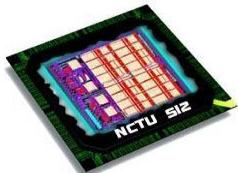


# Lab01 Cell Based Design & Combinational Circuit

---

Lecturer: Lin-Hung, Lai

System Integration and Silicon Implementation (Si2) Lab  
Institute of Electronics  
National Yang Ming Chiao Tung University, Hsinchu, Taiwan



# Outline

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



# Outline

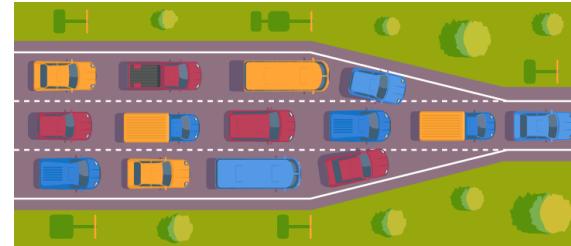
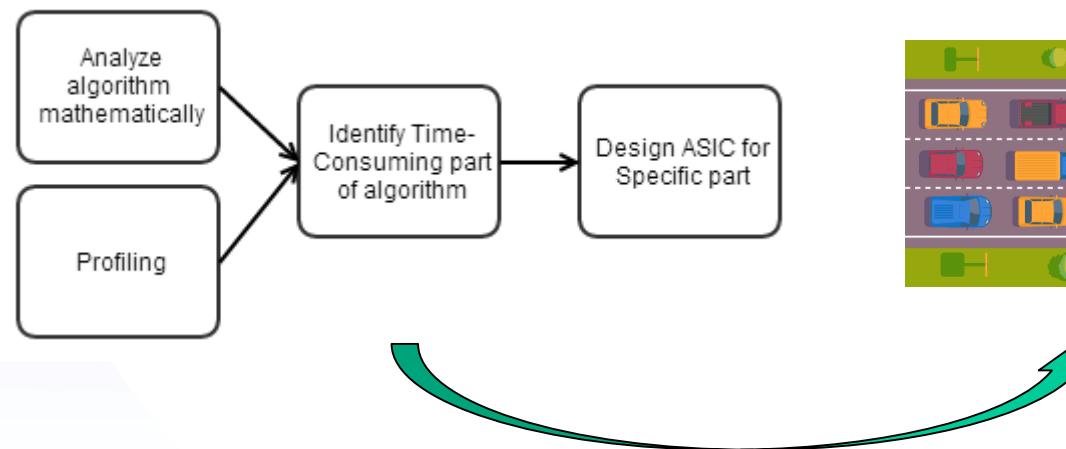
- ✓ **Section 1 Introduction to design flow**
- ✓ **Section 2 Basic Description of Verilog**
- ✓ **Section 3 Behavior Models of Combinational circuit**
- ✓ **Section 4 Simulations**



# 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



(Source: APM Zone)



# 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



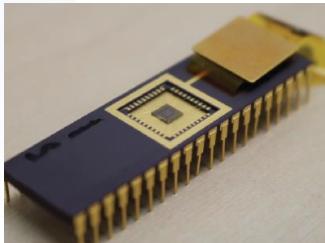
# How Does Hardware Accelerate System

## ✓ 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)



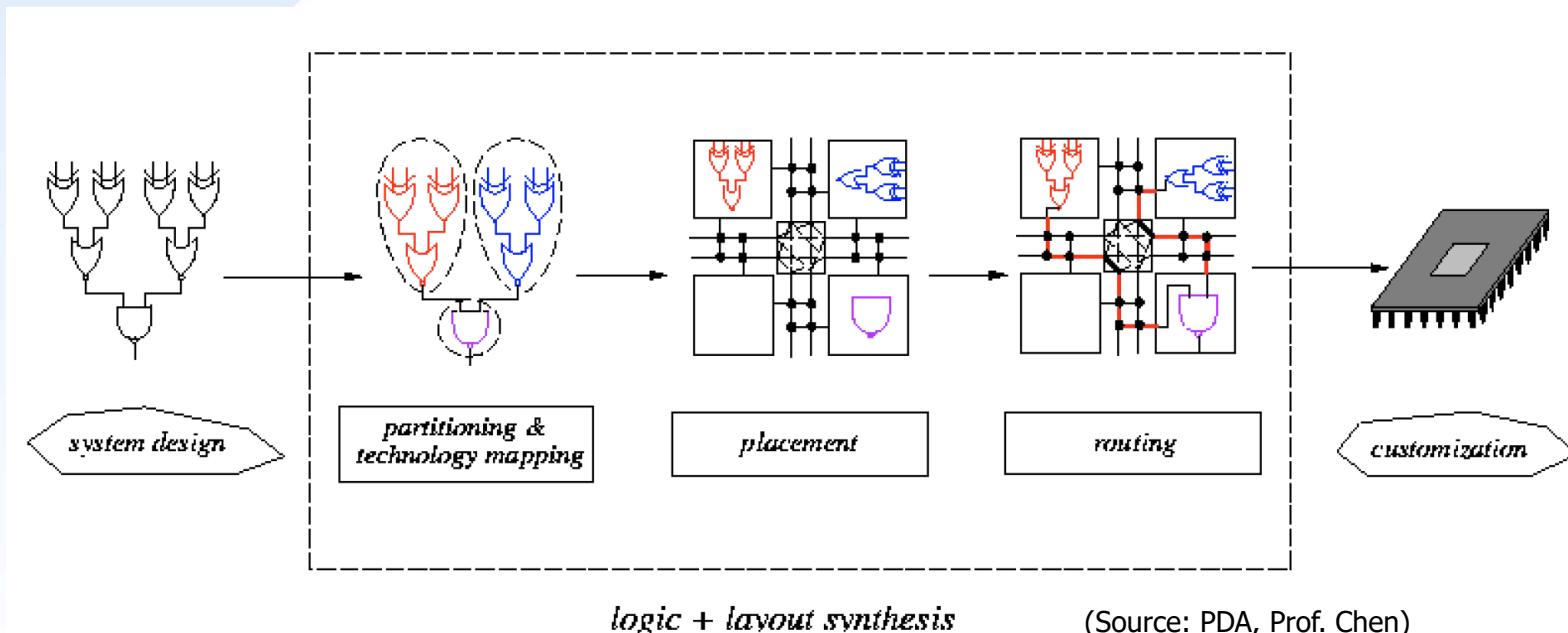
(Source: Xilinx)



# FPGA Example

## ✓ FPGA

- No fabrication is needed
- Limited routing resources



# 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

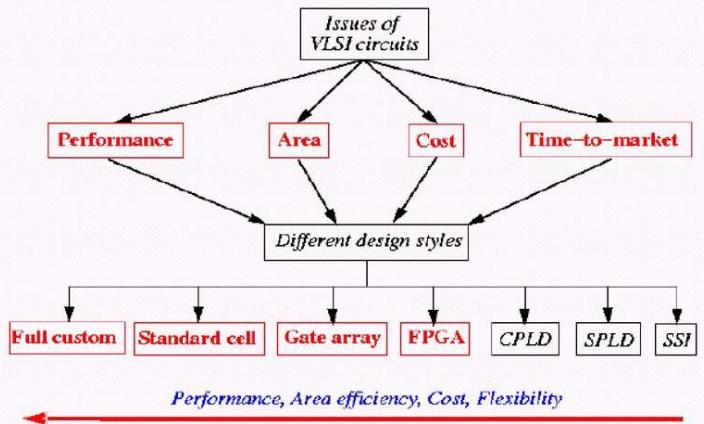


## ✓ Full-Custom Design Flow

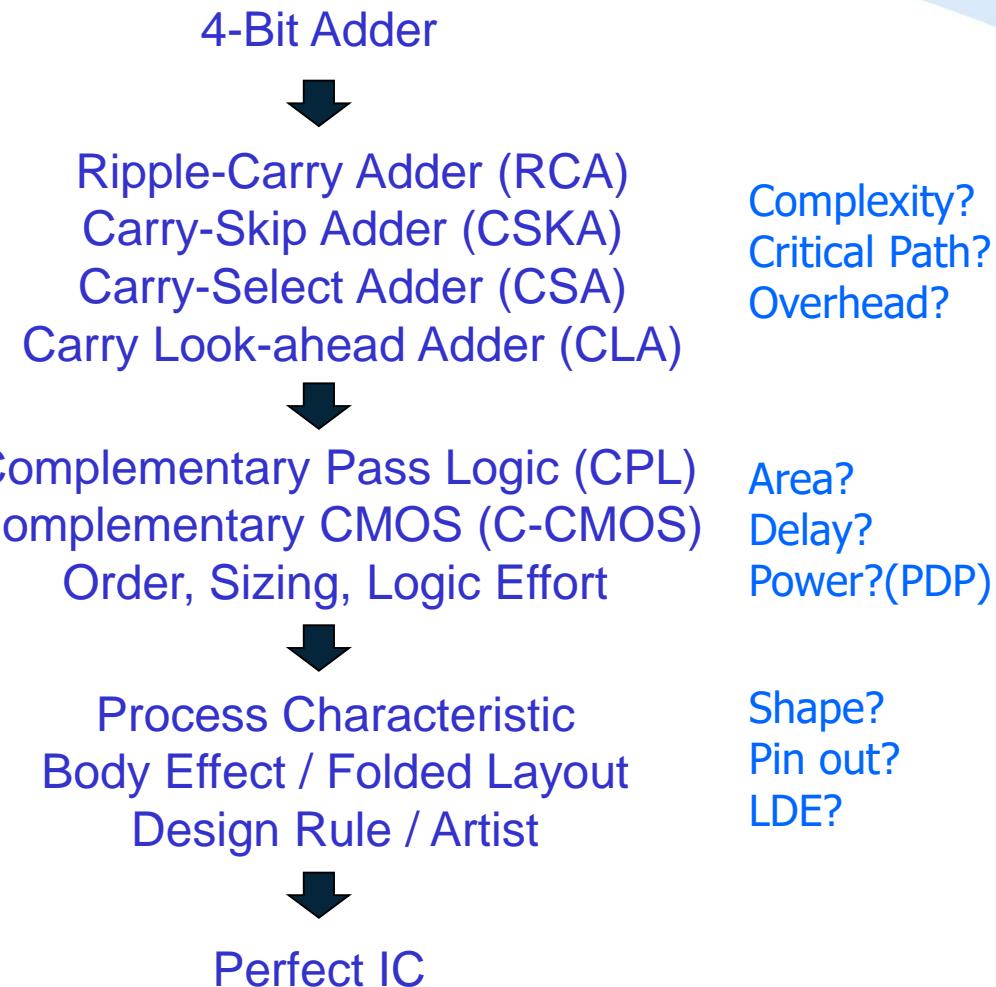
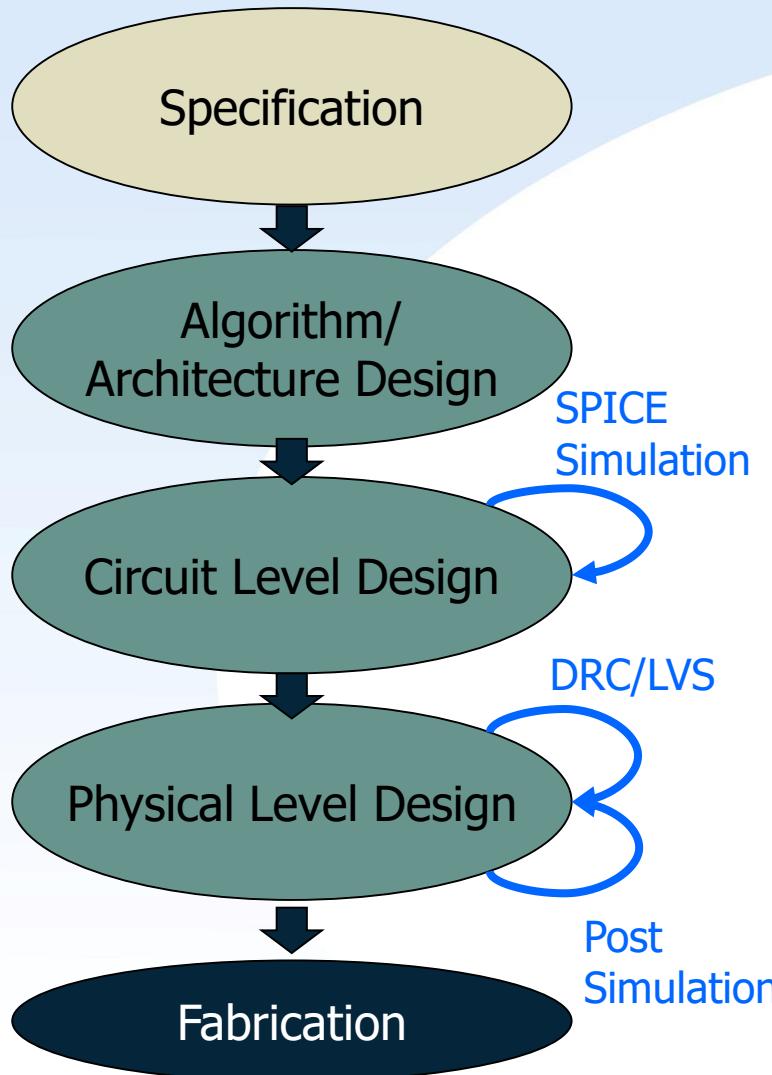
- Design every thing by yourself
- Not our focus

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

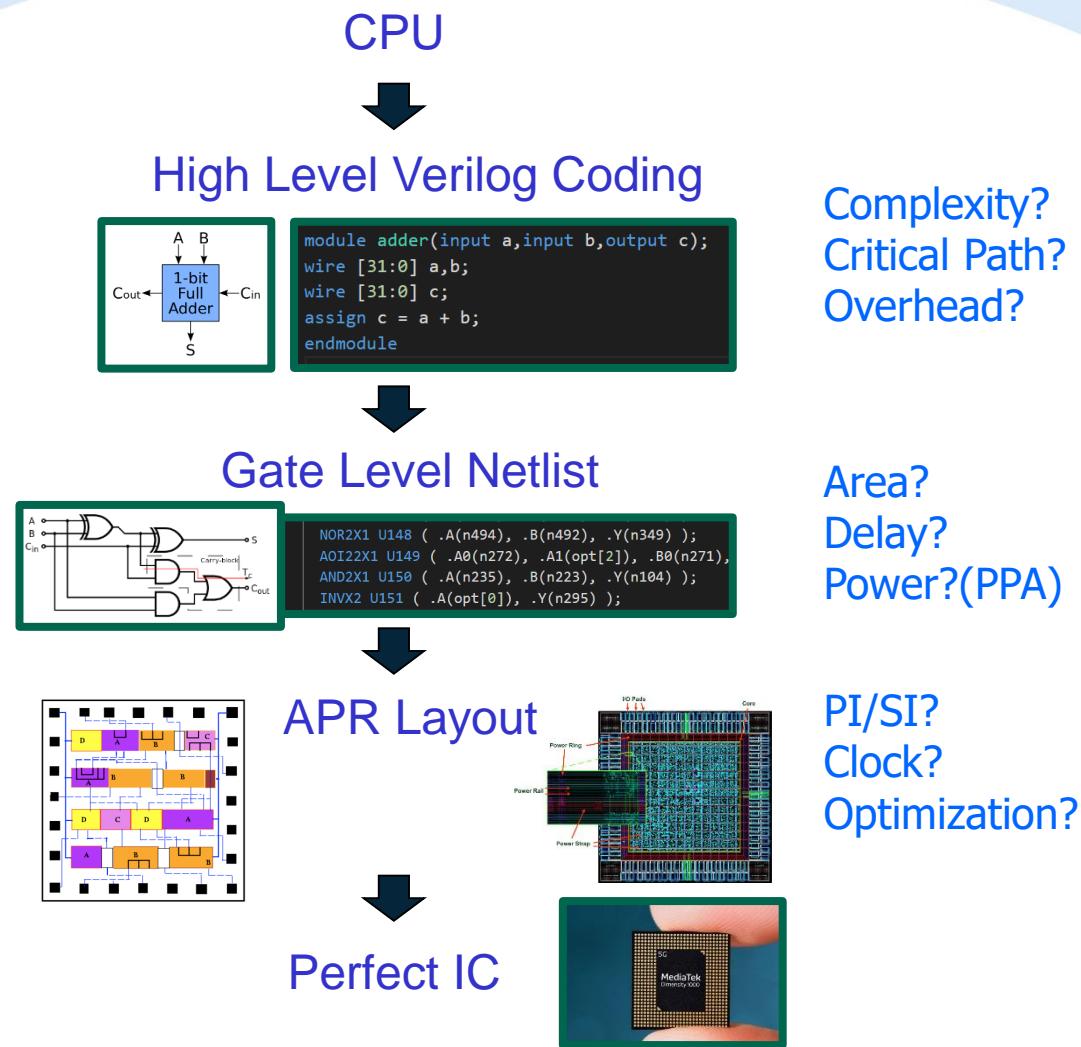
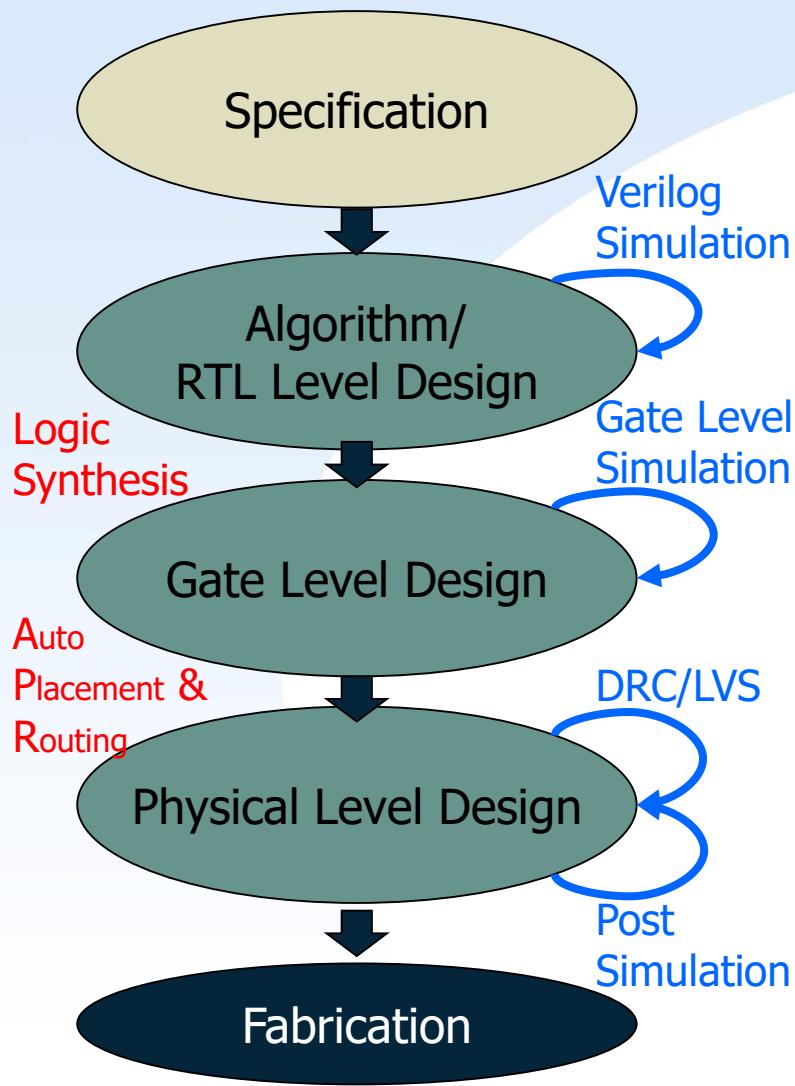
- Specific design styles shall require specific CAD tools



# Full Custom Design Flow

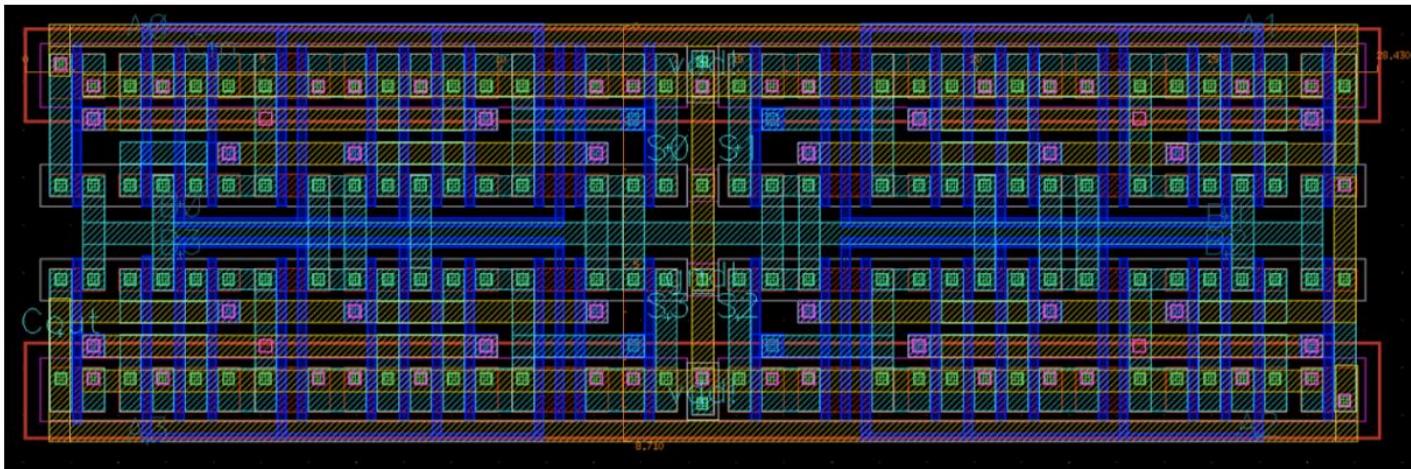


# Cell-Based Design Flow

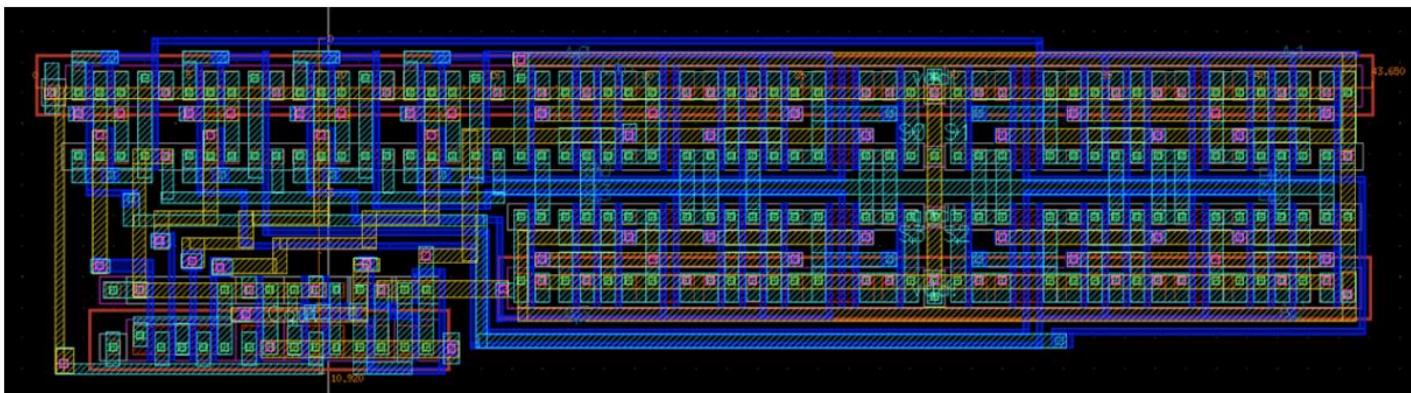


# Full-Custom Example

4-bit Carry Ripple Adder



4-bit Carry Skip Adder



# Design Ware Library (IP)



## DW01\_add

### Adder

Version, STAR and Download Information: [IP Directory](#)

### Features and Benefits

- Parameterized word length
- Carry-in and carry-out signals

### Description

DW01\_add adds two operands A and B with a carry-in CI to produce the output SUM with a carry-out CO.

### Revision History

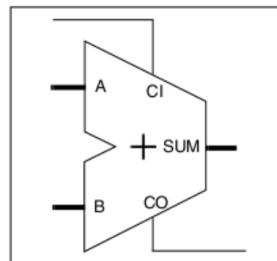


Table 1-1 Pin Description

Pin Name	Width	Direction	Function
A	width bits	Input	Input data
B	width bits	Input	Input data
CI	1 bit	Input	Carry-in
SUM	width bits	Output	Sum of (A + B + CI)
CO	1 bit	Output	Carry-out

Table 1-2 Parameter Description

Parameter	Values	Description
width	$\geq 1$	Word length of A, B, and SUM

Table 1-3 Synthesis Implementations<sup>a</sup>

Implementation	Function	License Feature Required
rpl	Ripple-carry synthesis model	none
cla	Carry-look-ahead synthesis model	none
pparch	Delay-optimized flexible parallel-prefix	DesignWare

Command: evince /RAID2/EDA/synopsys/synthesis/2020.09/dw/doc/datasheets/dw01\_add.pdf

### HDL Usage Through Component Instantiation - Verilog

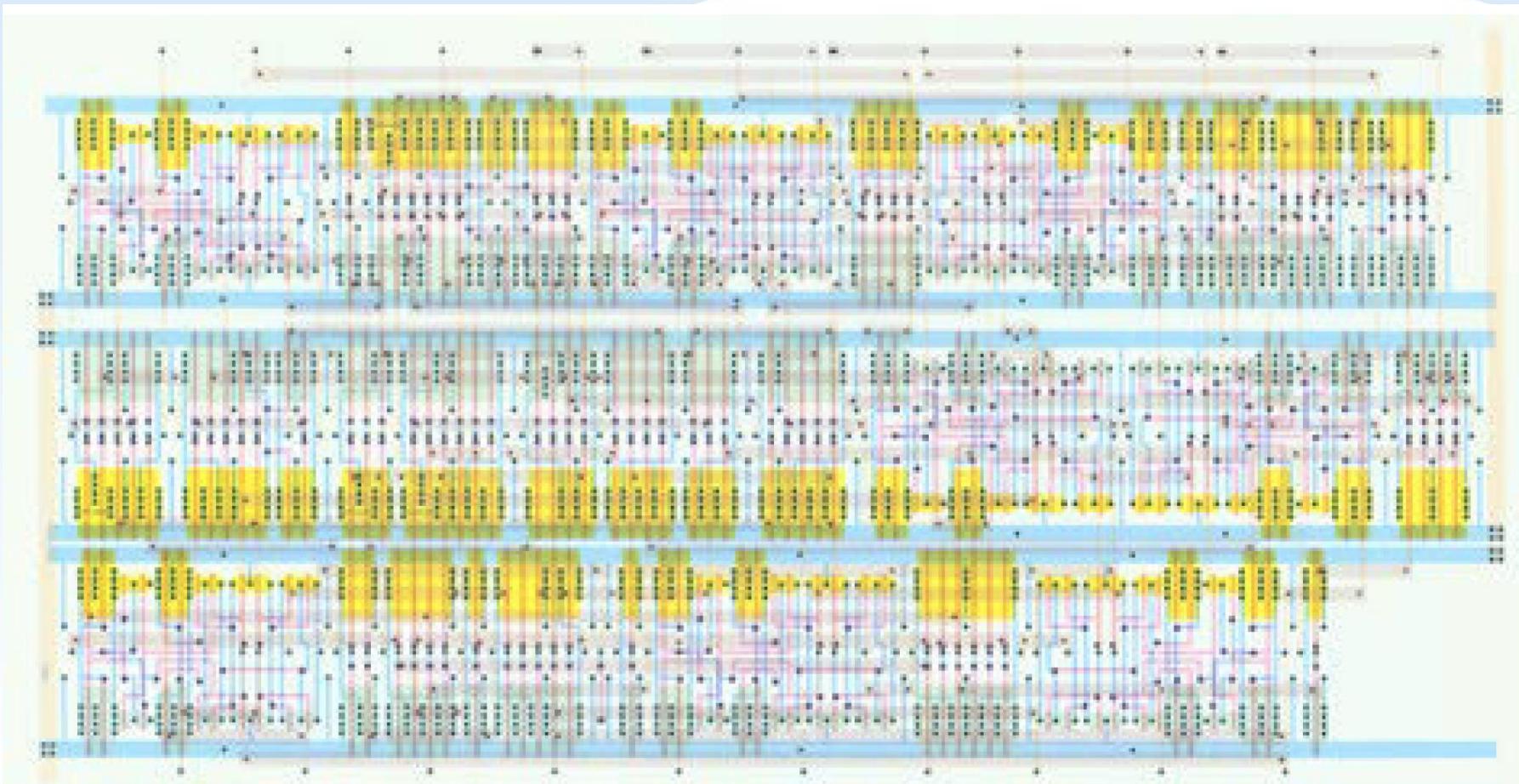
```
module DW01_add_inst( inst_A, inst_B, inst_CI, SUM_inst, CO_inst );  
  parameter width = 8;  
  
  input [width-1 : 0] inst_A;  
  input [width-1 : 0] inst_B;  
  input inst_CI;  
  output [width-1 : 0] SUM_inst;  
  output CO_inst;  
  
  // Instance of DW01_add  
  DW01_add #(width)  
    U1 (.A(inst_A), .B(inst_B), .CI(inst_CI), .SUM(SUM_inst), .CO(CO_inst) );  
endmodule
```

### HDL Usage Through Operator Inferencing - Verilog

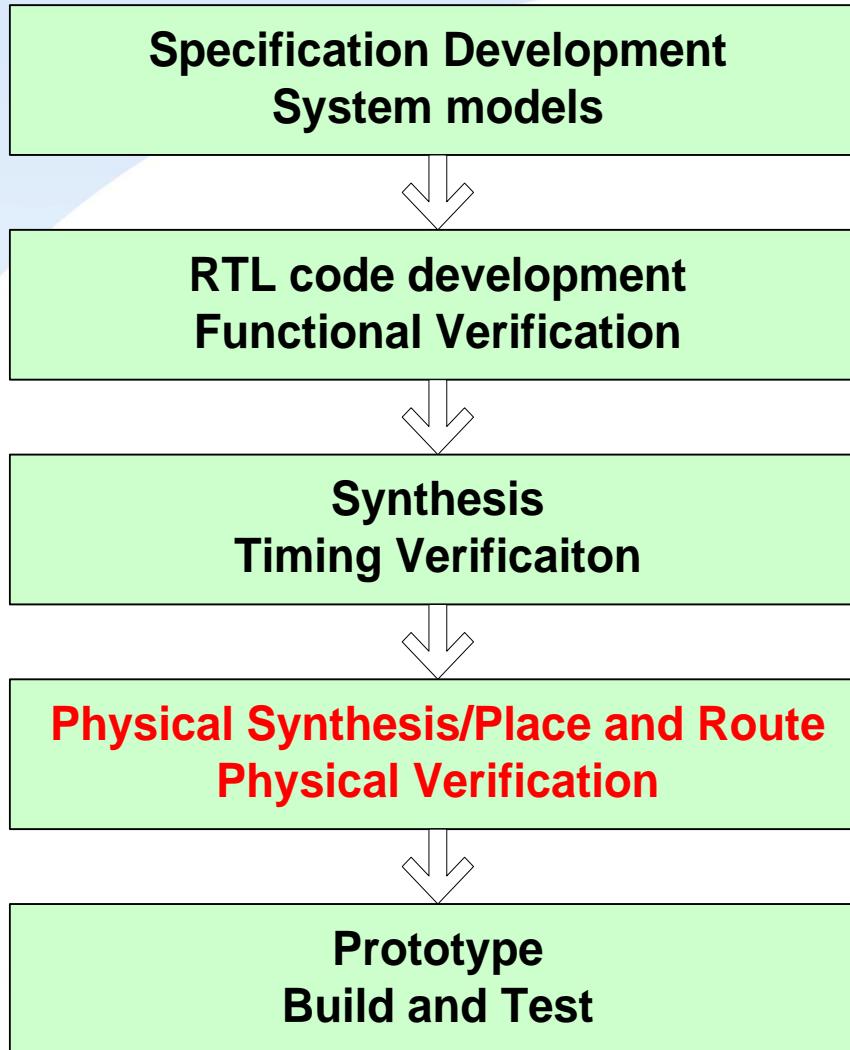
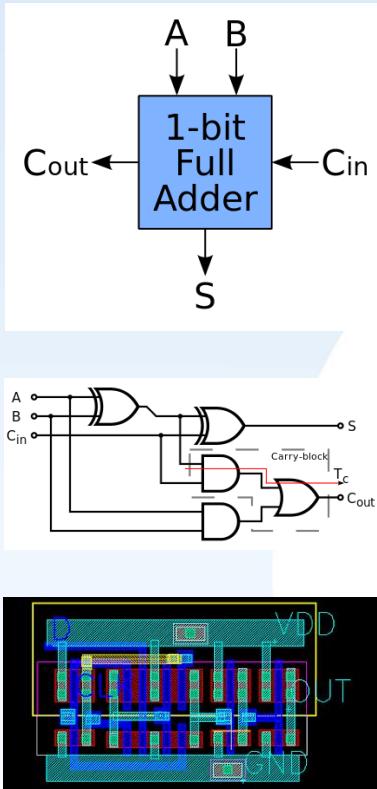
```
module DW01_add_oper(in1,in2,sum);  
  parameter wordlength = 8;  
  
  input [wordlength-1:0] in1,in2;  
  output [wordlength-1:0] sum;  
  
  assign sum = in1 + in2;  
endmodule
```



# Standard Cell Example



# Cell-based Design Flow



System Architecture

RTL 1

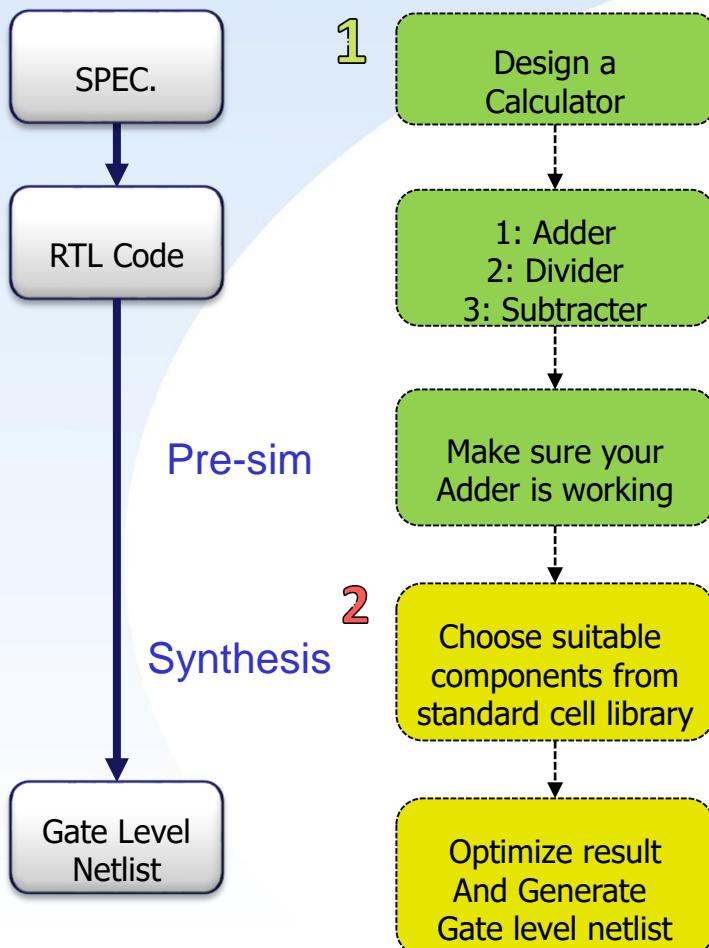
Synthesis 2

Physical Design 3

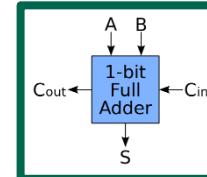
System Integration and  
Software Test



# Cell-based Design Flow - RTL to GATE



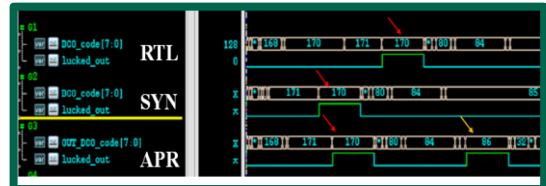
Specify input/output relationship



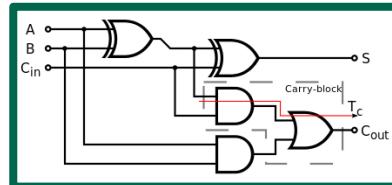
Write Verilog HDL Code

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

Use Cadence Tool  
Ncverilog, irun  
nWave



Run synthesis by  
Design Compiler



```
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) );
INVX1 U151 ( .A(opt[0]), .Y(n295) );
```



# Synopsys EDA Tool: Design Compiler

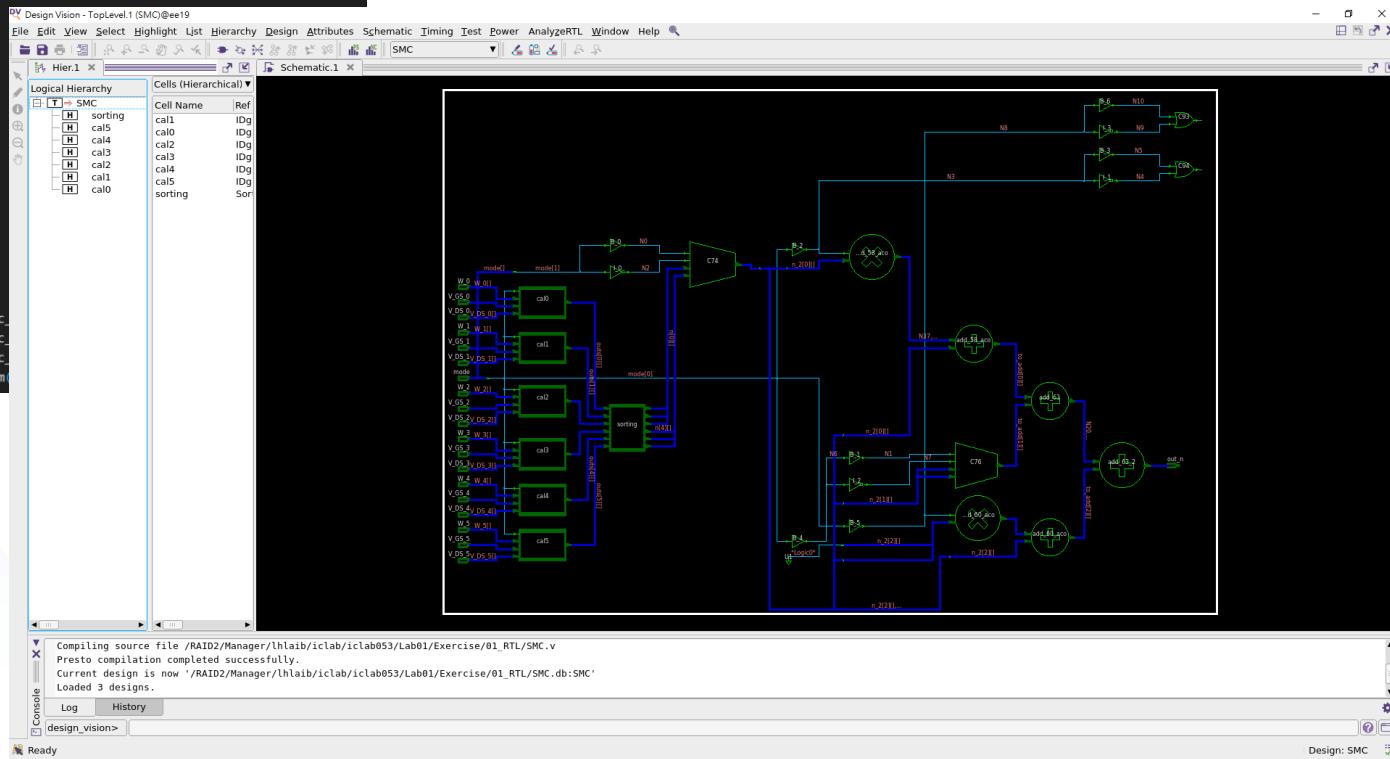
```
module CORE (
    in_n0,
    in_n1,
    opt,
    out_n
);
//-----
//Input, Output Declaration
//-----
input [2:0] in_n0, in_n1;
input opt;
output [3:0] out_n;

reg [3:0] in_n1_inv;
reg carry_in;
wire [3:0] tmp;

//-----write your code here-----

always@(*)
begin
if(opt)
    begin
    in_n1_inv = {1'b1, ~in_n1};
    carry_in = 1'b1;
    end
else
    begin
    in_n1_inv = {1'b0,in_n1};
    carry_in = 1'b0;
    end
end

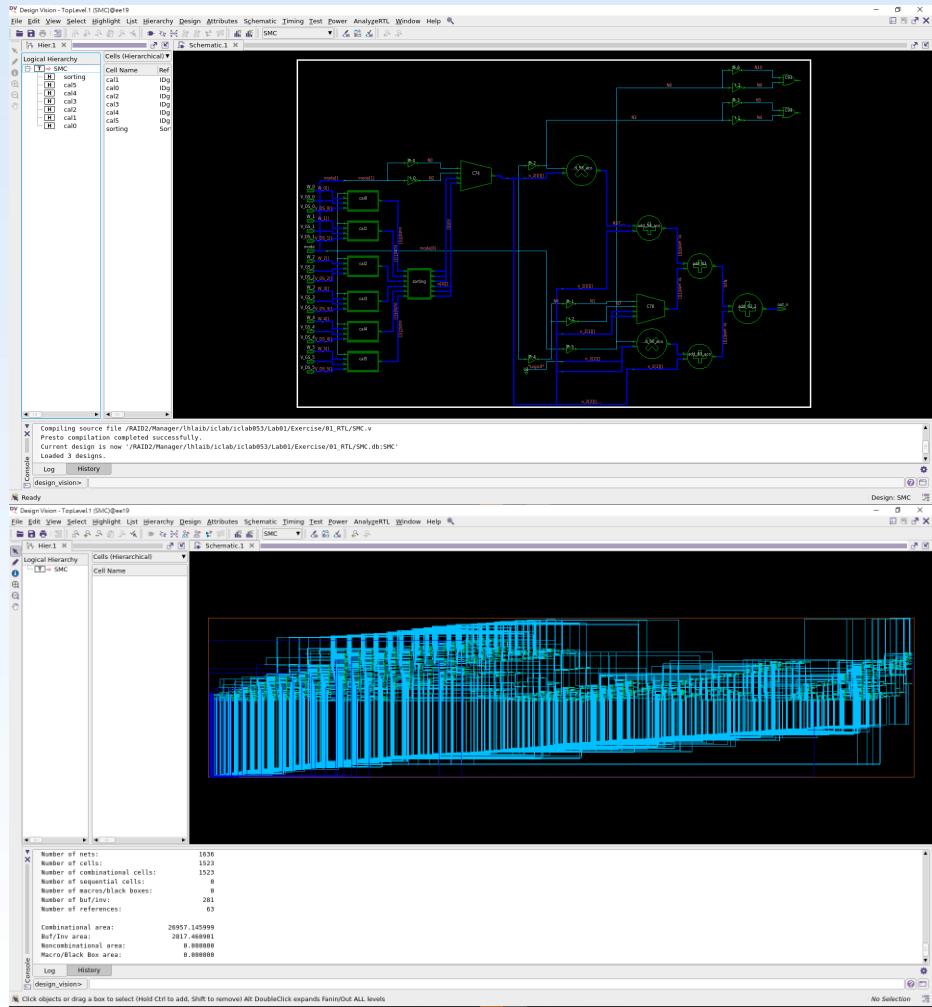
FA FA0(.a(in_n0[0]),.b(in_n1_inv[0]),.c
FA FA1(.a(in_n0[1]),.b(in_n1_inv[1]),.c
FA FA2(.a(in_n0[2]),.b(in_n1_inv[2]),.c
HA HA3(.a(tmp[2]),.b(in_n1_inv[3]),.sum
```



**GUI Version:** Design Vision  
**Command Line Version:** Design Compiler



# Synopsys EDA Tool: Design Compiler



## Get Area

```
*****
Report : area
Design : SMC
Version: T-2022.03
Date  : Tue Nov 8 22:56:27 2022
*****
Library(s) Used:
    slow (File: /RAID2/COURSE/iclab/iclabta01/umc018/Synthesis/slow.db)

Number of ports:                                66
Number of nets:                                 1636
Number of cells:                               1523
Number of combinational cells:                 1523
Number of sequential cells:                      0
Number of macros/black boxes:                  0
Number of buf/inv:                             281
Number of references:                           63

Combinational area:                            26957.145999
Buf/Inv area:                                2817.469901
Noncombinational area:                         0.000000
Macro/Black Box area:                          0.000000
Net Interconnect area:                        undefined (No wire load specified)

Total cell area:                             26957.145999
Total area:                                  undefined
```

## Get resources

```
*****
Report : resources
Design : SMC
Version: T-2022.03
Date  : Tue Nov 8 22:56:26 2022
*****
Resource Report for this hierarchy in file
/RAID2/Manager/lhlab/iclab/iclab03/Lab01/Exercise/01_RTL/SMC.v
=====
| Cell           | Module          | Parameters | Contained Operations |
| mult_x_3      | DW_mult_uns   | a_width=9  | mult_add_60_aco (SMC.v:60) |
| mult_x_4      | DW_mult_uns   | a_width=8  | mult_add_58_aco (SMC.v:58) |
| lte_x_5       | DW_cmp         | width=3   | cal0/lte_79 (SMC.v:79)     |
| sub_x_6       | DW01_sub      | width=5   | cal0/sub_84 (SMC.v:84)     |
| mult_x_7      | DW_mult_uns   | a_width=3  | cal0/mult_84 (SMC.v:84)   |
| mult_x_8      | DW_mult_uns   | a_width=3  | cal0/mult_87 (SMC.v:87)   |
| div_9          | DW_div_uns    | a_width=10 | cal0/div_87 (SMC.v:87)    |
| lte_x_13      | DW_cmp         | width=7   | sorting_lte_98 (SMC.v:98)  |
| lte_x_14      | DW_cmp         | width=7   | sorting_lte_99 (SMC.v:99)  |
| lte_x_15      | DW_cmp         | width=7   | sorting_lte_100 (SMC.v:100) |
| lte_x_16      | DW_cmp         | width=7   | sorting_lte_101 (SMC.v:101) |
```



# Synopsys EDA Tool: Design Compiler

## Get Critical Path

```
Information: Updating design information... (UID-85)
```

```
*****
```

```
Report : timing
      -path full
      -delay max
      -max_paths 1
```

```
Design : SMC
Version: T-2022.03
Date  : Tue Nov  8 22:56:26 2022
*****
```

```
Operating Conditions: slow  Library: slow
Wire Load Model Mode: top
```

```
Startpoint: V_GS_0[0] (input port)
Endpoint: out_n[9] (output port)
Path Group: default
Path Type: max
```

Point	Incr	Path
input external delay	0.00	0.00 f
V_GS_0[0] (in)	0.00	0.00 f
U1272/Y (INVX2)	0.07	0.07 r
U1260/Y (NAND2X1)	0.12	0.18 f
U1259/Y (NAND2X1)	0.23	0.42 r
U1856/Y (INVX2)	0.09	0.51 f
U797/Y (AOI22X1)	0.19	0.70 r
U795/Y (NOR2X1)	0.09	0.78 f
U1777/Y (OAI22X1)	0.39	1.18 r
U1358/Y (MXI2X1)	0.34	1.52 f
U777/Y (INVX1)	0.20	1.72 r
U774/Y (NOR2X1)	0.11	1.83 f
U1684/Y (NAND2X1)	0.18	2.02 r
U1683/Y (OAI21X1)	0.10	2.12 f
U1577/Y (AND2X1)	0.27	2.39 f
U1146/Y (NOR2X1)	0.14	2.53 r
U1825/C0 (ADDHXL)	0.22	2.75 r
U750/C0 (ADDFX1)	0.64	3.39 r
U1378/C0 (ADDFHX1)	0.41	3.80 r
U1864/C0 (ADDFHX1)	0.35	4.16 r
U1013/C0 (ADDFX1)	0.31	4.47 r

U699/Y (NAND2XL)	0.11	12.32 f
U880/Y (AOI2BB2X1)	0.28	12.59 r
U2043/CON (AFHCNX2)	0.14	12.74 f
U697/Y (AOI2BB1XL)	0.30	13.03 f
U877/Y (OAI22X1)	0.24	13.28 r
U2045/CON (AFHCNX2)	0.21	13.49 f
U2046/Y (OAI22X4)	0.25	13.74 r
U2047/Y (MXI2X1)	0.39	14.13 f
U2061/CON (AFHCNX2)	0.34	14.47 r
U817/Y (AOI21XL)	0.14	14.62 f
U693/Y (OAI22X1)	0.22	14.83 r
U761/Y (NAND2X1)	0.09	14.92 f
U760/Y (AOI22X1)	0.23	15.16 r
U759/Y (OAI21X1)	0.17	15.33 f
U2068/Y (OAI21X4)	0.22	15.55 r
U1354/Y (MXI2X1)	0.39	15.94 f
U2093/CON (AFHCNX2)	0.34	16.28 r
U637/Y (OAI22XL)	0.14	16.42 f
U687/Y (OAI21X1)	0.21	16.63 r
U685/Y (OAI21X1)	0.16	16.79 f
U859/CON (AFHCNX2)	0.20	16.99 r
U702/Y (NAND2X1)	0.27	17.26 f
U694/Y (INVX1)	0.21	17.46 r
U1730/Y (NAND2XL)	0.08	17.55 f
U1728/Y (OAI211XL)	0.18	17.73 r
U679/Y (AND2X1)	0.27	18.00 r
U678/S (ADDFX1)	0.63	18.63 f
U2143/S (ADDFHX1)	0.42	19.06 f
U567/Y (NOR2X1)	0.21	19.26 r
U2142/Y (OAI21X1)	0.14	19.40 f
U2153/Y (AOI21XL)	0.19	19.59 r
U636/Y (OAI21X1)	0.13	19.72 f
U628/Y (XOR2X1)	0.26	19.98 r
out_n[9] (out)	0.00	19.98 r
data arrival time		19.98
max_delay	20.00	20.00
output external delay	0.00	20.00
data required time		20.00
data required time		20.00
data arrival time		-19.98
slack (MET)		0.02



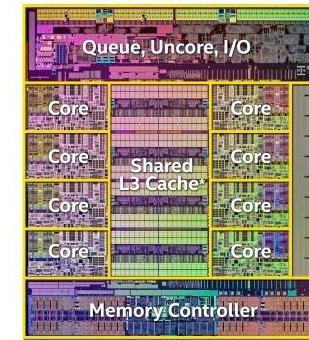
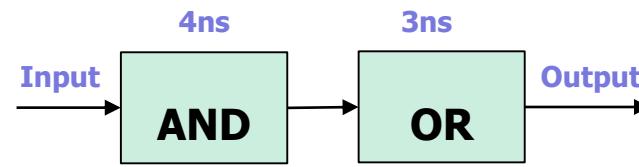
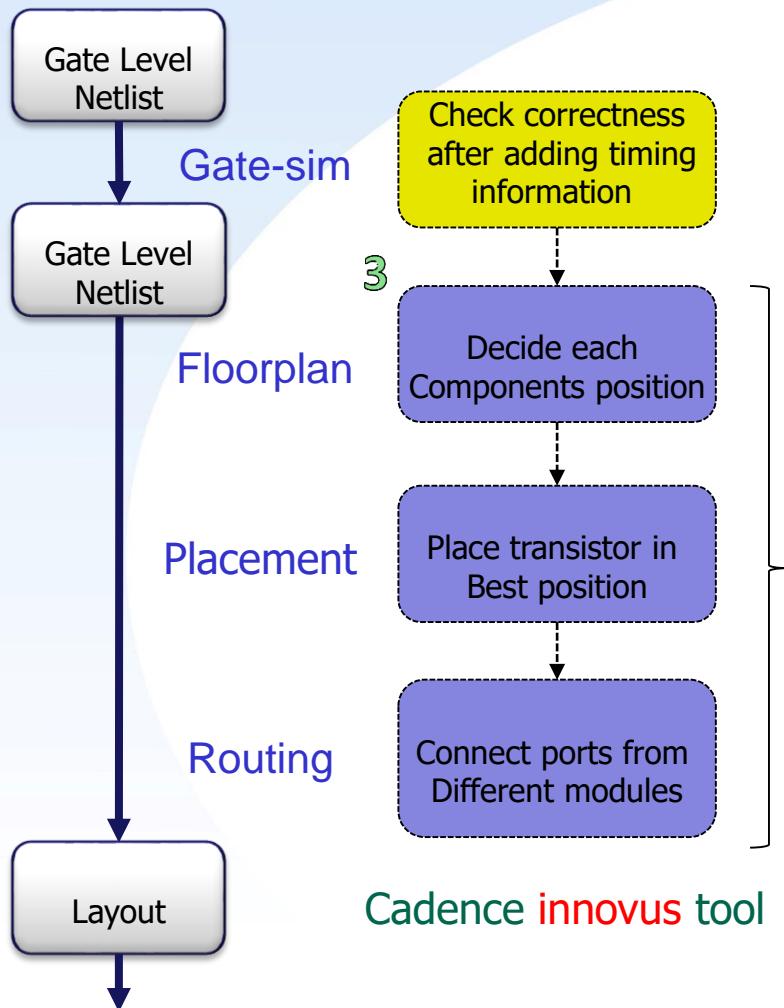
# Synopsys EDA Tool: Design Compiler

## Standard Delay Format: .sdf file

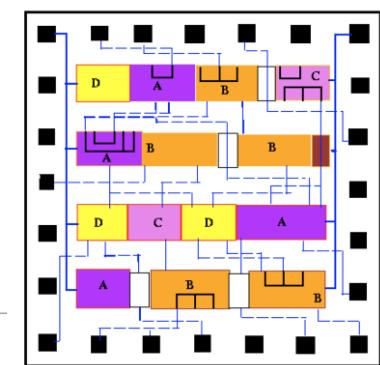
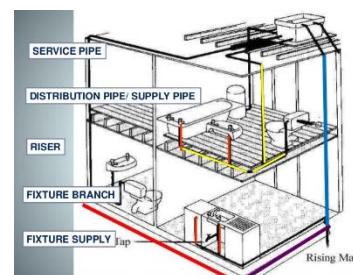
```
(CELL
  (CELLTYPE "ADDFX1")
  (INSTANCE U569)
  (DELAY
    (ABSOLUTE
      (COND B == 1'b1 (IOPATH A CO (0.525937:0.545691:0.545691) (0.484300:0.491862:0.491862)))
      (COND B == 1'b0 (IOPATH A CO (0.498356:0.520221:0.520221) (0.481282:0.488874:0.488874)))
      (IOPATH A CO (0.525937:0.545691:0.545691) (0.484737:0.492320:0.492320))
      (COND A == 1'b1 (IOPATH B CO (0.582731:0.583036:0.583036) (0.314695:0.315612:0.315612)))
      (COND A == 1'b0 (IOPATH B CO (0.364209:0.364409:0.364409) (0.520958:0.521962:0.521962)))
      (IOPATH B CO (0.582731:0.583036:0.583036) (0.520958:0.521962:0.521962))
      (IOPATH CI CO (0.336391:0.362851:0.362851) (0.357614:0.372495:0.372495))
      (COND B == 1'b0 && CI == 1'b0 (IOPATH A S (0.393408:0.414347:0.414347) (0.415707:0.422931:0.422931)))
      (COND B == 1'b1 && CI == 1'b1 (IOPATH A S (0.387539:0.407365:0.407365) (0.399195:0.406190:0.406190)))
      (COND B == 1'b0 && CI == 1'b1 (IOPATH A S (0.460406:0.468100:0.468100) (0.460216:0.481809:0.481809)))
      (COND B == 1'b1 && CI == 1'b0 (IOPATH A S (0.467909:0.475581:0.475581) (0.508563:0.528157:0.528157)))
      (IOPATH (posedge A) S (0.474953:0.518705:0.518705) (0.508563:0.528157:0.528157))
      (IOPATH (negedge A) S (0.467909:0.475581:0.475581) (0.504884:0.508891:0.508891))
      (COND A == 1'b0 && CI == 1'b0 (IOPATH B S (0.420326:0.420641:0.420641) (0.416258:0.417233:0.417233)))
      (COND A == 1'b1 && CI == 1'b1 (IOPATH B S (0.419939:0.420221:0.420221) (0.508525:0.509545:0.509545)))
      (COND A == 1'b0 && CI == 1'b1 (IOPATH B S (0.509318:0.510324:0.510324) (0.444515:0.444819:0.444819)))
      (COND A == 1'b1 && CI == 1'b0 (IOPATH B S (0.478825:0.479836:0.479836) (0.567383:0.567689:0.567689)))
      (IOPATH (posedge B) S (0.516636:0.516999:0.516999) (0.567383:0.567689:0.567689))
      (IOPATH (negedge B) S (0.509318:0.510324:0.510324) (0.561220:0.562068:0.562068))
      (COND (A == 1'b0 && B == 1'b0) ||| (A == 1'b1 && B == 1'b1) (IOPATH CI S (0.224687:0.246160:0.246160) (0.261392:0.276080:0.276080)))
      (COND (A == 1'b0 && B == 1'b1) ||| (A == 1'b1 && B == 1'b0) (IOPATH CI S (0.392451:0.407996:0.407996) (0.309505:0.360704:0.360704)))
      (IOPATH (posedge CI) S (0.397823:0.458590:0.458590) (0.309505:0.360704:0.360704))
      (IOPATH (negedge CI) S (0.392451:0.407996:0.407996) (0.304602:0.318789:0.318789))
    )
  )
  (CELL
    (CELLTYPE "NOR2X1")
    (INSTANCE U567)
    (DELAY
      (ABSOLUTE
        (IOPATH A Y (0.202379:0.205674:0.205674) (0.106812:0.108177:0.108177))
        (IOPATH B Y (0.221780:0.233445:0.233445) (0.118710:0.126118:0.126118))
      )
    )
  )
)
```



# Cell-based Design Flow – GATE to LAYOUT



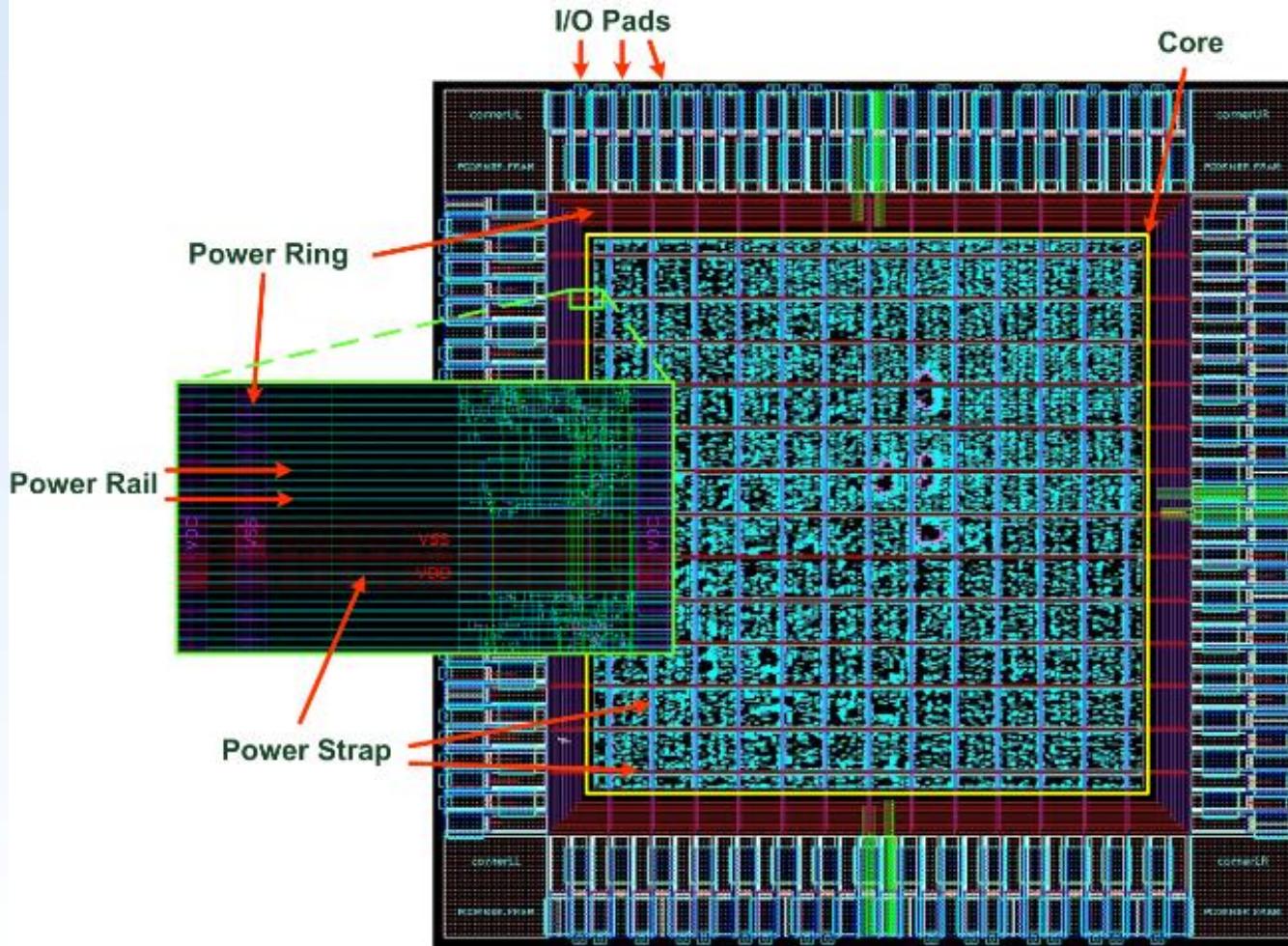
(Source: Intel i7-5960X processor floorplan)



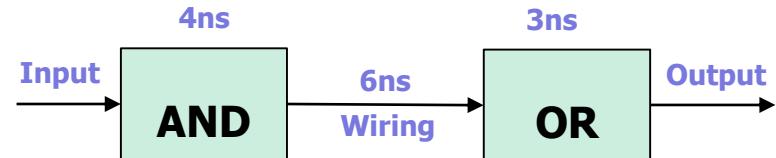
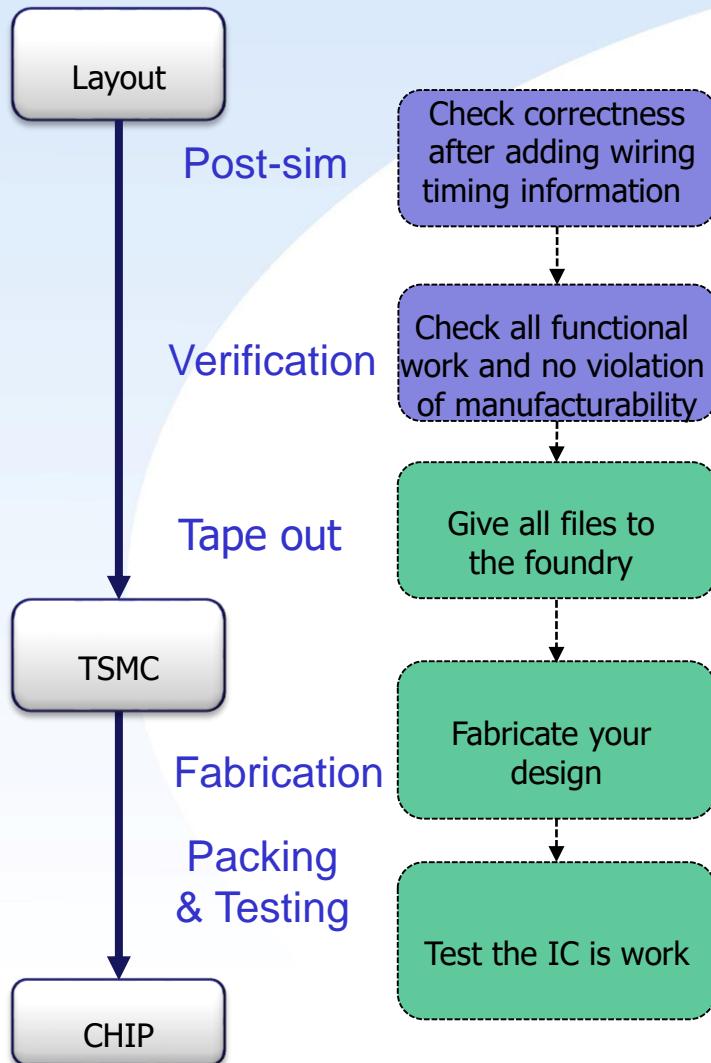
(Source: PDA, Prof. Chen)



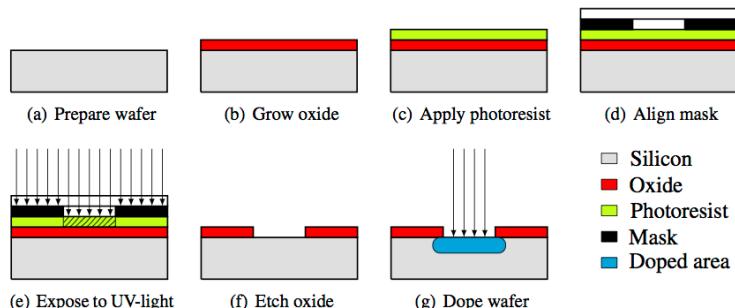
# Layout



# 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

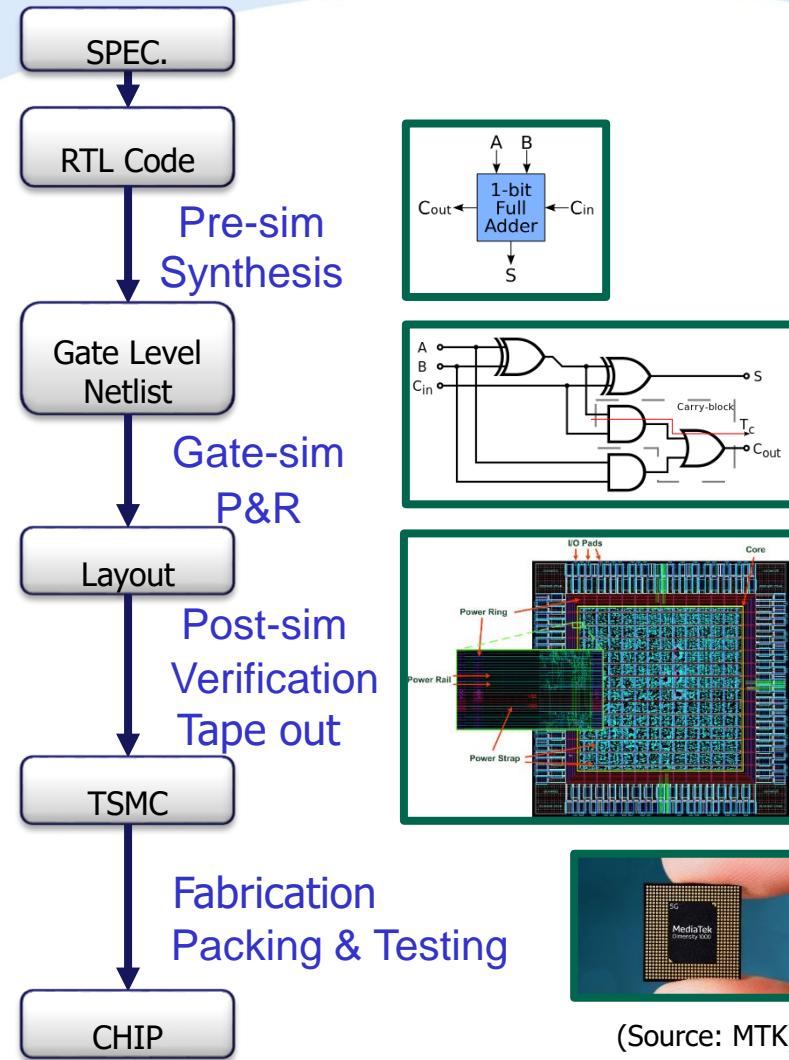
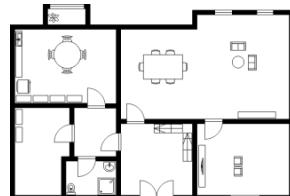
Choose the suitable  
Furniture based on  
Datasheet

## 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**



# Hardware vs. Software

Hardware Parallel	Software Sequential
<pre>module a();     b b1();     c c1();     ... endmodule  module b(); ... endmodule  module c(); ... endmodule</pre>	<pre>void a(){...} void b(){     a();     ... } void c(){     a();     b(); }</pre>



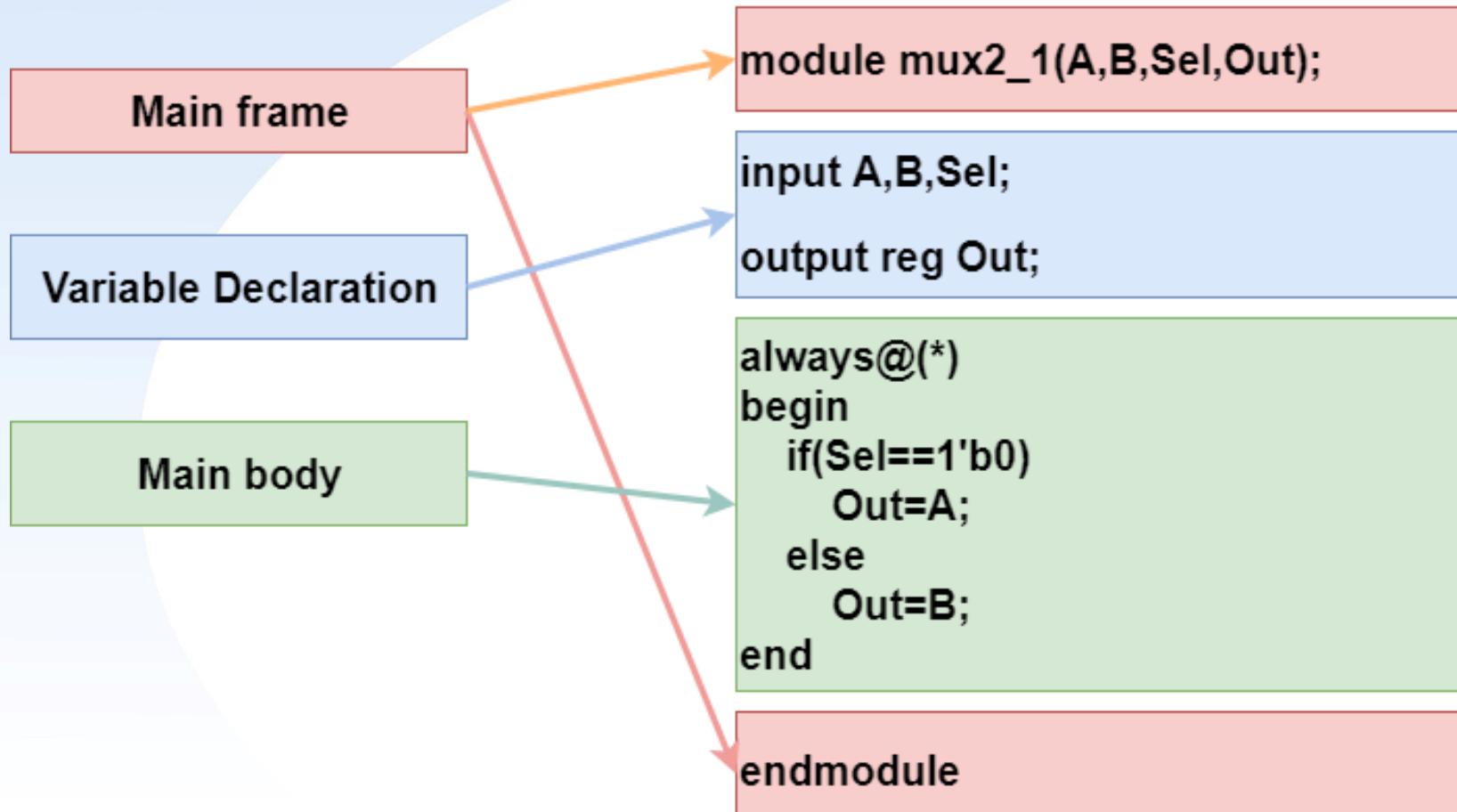
# Verilog

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



# Module

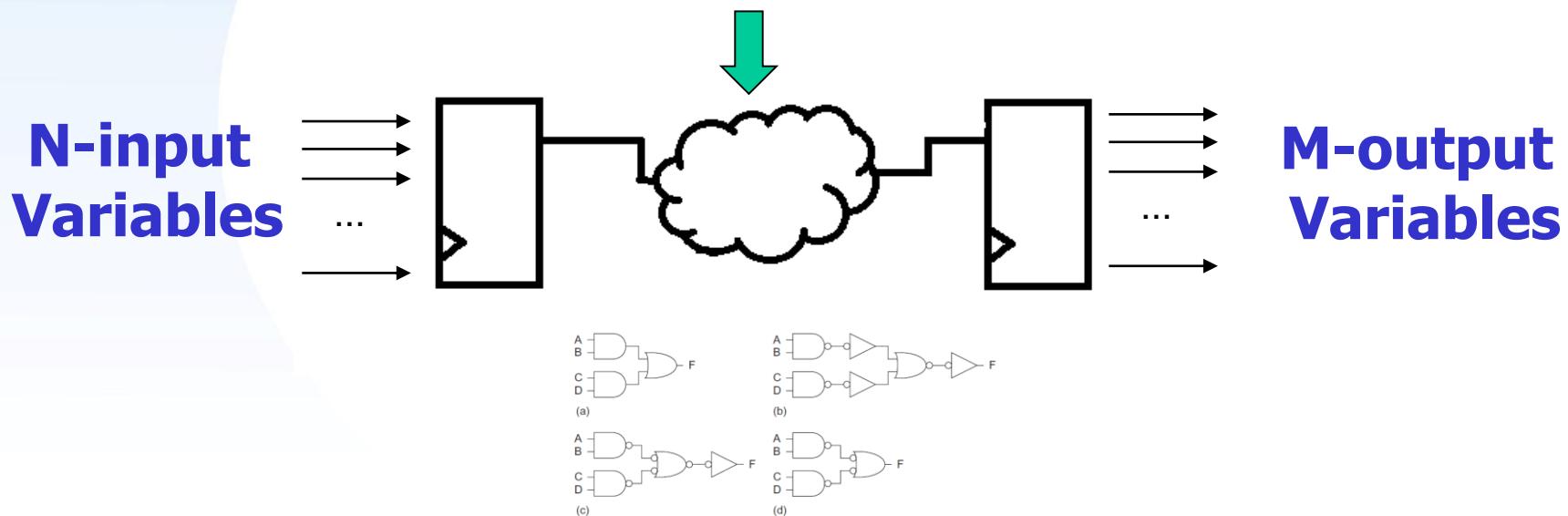
- ✓ All modules run **concurrently**



# Combinational Circuits

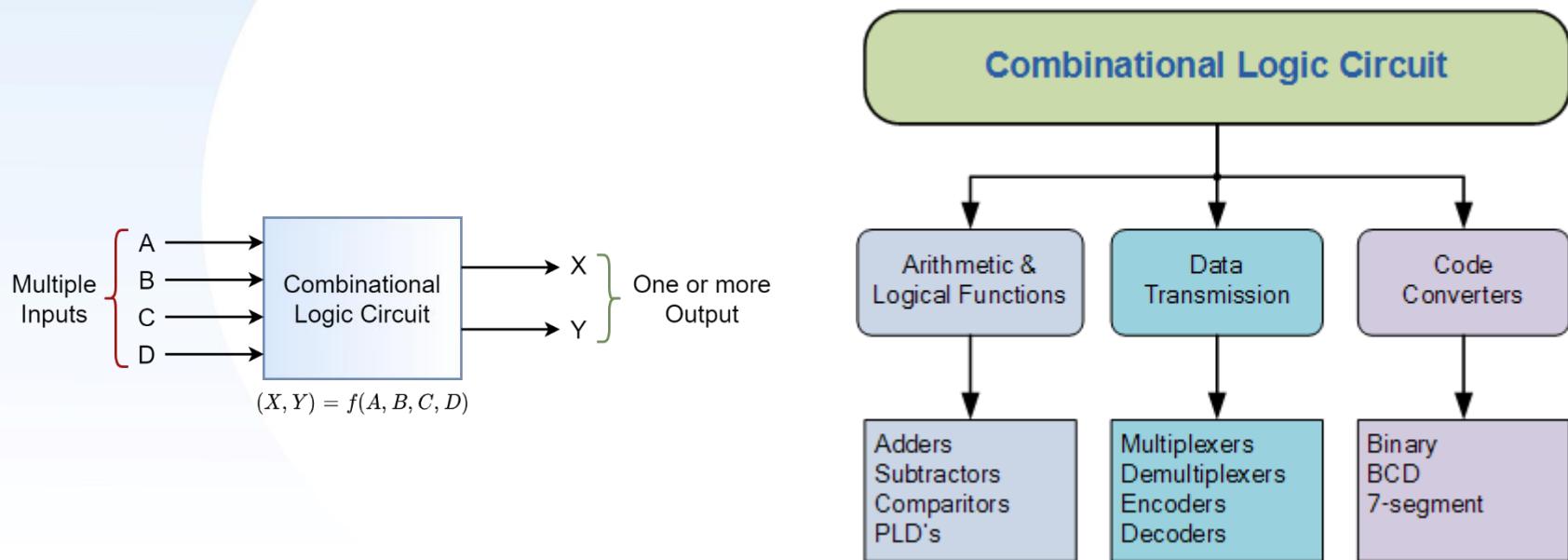
- ✓ The output of combinational circuit depends on the **present input only**.
- ✓ Combinational circuit can be used to do mathematical computation and circuit control.

## Combinational Logic Circuit



# 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.



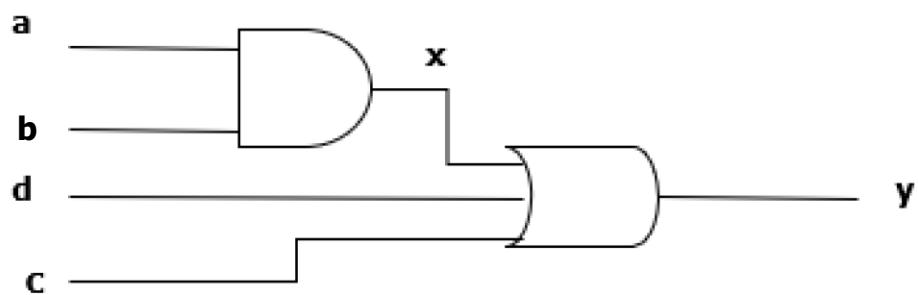
# Behavioral Modeling (1/3)

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

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

**trigger**

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



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

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

Which is better? ?



# Behavioral Modeling (2/3)

## ✓ Data Assignment

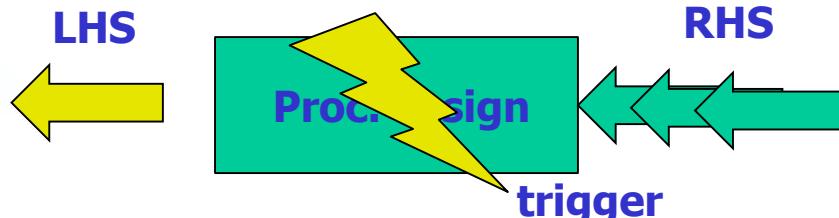
## ✓ Continuous Assignment → for wire assignment

- 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: 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

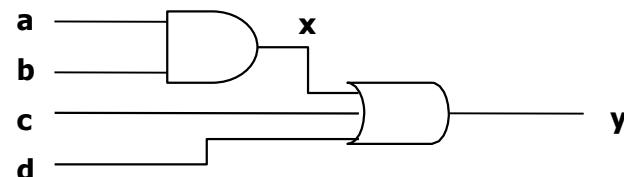
always@(a or b or c or d) begin
    y = x | c | d;
    x = a & b;
end // not in topological
// simulation-synthesis mismatch

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 or x)
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 !!

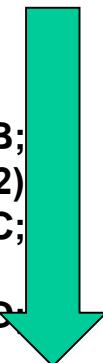


# Procedural block

✓ always, initial

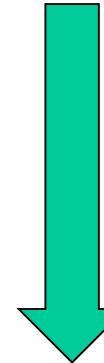
## DESIGN.v

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



## PATTERN.v

```
module PATTERN(sel,out,a,b);
    input out;
    output a,b,sel;
    reg a,b,sel,clk,reset;
    initial begin
        a=0;
        b=0;
        sel=0;
        reset=0;
        clk=0;
    end
```



# 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

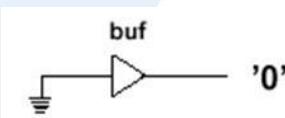
## ✓ Identical(similar) names for connected signals and ports



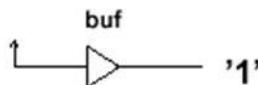
# Data Type (1/5)

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

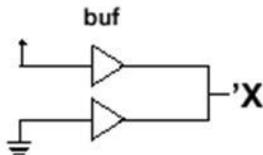
- **0,1:** means low or high signal
- **X :** unknown signal, means that we don't know whether the signal is 0 or 1
- **Z :** high-impedance, the signal is neither 0 nor 1.
- **Avoid X and Z !!!**



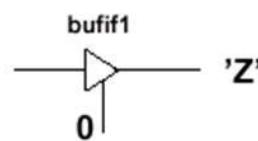
'0', Low, False, Logic Low, Ground,VSS,  
Negative Assertion



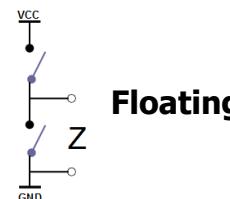
'1', High, True, Logic High, Power,  
VDD, VCC, Positive Assertion



'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 used 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)

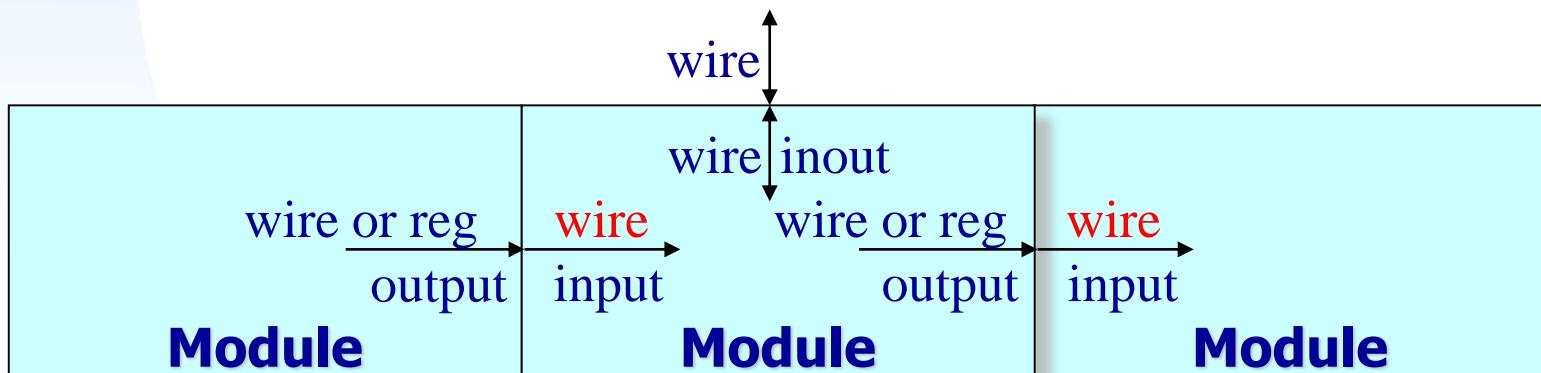
✓ Interface is defined by ports

- 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's 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!**

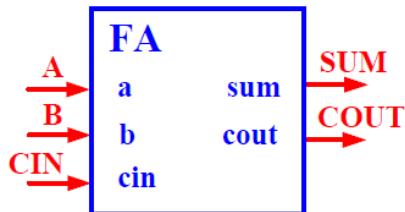


## ✓ Modules connect by name (explicit)

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

**Use this!!!**

**Name Mapping**



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

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

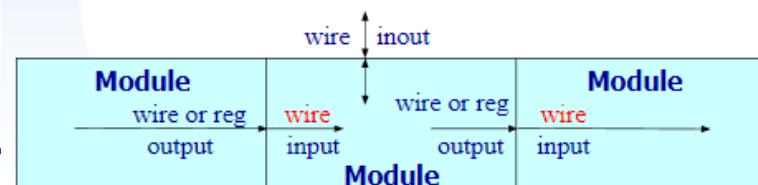
```
`include "mux.v"
module test;
    reg      out;           //incorrect define Wire for connection
    reg      a,b;
    reg      clk,sel,rst;

// 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 Name submodules differently
.....
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**)

- $A = B >> 2;$        $\rightarrow$  shift right 'B' by 2-bit (if  $B = 4'b1000$ ,  $A = 4'b0010$ )
  - $A = B << 2;$        $\rightarrow$  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;  
B = 4'b1000;  
A = B >>>2;      (A = 4'b1110 ,which is 1000 shifted to the right two positions and sign-filled.)



# 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

```
wire [2:0] a;  
wire [2:0] mod1;  
assign mod1 = a % 7;
```

Correct

unsigned    unsigned    signed  
(unsigned operation)

```
wire [2:0] a;  
wire signed [2:0] mod1;  
assign mod1 = a % 7;
```

Correct

signed    unsigned    signed  
(unsigned operation)

```
wire signed [2:0] a;  
wire [2:0] mod1;  
assign mod1 = a % 7;
```

The value may  
not in range 0~6

unsigned    signed    signed  
(signed operation)

```
wire signed [2:0] a;  
wire signed [2:0] mod1;  
assign mod1 = a % 7;
```

The value may  
not in range 0~6

signed    signed    signed  
(signed operation)



# Operators (4/4)

- ✓ Bitwise operators: perform bit-sliced operations on vectors
  - $\sim(4'b0101) = \{\sim0, \sim1, \sim0, \sim1\} = 4'b1010$
  - $4'b0101 \& 4'b0011 = 4'b0001$
- ✓ Logical operators: return one-bit (true/false) results
  - $!(4'b0101) = \sim1 = 1'b0$  , EX: if(a == c && a == c || !b != c )
- ✓ 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
$a ^ b$	XOR
$a \sim^ b$	XNOR

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

Reduction	
$\&a$	AND
$\sim\&$	NAND
$ $	OR
$\sim $	NOR
$^$	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 (? :)

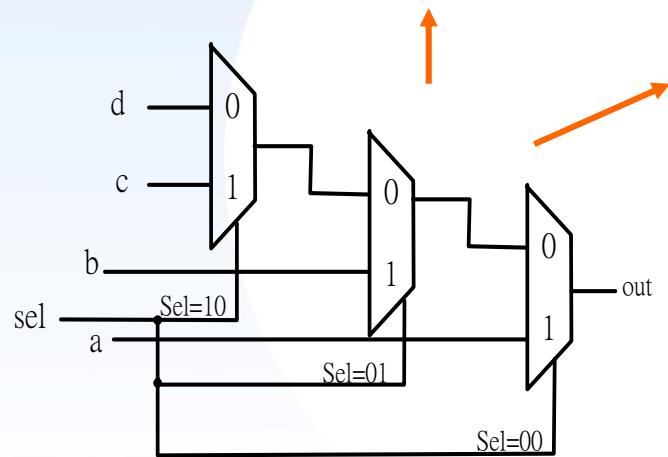
- $? : \rightarrow c = \text{sel} ? a : b;$
- used in cont. assignment
- same as if-else statement



# 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 )));
```

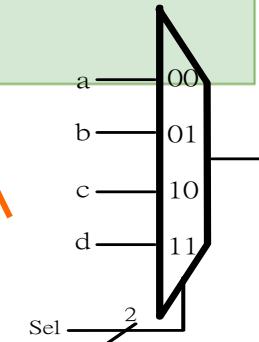


## If-then-else

```
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

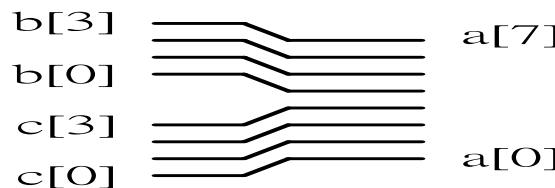
```
always@(*)  
begin  
case(Sel)  
2'b00: data = a;  
2'b01: data = b;  
2'b10: data = c;  
default: data = d;  
endcase  
end
```



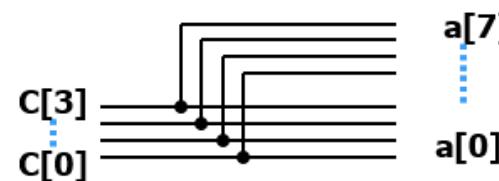
# Concatenation

## – Concatenation

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



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



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



# Gate-Level Modeling (1/3)

## ✓ Primitive logic gate

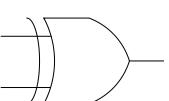
- and



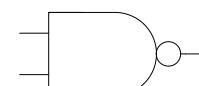
- or



- xor



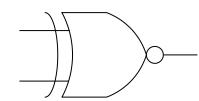
- nand



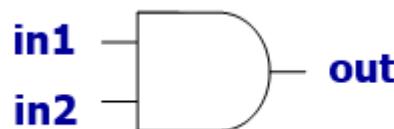
- nor



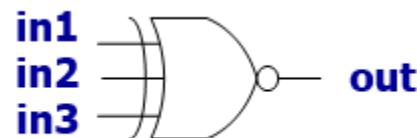
- xnor



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



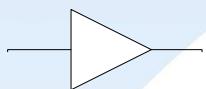
can use with multiple inputs → i.e. xor( out, in1, in2, in3 ) ;



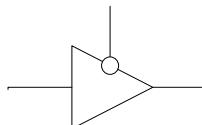
# Gate-Level Modeling (2/3)

## ✓ Primitive logic gate

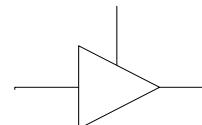
- buf,



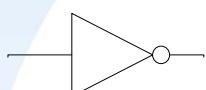
- bufif0,



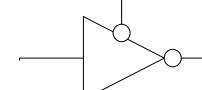
- bufif1



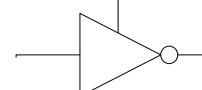
- not,



- notif0,

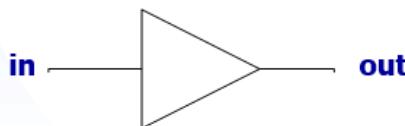


- notif1



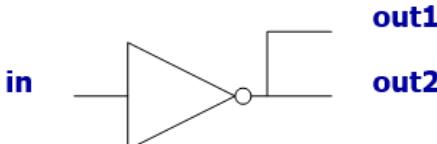
can use without instance name

→ i.e. buf( out, in ) ;



can use with multiple outputs

→ i.e. not( out1, out2 ,in) ;



# Behavioral Model v.s. Gate Level Model

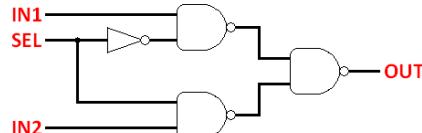
## ✓ Gate Level Model

```
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
```



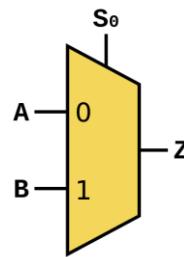
## ✓ Behavioral Model

```
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
```

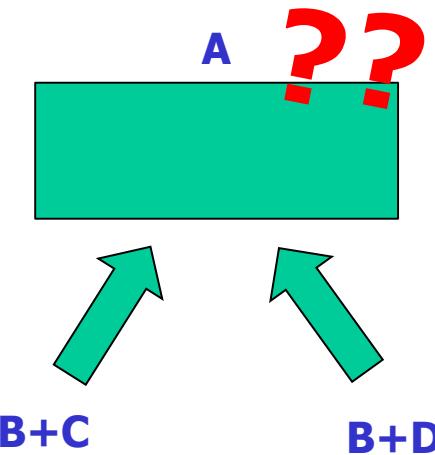
```
endmodule
```



# Coding style

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

```
always@* begin  
    A=B+C;  
end  
always@* begin  
    A=B+D; X  
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
```

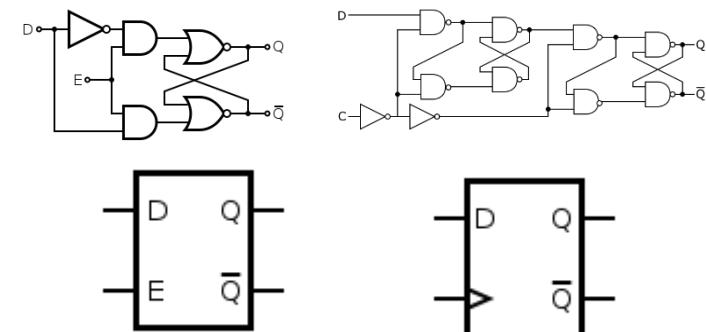


Figure. (a) D-latch (b) D flip-flop

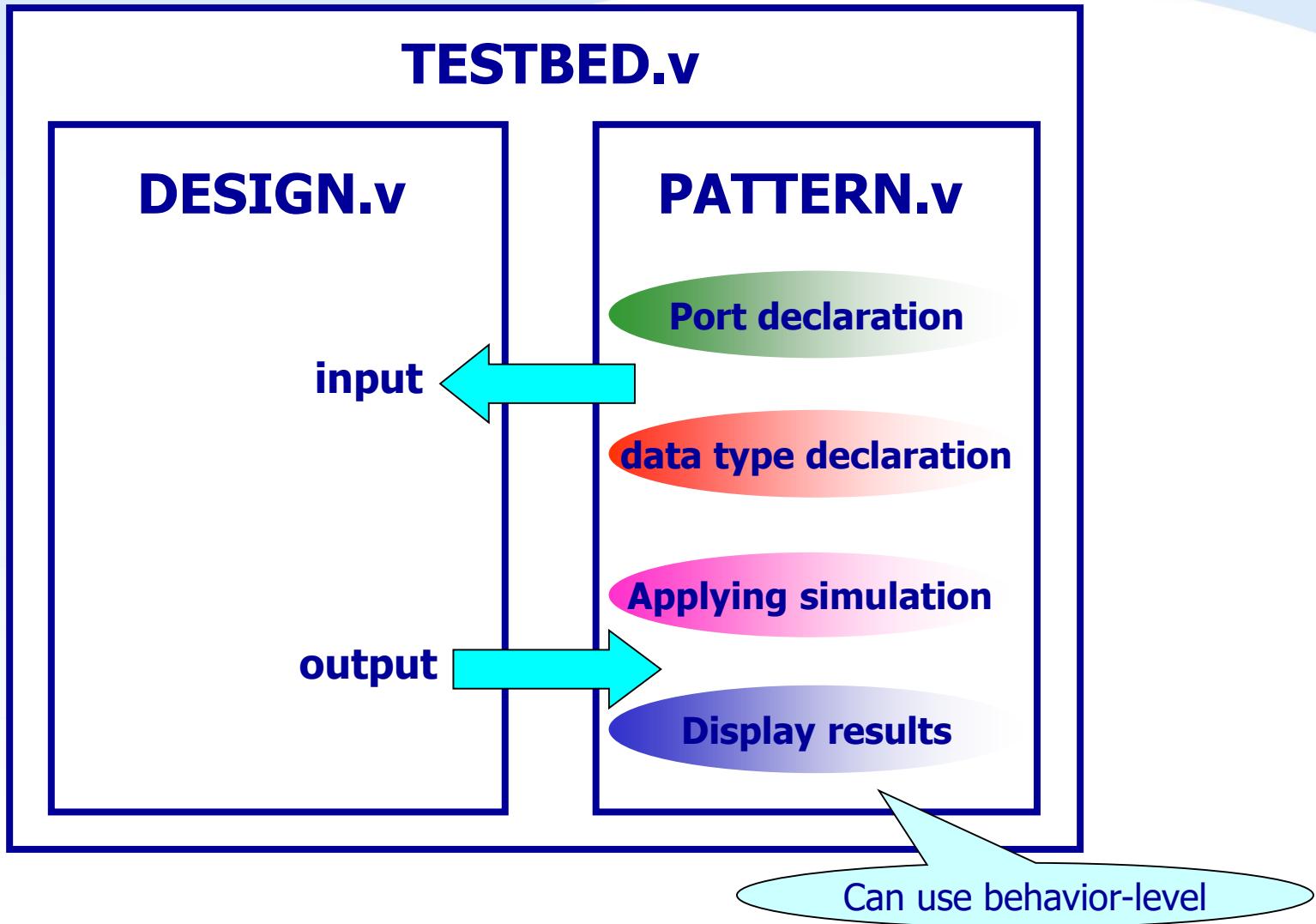


# Outline

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



# Simulation Environment



# 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));

endmodule
```

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
```

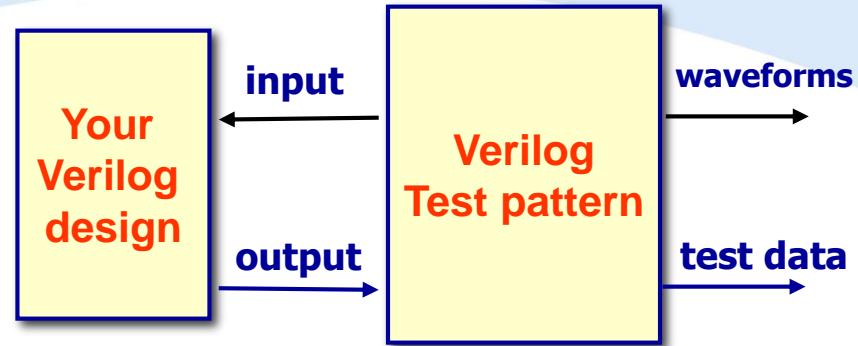
```
endmodule
```



# Simulation Environment (cont.)

## ✓ Simulation command

- Verilog compile
  - irun TESTBED.v -define RTL –debug
- Invoke nWave
  - nWave &
- Stop the simulation and continue the simulation
  - Ctrl+c → 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



```
VCS_RTL_SIM = vcs ${TIMESCALE} \
-j${num_CPU_cores} \
-sverilog \
+v2k \
-full64 \
-Mupdate \
-R \
-debug_access+all \
-y ${DW_SIM} \
+libext+.v \
-f ${source_file} \
-o ${output_file} \
-l ${log_file} \
-P ${VERDI}/share/PLI/VCS/linux64/novas.tab \
| ${VERDI}/share/PLI/VCS/linux64/pli.a \
+define+RTL \
+notimingchecks
```

```
VCS_GATE_SIM = vcs ${TIMESCALE} \
-j${num_CPU_cores} \
-sverilog \
+v2k \
-full64 \
-Mupdate \
-R \
-debug_access+all \
-f ${source_file} \
-o ${output_file} \
-l ${log_file} \
-P ${VERDI}/share/PLI/VCS/linux64/novas.tab \
| ${VERDI}/share/PLI/VCS/linux64/pli.a \
-v ${UMC018_SIM} \
+define+GATE \
-ntb_opts nontcgitch
```



# MakeFile for Simulation

```
#####
# Set your desired file names
#####
source_file="filelist.f"          # Name of your source file
output_file="simv"                # Desired output simulation file name
log_file="vcs.log"                # Desired log file name
fsdb_file="SMC.fsdb"              # Desired log file name

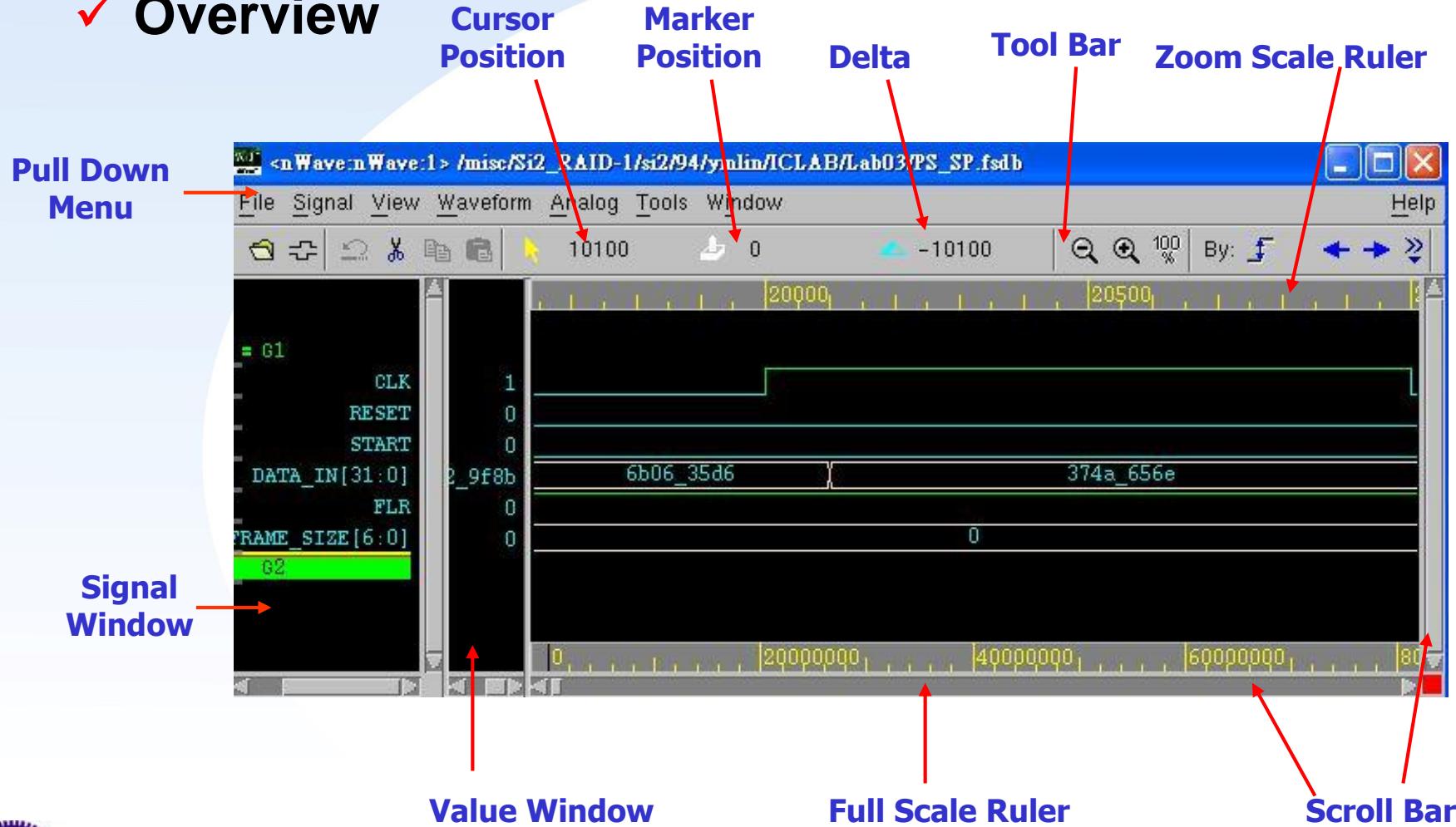
#####
# Default Setting
#####
num_CPU_cores="8"                # Number of CPU that used for VCS
TIMESCALE="-timescale=1ns/1fs"
TIMESCALE1="-timescale 1ns/1fs"
VERDI="/usr/cad/synopsys/verdi/2019.06/"
DW_SIM="/usr/cad/synopsys/synthesis/cur/dw/sim_ver/"
UMC018_SIM="/usr/cad/umc018/Verilog/umc18_neg.v"

# ====== irun ======      # ====== xrun ======
IRUN RTL SIM = irun -f ${source_file} \
-incdir ${DW_SIM} \
-loadpli1 debpli:novas_pli_boot \
-debug \
-notimingchecks \
-define RTL
# ====== 
# ====== 
IRUN GATE SIM = irun -f ${source_file} \
-override_precision \
${TIMESCALE1} \
-sdf_precision 1fs \
-v ${UMC018_SIM} \
-loadpli1 debpli:novas_pli_boot \
-debug \
-nontcgitch \
-define GATE
# ======      # ======
XRUN RTL SIM = xrun -f ${source_file} \
-mcl ${num_CPU_cores} \
-incdir ${DW_SIM} \
-loadpli1 debpli:novas_pli_boot \
-debug \
-notimingchecks \
-define RTL
# ====== 
# ====== 
XRUN GATE SIM = xrun -f ${source_file} \
-mcl ${num_CPU_cores} \
-override_precision \
${TIMESCALE1} \
-sdf_precision 1fs \
-v ${UMC018_SIM} \
-loadpli1 debpli:novas_pli_boot \
-debug \
-nontcgitch \
-define GATE
# ======      # ======
# ====== VCS ======
VCS RTL SIM = vcs ${TIMESCALE} \
-j${num_CPU_cores} \
-sverilog \
+v2k \
-full164 \
-Mupdate \
-R \
-debug_access+all \
-y ${DW_SIM} \
+libext+.v \
-f ${source_file} \
-o ${output_file} \
-l ${log_file} \
-P ${VERDI}/share/PLI/VCS/linux64/novas.tab \
| ${VERDI}/share/PLI/VCS/linux64/pli.a \
+define+RTL \
+notimingchecks
# ====== 
# ====== VCS ======
VCS GATE SIM = vcs ${TIMESCALE} \
-j${num_CPU_cores} \
-sverilog \
+v2k \
-full164 \
-Mupdate \
-R \
-debug_access+all \
-f ${source_file} \
-o ${output_file} \
-l ${log_file} \
-P ${VERDI}/share/PLI/VCS/linux64/novas.tab \
| ${VERDI}/share/PLI/VCS/linux64/pli.a \
-v ${UMC018_SIM} \
+define+GATE \
-ntb_opts nontcgitch
```



# nWave

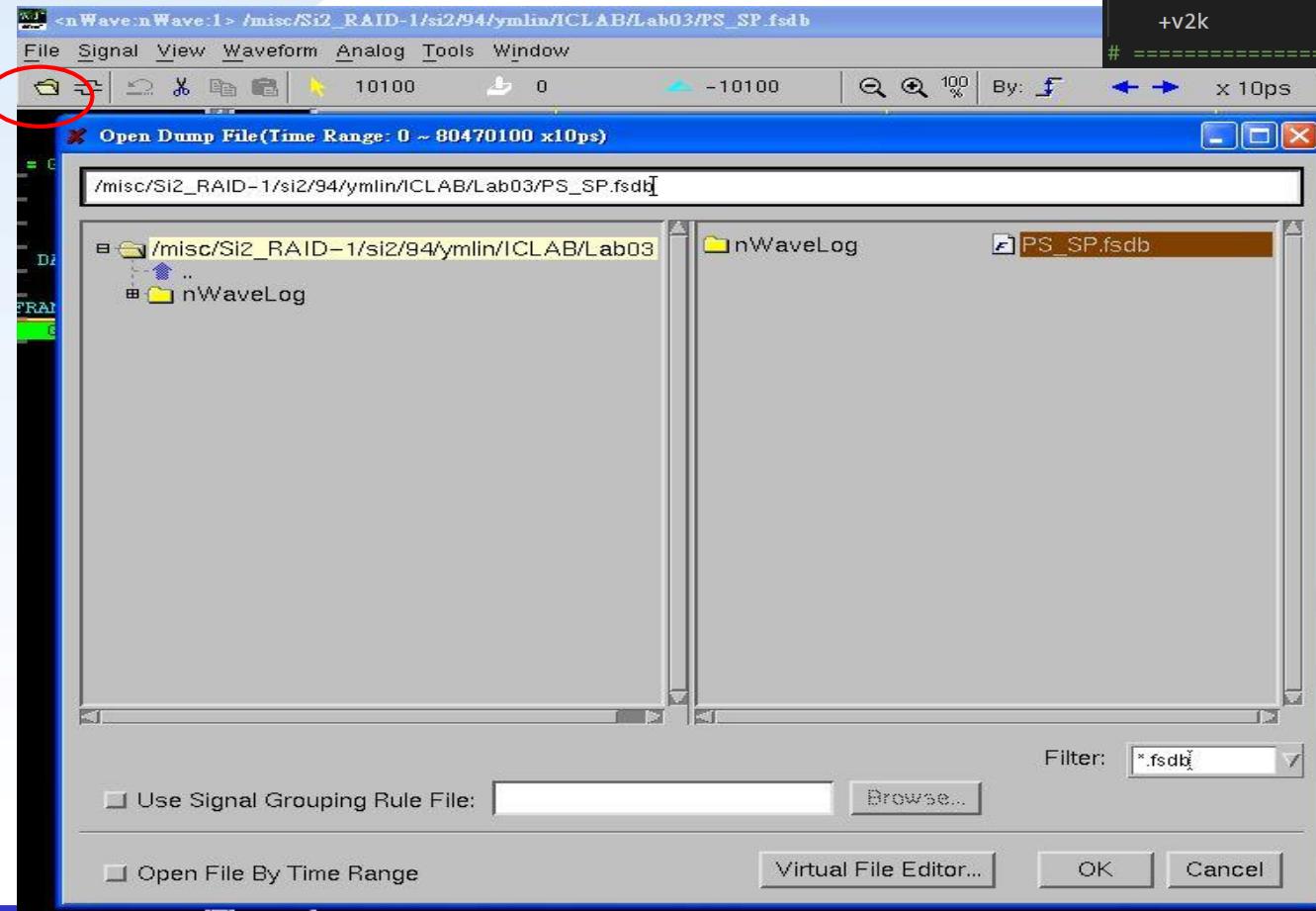
## ✓ Overview



# nWave (cont.)

## ✓ Open fsdb file

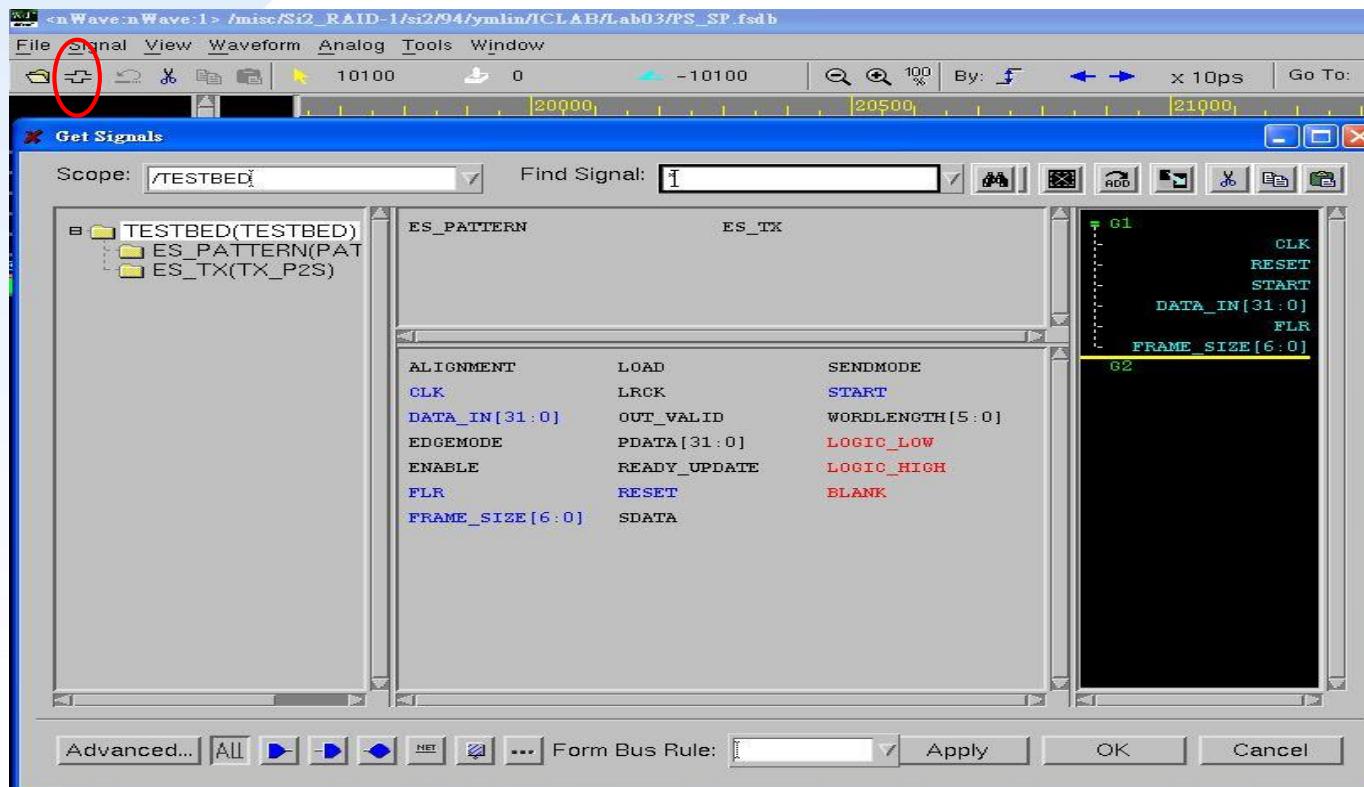
- Use **File → Open...** command



```
# ====== verdi ======
nWave_ON = nWave -ssf ${fsdb_file}
#
# ======
VERDI_ON = verdi -ssf ${fsdb_file} \
    -sv \
    -f ${source_file} \
    -nologo \
    +define+RTL \
    +v2k
# ======
```

## ✓ Get signal

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

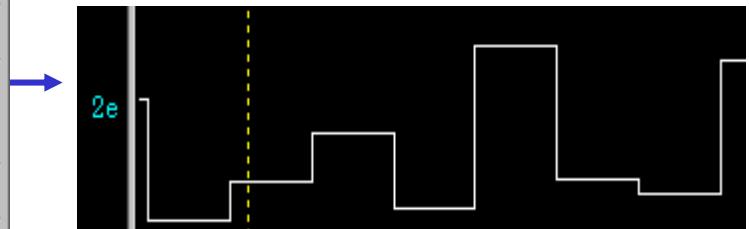
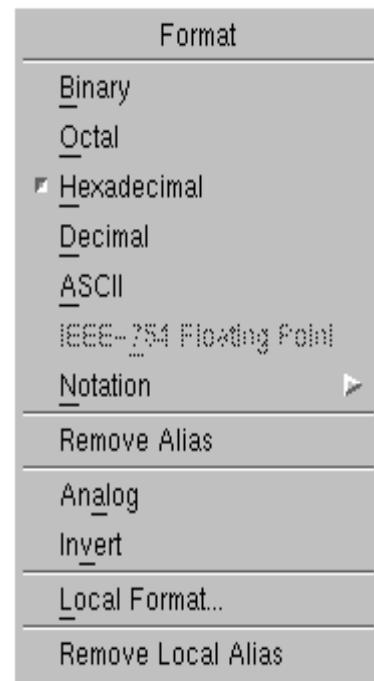


## ✓ Choose value format

- On the value window click Left Button

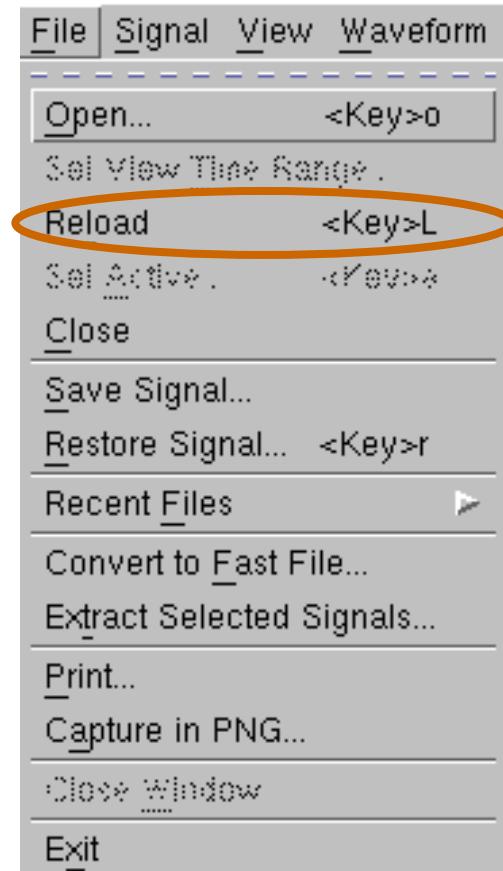


**Default : Hexadecimal**



## ✓ Reload nWave

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



# Q & A

✓ Any question ?



# 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

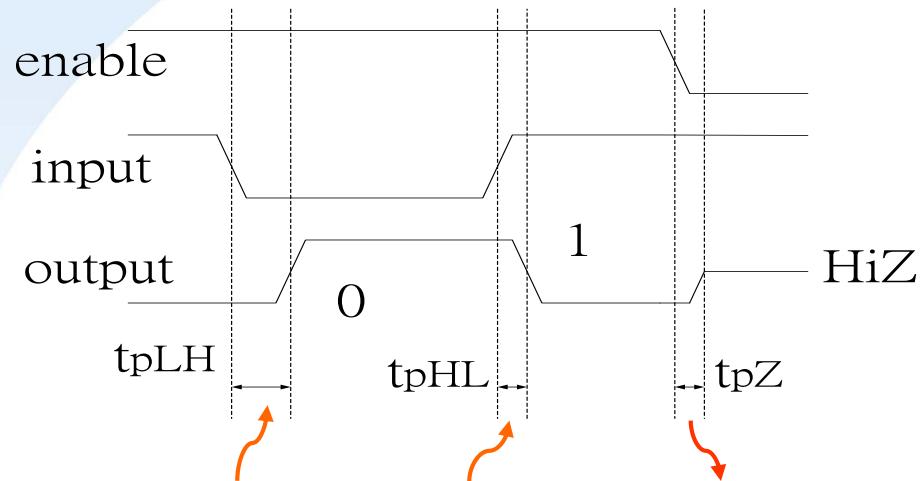
2021revised Lin-Hung Lai ([h123572119@gmail.com](mailto:h123572119@gmail.com))



# Appendix - Gate-Level Modeling with delay

## ✓ Gate Delays

- Rise Delays, Fall Delays, and Turn-off Delays



```
gate #( rise_delay, fall_delay, turnoff_delay ) a3( out, in, enable ) ;
gate #( rise_delay, fall_delay )           a2( out, i1, i2 ) ;
Only bufif0, bufif1, notif0, notif1 have turn-off delays
i.e. bufif1 #(3,4,5) ( out, in, enable ); //rise=3,fall=4,turnoff=5
```



# Appendix - Gate-Level Modeling with delay

## ✓ Gate Delays

- Min/Typ/Max delay time
  - 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



# Appendix-Simulation Command

## ✓ Dump a FSDB file for debug

### – General debussy waveform generator

- \$fsdbDumpfile("file\_name.fsdb");
- \$fsdbDumpvars;

### – 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

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

- Dump an fsdb file
- Dump all values

## ✓ Put the above command in an initial block

