently**
- ✓ Encapsulation makes the model available for instantiation in other modules



Identifier and Comment

- ✓ Verilog is a **case sensitive** language
- ✓ Terminate lines with **semicolon ;**
- ✓ Identifiers
 - starts **only** with a letter or an **_**(underline), can be any sequence of letters, digits, \$, _ .
 - case-sensitive !
 - e.g. shiftreg_a
 - _bus3
 - n\$657
 - 12_reg → **illegal !!!!**
- ✓ Comments
 - single line : **//**
 - multiple line : **/* ... */**



Naming Conventions

- ✓ Consistent naming convention for the design
- ✓ Lowercase letters for signal names
- ✓ Uppercase letters for constants
- ✓ *clk* sub-string for clocks
- ✓ *rst* sub-string for resets
- ✓ Suffix
 - *_n* for active-low, *_z* for tri-state, *_a* for async , ...
- ✓ *[name]_cs* for current state, *[name]_ns* for next state
 - or *[name]_r* and *[name]_w*
- ✓ Identical(similar) names for connected signals and ports
- ✓ Consistency within group, division and corporation

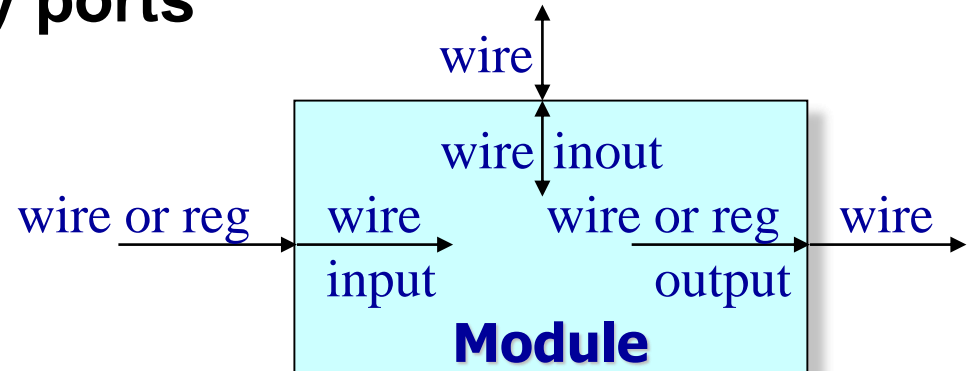


Port

✓ Interface is defined by ports

✓ Port declaration

- input : input port
- output : output port
- inout : bidirectional 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 ***[Tri-state]***

Port : Module Connection

✓ Port ordering

- one port per line with appropriate comments
- inputs first then outputs
- clocks, resets, enables, other controls, address bus, then data bus

✓ Modules connected by port order (implicit)

- Here order shall match correctly. Normally, it not a good idea to connect ports implicitly. It **could cause problem in debugging** when any new port is added or deleted.
- e.g. : FA U01(A, B, CIN, SUM, COUT);

✓ Modules connect by name (explicit)



Use this!!!

- Use **named** mapping instead of **positional** mapping
- name shall match correctly.
- e.g. : FA U01(.a(A), .b(B), .cin(CIN), .sum(SUM), .cout(COUT));
foo u_foo1(4'h2, 4'h5, 4'h8) ; X



Port : Examples

```
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,
//only reg data type can be assigned value
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
```

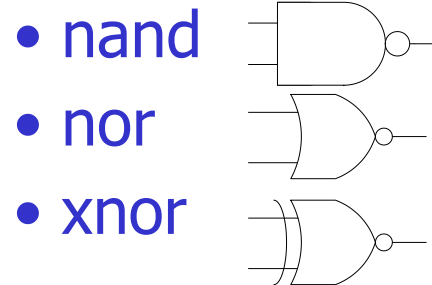
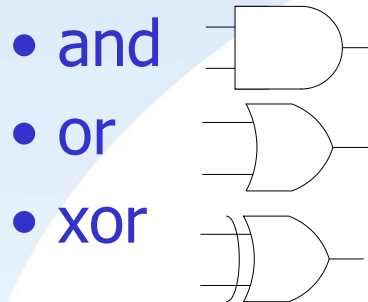
【 test module 】



Gate-Level Modeling

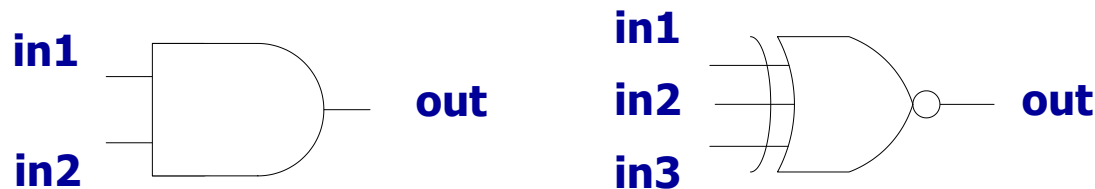
✓ Primitive logic gate

— and/or gates



-- can use without instance name → i.e. `and(out, in1, in2) ;`

-- can use with multiple inputs → i.e. `xor(out, in1, in2, in3) ;`

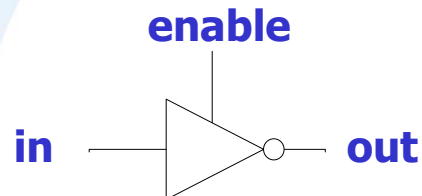


Gate-Level Modeling (cont.)

✓ Gate Delays

- Rise, Fall, and Turn-off Delays ← **Why do we need these?**

- Rise Delays
- Fall Delays
- Turn-off Delays



```
notif1 ( out, in, enable );
```

```
gate #( each_delay )
```

```
gate #( rise_delay, fall_delay )
```

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

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

```
i.e. and #(3,4) ( out, i1, i2 );
```

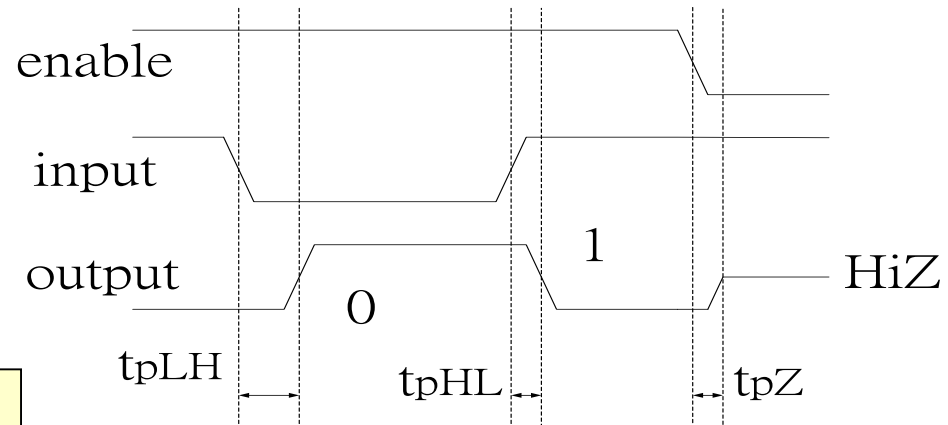
```
i.e. bufif1 #(3,4,5) ( out, in, enable );
```

```
a1( out, i1, i2 );
```

```
a2( out, i1, i2 );
```


```
//rise=3,fall=4
```

```
//rise=3,fall=4,turnoff=5
```



Gate-Level Modeling (cont.)

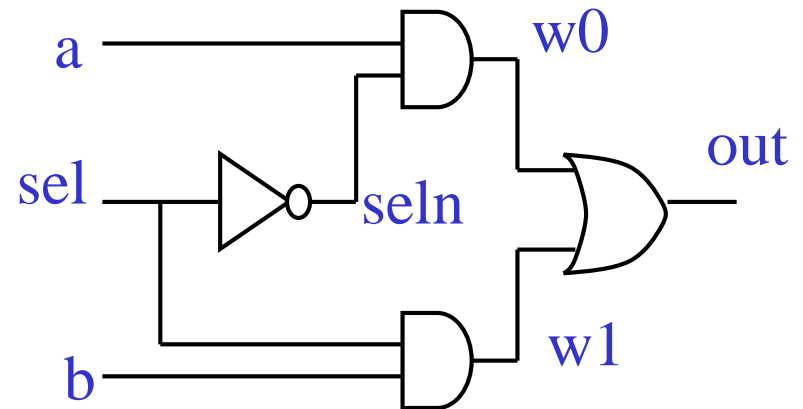
✓ Gate Delays

- Min/Typ/Max delay time  **Why do we need these?**
 - Min/Typ/Max : expectation minimum/typical/maximum delay
gate #(mindelay:typdelay:maxdelay) b(out, i1, i2);
i.e. nand #(1:2:3) (out , in1 , in2);
- Combine min/typ/max and rise/fall/turn-off delays
i.e. 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



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
```



Number

✓ 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>: 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!)
- Extension:
 - 12'hz → zzzz zzzz zzzz; 6'bx → xx xxxx; 8'b0 → 0000 0000; 8'b1 → 0000 0001

– Negative : -<size>'<base><value>

- e.g. -8'd3 → legal 8'd-3 → illegal



Operator

✓ Operators

– Arithmetic Description

- $A = B + C;$
- $A = B - C;$
- $A = B * C;$
- $A = B / C;$
- $A = B \% C;$ modulus → some synthesis tools don't support this operator

– 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**)

- $A = B >>> 2;$ " $>>>$ ", " $<<<$ " are used only for '**signed**' data type in Verilog 2001
- $A = B <<< 2;$
- e.g. $B = 4'b1000;$ ($A = 4'b1110$, which is 1000 shifted to the right two positions and sign-filled.)
 $A = B >>> 2;$



Operator

✓ Bit-wise Operator

- NOT: $A = \sim B;$
- AND: $A = B \& C;$
- OR: $A = B | C;$
- XOR: $A = B \wedge C;$
- e.g. $4'b1001 | 4'b1100 \rightarrow 4'b1101$

✓ Logical Operators: return 1-bit true/false

- NOT: $A = ! B;$
- AND: $A = B \&\& C;$
- OR: $A = B || C;$
- e.g. $4'b1001 || 4'b1100 \rightarrow \text{true}, 1'b1$

✓ Conditional Description

- if else
- case endcase
- $? : \rightarrow c = \text{sel} ? a : b;$
`// if (sel==1'b1)`
`// c = a;`
`// else`
`// c = b;`

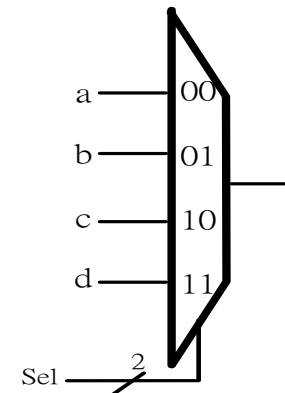
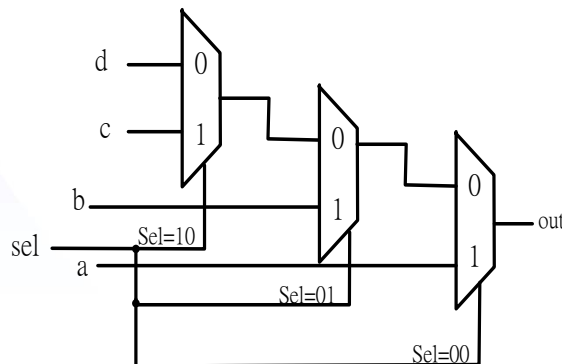
✓ Relational and equality(conditional)

- $\leq, <, >, \geq, ==, !=$
- i.e. $\text{if}((a \leq b) \&\& (c == d) || (e > f))$



If-then-else vs. case

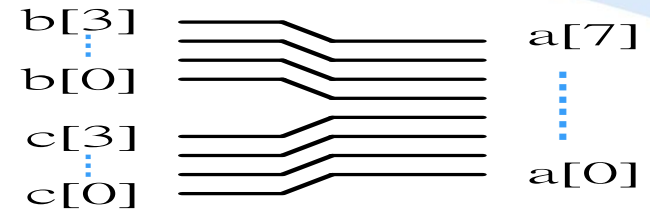
- ✓ ***if-then-else*** often infers a cascaded encoder
 - inputs signals with different arrival time
- ✓ ***case*** infers a single-level mux
 - *case* is better if priority encoding is not required
 - *case* is generally simulated faster than *if-then-else*
- ✓ ***conditional assignment* (? :)**
 - infers a mux with slower simulation performance
 - better avoided



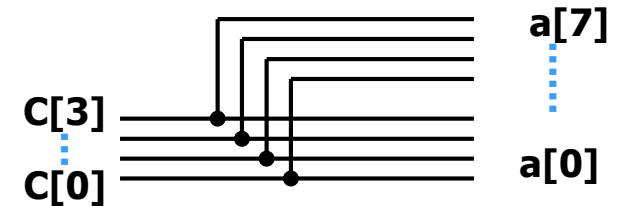
Concatenation

Concatenation

- $\{ \}$ $\rightarrow a = \{b, c\};$



- $\{\{ \}\}$ $\rightarrow a = \{2\{c\}\};$



- $a[4:0] = \{b[3:0], 1'b0\}; \Leftrightarrow a = b \ll 1;$

Data Type

- ✓ 4-value logic system in Verilog: 0, 1, x or X, z or Z
- ✓ Declaration Syntax <data_type>[<MSB> : <LSB>]<list_of_identifier>
 - **Nets** : represent physical connections between devices (**default=z**)
 - represent connections between things (ex: wire)
 - Cannot be assigned in an initial or always block
 - **Register** : represent abstract data storage element (**default=x**)
 - represent data storage (ex: reg)
 - 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

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



Data Type : Vector and Array

- Vectors : the wire and register can be represented as a vector

- wire [7:0] vec; → 8-bit bus
- reg [0:7] vec; → vec[0] is the MSB

- Arrays : <array_name>[<subscript>]

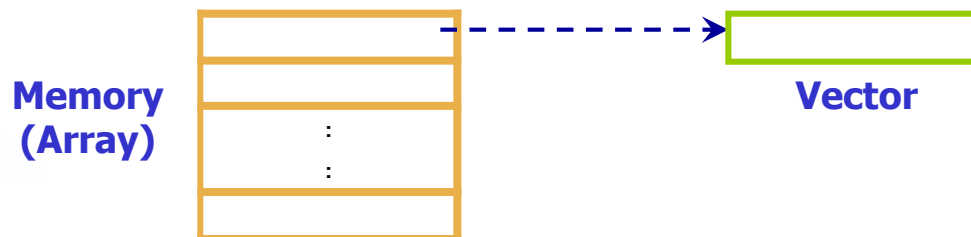
→ It isn't well for the backend verifications

- integer mem[0:7] → (8x32)-bit mem
- reg [7:0] mem[0:1023] → Memories!! (1k - 1byte)

For this reason, we do not use array as memory, Memory component will be introduced later

- What's difference between Vector and Array?

- **Vector** : single-element with multiple-bit
- **Array** : multiple-element with multiple-bit



Data Type : Signed and Unsigned

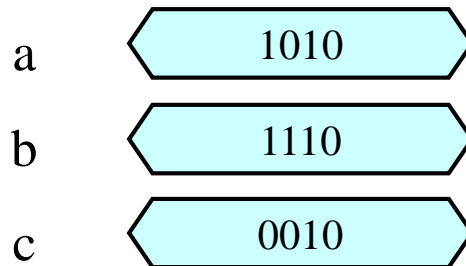
✓ Signed & Un-signed

— Verilog-1995

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

— Verilog-2001

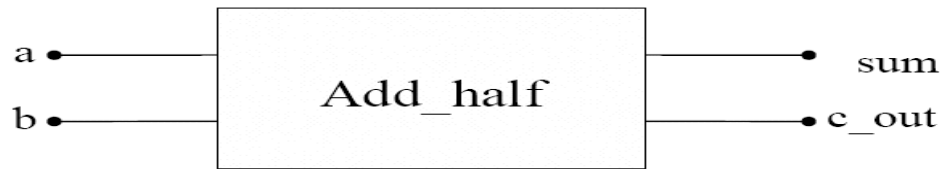
- allows **reg** variables and all net data types to be declared using the reserved keyword **signed**



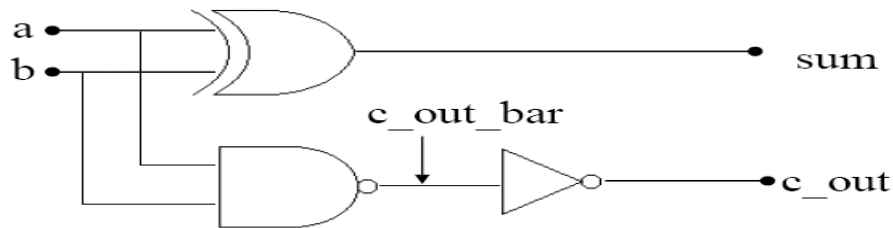
```
module verilog2001(a, b, c);  
  
output signed [3:0] b;  
output [3:0] c;  
input signed [3:0] a;  
  
assign b = a >>> 2; //arithmetic shift  
assign c = a >> 2; //bit-wise shift  
  
endmodule
```



Example



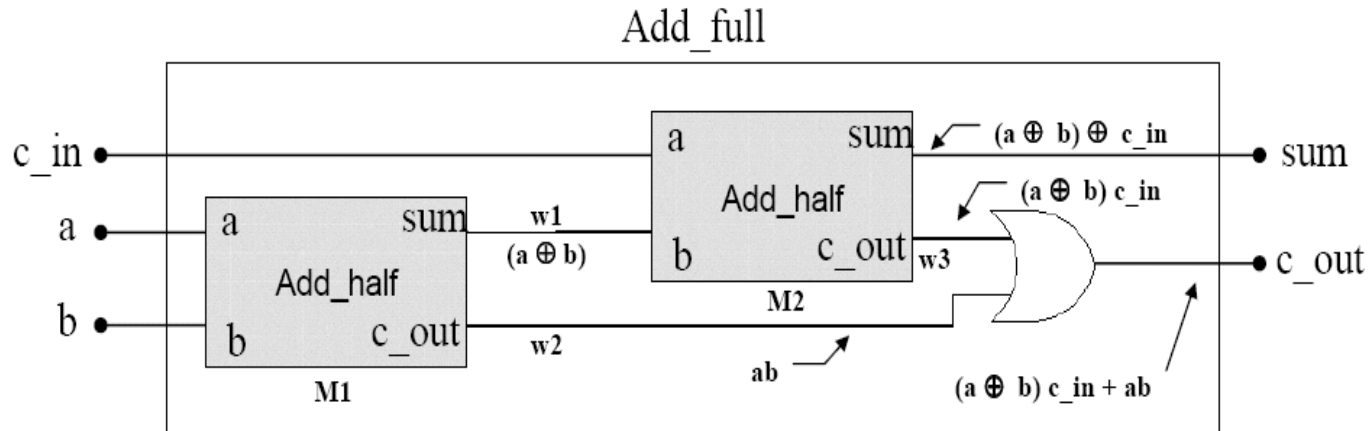
Block Diagram



Schematic

```
module Add_half(a, b, sum, c_out);  
input  a, b;  
output sum, c_out;  
wire  c_out_bar;  
xor (sum, a, b);  
and (c_out_bar, a, b);  
not (c_out, c_out_bar);  
endmodule
```

Example (cont.)



```
module Add_full(a, b, c_in, sum, c_out);  
input  a, b, c_in;  
output sum, c_out;  
wire  w1, w2, w3;  
Add_half M1(.a(a), .b(b), .sum(w1), .c_out(w2));  
Add_half M2(.a(w1), .b(c_in), .sum(sum), .c_out(w3));  
or (c_out, w2, w3);  
endmodule
```

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 : 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,  
//only reg data type can be assigned value  
always@(posedge reset or posedge clk)  
begin  
        if(reset==1'b1) out <= 0;  
        else out <= c;  
end  
endmodule
```



Data Assignment (cont.)

In the behavior-level modeling, Verilog code is just like C language. In the behavior level, it uses two essential statements.

✓ 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
```

Don't forget the corresponding hardware when writing design!



SECTION 3

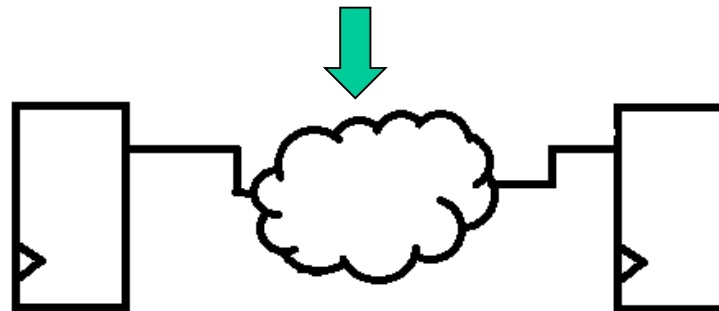
BEHAVIOR MODELS OF COMBINATIONAL CIRCUIT



Combinational Circuits

- ✓ The output of combinational circuit depends on the **present input** only.
(This property contrast to the sequential circuit which will be introduced in next chapter)
- ✓ Combinational circuit can be used to do mathematical computation and circuit control.

Combinational circuit



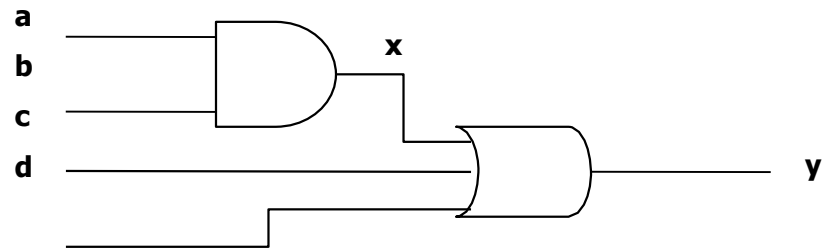
Behavioral Modeling – Combinational Blocks

✓ Using **always** construct

- assignment should be applied in **topological** order
- `always@*begin`
- `x = a & b;`
- `y = x | c | d;`
- `end`

✓ Using **continuous assignments**

- ✓ `assign y = x | c | d ;`
- `assign x = a & b ;`



Combinational Circuits – Continuous Assignment (cont.)

✓ LHS should be “wire” type

✓ Logic Assignments

- Ex: // b=5'b01110 c=5'b00101
- assign a = !b ; // a=1'b0
- assign a = ~b ; // a=5'b10001
- assign a = b & c ; // a=5'b00100

✓ Arithmetic Assignments

- Ex: // b=5'b01110 c=5'b00101
- assign a = - b ; // a=5'd -14
- assign a = b * c ; // a=10'd 70

<Example> (for logical and bitwise operators)

```
module LOG_BIT_OP(A, B, LOG1, LOG2, LOG3, BIT1, BIT2, RED1, RED2);
  input [3:0] A, B;
  output LOG1, LOG2, LOG3;
  output [3:0] BIT1, BIT2;
  output RED1, RED2;
  //A = 4'b0000;
  //B = 4'b1010;
  assign LOG1 = A && B; //LOG1 == 0
  assign LOG2 = A || B; //LOG2 == 1
  assign LOG3 = !A; //LOG3 == 1
  assign BIT1 = A & B; //BIT1 == 0000
  assign BIT2 = A | B; //BIT2 == 1010
  assign RED1 = &A; //RED1 == 0
  assign RED2 = |B; //RED2 == 1
endmodule
```

```
module ARI_OP (A, B, A_ADD_B, A_SUB_B, A_MUL_B, A_DIV_B, A_MOD_B);
  input [7:0] A, B;
  output [7:0] A_ADD_B, A_SUB_B, A_MUL_B, A_DIV_B, A_MOD_B;
  //A = 8'd3;
  //B = 8'd4;
  assign A_ADD_B = A + B; //A_ADD_B = 7
  assign A_SUB_B = A - B; //A_SUB_B = -1(8'b1111_1111)
  assign A_MUL_B = A * B; //A_MUL_B = 12
  assign A_DIV_B = A / B; //A_DIV_B = 0
  assign A_MOD_B = A % B; //A_MOD_B = 3
endmodule
```



Useful Boolean Operators

- ✓ **Bitwise operators** perform bit-sliced operations on vectors
 - $\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = 4'b1010$
 - $4'b0101 \& 4'b0011 = 4'b0001$
- ✓ **Logical operators** return one-bit (true/false) results
 - $!(4'b0101) = \sim 1 = 1'b0$
- ✓ **Reduction operators** act on each bit of a single input vector
 - $\&(4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0$
- ✓ **Comparison operators** perform a Boolean test on two arguments

Bitwise	
$\sim a$	NOT
$a \& b$	AND
$a b$	OR
$a \wedge b$	XOR
$a \sim\wedge b$	XNOR

Logical	
$!a$	NOT
$a \&\& b$	AND
$a b$	OR

Reduction	
$\&a$	AND
$\sim\&$	NAND
$ $	OR
$\sim $	NOR
\wedge	XOR

Comparison	
$a < b$ $a > b$ $a \leq b$ $a \geq b$	Relational
$a == b$ $a != b$	[in]equality returns x when x or z in bits. Else returns 0 or 1
$a === b$ $a !== b$	case [in]equality returns 0 or 1 based on bit by bit comparison

Note distinction between $\sim a$ and $!a$



Combination Circuits – always construct

- ✓ Procedural assignments update the value of **reg** variables under the control of the procedural flow.
- ✓ Updating reg, integer, time, memory variables.
 - Occur within always, initial, task and function.
 - i.e. `reg a,clk; //Note "reg", not "wire"`
 - `always#5 clk = ~clk; // procedural assignment`
 - i.e. `always @ (b) // procedural assignment with triggers`
`a = ~b;`



Combination Circuits – always construct

✓ Using blocking assignments in always construct

- ✓ The “always” block runs once
- ✓ whenever a signal in the sensitivity list changes value

```
1  a = b;  
2  out_d = 0;  
3  {carry,sum} = in + sum_in;
```

```
always@(a or b or c) begin  
    x = a & b;  
    y = x | c | d;  
end  
// simulation-synthesis mismatch  
  
always@(a or b or c or d) begin  
    y = x | c | d;  
    x = a & b;  
end // not in topological  
// simulation-synthesis mismatch  
  
always@(a or b or c or d or e)  
begin  
    x = a & b;  
    y = x | c | d;  
end  
//performance loss
```

```
always@(a or b or c or d)  
begin  
    x = a & b;  
    y = x | c | d;  
end  
// best final  
  
always@(a or b or c or d or x)  
begin  
    x = a & b;  
    y = x | c | d;  
end  
// correct
```

```
always@*  
begin  
    x = a & b;  
    y = x | c | d;  
end  
// use this!!
```

Better !!

Coding style

- ✓ Data has to be described in one always block
- ✓ Don't use initial block for synthesis
- ✓ Data bit order can't vary
- ✓ Avoid combinational loop

```
always@* begin
```

```
  A=B+C;
```

```
end
```

```
always@* begin
```

```
  A=B+D;
```

```
end
```



```
initial begin
```

```
  A=B;
```

```
  B=C;
```

```
end
```



```
reg [15:0] a;
```

```
reg [3:0] b;
```

```
assign a[b]=0;
```



```
always@* begin
```

```
  A=B;
```

```
  B=A;
```

```
end
```



Examples

```
module CORE (A,B,MODE,OUT);  
input[3:0] A,B;  
input[1:0] MODE;  
output reg signed[6:0] OUT;  
  
always@* begin  
  
    case(MODE)  
        2'b00: OUT = A+B;  
        2'b01: OUT = A-B;  
        2'b10: OUT = A*B;  
        2'b11: OUT = A/B;  
    endcase  
end  
  
endmodule
```

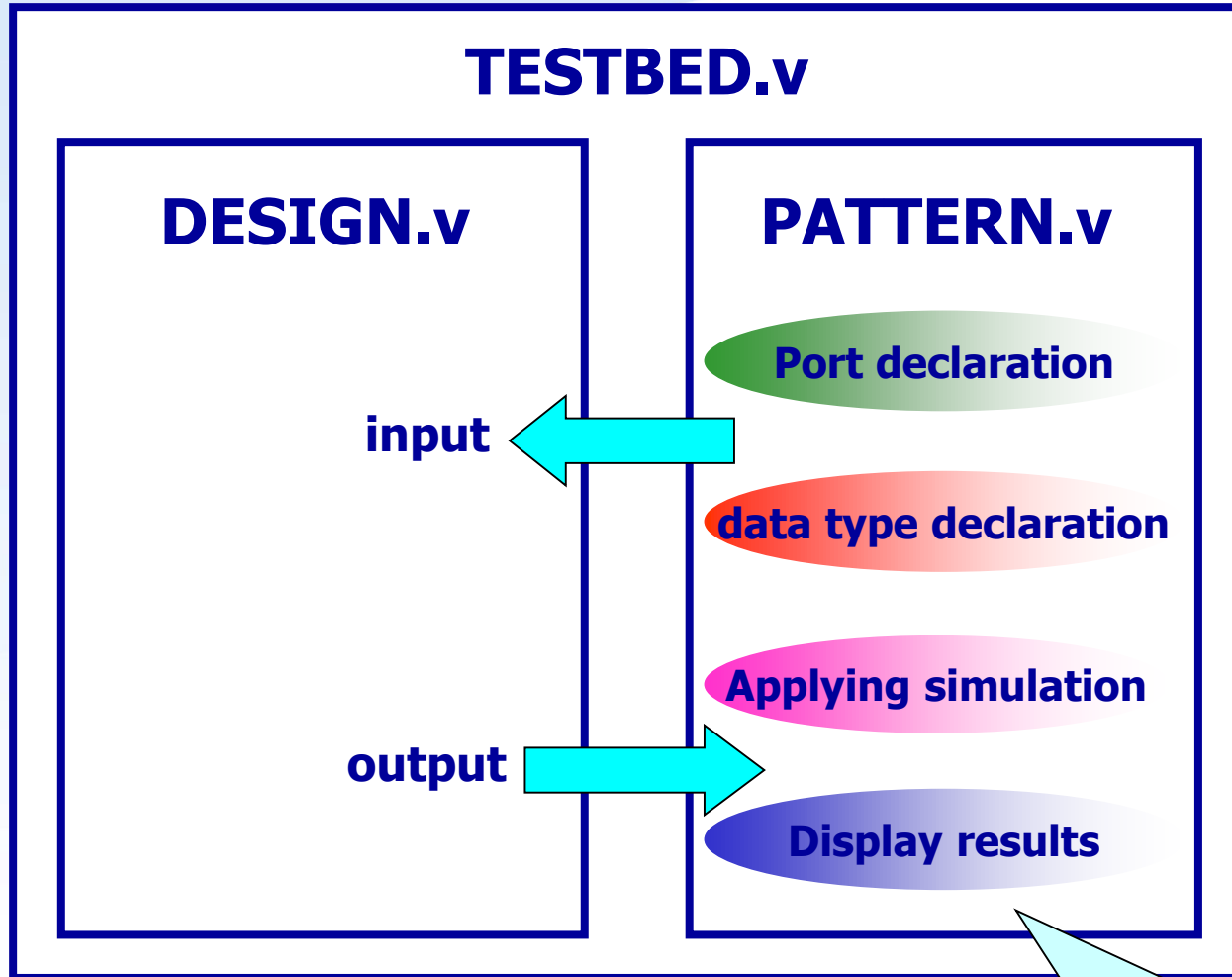


SECTION 4

SIMULATIONS



Simulation Environment



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
```

← Just like a breadboard

← Putting devices on the board and connect them together!

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
```



Simulation Environment (cont.)

✓ Dump a FSDB file for debug

– General debussy waveform generator

- `$fsdbDumpfile("file_name.fsdb");`
- `$fsdbDumpvars;`

fsdb is a file format that contains information of the waveform during the simulation

- Dump an fsdb file
- 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

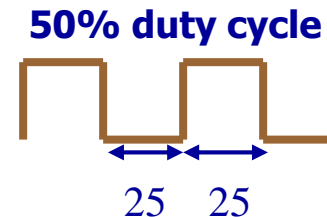
✓ Put the above command in an initial block



Simulation Environment (cont.)

✓ Clock Generation

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



✓ Display simulation result

– Texture format output

- `$display("Output #%d=%d ", count , value);`
 - Displays the values of the argument list whenever any of the arguments change except `$time`.
- `$monitor($time,"clk=%b out=%b\n",clk,out);`
 - Prints out the current values of the signals in the argument list

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

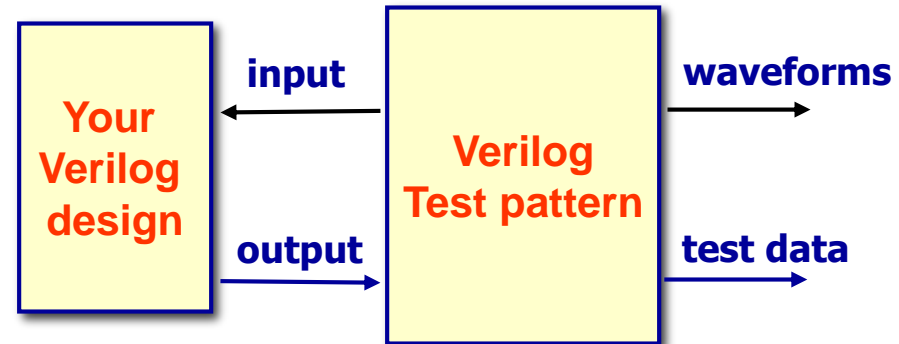
- \$random
 - \$random(seed)
- e.g. in1=\$random;
in2=\$random(37);

✓ Control command

- \$finish;
//finish the simulation
- \$stop;
//stop the simulation

✓ Simulation command

- Verilog compile
 - ncverilog test_file_name.v
 - ncverilog test_file_name.v +access+wr (dump the waveform)
- Debussy waveform generation
 - nWave &
- Stop the simulation and continue the simulation
 - Ctrl+z → Suspend the simulation at anytime you want. (not terminate yet!)
 - . → Continue if you stop your simulation by \$stop command
 - jobs → Here you can see the jobs which are processing with a index on the left [JOB_INDEX]



- kill → Use the command "kill %JOB_INDEX" to terminate the job



Overview

✓ Verdi

- An HDL Debug & Analysis tool developed by NOVAS Software, Inc. It mainly contains the following three parts.
- **nTrace** : A source code viewer and analyzer that can display the design hierarchy and source code (Verilog, VHDL, SysmVerilog, SystemC, PSL, OVA, mixed) for selected design blocks.
- **nWave** : A state-of-the-art graphical waveform viewer and analyzer that is fully integrated with Verdi's source code, schematic, and flow views.
- **nSchema** : A schematic viewer and analyzer that generates interactive debug-specific logic diagrams showing the structure of selected portions of a design.

✓ Invoke nWave

- By command : nWave &



nWave

✓ Overview

Cursor
Position

Marker
Position

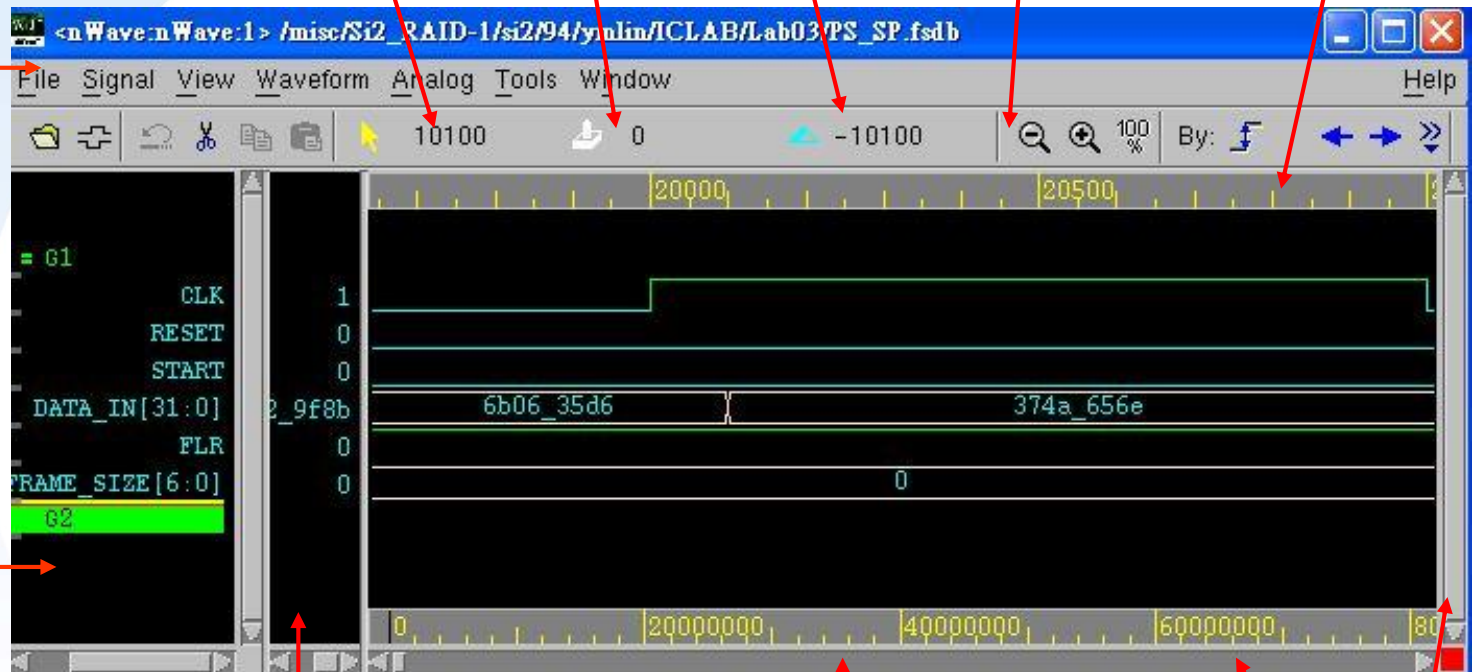
Delta

Tool Bar

Zoom Scale Ruler

Pull Down
Menu

Signal
Window



Value Window

Full Scale Ruler

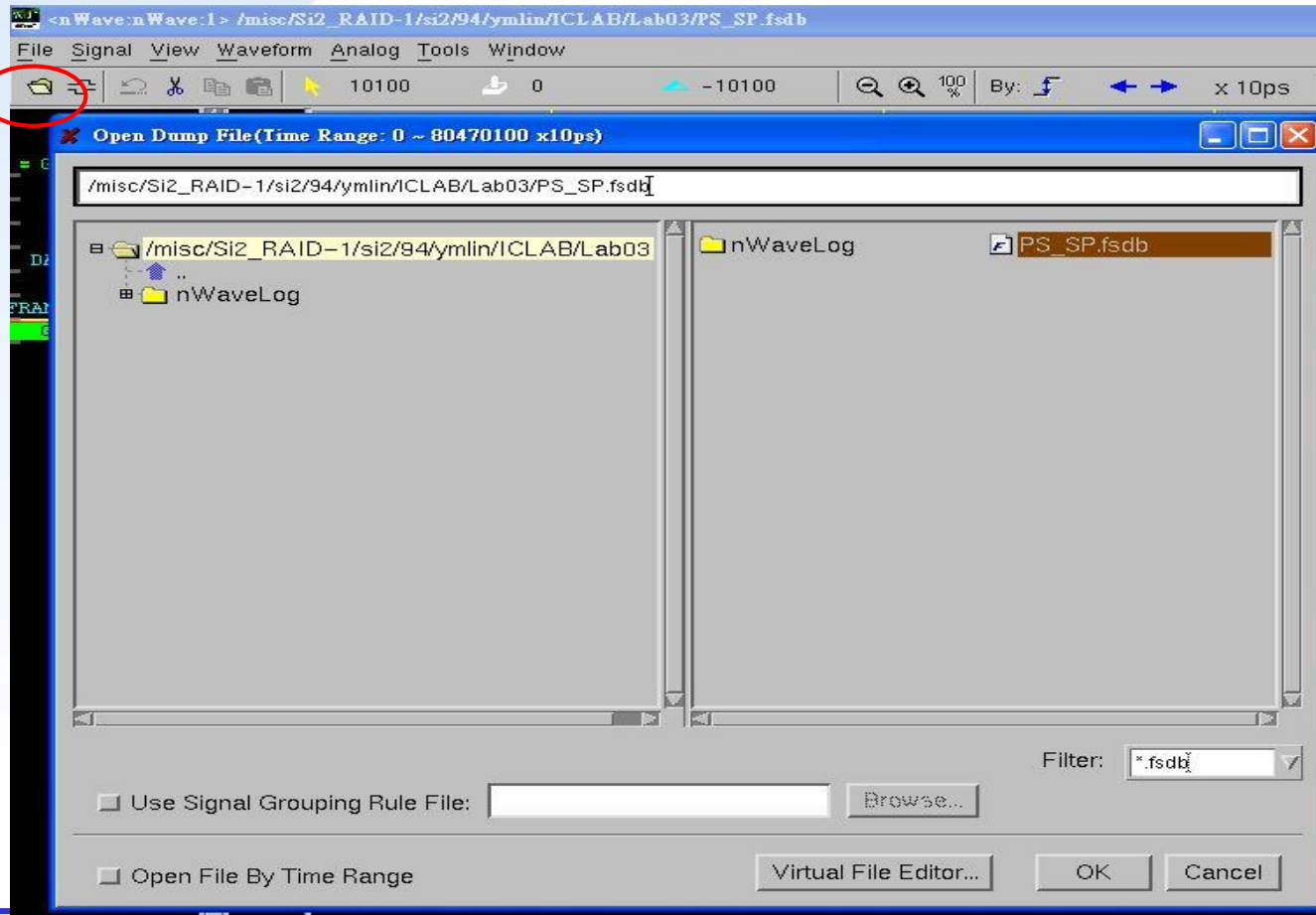
Scroll Bar



nWave (cont.)

✓ Open fsdb file

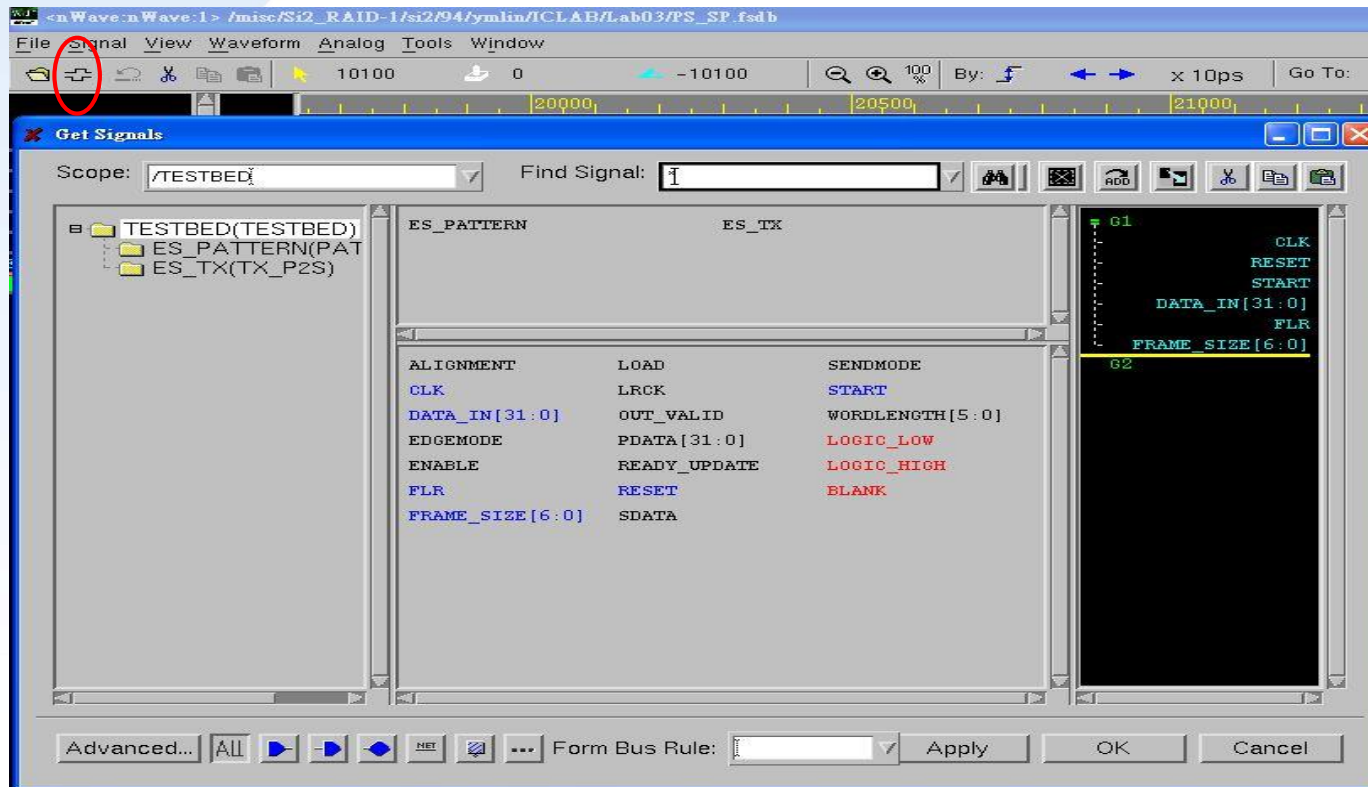
- Use **File → Open...** command



nWave (cont.)

✓ Get signal

- Use **Signal** → **Get Signals...** command

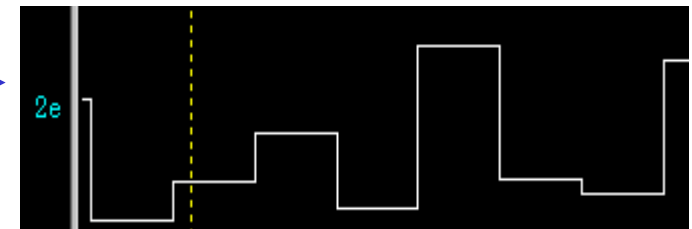
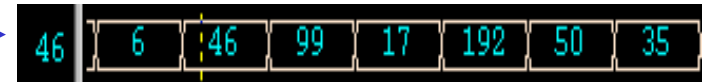
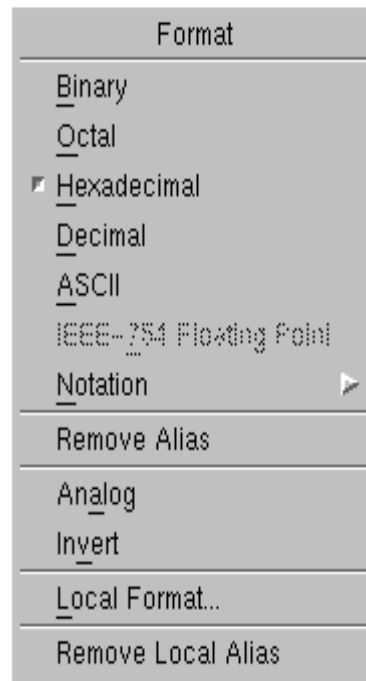


✓ Choose value format

- On the value window click Left Button



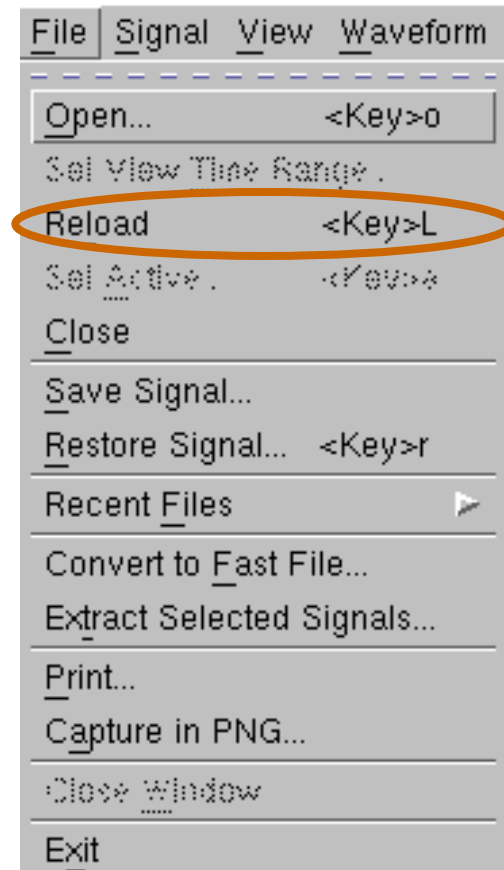
Default : Hexadecimal



nWave (cont.)

✓ Reload nWave

- Update fsdb file in Debussy database
 - **File → Reload**
 - **Hot key → L (shift + I)**



Authors

2004 Chia-Hao Lee (matchbox@si2lab.org)

2006revised Yi-Min Lin (ymlin@si2lab.org)

2008revised Chien-Ying Yu (cyyu@si2lab.org)

2008revised Chi-Heng Yang (kevin@oasis.ee.nctu.edu.tw)

2009revised Yung-Chih Chen (ycchen@oasis.ee.nctu.edu.tw)

2010revised Liang-Chi Chiu (oboe.ee98g@nctu.edu.tw)

2012revised Shyh-Jye Jou

2014revised Sung-Shine Lee (sungshil@si2lab.org)

