

# Introduction to IC Design & Combinational Circuit

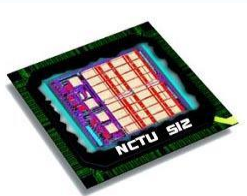
---

Lecturer: Tzu-Hsuan, Hung

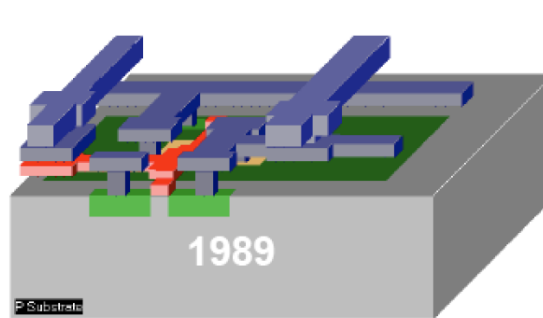
System Integration and Silicon Implementation (Si2) Lab

Institute of Electronics

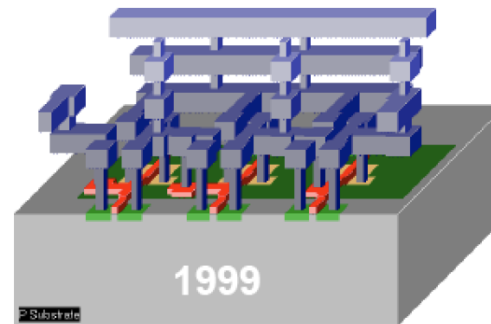
National Yang Ming Chiao Tung University, Hsinchu, Taiwan



# I ntegrated C ircuit



0.8 $\mu$ m CMOS



0.18 $\mu$ m CMOS

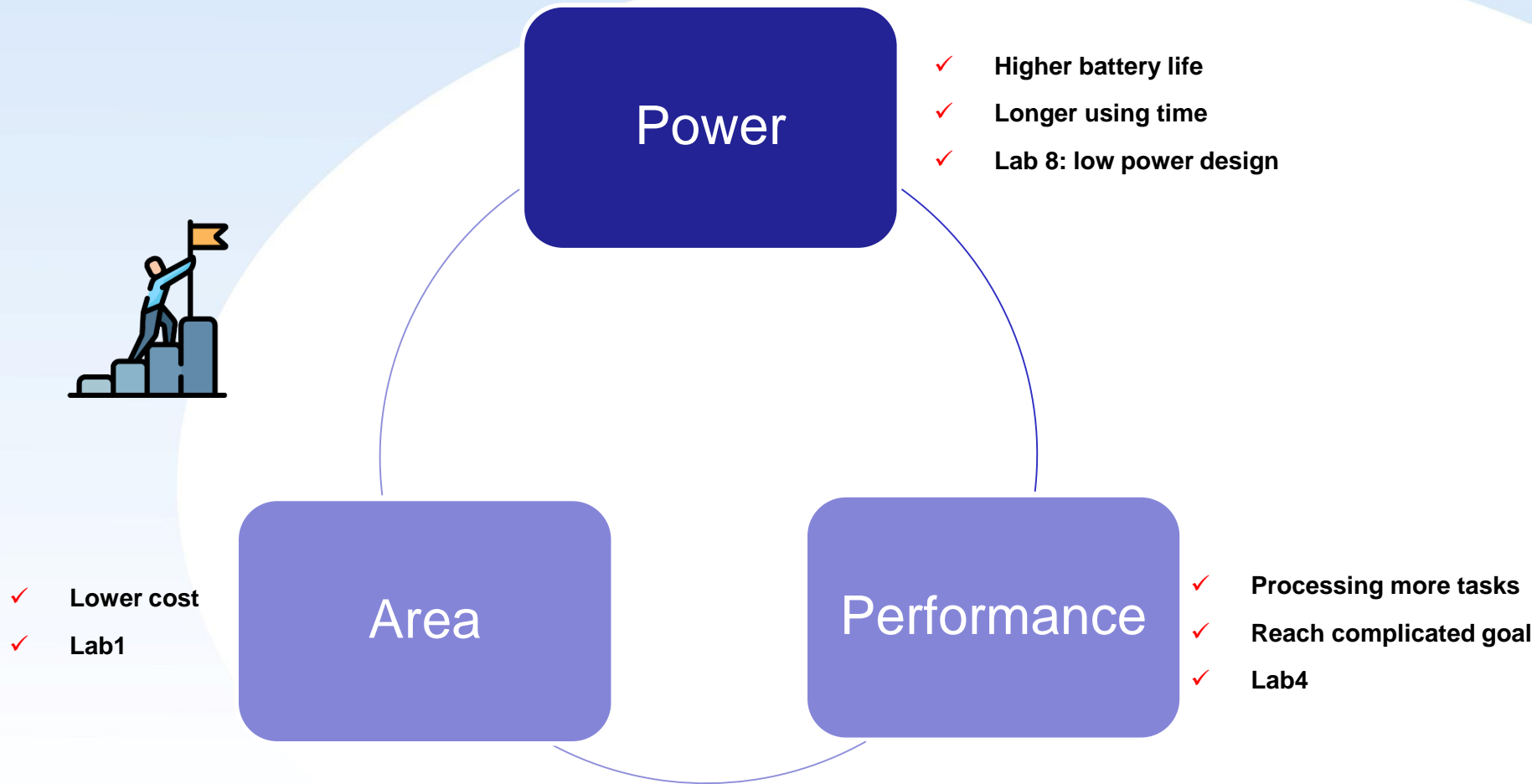
Technology:	0.8 $\mu$ m	0.18 $\mu$ m	0.07 $\mu$ m
# of Metal layers:	2~3	6	8-9
G.W. Aspect ratio (t/w):	~0.8	~1.8	~2.7
Wire length(m/chip):	~130	~1,480	~10,000

**Interconnects Start to Dominate  
Cost and Performance**

**Interconnect starts to be main  
design constraints**

Source: L.-R. Zheng, KTH

# Performance Power Area



# Outline

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



# Outline

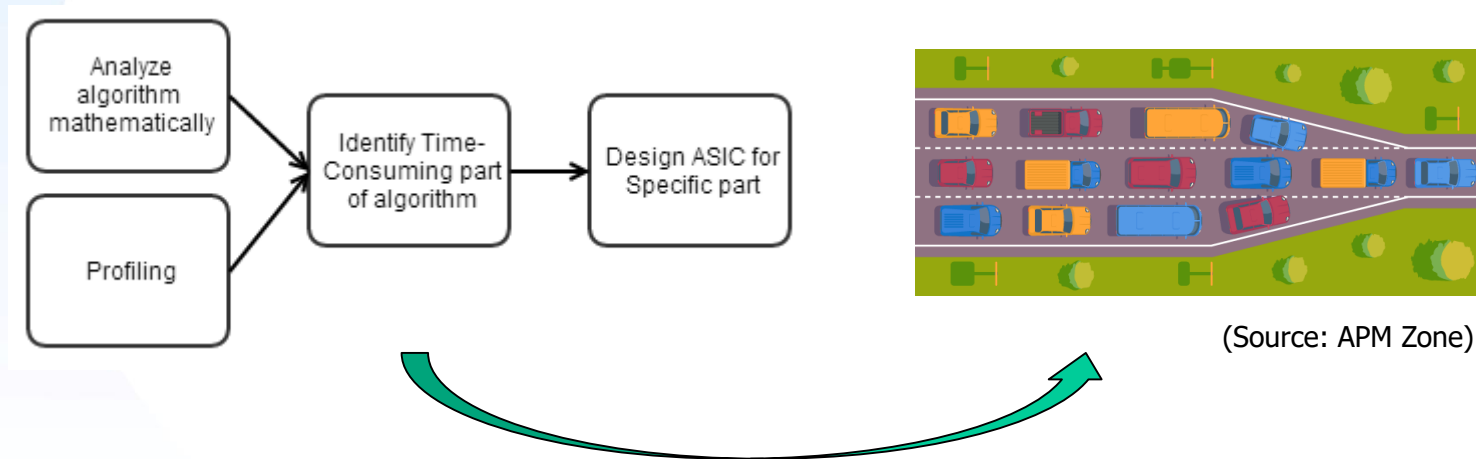
- ✓ **Section 1 Introduction to design flow**
- ✓ Section 2 Basic Description of Verilog
- ✓ Section 3 Behavior Models of Combinational circuit
- ✓ Section 4 Simulations



# 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



# How Does Hardware Accelerate System - Example

✓ **An algorithm contains steps:**

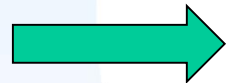
– (1) → (2) → (3) → (4)

✓ **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



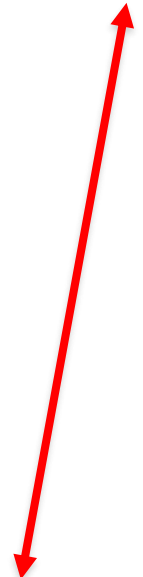
Make ASIC for (3), easily accelerated by 100x

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

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



takes 31s



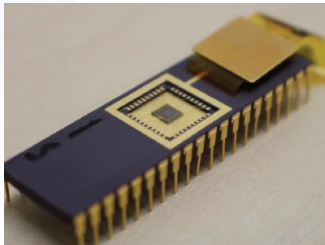
# How Does Hardware Accelerate System

## ✓ 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

## ✓ Field-Programmable Gate Array(FPGA)

- As implied by the name itself, the FPGA is field programmable.
- FPGA working as a microprocessor can be reprogrammed to function as the graphics card in the field, as opposed to in the semiconductor foundries.



(Source: sigenics)



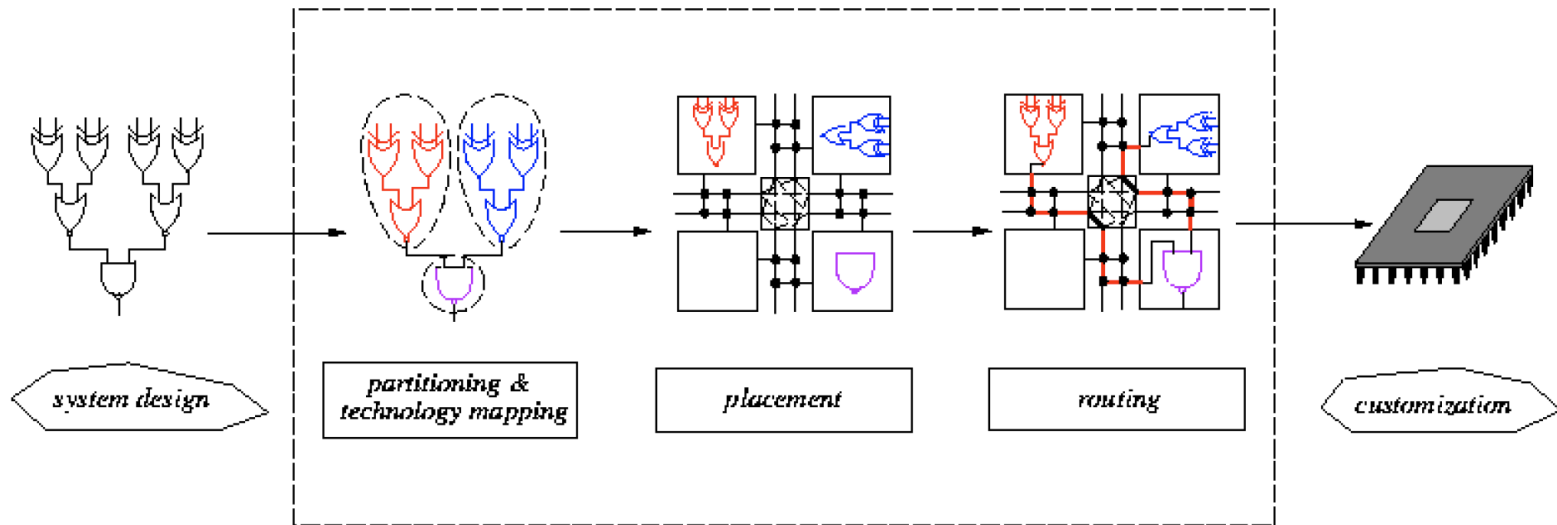
(Source: Xilinx)



# FPGA Example

## ✓ FPGA

- No fabrication is needed
- Limited routing resources



*logic + layout synthesis*

(Source: PDA, Prof. Chen)

# ASIC Example

## ✓ Cell-based Design Flow

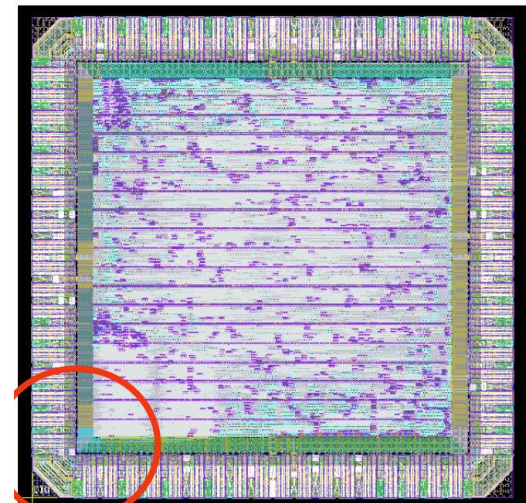
- use **pre-designed** logic cells (known as standard cells) and micro cells (e.g. microcontroller)
- designers save time, money, and reduce risk



## ✓ Full-Custom Design Flow

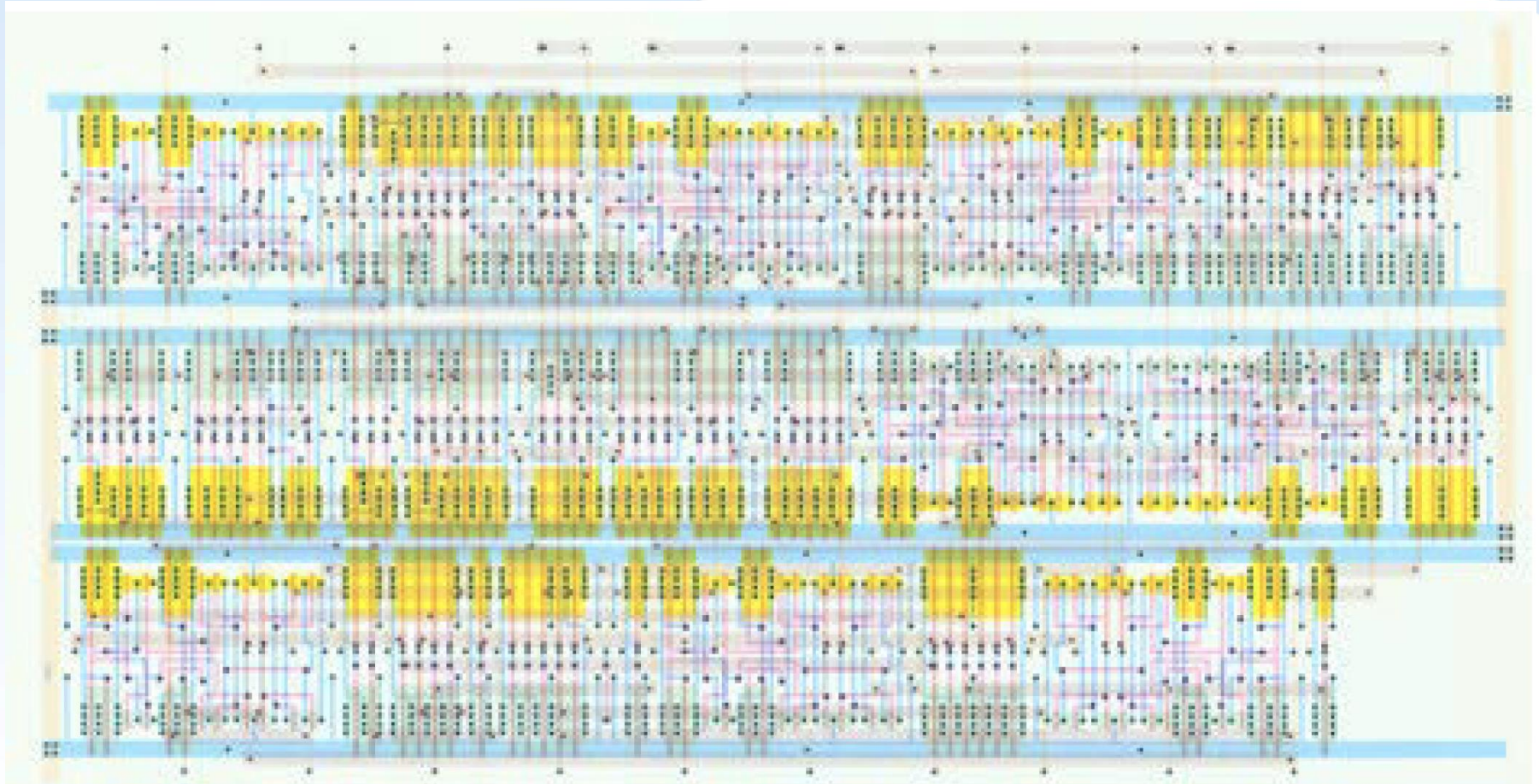
- Design every thing by yourself
- Not our focus

	Cell-based	Full-Custom
Pro.	Design speed is fast	Large design freedom
Con.	Less design freedom	Design speed is slow



(Source: TSRI)

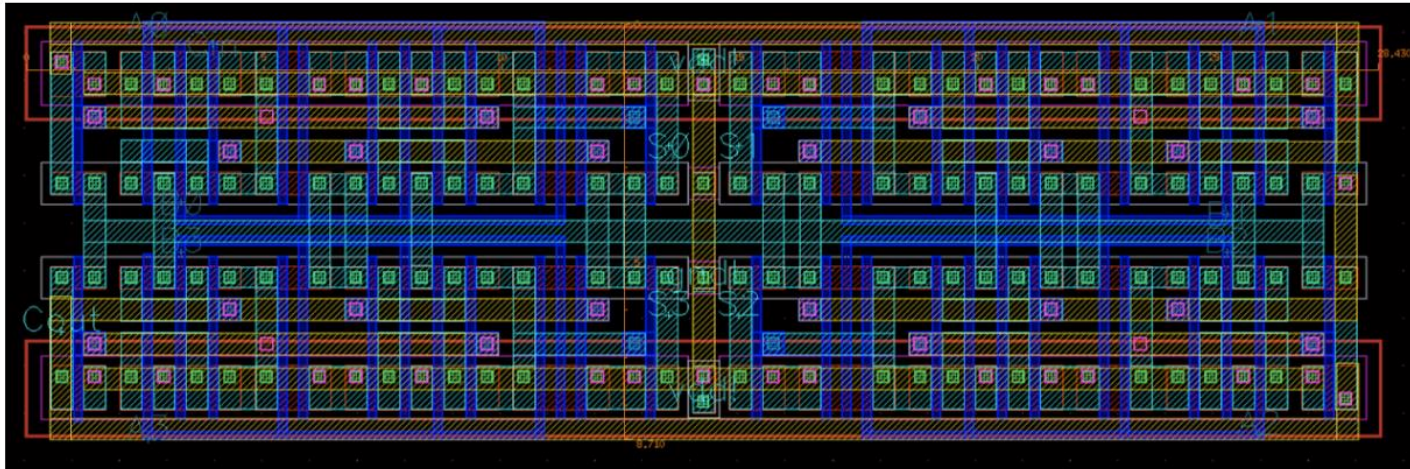
# Standard Cell Example



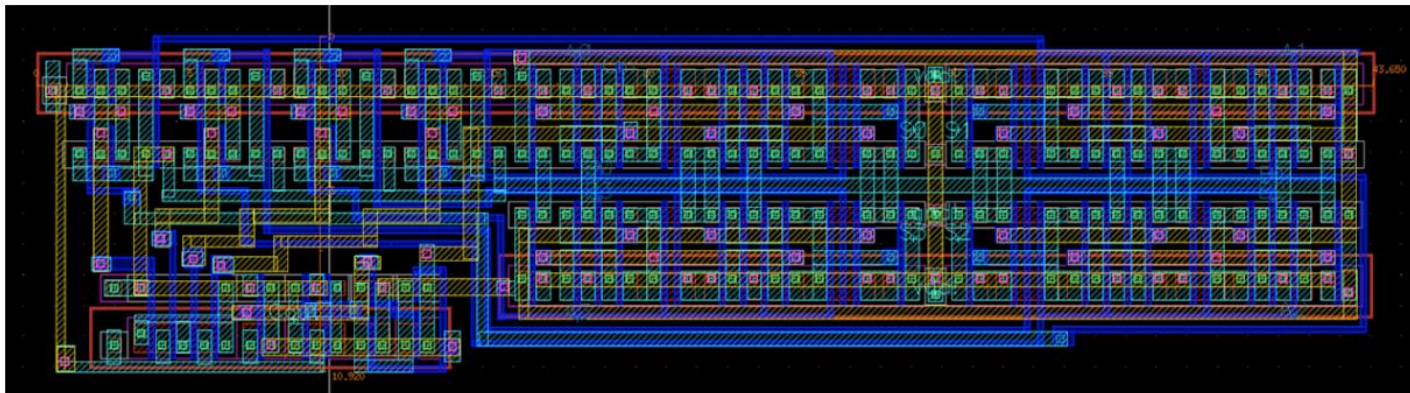


# Full-Custom Example

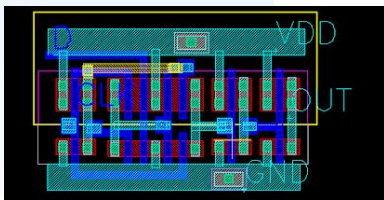
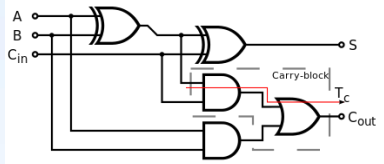
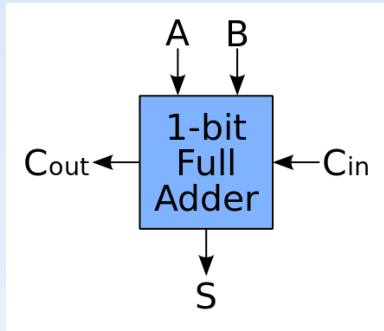
## 4-bit Carry Ripple Adder



## 4-bit Carry Skip Adder



# Cell-based Design Flow



**Specification Development**  
**System models**

System Architecture

**RTL code development**  
**Functional Verification**

RTL **1**

**Synthesis**  
**Timing Verificaiton**

Synthesis **2**

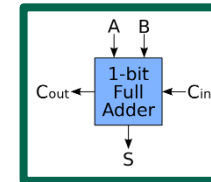
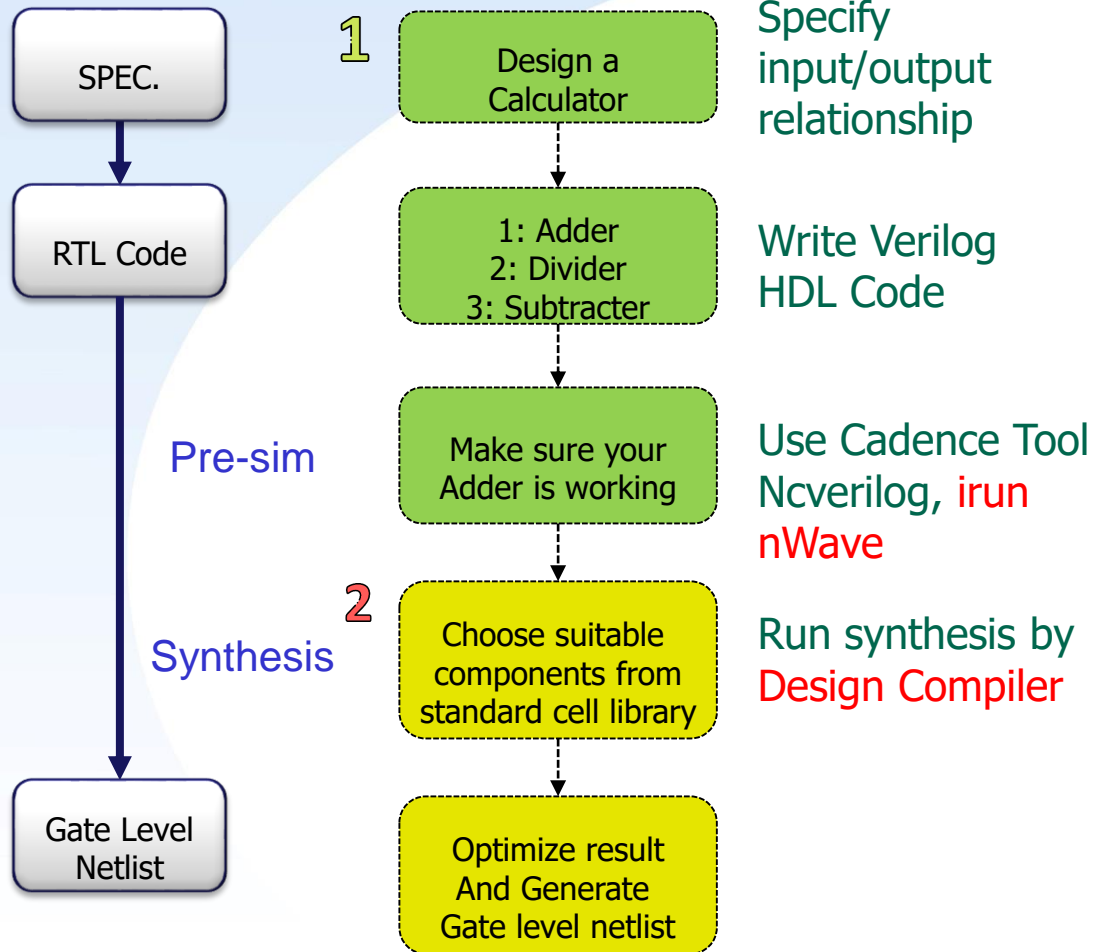
**Physical Synthesis/Place and Route**  
**Physical Verification**

Physical Design **3**

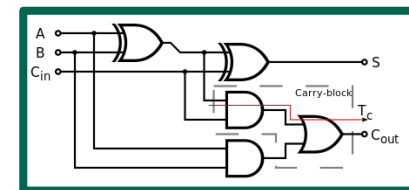
**Prototype**  
**Build and Test**

System Integration and  
Software Test

# Cell-based Design Flow - RTL to GATE



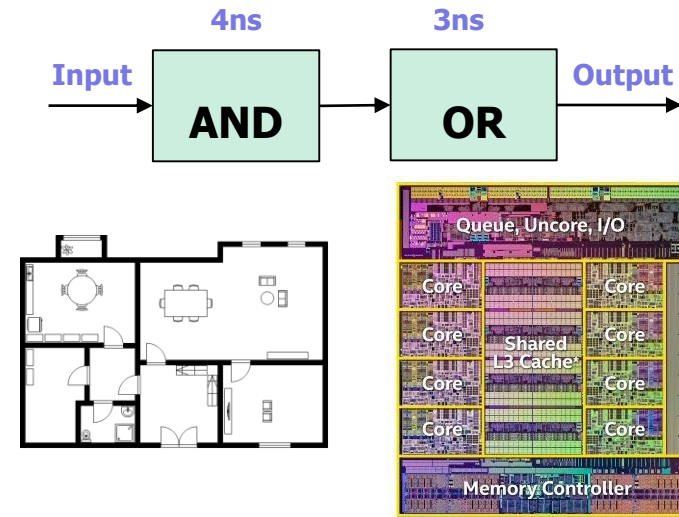
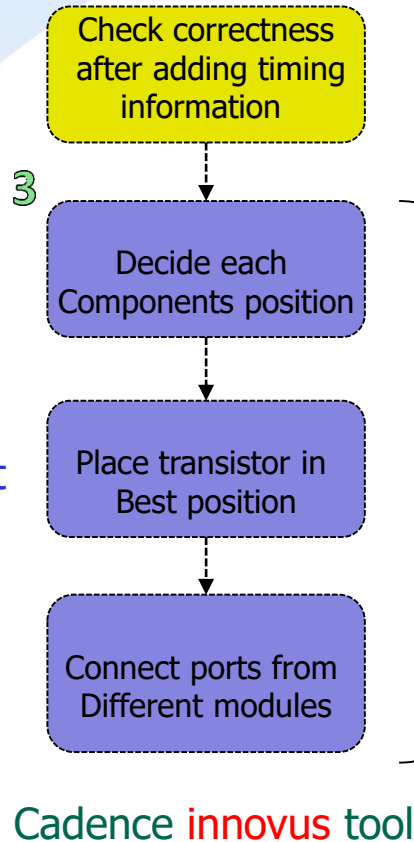
```
module adder(input a,input b,output c);
wire [31:0] a,b;
wire [31:0] c;
assign c = a + b;
endmodule
```



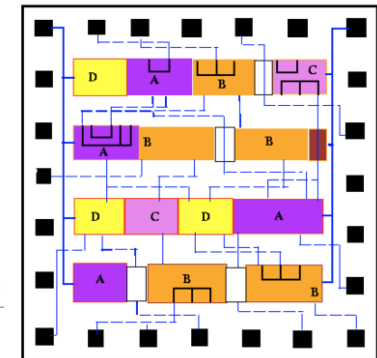
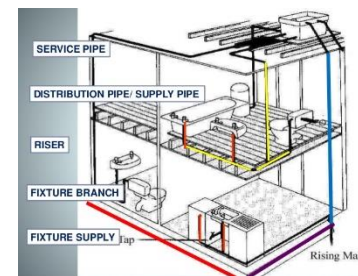
```
NOR2X1 U148 ( .A(n494), .B(n492), .Y(n349) );
AOI22X1 U149 ( .A0(n272), .A1(opt[2]), .B0(n271),
AND2X1 U150 ( .A(n235), .B(n223), .Y(n104) );
INVX2 U151 ( .A(opt[0]), .Y(n295) );
```



# Cell-based Design Flow – GATE to LAYOUT



(Source: Intel i7-5960X processor floorplan)

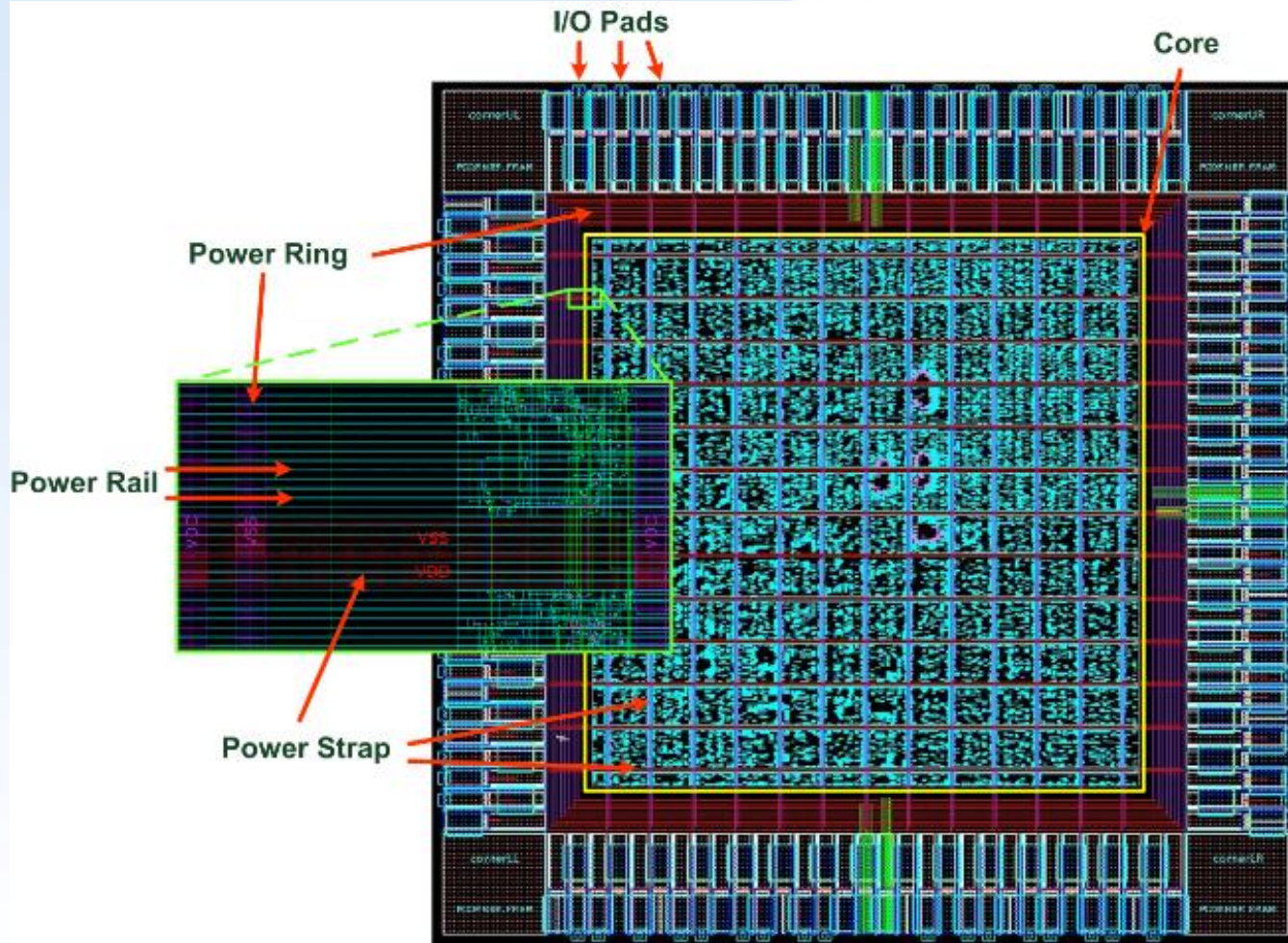


(Source: PDA, Prof. Chen)



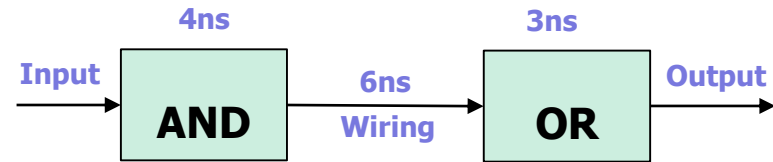
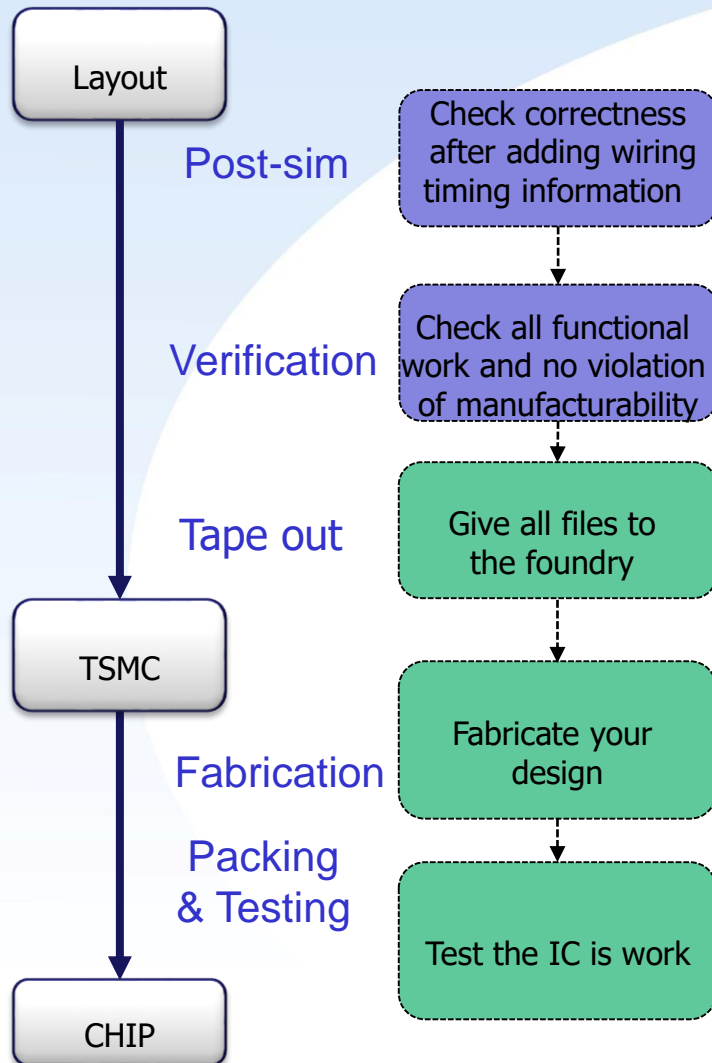


# Layout

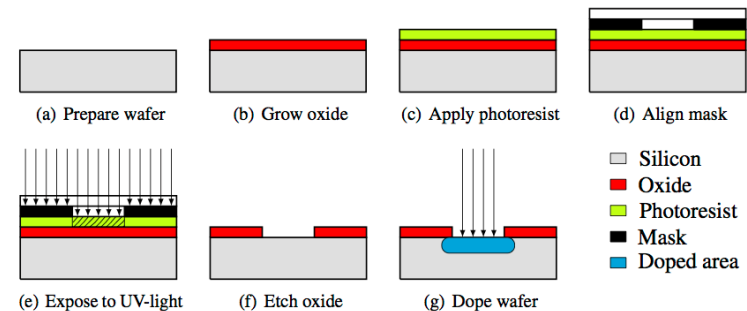




# Cell-based Design Flow – LAYOUT to CHIP



Design rule check (DRC)  
Layout verse schematic(LVS)



(Source: MTK)



# Cell-based Design Flow Summary

## RTL Design

Plan the furniture  
You want to own  
in your room

## Synthesis

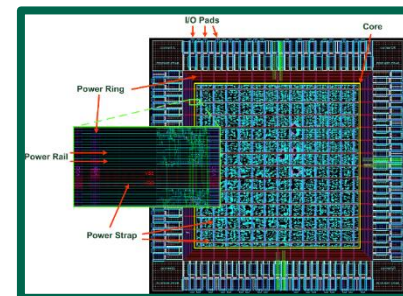
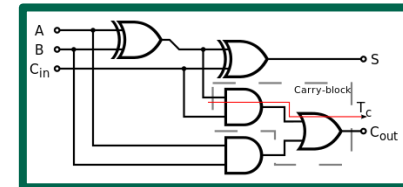
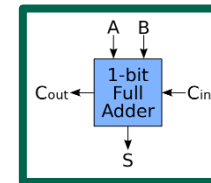
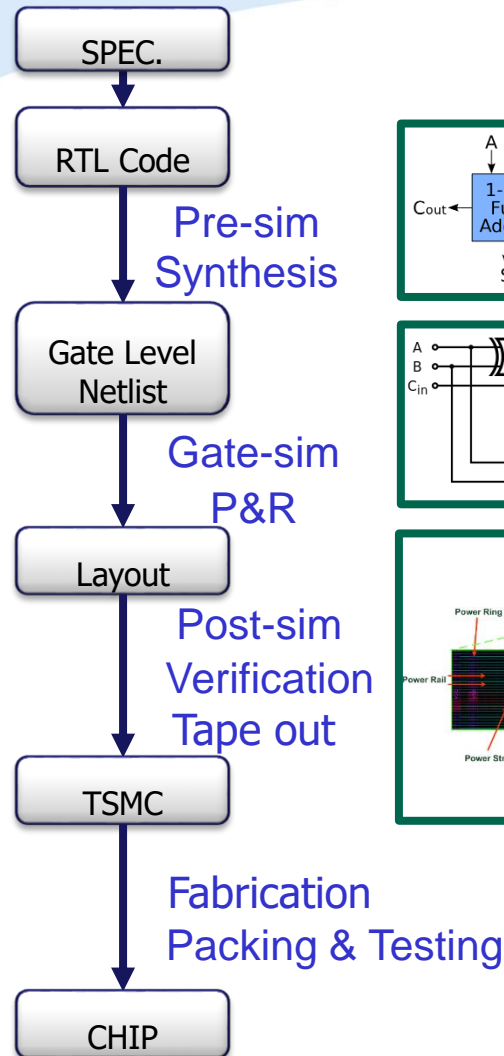
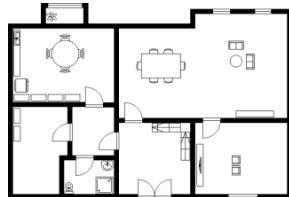
Choose the suitable  
Furniture based on  
Datasheet

## P&R

Produce Floorplan  
And layout of  
your room

## Fabrication

Construction  
And  
Get your new room



(Source: MTK)

# Cell-based Design Tools

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

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

## ✓ Hardware based description language

- System C
- SystemVerilog
- Verilog
- ...

## ✓ RTL simulation and debug

- NC-Verilog, irun
- 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

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



# Outline

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



# 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**



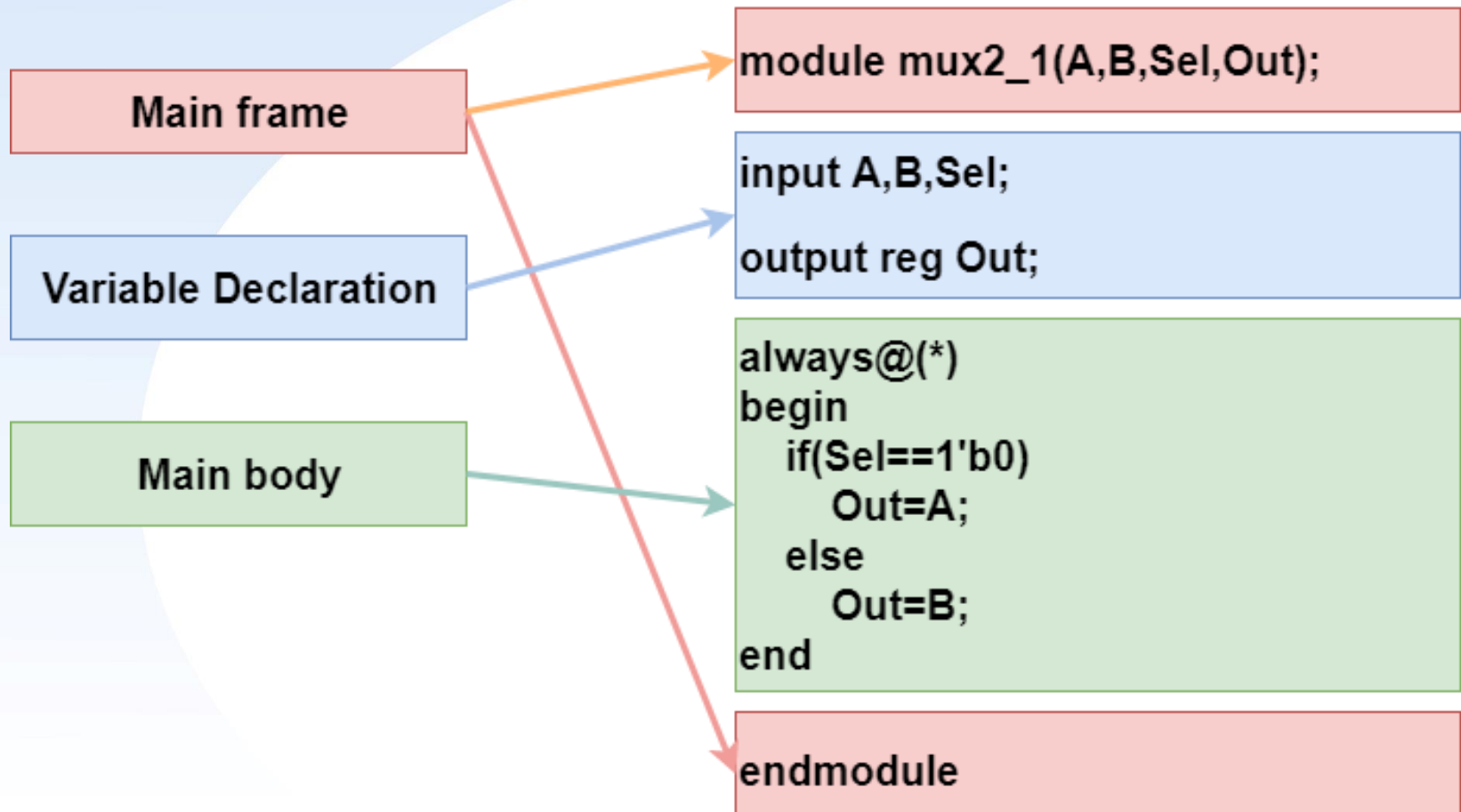
# Verilog

- ✓ **Basic Language Rules**
- ✓ **Data type**
- ✓ **Port Declare and Connect**
- ✓ **Number Representation**
- ✓ **Operators**
- ✓ **Conditional Description**
- ✓ **Concatenation**



# Module

- ✓ All modules run **concurrently**

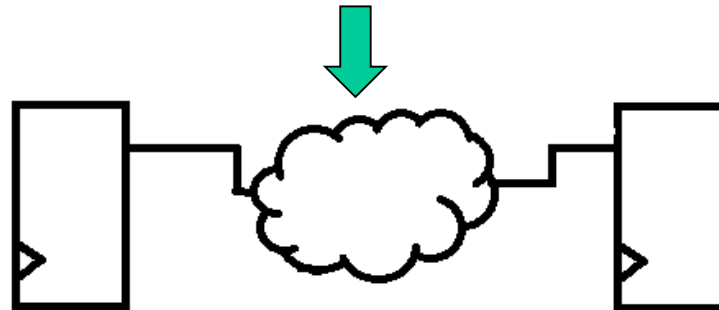




# Combinational Circuits

- ✓ The output of combinational circuit depends on the **present input** only.
- ✓ Combinational circuit can be used to do mathematical computation and circuit control.

Combinational circuit

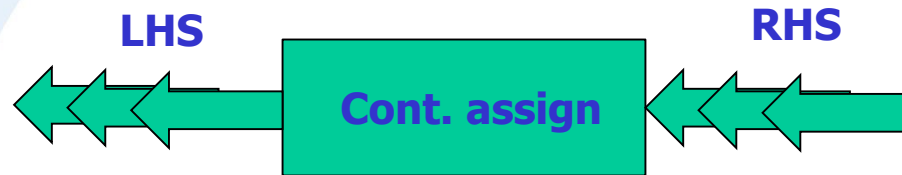


# Behavioral Modeling (1/3)

## ✓ Data Assignment

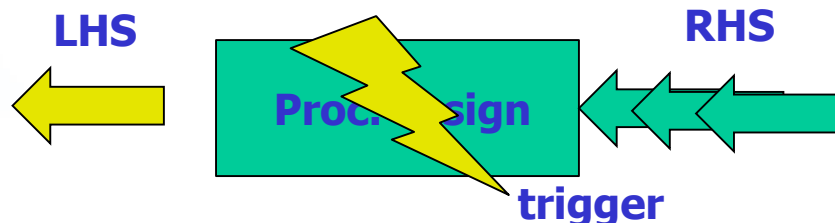
## ✓ Continuous Assignment → for wire assignment

- Imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS. => **assign**



## ✓ 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.



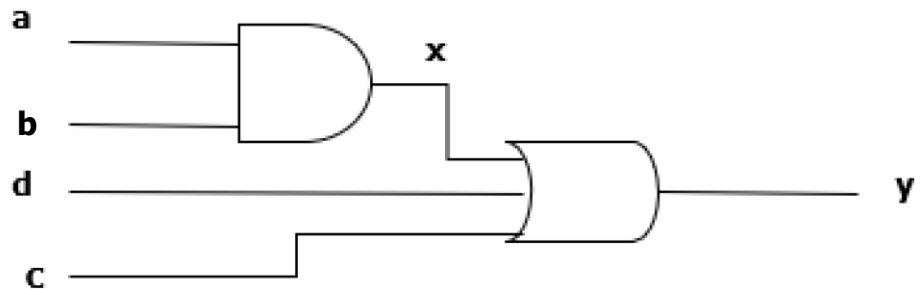
# Behavioral Modeling (2/3)

## ✓ Using **always** construct (Proc. assignment)

- assignment should be applied in **topological** order
- Simulation from top to down

– always@(a,b,~~c~~d) begin  
– x = a & b;  
– y = x | c | d;  
– end

**trigger**



## ✓ Using **assign** construct (Cont. assignments)

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

Which is better? **?**

# Behavioral Modeling: Example (3/3)

- ✓ Using blocking assignments in always construct
  - ✓ The "always" block runs once whenever a signal in the
  - ✓ **Sensitivity list** changes value (trigger)

```
always@(a or b or c) begin
    x = a & b;
    y = x | c | d;
end
// simulation-synthesis mismatch

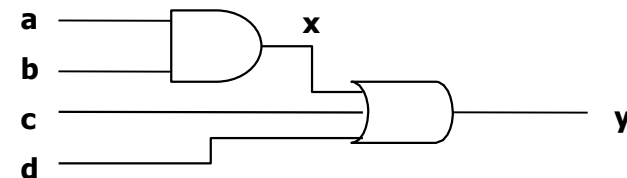
always@(a or b or c or d) begin
    y = x | c | d;
    x = a & b;
end // not in topological
// simulation-synthesis mismatch

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!!
```



# Basic Language Rules

- ✓ **Terminate lines with **semicolon ;****
- ✓ **Identifiers**
  - Verilog is a **case sensitive** language
    - C\_out\_bar and C\_OUT\_BAR: two different identifiers
  - Starts only with a letter or an \_(underline), can be any sequence of letters, digits, \$, \_ .
    - e.g. 12\_reg → **illegal !!!!**
- ✓ **Comments**
  - single line : //
  - multiple line : /\* ... \*/



# Naming Conventions

## ✓ Common

- Lowercase letters for **signal** names
- Uppercase letters for **constants**

## ✓ Abbreviation

- **clk** sub-string for clocks
- **rst** sub-string for resets

## ✓ Suffix

- **\_n** for active-low, **\_z** for tri-state, **\_a** for async , ...
- Ex: **rst\_n** => reset circuit at active-low

## ✓ State Machine

- **[name]\_cs** for current state, **[name]\_ns** for next state

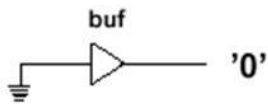
## ✓ Identical(similar) names for connected signals and ports



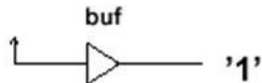
# Data Type (1/5)

## ✓ 4-value logic system in Verilog: 0, 1, X, or Z

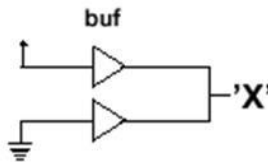
- **0,1:** means low or high signal
- **X:** unknown signal, means that we don't know whether the signal is 0 or 1
- **Z:** high-impedance, the signal is neither 0 nor 1.
- **Avoid X and Z !!!**



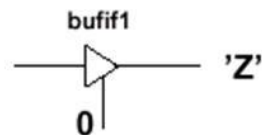
'0', Low, False, Logic Low, Ground, VSS,  
Negative Assertion



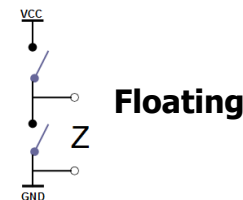
'1', High, True, Logic High, Power,  
VDD, VCC, Positive Assertion



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



HiZ, High Impedance, Tri- Stated,



Floating

# Data Type (2/5)

## ✓ **Wire** (default = **Z** (high impedance, floating net))

A wire cannot store value, often use in combination circuit

### 1. Represent port connection between devices

```
wire clk,A,B;  
BBQ bbq1(.clk(clk),.meat(A),.vegetable(B))  
BBQ bbq2(.clk(clk),.meat(A),.vegetable(B))
```

### 2. Can not be use in procedure assignment: 'initial' or 'always'

```
wire C;  
always@(*) begin  
    C = a+b; // wrong, C should be reg data type (X)  
end
```

### 3. Only use in continuous assignment: 'assign'

```
wire C;  
assign C = a+b; // correct (O)
```





# Data Type (3/5)

## ✓ Registers (default = **X** (unknown, should be initialized) )

A reg is a simple Verilog, variable-type register  
represent abstract data storage element

Hold their value until explicitly assigned in an initial or always block

1. Only use in procedure assignment: 'initial' or 'always'
2. Does not imply a physical register

- EX1

```
reg C;  
always@(*) begin  
    C = a+b;           // (O) This reg does not imply a physical register  
end
```

- EX2

```
reg C;  
Always@(posedge clk) begin  
    C <= a+b;          // (O) This reg imply a physical register  
end
```



# Data Type (4/5)

## Wire

1. Port connection(in/out)
2. Assign (cont. assignment)
3. Can declared as vector
4. Often use in Comb. circuit

## Reg

1. Port output(register out)
2. Always block (Proc. assignment)
3. Can declared as vector
4. Often use in Sequ. circuit



# Data Type (5/5)

## ✓ Vectors and Arrays : the **wire** and **reg** can be represented as a vector

- Vectors: single-element with multiple-bit
  - wire [7:0] vec; → 8-bit
- Arrays: multiple-element with multiple-bit
  - It isn't well for the backend verifications
  - 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

# Port Declare and Connect (1/3)

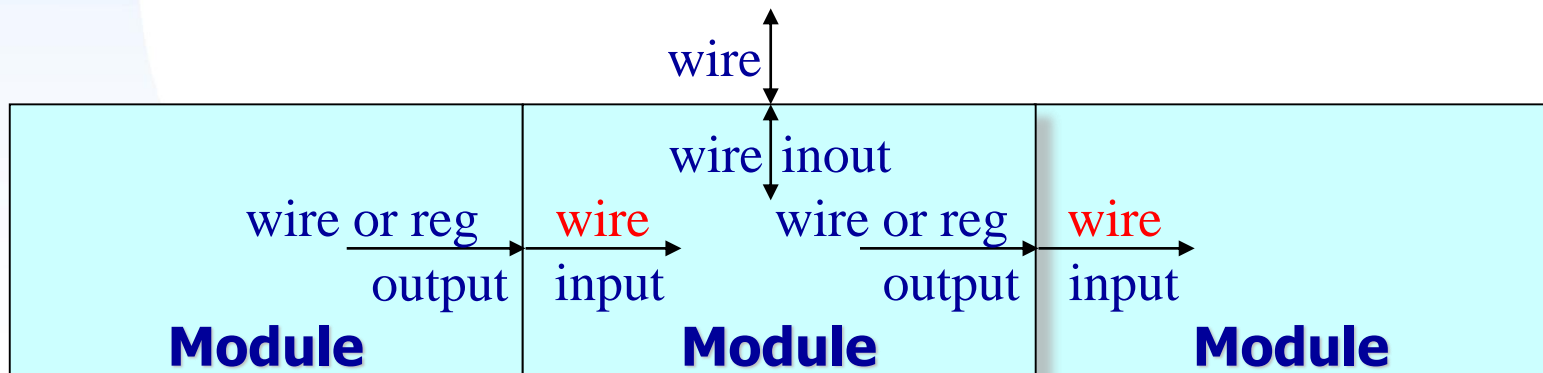
## ✓ Interface is defined by ports

### – Port inside module declaration

- ◆ **input** : input port, **only wire** can be assigned
- ◆ **output** : output port, **wire/reg** can be assigned
- ◆ **inout** : bidirectional port, **only wire** can be assigned

### – Port connection outside module

- ◆ **input** : wire or reg can be assigned to connect into the module
- ◆ **output** : only wire can be assigned to connect out of the module
- ◆ **inout** : register assignment is forbidden neither in module nor out of module [Tri-state]



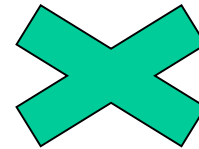
# Port Declare and Connect (2/3)

## ✓ Modules connected by port order (implicit)

- Order must 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 );`

**Order is vital!**



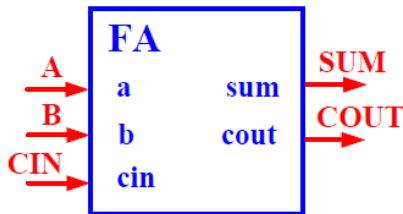
## ✓ Modules connect by name (explicit)



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) );`

**Name Mapping**



# Port Declare and Connect : Example (3/3)

```
module MUX2_1(out,a,b,sel,clk,rst);  
input      sel,clk,rst;  
input      a,b;  
output     out;  
wire       c;  
reg        a,b;  
reg        out;
```

**Wire for input**

//incorrect define

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

```
//Procedural assignment,  
//only reg data type can be assigned value  
always@(posedge rst or posedge clk)  
begin
```

```
    if(reset==1'b1) out <= 0;  
    else out <= c;
```

```
end  
endmodule
```

**【 sub module 】**

```
`include "mux.v"  
module test;  
reg        out;  
reg        a,b;  
reg        clk,sel,rst;
```

**Wire for port connection**

//incorrect define

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

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

```
initial begin
```

```
.....
```

```
end  
endmodule
```

**【 test module 】**



# Number Representation (1/2)

## ✓ Number Representation

### – Format: <size>'<base><value>

- Base format: b(binary), o(octal), d(decimal) or h(hexadecimal)
  - ◆ e.g. 4'd10 → 4-bit, 10, decimal
- If <size> is smaller than <value>, left-most bits of <value> are truncated
  - ◆ e.g. 6'hca → 6-bit, store as 6'b001010 (truncated, not 11001010!)
- If <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'
  - ◆ e.g. 12'hz → zzzz zzzz zzzz; 6'bx → xx xxxx;  
8'b0 → 0000 0000; 8'b1 → 0000 0001;
  - ◆ e.g. 6'ha → 6-bit, store as 6'b001010 (filled with 2-bit '0' on left!)
- Default size is 32-bits decimal number
  - ◆ e.g. 11 => 32'd11 (integer type)



# Number Representation(2/2)

## ✓ Number Representation

### – Signed Value (Verilog-2001)

- By default the signal is unsigned → Declare with keyword “signed”
  - ◆ e.g. wire **signed** [7:0] a;
- Negative : -<size>'<base><value>
  - ◆ e.g. -8'd3 → legal, 8'd-3 → illegal
  - ◆ A 3-bit signed value would be declared as wire signed [2:0] A

Decimal Value	Signed Representation (2's complement)
3	3'b011
2	3'b010
1	3'b001
0	3'b000
-1	3'b111
-2	3'b110
-3	3'b101
-4	3'b100





# Operators (1/4)

## ✓ Operators

### – Arithmetic Description

- $A = B + C;$
- $A = B - C;$
- $A = B * C;$
- $A = B / C;$
- $A = B \% C;$  (modulus)

### – Shift Operator (**logical**)

- $A = B >> 2;$       ➔ shift right 'B' by 2-bit (if  $B = 4'b1000$ ,  $A = 4'b0010$ )
- $A = B << 2;$       ➔ shift left 'B' by 2-bit (if  $B = 4'b0001$ ,  $A = 4'b0100$ )

### – Shift Operator (**arithmetic**)

- $A = B >>> 2;$       " $>>>$ ", " $<<<$ " are used only for '**signed**' data type in Verilog 2001
- $A = B <<< 2;$
- e.g. wire signed [3:0] A,B;      ( $A = 4'b1110$ , which is 1000 shifted to the right two positions and sign-filled.)  
     $B = 4'b1000;$   
     $A = B >>> 2;$



# Operators (2/4)

## ✓ Unsigned Operation

```
wire [7:0] a,b;  
wire [3:0] c,  
wire [8:0] sum1, sum2, sum3, sum4;
```

```
assign sum1 = a + b;  
assign sum2 = a + c;  
assign sum3 = a + {4{1'b0}, c};
```

a and b are same width =>  
can be applied to signed and unsigned

reg type is regard as unsigned =>  
automatic 0 extension

manual 0 extension

## ✓ Signed Operation

```
wire signed [7:0] a,b;  
wire signed [3:0] c_sign;  
wire signed [8:0] sum1, sum4;
```

```
wire [7:0] a,b;  
wire [3:0] c,  
wire [7:0] sum1, sum4;
```

```
assign sum1 = a + b;  
assign sum4 = a + c_sign;
```

the same !!

```
assign sum4 = a + {4{c[3]}, c};
```

a and b are same width =>  
can be applied to signed and unsigned

c\_sign is signed type =>  
automatic signed extension

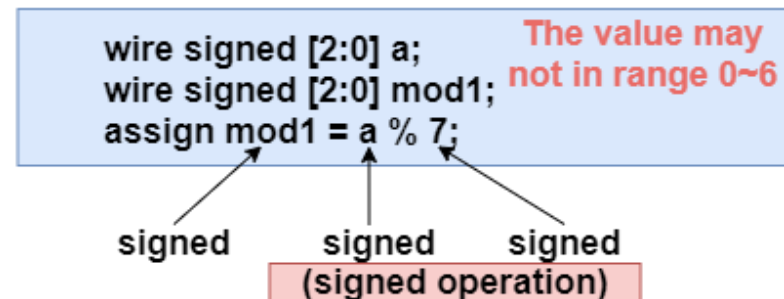
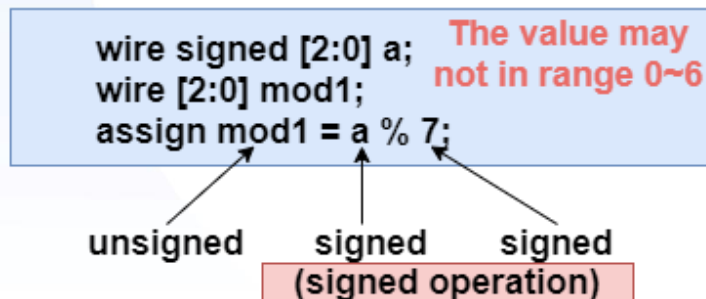
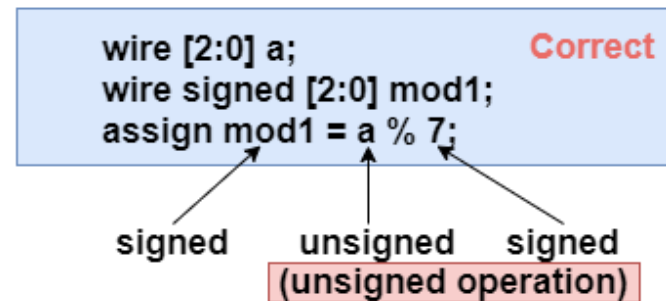
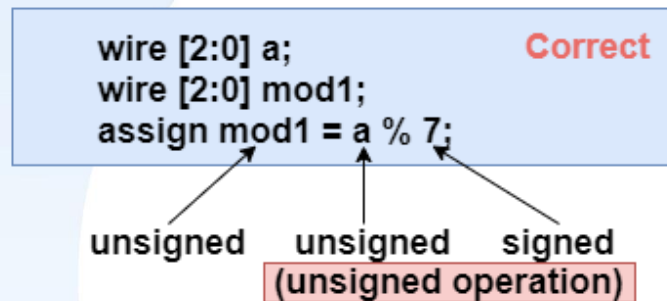
manual signed extension



# Operators (3/4)

## ✓ Unsigned / Signed Mix Operation

- If there are one unsigned operator, the operation will be regard as **unsigned**
- Example:
  - Goal: Number need to be in 0~6



# Operators (4/4)

- ✓ **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  $!a$*



# Conditional Description (1/2)

- ✓ **If-then-else** often infers a cascaded encoder
  - inputs signals with different arrival time
  - Priority inferred
  - used in proc. assignment
- ✓ **case** infers a single-level mux
  - case is better if priority encoding is not required
  - case is generally simulated **faster** than if-then-else
  - used in proc. assignment
- ✓ **conditional assignment (? :)**
  - $? : \rightarrow c = \text{sel } ? a : b;$
  - used in cont. assignment
  - same as if-else statement



# Conditional Description: Example (2/2)

## Conditional Assignment (?:)

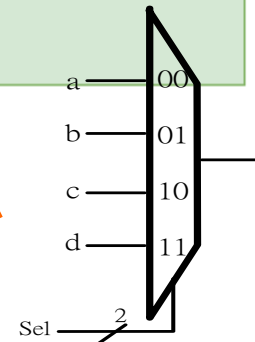
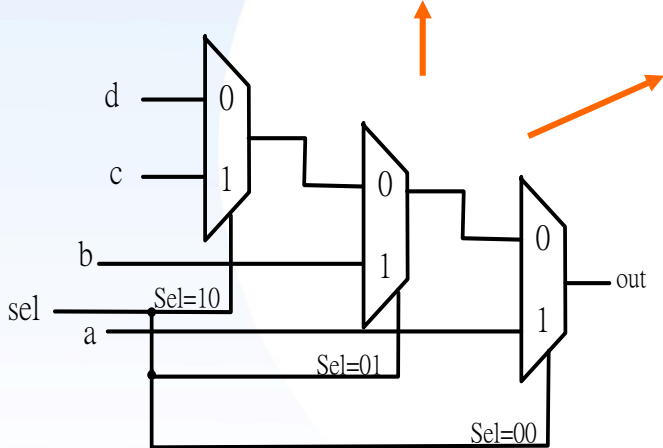
```
assign data=(Sel==2'b00) ? a:  
((Sel==2'b01) ? b:  
((Sel==2'b10) ? c:  
((Sel==2'b11) ? d)));
```

## If-then-else

```
always@ (*)  
begin  
    if(Sel == 2'b00)  
        data=a;  
    else if(Sel== 2'b01)  
        data=b;  
    else if(Sel==2'b10)  
        data=c;  
    else  
        data=d;  
end
```

## case

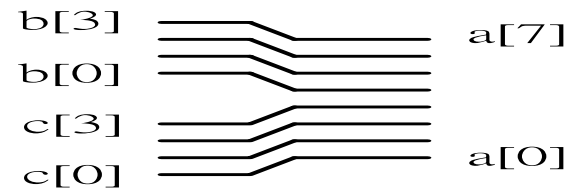
```
always@(*)  
begin  
    case(Sel)  
        2'b00: data = a;  
        2'b01: data = b;  
        2'b10: data = c;  
        default: data = d;  
    endcase  
end
```



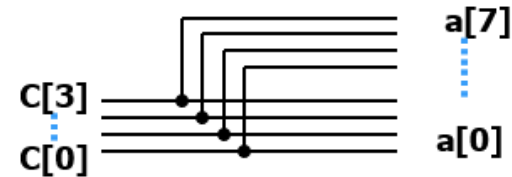
# Concatenation

## Concatenation

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



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

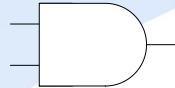


- Ex.  $a[4:0] = \{b[3:0], 1'b0\}; \Leftrightarrow a = b \ll 1;$

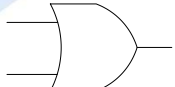
# Gate-Level Modeling (1/3)

## ✓ Primitive logic gate

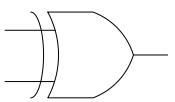
- and



- or



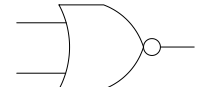
- xor



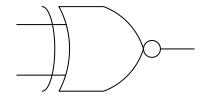
- nand



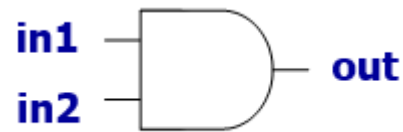
- nor



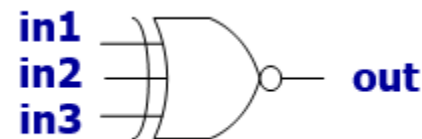
- xnor



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



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

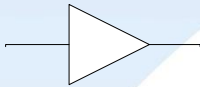




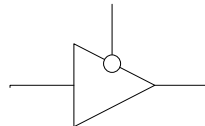
# Gate-Level Modeling (2/3)

## ✓ Primitive logic gate

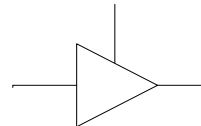
- buf,



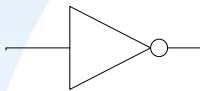
- bufif0,



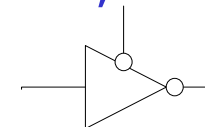
- bufif1



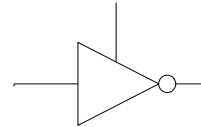
- not,



- notif0,

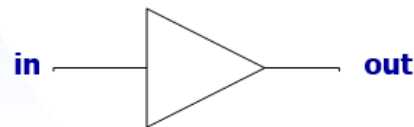


- notif1



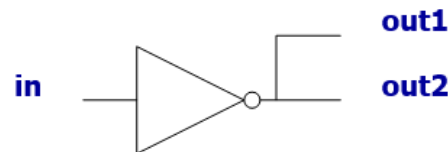
can use without instance name

→ i.e. buf( out, in ) ;



can use with multiple outputs

→ i.e. not( out1, out2 ,in) ;



# Behavioral Model v.s. Gate Level Model

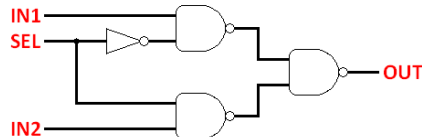
## ✓ Gate Level Model

```
module mux2_1(A,B,Sel,Out);
```

```
input A,B,Sel;  
output Out;  
wire Sel_n, and_out1, and_out2;
```

```
not (Sel_n, Sel);  
and (and_out1, A, Sel_n);  
and (and_out2, B, Sel);  
or (Out, and_out1, and_out2);
```

```
endmodule
```



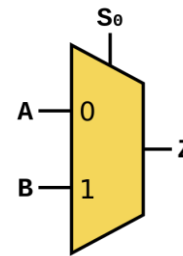
## ✓ Behavioral Model

```
module mux2_1(A,B,Sel,Out);
```

```
input A,B,Sel;  
output reg Out;
```

```
always@(*)  
begin  
    if(Sel==1'b0)  
        Out=A;  
    else  
        Out=B;  
    end  
end
```

```
endmodule
```



# Coding style

- ✓ Data has to be described in one always block
  - Multi-driver (not synthesizable)

```
always@* begin
```

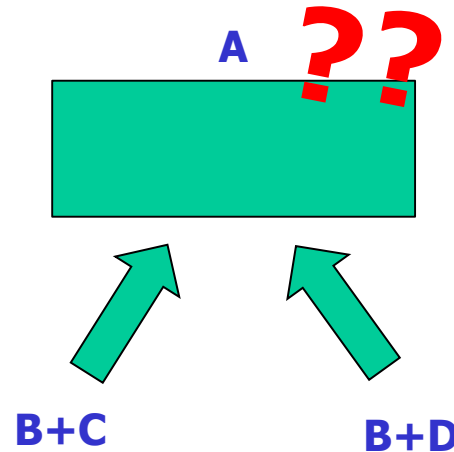
```
    A=B+C;
```

```
end
```

```
always@* begin
```

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

```
end
```



# Coding style

- ✓ Don't use initial block in your design for synthesis

```
initial begin
```

```
  A=B;
```

```
  B=C;
```

```
end
```



Initial use for PATTERN only!!

# Coding style

## ✓ Avoid combinational loop

- May synthesis a Latch in your circuit !! (Latch is non-edge triggered, avoid)

```
always@* begin
  A=B;
  B=A;
end
```



```
always@* begin
  case(Q)
    2'd0: A=B;
    2'd1: A=C;
  endcase
end
```



```
always@* begin
  if(Q==2'd1)
    A=B;
  else if(Q==2'd2)
    A=C;
end
```



```
always@* begin
  case(Q)
    2'd0: A=B;
    2'd1: A=C;
    default: A=D;
  endcase
end
```



```
always@* begin
  if(Q==2'd1)
    A=B;
  else if(Q==2'd2)
    A=C;
  else A=D;
end
```

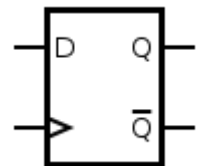
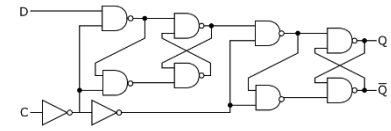
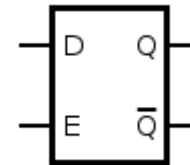
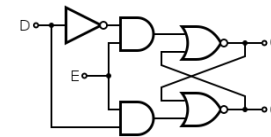
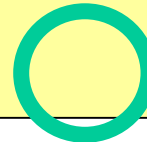


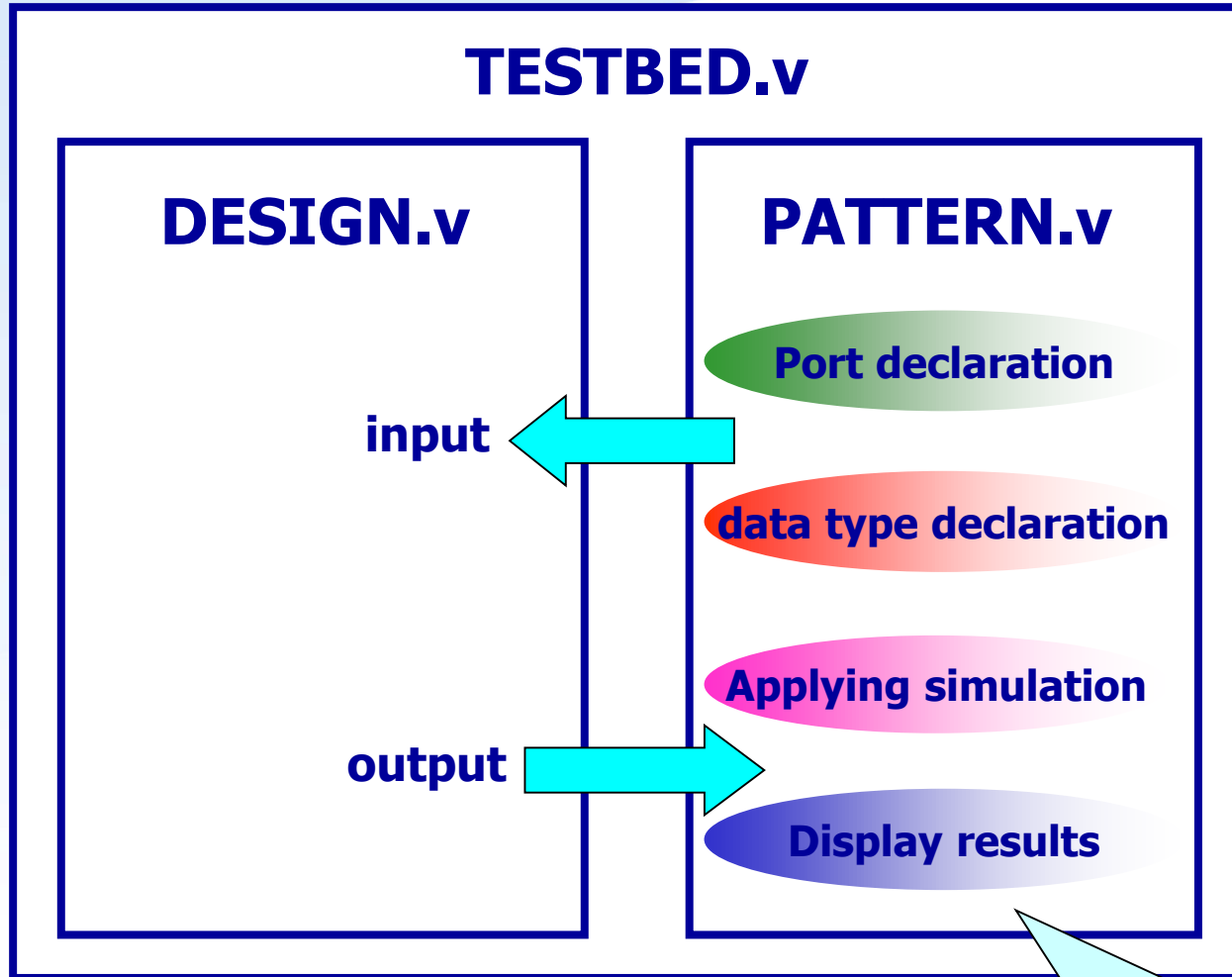
Figure. (a) D-latch (b) D flip-flop

# Outline

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

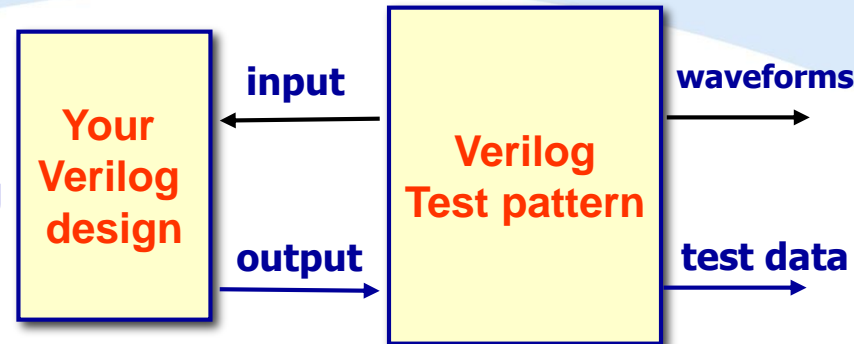
```
endmodule
```



# Simulation Environment (cont.)

## ✓ Simulation command

- Verilog compile
  - `irun TESTBED.v -define RTL -debug`
- Invoke nWave
  - `nWave &`
- Stop the simulation and continue the simulation
  - `Ctrl+z` → Suspend the simulation at anytime you want. (not terminate yet!)
  - `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



# nWave

## ✓ Overview

Cursor  
Position

Marker  
Position

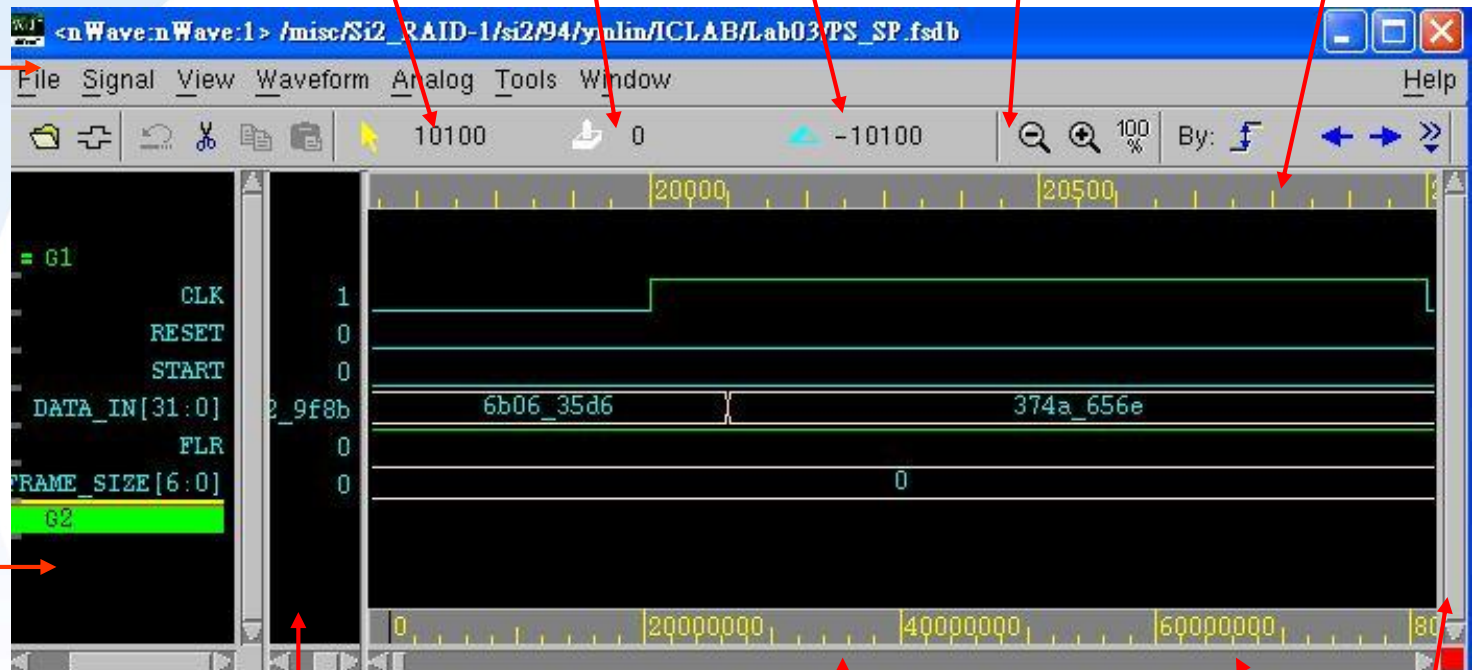
Delta

Tool Bar

Zoom Scale Ruler

Pull Down  
Menu

Signal  
Window



Value Window

Full Scale Ruler

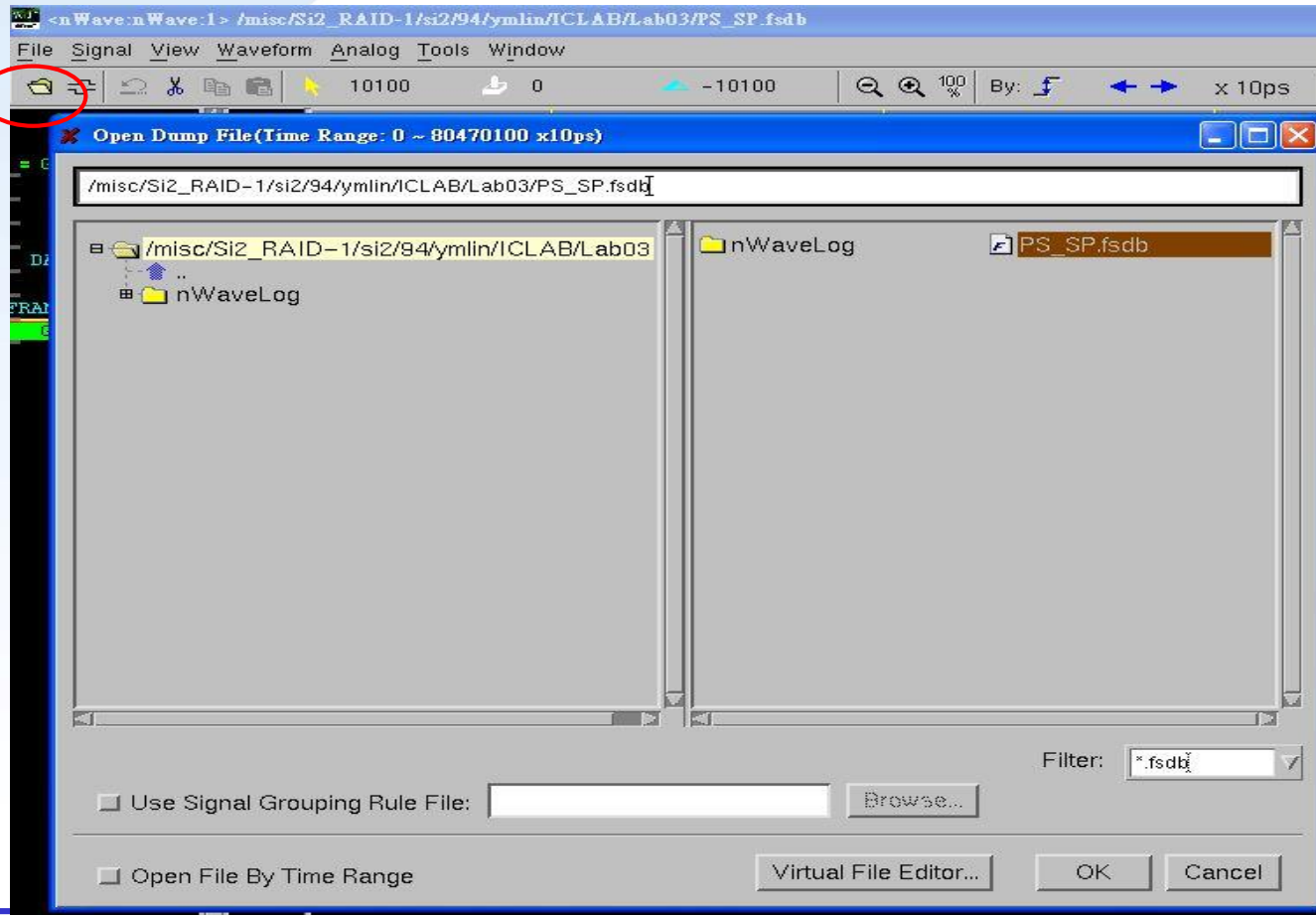
Scroll Bar



# nWave (cont.)

## ✓ Open fsdb file

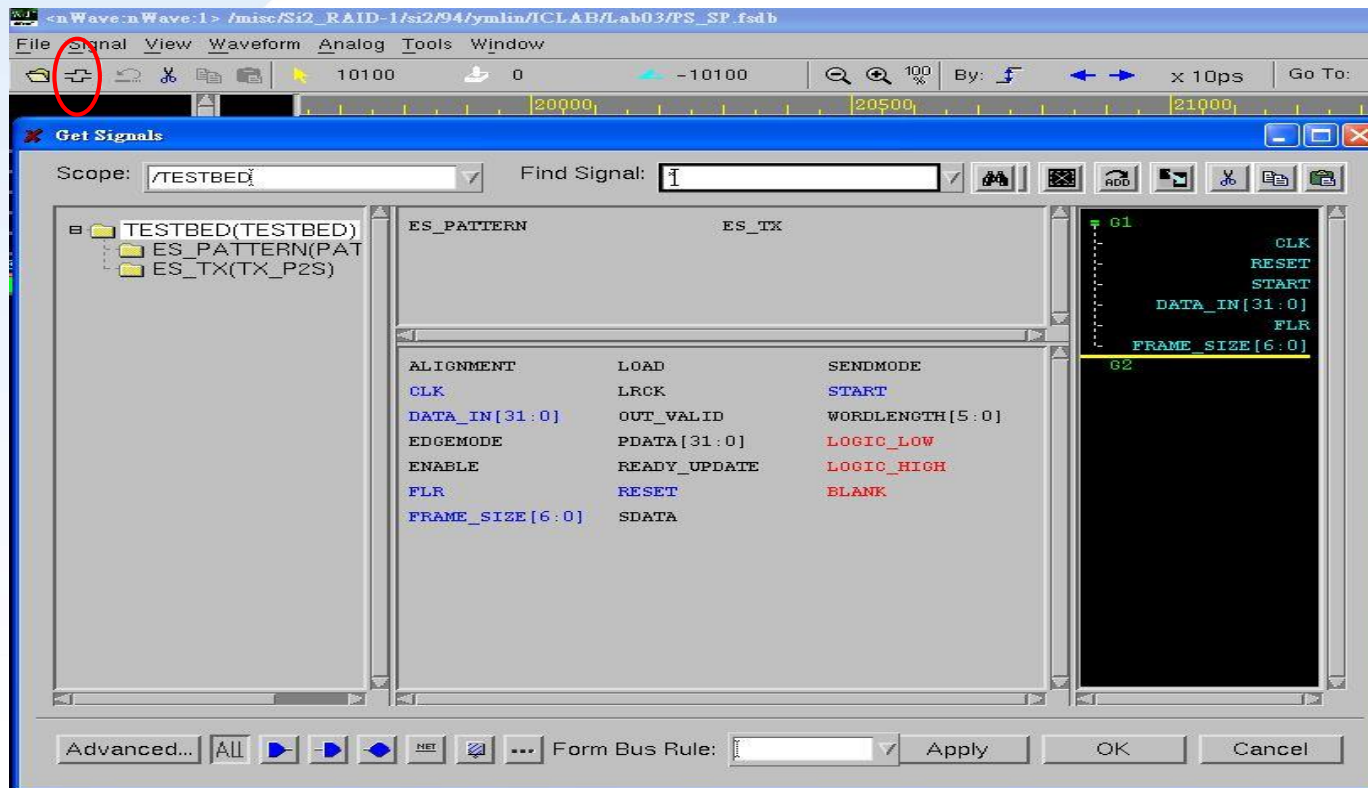
- Use **File** → **Open...** command



# nWave (cont.)

## ✓ Get signal

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

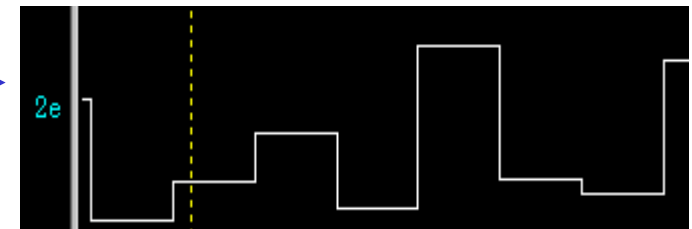
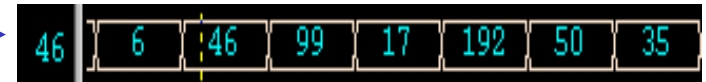
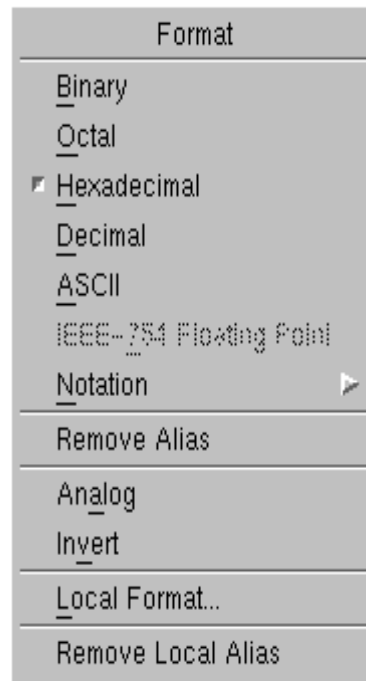


## ✓ Choose value format

- On the value window click Left Button



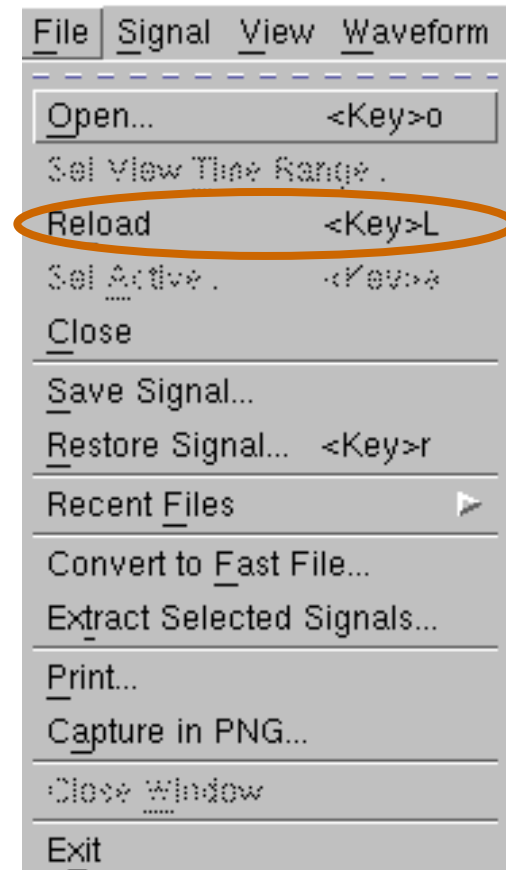
**Default : Hexadecimal**



# 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](mailto:matchbox@si2lab.org))**

**2006revised Yi-Min Lin ([ymlin@si2lab.org](mailto:ymlin@si2lab.org))**

**2008revised Chien-Ying Yu ([cyyu@si2lab.org](mailto:cyyu@si2lab.org))**

**2008revised Chi-Heng Yang ([kevin@oasis.ee.nctu.edu.tw](mailto:kevin@oasis.ee.nctu.edu.tw))**

**2009revised Yung-Chih Chen ([ycchen@oasis.ee.nctu.edu.tw](mailto:ycchen@oasis.ee.nctu.edu.tw))**

**2010revised Liang-Chi Chiu ([oboe.ee98g@nctu.edu.tw](mailto:oboe.ee98g@nctu.edu.tw))**

**2012revised Shyh-Jye Jou**

**2014revised Sung-Shine Lee ([sungshil@si2lab.org](mailto:sungshil@si2lab.org))**

**2018revised Po-Yu Huang**

**2019revised Wei Chiang**

**2020revised Ya-Yun Hou / Lien-Feng Hsu**

**2021revised Lin-Hung Lai ([h123572119@gmail.com](mailto:h123572119@gmail.com))**

**2022revised Tzu\_Hsuan Hung ([davidhung.c@nycu.edu.tw](mailto:davidhung.c@nycu.edu.tw))**

