

# Computer-Aided VLSI System Design

## Chap.1-1 Fundamentals of Hardware Description Language

Lecturer: 張惇宥

*Graduate Institute of Electronics Engineering, National Taiwan University*



NTU GIEE



# Verilog Course Overview

## ❖ Chapter 1

- Fundamentals of HDL
- Language element
- Structural level modeling
- RTL (Dataflow) modeling

## ❖ Chapter 2

- Signing and Sizing
- Behavioral level modeling

## ❖ Chapter 3

- Synthesizable verilog coding
- Debugging and testbench

## ❖ Chapter 4

- Architecture improvement of timing, area, and power
- From spec. to circuit



# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



# Hardware Description Language

*From Wikipedia*

- ❖ Hardware Description Language (HDL) is any language from a class of computer languages and/or programming languages for formal description of electronic circuits, and more specifically, **digital logic**.
- ❖ HDL can
  - **Describe** the circuit's operation, design, organization
  - **Verify** its operation by means of simulation.
- ❖ Now HDLs usually merge Hardware Verification Language, which is used to verify the described circuits.
- ❖ Supporting discrete-event (digital) or continuous-time (analog) modeling, e.g.:
  - ❖ SPICE, Verilog HDL, VHDL, SystemC



# High-Level Programming Language

- ❖ It's possible to describe hardware (operation, structure, timing, and testing methods) in C/C++/Java, why do we use HDL?
- ❖ The **efficiency** (to model/verify) does matter.
  - Native support to **concurrency**
  - Native support to the simulation of the **progress of time**
  - Native support to simulate the model of **system**
- ❖ The required level of detail determines the language we use.



# List of HDL for Digital Circuits

## ❖ Verilog

## ❖ **VHDL**

- ❖ Advanced Boolean Expression Language (ABEL)
- ❖ AHDL (Altera HDL, a proprietary language from Altera)
- ❖ Atom (behavioral synthesis and high-level HDL based on Haskell)
- ❖ Bluespec (high-level HDL originally based on Haskell, now with a SystemVerilog syntax)
- ❖ Confluence (a functional HDL; has been discontinued)
- ❖ CUPL (a proprietary language from Logical Devices, Inc.)
- ❖ Handel-C (a C-like design language)
- ❖ C-to-Verilog (Converts C to Verilog)
- ❖ HDCaml (based on Objective Caml)
- ❖ Hardware Join Java (based on Join Java)
- ❖ HML (based on SML)
- ❖ Hydra (based on Haskell)
- ❖ Impulse C (another C-like language)
- ❖ JHDL (based on Java)
- ❖ Lava (based on Haskell)
- ❖ Lola (a simple language used for teaching)
- ❖ MyHDL (based on Python)

- ❖ PALASM (for Programmable Array Logic (PAL) devices)
- ❖ Ruby (hardware description language)
- ❖ RHDL (based on the Ruby programming language) SDL based on Tcl.
- ❖ CoWareC, a C-based HDL by CoWare. Now discontinued in favor of SystemC

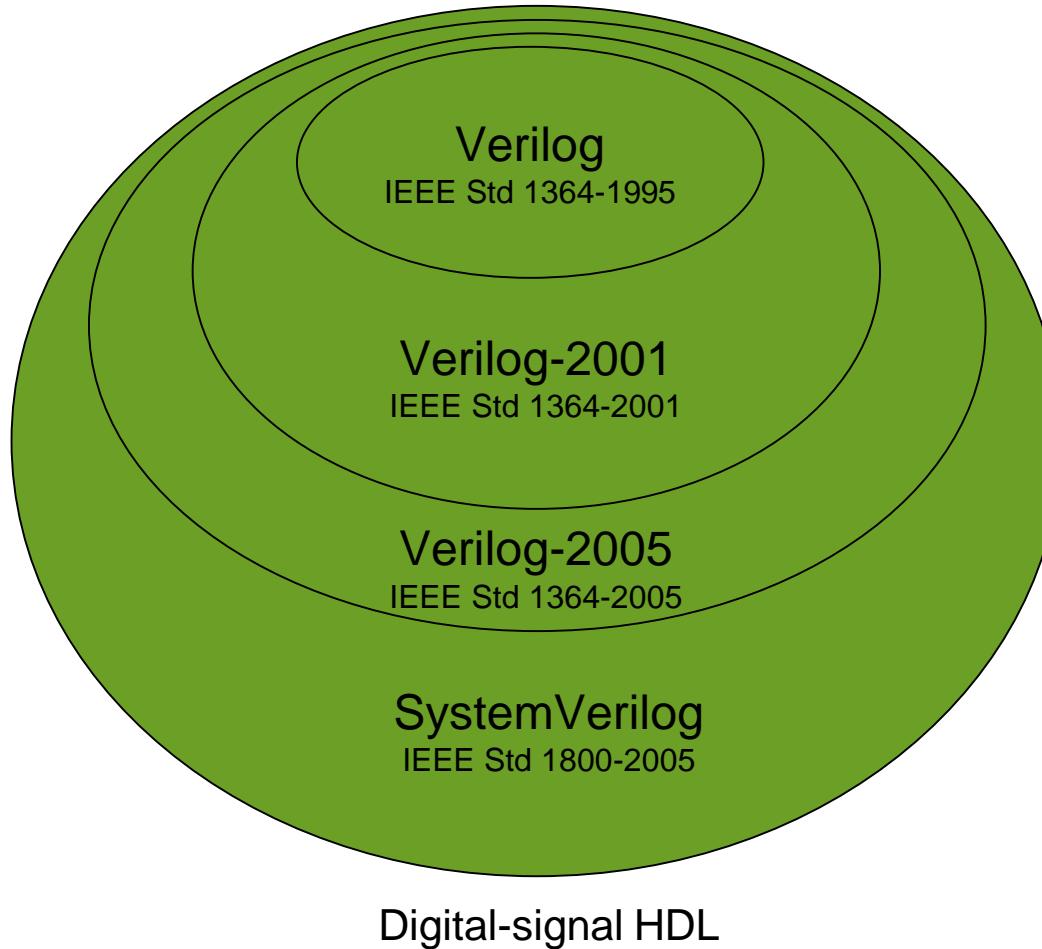
❖ **SystemVerilog**, a superset of Verilog, with enhancements to address system-level design and verification

❖ **SystemC**, a standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level

- ❖ SystemTCL, SDL based on Tcl.



# History/Branch of Verilog





# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



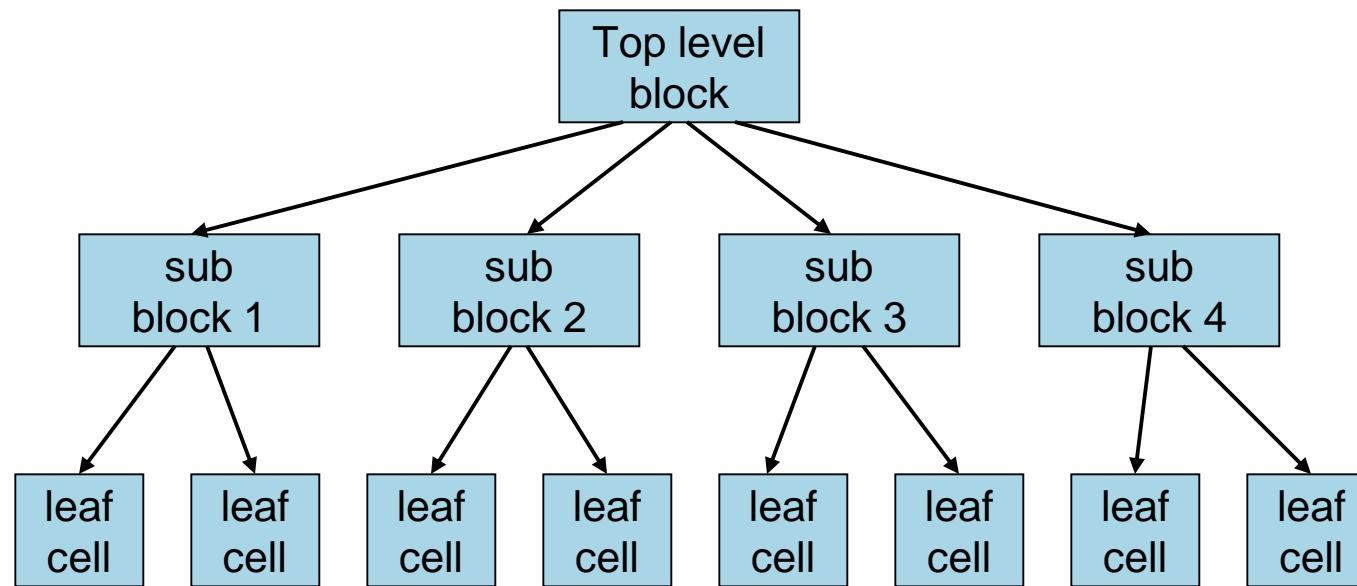
# Hierarchical Modeling Concept

- ❖ Introduce *top-down* and *bottom-up* design methodologies
- ❖ Introduce *module* concept and encapsulation for hierarchical modeling
- ❖ Explain differences between modules and module instances in Verilog



# Top-down Design Methodology

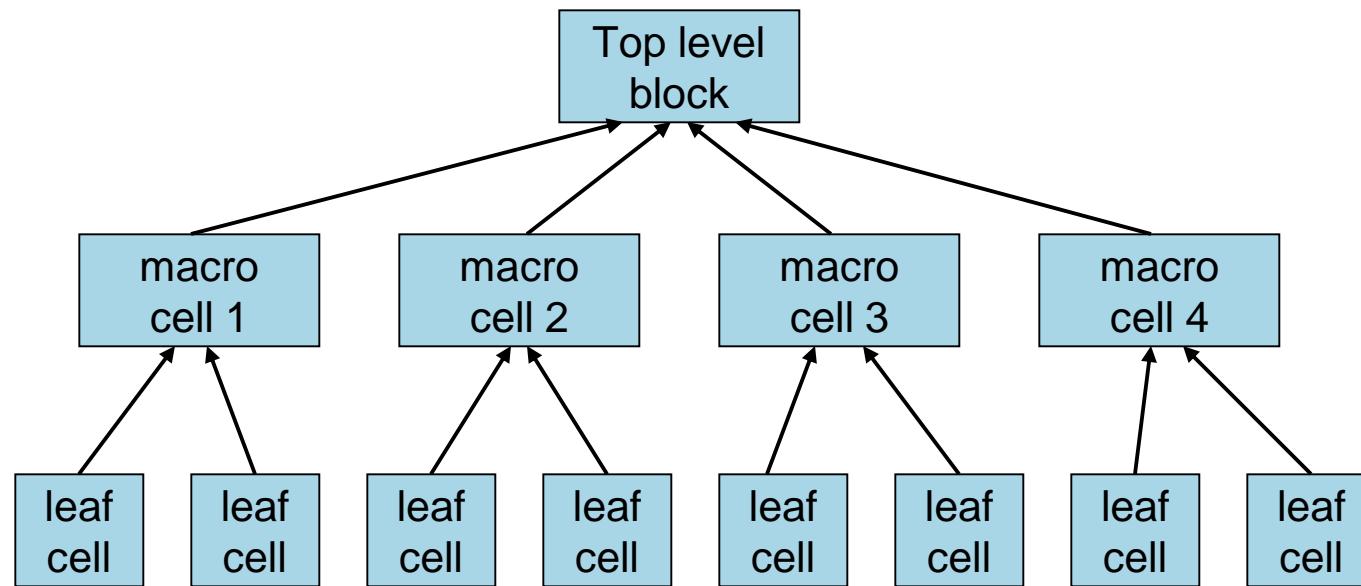
- ❖ We define the top-level block and identify the sub-blocks necessary to build the top-level block.
- ❖ We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.





# Bottom-up Design Methodology

- ❖ We first identify the building block that are available to us.
- ❖ We build bigger cells, using these building blocks.
- ❖ These cells are then used for higher-level blocks until we build the top-level block in the design.

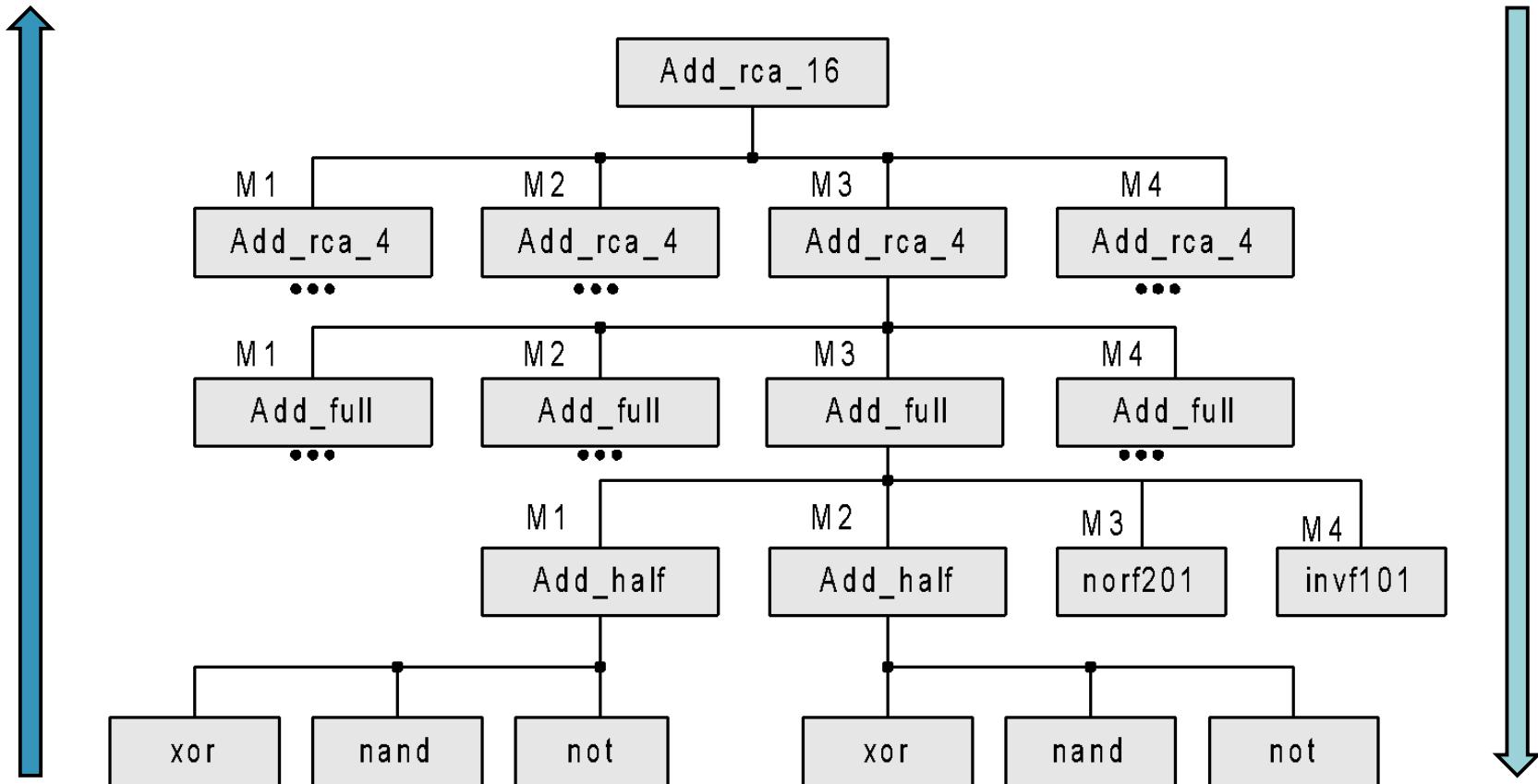




# Example: 16-bit Adder

conquer

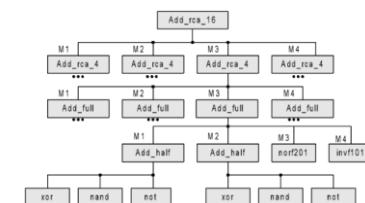
divide





# Hierarchical Modeling in Verilog

- ❖ A Verilog design consists of a hierarchy of modules.
- ❖ **Modules** encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional **ports**.
- ❖ Internally, a module can contain any combination of the following
  - net/variable declarations (wire, reg, integer, etc.)
  - concurrent and sequential statement blocks
  - instances of other modules (sub-hierarchies).





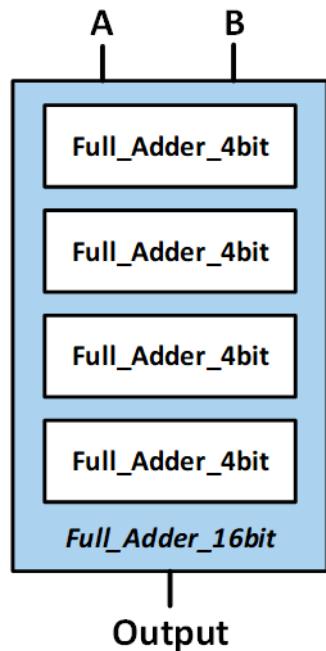
# Module

- ❖ Basic building block in Verilog.
- ❖ Module
  1. Created by “**declaration**” (**can’t be nested**)
  2. Used by “**instantiation**”
- ❖ Interface is defined by ports
- ❖ May contain instances of other modules
- ❖ All modules run concurrently



# Module Declaration (1/2)

- ❖ Module Declaration encapsulates structural and functional details in a module



```
module <Module Name> (<PortName List>);  
  
    // Structural part  
    <List of Ports>  
    <Lists of Nets and Registers>  
    <SubModule>  
  
    // Behavior part  
    <Timing Control Statements>  
    <Parameter/Value Assignments>  
    <System Task>  
endmodule
```

- ❖ Encapsulation makes the model available for instantiation in other modules



## Module Declaration (2/2)

- ❖ The descriptions of the logic can be placed inside modules
- ❖ Modules can represent:
  - A physical block such as an ASIC standard cell
  - A logical block such as the GPU portion of an SoC design
  - The complete system
- ❖ Every module description starts with the keyword ***module***, has a name, and ends with the keyword ***endmodule***

```
module FullAdder16 (O, A, B);
    input [15:0] A, B;
    output [15:0] O;

    ... // logic description
endmodule
```



# Module Ports

- ❖ Modules communicate with each other through **ports**
- ❖ There are three kinds of ports: input, output, and inout
- ❖ There are two ways to declare module ports
  1. List a module's ports in parentheses "()" after the module name, and declare ports in the module description
  2. List and declare a module's ports in parentheses "()" after the module name at the same time

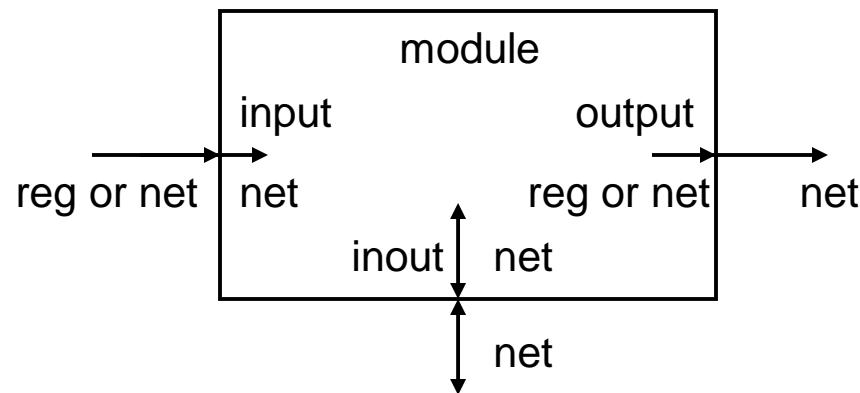
```
module FullAdder16 (O, A, B);
    input [15:0] A, B;
    output [15:0] O;
    ...
endmodule
```

```
module FullAdder16 (
    input [15:0] A,
    input [15:0] B,
    output [15:0] O
);
    ...
endmodule
```



# Port Declaration

- ❖ Three port types
  - Input port
    - input a;
  - Output port
    - output b;
  - Bi-direction port
    - inout c;





## Module Instantiation (2/2)

adder adder\_0 ( out0 , in1 , in2 );  
Module name Instance name

```
module adder(out,in1,in2);
    output  out;
    input   in1, in2;

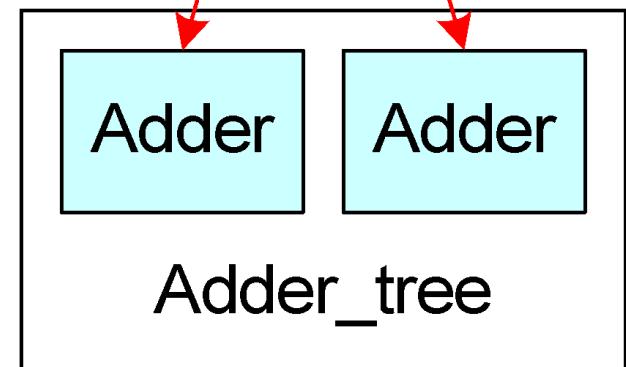
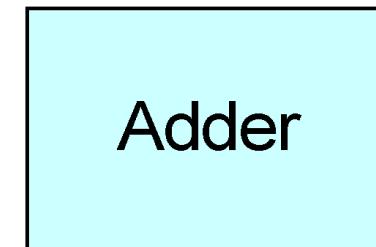
    assign  out=in1 + in2;
endmodule
```

instance  
example

```
module adder_tree (out0,out1,in1,in2,in3,in4);
    output  out0,out1;
    input   in1,in2,in3,in4;

    adder  add_0 (out0,in1,in2);
    adder  add_1 (out1,in3,in4);

endmodule
```

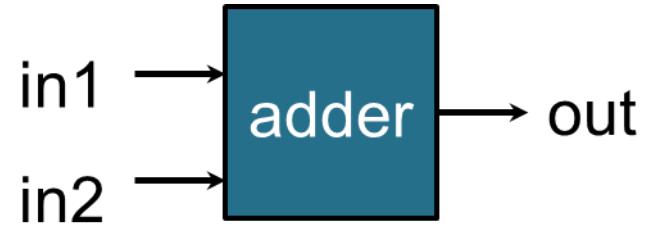




# Port Connection

```
module adder (out,in1,in2);
    output out;
    input  in1 , in2;

    assign out = in1 + in2;
endmodule
```



- Connect module ports by *order list*
  - adder adder\_0 ( C , A , B ); //  $C = A + B$
- Connect module ports by *name (Recommended)*
  - Usage: .PortName (NetName)
  - adder adder\_1 ( .out(C) , .in1(A) , .in2(B) );
- Not fully connected (**Avoid**)
  - adder adder\_2 ( .out(C) , .in1(A) , .in2() );



# Outline

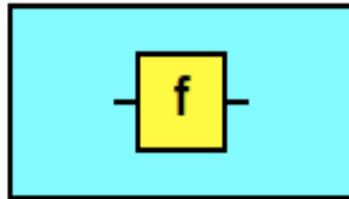
- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



# Cell-Based Design and

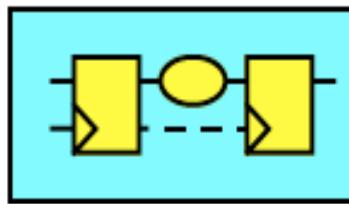
## Levels of Modeling

Behavioral Level



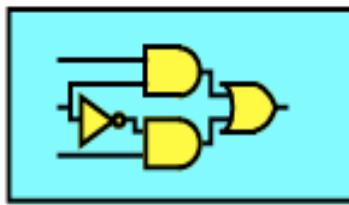
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

Structural/Gate Level



Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library

Transistor/Physical Level

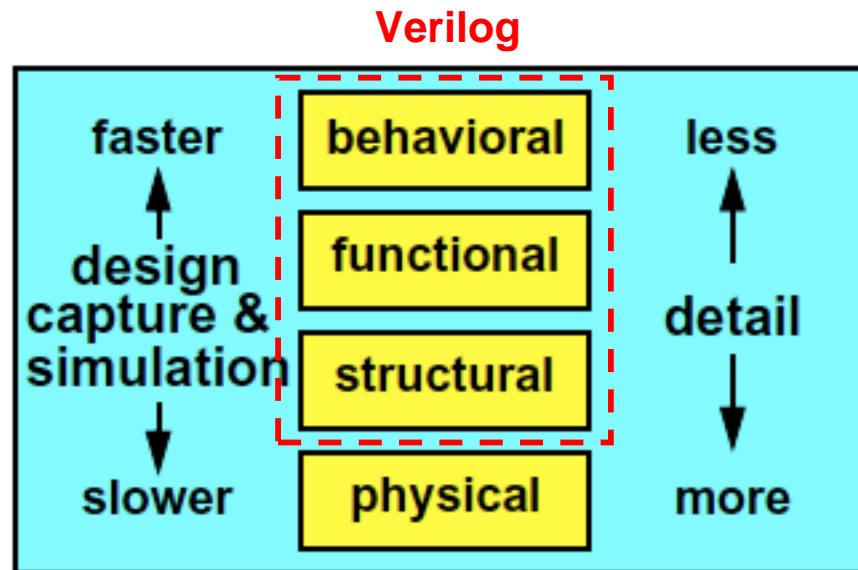


Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE



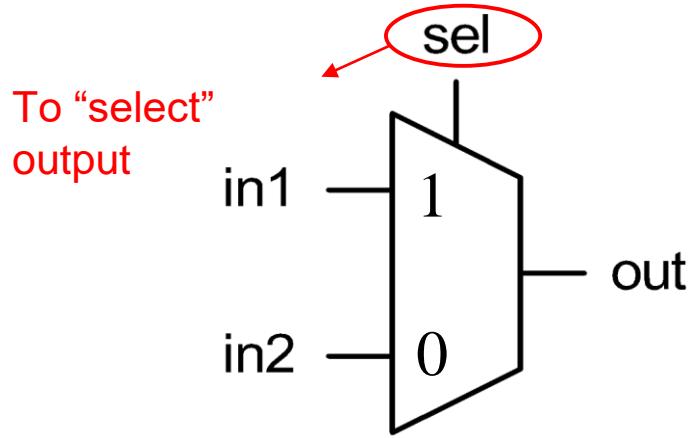
# Tradeoffs Among Modeling Levels

- ❖ Each level of modeling permits modeling at a higher or lower level of detail. More detail means more effort for designers and the simulator.





# An Example - 1-bit Multiplexer in Behavioral Level



```
module mux2 (out, in1, in2, sel);
    input in1, in2, sel;
    output reg out;

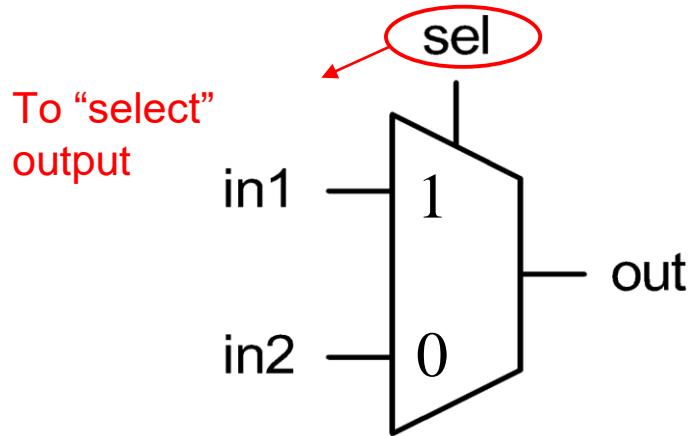
    always@(*) begin
        if (sel) out = in1;
        else      out = in2;
    end

endmodule
```

Behavioral Level modeling code != non-synthesizable code



# An Example - 1-bit Multiplexer in RTL Level



```
module mux2(out,in1,in2,sel);
    output out;
    input in1,in2,sel;
    assign out=sel?in1:in2;
endmodule
```

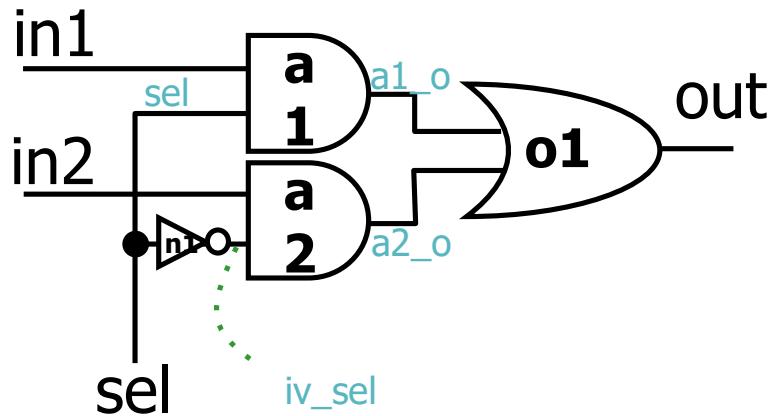
Continuous assignment

**out = sel? In1 : in2**

**RTL: describe logic/arithmetic function between input node and output node**



# An Example - 1-bit Multiplexer in Gate Level



```
module mux2(out,in1,in2,sel);
    output out;
    input in1,in2,sel;
    wire iv_sel, a1_o, a2_o;
    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or o1(out,a1_o,a2_o);
endmodule
```

Gate Level: you see only netlist (gates and wires) in the code



# Gate Level Modeling

## ❖ Steps

- Develop the Boolean function of output
- Draw the circuit with logic gates/primitives
- Connect gates/primitives with net (usually wire)



# Summary: Verilog Basic Cell

## ❖ Verilog Basic Components

## ❖ Declarations

- port declarations
- nets or reg declarations
- parameter declarations

## ❖ Instantiations & Assignment

- gate instantiations
- module instantiations
- continuous assignments

## ❖ Behavioral modeling

- Function definitions
- always blocks
- task statements

```
module test (out, in1, in2, sel);
    // declaration
    input in1, in2, sel;
    output out;
    wire sel_inv;
    reg   out_r;

    // instantiation
    INV u_inv (sel_inv, sel);
    // assignment
    assign out = out_r;

    // behavioral modeling
    always@(*) begin
        if (sel_inv) out_r = in1;
        else         out_r = in2;
    end

endmodule
```



# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



# Verilog Language Rules

- ❖ Verilog is a **case sensitive** language (with a few exceptions)
  - **Avoid to use**
- ❖ Terminate lines with semicolon ;
- ❖ Single line comments:
  - // A single-line comment goes here
- ❖ Multi-line comments:
  - /\* Multi-line comments like this
  - Multi-line comments like this \*/



# Identifiers

- ❖ Identifiers are used to give an object, such as a register or a module, a **name** so that it can be referenced from other places in the code
- ❖ **Identifiers** are a space-free sequence of symbols
  - upper and lower case letters from the alphabet
  - digits (0, 1, ..., 9)
  - underscore ( \_ )
  - \$ symbol (for system tasks)
  - Max length of 1024 symbols
- ❖ The first character of identifiers should be a letter or underscore
  - Should not be a digit or \$

• `m_axi_address`, `_psum`    • `8bit_result`, `a+b`, `$n321`

correct

wrong



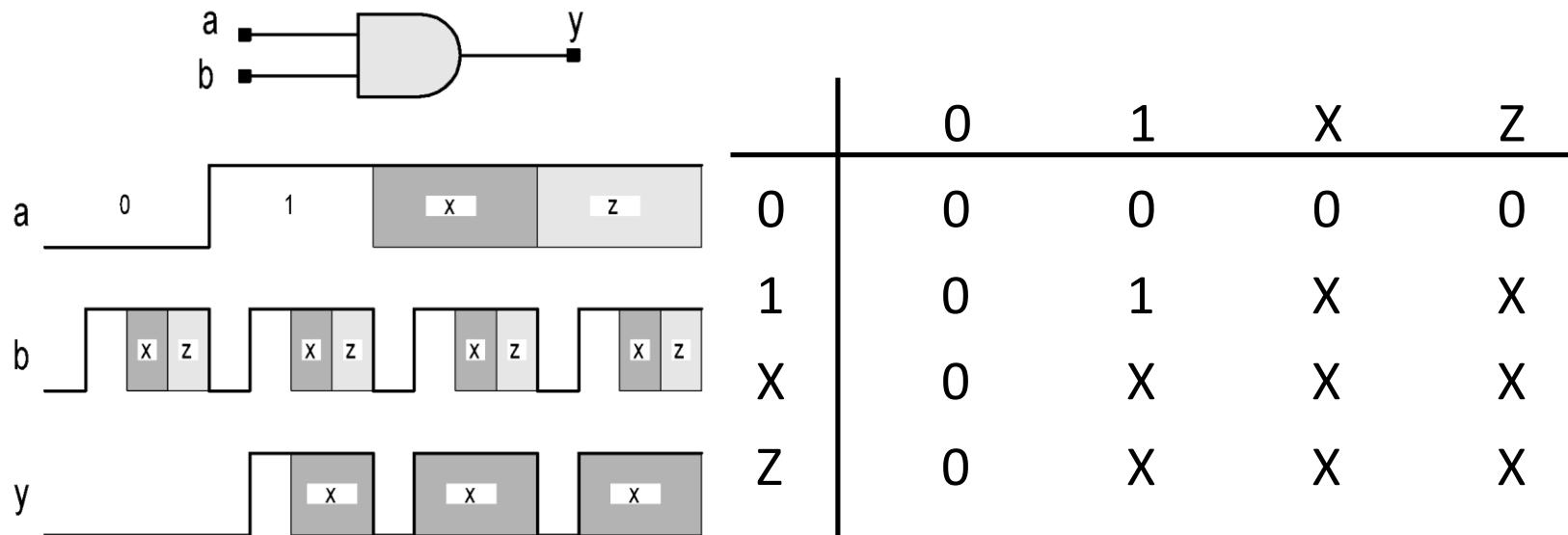
# Four-Valued Logic System

- ❖ Verilog's nets and registers hold four-valued data
  - 0 represent a logic **low** or **false** condition
  - 1 represent a logic **high** or **true** condition
  - z
    - Output of an undriven tri-state driver – high-impedance value
    - Models case where nothing is setting a wire's value
  - x
    - Models when the simulator can't decide the value – uninitialized or unknown logic value
      - Initial state of registers
      - When a wire is being driven to 0 and 1 simultaneously
- ❖ Unknown value would propagate
  - Unknown input would cause unknown output, and make following stage all become unknown



# Logic System in Verilog

- ❖ Four values: 0, 1, x or X, z or Z // Not case sensitive here
  - The logic value **x** denotes an unknown (ambiguous) value
  - The logic value **z** denotes a high impedance
- ❖ Primitives (logic gate) have built-in logic
- ❖ Simulators describe 4-value logic





# Constants Specified Methods

- ❖ Typically constants are used to specify conditions, states, width of vector, entry number of array, and delay

1. parameter                                 parameter BUS\_WIDTH = 8;
2. `define compiler directive                  `define BUS\_WIDTH 8
3. localparam                                 localparam BUS\_WIDTH = 8;

- ❖ Example:

```
module var_mux(out, v0, v1, sel);
  parameter width = 2, flag = 1'b1;
  output [width-1:0] out;
  input [width-1:0] v0, v1;
  input sel;

  assign out = (sel==flag) ? v1 : v0;
endmodule
```

Good for flexibility and reusability

- If sel = 1, then v1 will be assigned to out;
- If sel = 0, then v0 will be assigned to out;



# Parameters

- ❖ Using parameters to declare run-time constants
- ❖ Parameter can only be known in the module it is defined
- ❖ Parameter definition syntax is
  - `parameter <list_of_assignments>`
- ❖ List of assignments is separated by comma

```
module var_mux(out, v0, v1, sel);
    parameter width = 2, flag = 1'b1,
              file = "../golden_0.dat";
    output [width-1:0] out;
    input [width-1:0] v0, v1;
    input sel;

    assign out = (sel==flag) ? v1 : v0;
endmodule
```



# Overriding the Values of Parameters

- ❖ You can use **defparam** to group all parameter value override assignment in one module.

```
module top;
.....
wire [1:0] a_out, a0, a1;
wire [3:0] b_out, b0, b1;
wire [2:0] c_out, c0, c1;

var_mux U0(a_out, a0, a1, sel);
var_mux U1(b_out, b0, b1, sel);
var_mux U2(c_out, c0, c1, sel);

.....
endmodule
```

```
defparam
    top.U0.width = 2;
    top.U0.delay = 1;
    top.U1.width = 4;
    top.U1.delay = 2;
    top.U2.width = 3;
    top.U2.delay = 1;
```

Verilog 2001:

```
var_mux #( .width(2), .delay(1))
    U0 (.out(a_out), .v0(a0), .v1(a1), .sel(sel))
```



# Compiler Directive: `define (1/2)

- ❖ The `define compiler directive performs a simple text-substitution

- Substituted at compile time

```
`define <macro_name> <macro_text>
```

- ❖ To remove the definition of a macro: `undef <macro\_name>
- ❖ Using `define to define “global parameter”

```
`define width 2
module test(out, v0, v1,);
output [`width-1:0] out;
input  [`width-1:0] v0, v1;
endmodule
```



# Compiler Directive: `define (2/2)

- ❖ Using `define to improve code readability
  - Example: **Gray** = (77\*R+150\*G+29\*B) >> 8;

```
`define rgb2gray(R, G, B) (77*R+150*G+29*B) >> 8
module test;
...
Gray = `rgb2gray(R, G, B);
...
endmodule
```

- ❖ Using `define to aggregate all the global constant in a single file

```
`include "define.vh"
module test(out, v0, v1,);
output [`width-1:0] out;
input  [`width-1:0] v0, v1;
endmodule
```

*test.v*

```
`define width 2
`define flag 1
...
define.vh
```



# Compiler Directive: `include

- ❖ Using the **`include`** compiler directive to insert the contents of an entire file

```
`include "define.vh"  
module test(out, v0, v1,);  
output [`width-1:0] out;  
input [`width-1:0] v0, v1;  
endmodule
```

```
`define width 2  
`define flag 1  
...  
module test(out, v0, v1,);  
output [`width-1:0] out;  
input [`width-1:0] v0, v1;  
endmodule
```

```
`define width 2  
`define flag 1  
...  
define.vh
```

Simulation: (if *define.vh* and *test.v* are not in the same directory)  
**ncverilog ... +incdir+<dir\_of\_define.vh>**



## localparam

- ❖ Similar to parameter, but the value **can't be modified** by parameter redefinition or by defparam statement
- ❖ Protect it from accidental or incorrect redefinition

```
localparam S_IDLE      = 4'b0001,  
           S_START     = 4'b0010,  
           S_EXEC      = 4'b0100,  
           S_FINISH    = 4'b1000;
```



# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



# Data Types

- ❖ **nets** are further divided into several net types
  - wire, wand, wor, tri, triand, trior, supply0, supply1
- ❖ **registers** – variable to store a logic value for event-driven simulation - reg
- ❖ **integer** - supports computation 32-bits signed
- ❖ **time** - stores time 64-bit unsigned
- ❖ **real** - stores values as real numbers
- ❖ **realtime** - stores time values as real numbers
- ❖ **event** – an event data type



# Net Types

- ❖ The most common and important net types
  - **wire (synthesizable)** and **tri**
- ❖ Verilog propagates the new value to the net automatically while the driver on the net change value
- ❖ Net type **wire** and **tri** are identical
  - Using tri to indicate that the net can be driven by high impedance
- ❖ Other net types
  - **wand**, **wor**, **triand**, and **trior**
    - for multiple drivers that are wired-anded and wired-ored
  - **tri0** and **tri1**
    - pull down and pull up



# Register Types

- ❖ **reg (synthesizable)**
  - any size, unsigned
- ❖ **integer (not synthesizable)**
  - `integer a,b; // declaration`
  - 32-bit signed (2's complement)
- ❖ **time (not synthesizable)**
  - 64-bit unsigned integer variable
- ❖ **real (not synthesizable)**
  - `Real c,d; //declaration`
  - 64-bit floating-point number
  - Defaults to an initial value of 0



# Integer, Time, & Real

- ❖ Data types not for hardware description
  - For **simulation** control, data, timing extraction.
- ❖ integer counter;
  - `initial` counter = -1;
- ❖ time sim\_time;
  - `initial` sim\_time = \$time;
- ❖ real delta;
  - `initial` delta= 4e10;



# Wire & Reg

## ❖ wire

- Physical wires in a circuit
- Cannot assign a value to a wire within a function or a begin...end block
- A wire does not store its value, it must be driven by
  - by connecting the wire to the output of a gate or module
  - by assigning a value to the wire in a continuous assignment
- An **un-driven** wire defaults to a value of **Z** (high impedance).
- Input, output, inout port declaration -- wire data type (default)

```
output out;
input in1,in2,sel;
reg out;
```



# Wire & Reg

- ❖ reg
  - A *event driven* variable in Verilog
- ❖ Use of “reg” data type is **not** exactly stands for a really register.
- ❖ Use of wire & reg
  - When use “wire”  $\Rightarrow$  usually use “**assign**” and “**assign**” **does not** appear in “**always**” block
  - When use “reg”  $\Rightarrow$  only use “ $a=b$ ” , always appear in “**always**” block

```
module test(a,b,c,d);
  input a,b;
  output c,d;
  reg d;
  assign c=a;
  always @(b)
    d=b;
endmodule
```



# Data Type - Examples

Not specify  $\Rightarrow$  default 1 bit

```
reg             a;           // scalar register
wand    b;           // scalar net of type "wand"
reg      [3:0] c;       // 4-bit register
tri     [7:0] bus;     // tri-state 8-bit bus
reg      [1:4] d;       // 4-bit
```



# Vector

- ❖ wire and reg can be defined as vector, default is 1bit
- ❖ vector is a multi-bits element
- ❖ Format: **[High#:Low#]** or **[Low#:High#]**
- ❖ The most significant bit is always on the left
- ❖ Constant part-select: `vect[msb_expr:lsb_expr]`
- ❖ Both `msb_expr` and `lsb_expr` should be constant

```
wire a;           // scalar net variable, default
wire [7:0] bus;  // 8-bit bus
reg clock;       // scalar register, default
reg [0:23] addr; // Vector register, virtual address 24 bits wide
```

```
bus[7]      // bit #7 of vector bus
bus[2:0]    // Three least significant bits of vector bus
            // using bus[0:2] is illegal because the significant bit should
            // always be on the left of a range specification
addr[0:1]   // Two most significant bits of vector addr
```



# Vector Indexed Part-Select

- ❖ Using base and width to perform part select
- ❖ Base shall be an integer, and width should be a positive constant

```
reg [15:0] big_vect;
reg [0:15] little_vect;

big_vect[lsb_base_expr +: width_expr]
little_vect[msb_base_expr +: width_expr]
big_vect[msb_base_expr -: width_expr]
little_vect[lsb_base_expr -: width_expr]
```

```
reg [63: 0] dword;
integer sel;

dword[ 0 +: 8] // == dword[ 7 : 0]
dword[15 -: 8] // == dword[15 : 8]

dword[8*sel +: 8] // variable part-select with fixed width
```



# Array

- ❖ <type> [MSB:LSB] <name> [first\_addr:last\_addr]
- ❖ Arrays are allowed in Verilog for reg, integer, time, and vector register data types.

```
integer    count[0:7];          // An array of 8 count variables
reg        bool[31:0];         // Array of 32 one-bit Boolean register variables
time       chk_ptr[1:100];       // Array of 100 time checkpoint variables
reg [4:0]   port_id[0:7];       // Array of 8 port_id, each port_id is 5 bits wide
integer    matrix[4:0][4:0]     // Two dimension array
```

```
count[5]           // 5th element of array of count variables
chk_ptr[100]        // 100th time check point value
port_id[3]          // 3rd element of port_id array. This is a 5-bit value
```

```
port_id[3][2:0] // part select for certain element in array
```



# Memories

- ❖ In a digital simulation, one often needs to model register files, RAMs, and ROMs.
- ❖ Memories are modeled in Verilog simply as an array of registers.
- ❖ Each element of the array is known as a word, each word can be one or more bits.
- ❖ It is important to differentiate between
  - n 1-bit registers
  - One n-bit register

```
reg mem1bit [0:1023];           // Memory mem1bit with 1K 1-bit words  
reg [7:0] mem1byte [0:1023]; // Memory mem1byte with 1K 8-bit words
```

```
mem1bit[255]      // Fetches 1 bit word whose address is 255  
Mem1byte[511]      // Fetches 1 byte word whose address is 511
```

```
reg [1:n] rega; // An n-bit register is not the same  
reg mema [1:n]; // as a memory of n 1-bit registers
```



# Arrays Extended after Verilog-2005

- ❖ In Verilog-1995, only reg, integer, and time can be declared as array. Array is limited to 1D.
- ❖ In Verilog-2005, arrays of reg, real, realtime, and any type of net are allowed.

```
// bit                                // 1D array: 8bit×8
reg r_1bit;                           wire [7:0] w_net [7:0];

// vector                             // 2D array: 4bit×8×8
reg [3:0] r_4bit_vec;                 reg [3:0] row_col_addr [0:7][0:7];

// 1D array: memory: 32bit×8        // 3D array: 100×16×4 float variables
reg [31:0] r_memory [7:0];           real float_array [0:99][1:16][10:13];
```

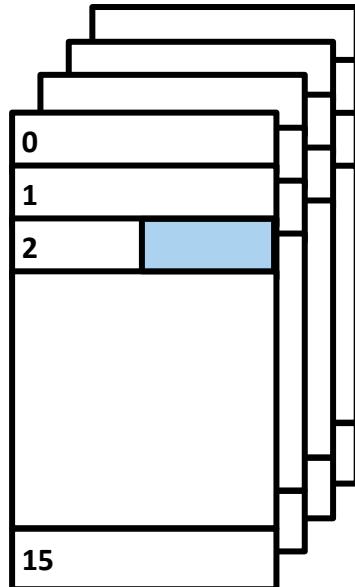


# Multibank Memory

- ❖ An 4-bank 8bit×16 memory

```
reg [7:0] mem [0:15][0:3]
```

- ❖ Selecting the four least significant bits at address 2 in bank 0



```
mem[0][2][0:3] // wrong  
mem[2][0][3:0] // correct
```



# Strings

- ❖ String: a sequence of 8-bits ASCII values

```
module string;
    reg [8*14:1] strvar;
    initial
        begin
            strvar = "Hello World"; // stored as 000000486561...726c64
        end
    endmodule
```

- ❖ Special characters

|    |               |      |                 |
|----|---------------|------|-----------------|
| \n | ⇒ newline     | \t   | ⇒ tab character |
| \\ | ⇒ \ character | \"   | ⇒ " character   |
| %% | ⇒ % character | \abc | ⇒ ASCII code    |



# Integer Constant Representation

- ❖ Format: <size>'<base\_format><number>
  - <size> - decimal specification of number of bits
    - default is unsized and machine-dependent but at least 32 bits
  - <base format> - ' followed by arithmetic base of number
    - <d> <D> - decimal - default base if no <base\_format> given
    - <h> <H> - hexadecimal
    - <o> <O> - octal
    - <b> <B> - binary
  - <number> - value given in base of <base\_format>
    - \_ can be used for reading clarity
    - If first character of sized, binary number 0, 1, x or z, will extend 0, 1, x or z (defined later!)
  - Negative number (2's complement)
    - Negative sign shall be placed before <size>



# Truncation and Extension

- ❖ Verilog truncates the most significant bits when you specify the size to be smaller than the number entered

$$2'b0110 \Rightarrow 2'b10$$

- ❖ Extension: padding zeros, x, or z if the size of number is smaller than the size specified

8'b010 // 0 padding to fill the MSB, 8'b00000010

8'b110 // 0 padding to fill the MSB, 8'b00000110

7'bx11 // x padding to fill the MSB, 7'bxxxxxx11

7'bz11 // x padding to fill the MSB, 7'bzzzzz11

```
reg [1:0] a;
```

```
reg [82:0] b;
```

```
a = 'd5 // gives 01
```

```
b = 'd5 // gives {{80{1'b0}}, 3'b101}
```



# Integer Constant Example

## ❖ Examples:

- 6'b1010\_111      gives 010111
- 8'b0110            gives 00000110
- 4'bx01            gives xx01
- 16'H3AB            gives 0000\_0011\_1010\_1011
- 24                  gives 0...0011000    **(default as signed unsized decimal)**
- 5'O36            gives 1\_1110
- 16'Hx            gives xxxxxxxxxxxxxxxxxx
- 8'hz                  gives zzzzzzzz
- 'hff                  gives 0...0\_1111\_1111
- 8'd-6                  **illegal**
- -8'd6                gives 1111\_1010 (2's complement of 6)
- 4'shf                  gives 1111
- -4'shf                gives -(-4'd 1), or 0001



# Real Constant Representation

- ❖ In Verilog, **real** constant can be represented in
  - Decimal                            <int>.<fraction>
  - Scientific format                <mantissa><e or E><exp>
- ❖ There shall be at least one digit on each side of the decimal point (Ex: .012 is illegal)
- ❖ Example
  - 12                                    gives 32-bit unsigned decimal
  - 1.2E10                            gives  $1.2 \times 10^{10}$
  - 236.1\_51e-2                    gives  $235.151 \times 10^{-2}$
  - .12                                    illegal
  - .3E4                                    illegal



# Vector Concatenations

- ❖ A easy way to group vectors into a larger vector

| Representation          | Meanings   |
|-------------------------|--|
| {cout, sum}             | {cout, sum}                                      |
| {b[7:4],c[3:0]}         | {b[7], b[6], b[5], b[4], c[3], c[2], c[1], c[0]} |
| {a,b[3:1],c,2'b10}      | {a, b[3], b[2], b[1], c, 1'b1, 1'b0}             |
| {4{2'b01}}              | 8'b01010101                                      |
| {b, {3{a, b}}}          | {b, a, b, a, b, a, b}                            |
| { {8{byte[7]}}, byte }  | Sign extension                                   |
| { {P-8{byte[7]}}, byte} | Sign extension to P-bit (P>8)                    |



# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



# Primitives

- ❖ Primitives are modules ready to be instanced
- ❖ **Smallest modeling block for simulator**
  - Behavior as software execution in simulator, not hardware description
- ❖ Verilog build-in primitive gate
  - *and, or, not, buf, xor, nand, nor, xnor*
  - `prim_name inst_name( output, in0, in1,.... );`
- ❖ User defined primitive (UDP)
  - building block defined by designer



# Outline

- ❖ Overview and History
- ❖ Hierarchical Design Methodology
- ❖ Levels of Modeling
  - ❖ Behavioral Level Modeling
  - ❖ Register Transfer Level (RTL) Modeling
  - ❖ Structural/Gate Level Modeling
- ❖ Language Elements
  - ❖ Lexical Conventions
  - ❖ Data Type
  - ❖ Primitive
  - ❖ Timing and Delay



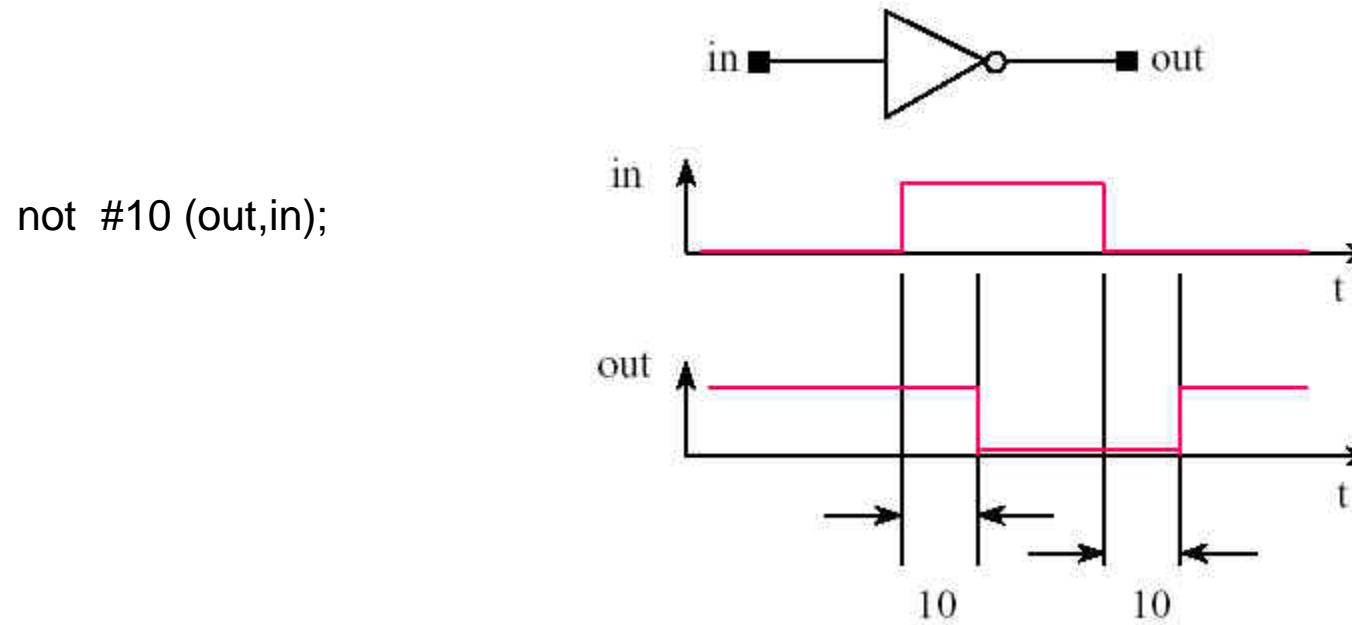
## Timing and Delay (for verification)

- ❖ Functional verification of hardware is used to verify functionality of the designed circuit.
- ❖ However, blocks in real hardware have delays associated with the logic elements and paths in them.
- ❖ To model these delay, we use timing / delay description in Verilog: **#**
- ❖ Then we can check whether the total circuit meets the timing requirements, given delay specifications of the blocks.



# Delay Specification in Primitives

- ❖ Delay specification defines the propagation delay of that primitive gate.



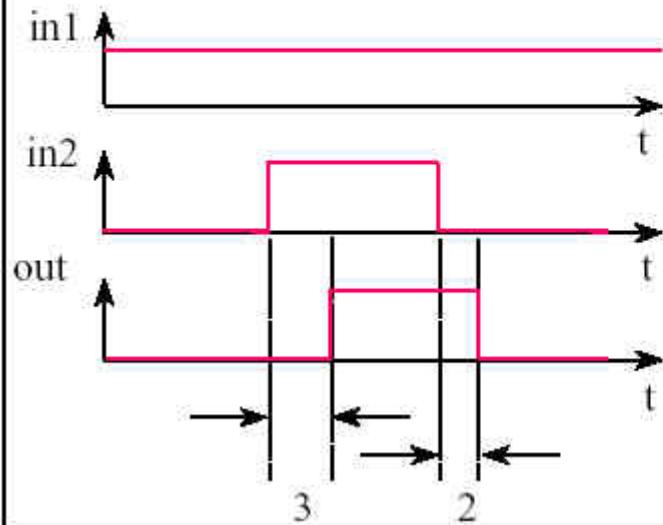
not #10 (out,in);



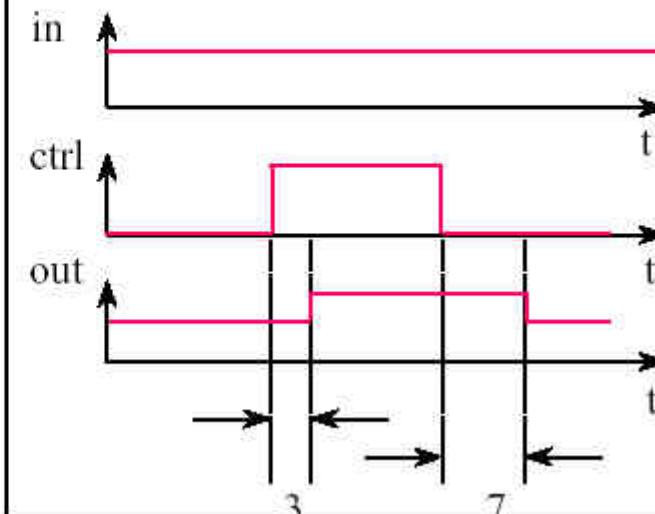
# Delay Specification in Primitives

- ❖ Verilog supports (rise, fall, turn-off) delay specification.

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



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





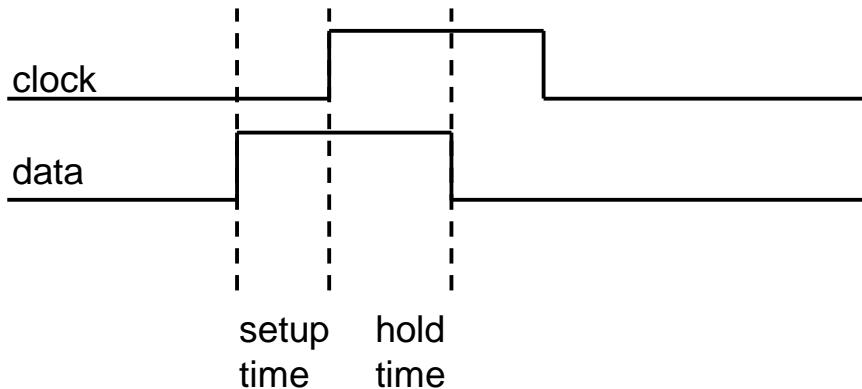
# Delay Specification in Primitives

- ❖ All delay specification in Verilog can be specified as *(minimum : typical : maximum)* delay
- ❖ Examples
  - *(min:typ:max)* delay specification of all transition
    - or #(3.2:4.0:6.3) U0(out, in1, in2);
  - *(min:typ:max)* delay specification of RISE transition and FALL transition
    - nand #(1.0:1.2:1.5,2.3:3.5:4.7) U1(out, in1, in2);
  - *(min:typ:max)* delay specification of RISE transition, FALL transition, and turn-off transition
    - bufif1 #(2.5:3:3.4,2:3:3.5,5:7:8) U2(out,in,ctrl);



# Timing Checks (For Testbench)

- ❖ setup time and hold time checks

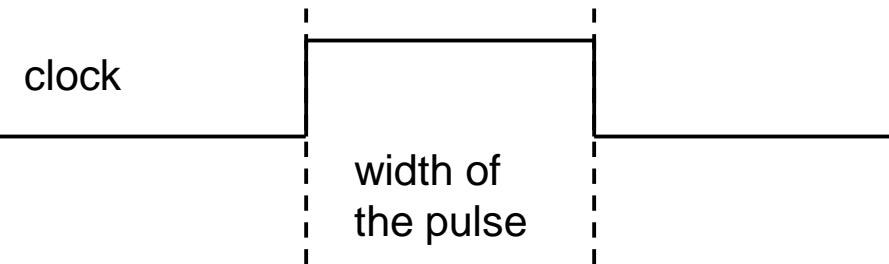


```
specify  
  $setup(data, posedge clock, 3);  
endspecify
```

```
specify  
  $hold(posedge clock, data, 5);  
endspecify
```

- ❖ Width check

- ❖ Sometimes it is necessary to check the width of a pulse.



```
specify  
  $width(posedge clock, 6);  
endspecify
```

# Computer-Aided VLSI System Design

## Chap.1-2 Logic Design at Register-Transfer Level

Lecturer: 張惇宥

*Graduate Institute of Electronics Engineering, National Taiwan University*



NTU GIEE



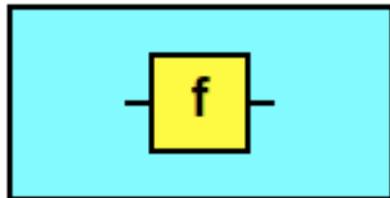
# Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Verilog Syntax of RTL
  - Operands
  - Operators



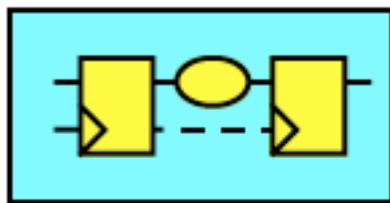
# Brief Review of Modeling Levels

Behavioral Level



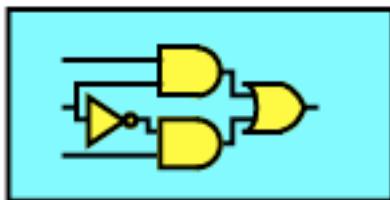
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



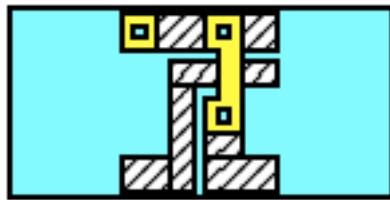
Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

Structural/Gate Level



Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library

Transistor/Physical Level



Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE



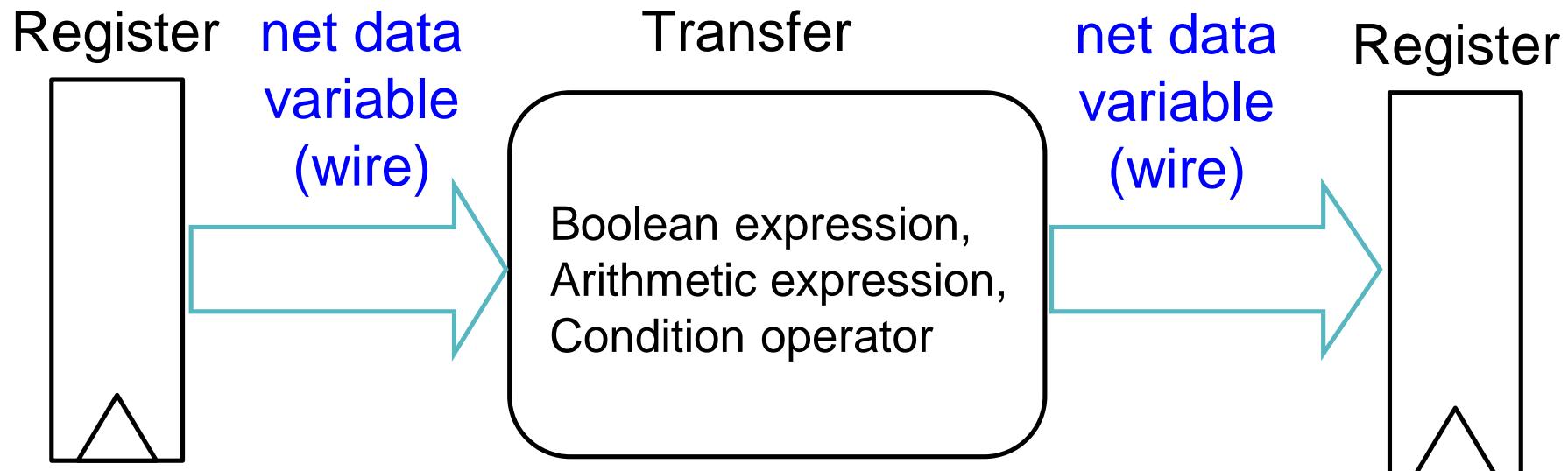
# What is Register Transfer Level?

- ❖ In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a **synchronous digital circuit**.
- ❖ In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers and the logical operations performed on those signals.
- ❖ Modern RTL code: “**Any code that is synthesizable is called RTL code**”



## Description at RT-Level

- ❖ Register (memory, usually D Flip-Flops)
- ❖ Transfer (operation, **combinational**)





# Procedures to Design at RT-Level

## ❖ Design partition

- Sequential circuit (register part)
- Combinational circuit (transfer part)

## ❖ Net declaration

- I/O ports
- Net variable

## ❖ Transfer Circuit design

- Logical operator
- Arithmetical operator
- Conditional operator



# Example

```
wire [7:0] in1;
wire [7:0] in2;
wire [7:0] in3;                                //input signals of transfer part

wire [7:0] out1;
wire [7:0] out2;
wire [7:0] out3;                                //output signals of transfer part

assign out1 = in1 & in2;
assign out2 = in1 + in3;
assign out3 = in1[0]==1'b1 ? in2 : in3;      //transfer description
```



# Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Verilog Syntax of RTL
  - Operands
  - Operators
- ❖ Continuous Assignments for Combinational Circuits



# Operands

- ❖ Data types **wire** & **reg** are used for operands in RTL/Behavioral Level description
- ❖ Since all metal wires in circuits only represent 0 or 1, we should consider all the variables in binary
  - Unsigned: Ordinary binary
  - Signed: **2's complement** binary
  - Since there's only 0/1 for each bit, **whether the variable is unsigned or signed is up to your recognition**
  - Simulators do all operations in binary (i.e., 0/1). If you check all the operations in binary, you'll never get wrong arithmetic/logic results

```
reg [4:0] a;  
reg signed [4:0] a;
```



# Operators

|                                |                                    |
|--------------------------------|------------------------------------|
| Concatenation and replications | {,}                                |
| Negation                       | !, ~                               |
| Unary reduction                | &,  , ^, ^~ ...                    |
| Arithmetic                     | + , - , * , / , %                  |
| Shift                          | >> , <<                            |
| Relational                     | < , <= , > , >=                    |
| Equality                       | == , != , <b>====</b> , <b>!==</b> |
| Bitwise                        | ~, &,  , ^                         |
| Logical                        | && ,                               |
| Conditional                    | ? :                                |



# Concatenation & Replication

## ❖ Concatenation and Replication Operator ({}))

### **Concatenation operator in LHS**

```
module add_32 (co, sum, a, b, ci);  
    output co;  
    output [31:0] sum;  
    input  [31:0] a, b;  
    input  ci;  
    assign #100 {co, sum} = a + b + ci;  
endmodule
```

### **Bit replication to produce *01010101***

```
assign byte = {4{2'b01}};
```

### **Sign Extension**

```
assign word = {{8{byte[7]}}, byte};
```



# Negation Operators

- ❖ The logical negation operator (!)
  - produces a 0, 1, or X scalar value
- ❖ The bitwise negation operator (~)
  - inverts each individual bit of the operand

```
module negation;
    initial begin
        $displayb( !4'b0100 ); // 0
        $displayb( !4'b0000 ); // 1
        $displayb( !4'b00z0 ); // x
        $displayb( !4'b000x ); // x
        $displayb( ~4'b01zx ); // 10xx
    end
endmodule
```



# Unary Reduction Operators

- ❖ The unary reduction operators (**&**, **|**, **^**, **^~**) produce a 0, 1, or X scalar value.

```
module reduction;
    initial begin
        $displayb ( &4'b1110 ); // 0
        $displayb ( &4'b1111 ); // 1
        $displayb ( &4'b111z ); // x
        $displayb ( &4'b111x ); // x
        $displayb ( |4'b0000 ); // 0
        $displayb ( |4'b0001 ); // 1
        $displayb ( |4'b000z ); // x
        $displayb ( |4'b000x ); // x
        $displayb ( ^4'b1111 ); // 0
        $displayb ( ^4'b1110 ); // 1
        $displayb ( ^4'b111z ); // x
        $displayb ( ^4'b111x ); // x
        $displayb ( ^~4'b1111 ); // 1
        $displayb ( ^~4'b1110 ); // 0
        $displayb ( ^~4'b111z ); // x
        $displayb ( ^~4'b111x ); // x
    end
endmodule
```



# Arithmetic Operators

- ❖ The arithmetic operators (\*\*, \*, /, %, +, -)
  - Produce numerical or unknown results
  - Integer division discards any remainder
  - An unknown operand produces an unknown result
  - Assignment of a signed value to an unsigned register is 2's-complement

```
module arithmetic;
    initial begin
        $display( -3 * 5 ); // -15
        $display( -3 / 2 ); // -1
        $display( -3 % 2 ); // -1
        $display( -3 + 2 ); // -1
        $display( 2 - 3 ); // -1
        $displayh( 32'hfffffd / 2 ); // 7fffffe
        $displayb( 2 * 1'bx); // xx...
    end
endmodule
```

Treat the all signal as binary signal !!!

How to deal with fraction number???

→ Fixed point representation



# Example (Arithmetic)

```
module DUT (sum,diff1,diff2,negA,A,B);
    output [4:0] sum,diff1,diff2,negA;
    input  [3:0] A , B

    assign sum = A + B;
    assign diff1 = A - B;
    assign diff2 = B - A;
    assign neg = -A ;
endmodule
```

| t_sim | A    | B    | sum   | diff1 | diff2 | negA  |
|-------|------|------|-------|-------|-------|-------|
| 5     | 5    | 2    | 7     | 3     | 29    | 27    |
| 15    | 0101 | 0010 | 00111 | 00011 | 11101 | 11011 |



# Shift Operators

## ❖ Shift operator

- “**>>**” logical shift right
- “**<<**” logical shift left
- “**>>>**” arithmetic shift right (fills in signed bits if the signal is signed)
- “**<<<**” arithmetic shift left

## ❖ Treat the right operand as unsigned

```
module shift;
    initial begin
        $displayb(8'b00011000 << 2) ; // 01100000
        $displayb(8'b00011000 >> 2) ; // 00000110
        $displayb(8'b00011000 >> -2) ; // 00000000
        $displayb(8'b00011000 >> 1'bx) ; // xxxxxxxx
    end
endmodule
$displayb(8'sb11001100 >>> 2); // 11110011
$displayb(8'sb11001100 <<< 2); // 00110000
```

-2 = 1110 (2's comp)  
→ Take it as unsigned  
→ 1110 = 14



# Relational Operators

- ❖ Relational operators (< , <= , >= , >)

```
module relational;
    initial begin
        $displayb ( 4'b1010 < 4'b0110 ) ; // 0
        $displayb ( 4'b0010 <= 4'b0010 ) ; // 1
        $displayb ( 4'b1010 < 4'b0x10 ) ; // x
        $displayb ( 4'b0010 <= 4'b0x10 ) ; // x
        $displayb (| 4'b1010 >= 4'b1x10 ) ; // x
        $displayb ( 4'b1x10 > 4'b1x10 ) ; // x
        $displayb ( 4'b1z10 > 4'b1z10 ) ; // x
    end
endmodule
```



# Equality Operators

## ❖ Equality operators

- “`==`”, “`!=`” don't perform a definitive match for Z or X.
- “`==>`”, “`!=>`” does perform a definitive match for Z or X.

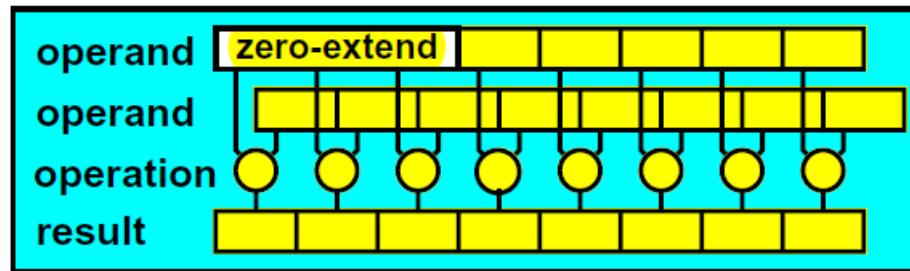
```
module equality;
    initial begin
        $displayb ( 4'b0011 == 4'b1010 ); // 0
        $displayb ( 4'b0011 != 4'b1x10 ); // 1
        $displayb ( 4'b1010 == 4'b1x10 ); // x
        $displayb ( 4'b1x10 == 4'b1x10 ); // x
        $displayb ( 4'b1z10 == 4'b1z10 ); // x
    end
endmodule
```

```
module identity(); NON-SYNTHESIZABLE
    initial begin
        $displayb ( 4'b01zx === 4'b01zx ); // 1
        $displayb ( 4'b01zx !== 4'b01zx ); // 0
        $displayb ( 4'b01zx === 4'b00zx ); // 0
        $displayb ( 4'b01zx !== 4'b11zx ); // 1
    end
endmodule
```



# Bit-Wise Operators

- ❖ Bit-wise operators ( $\&$ ,  $|$ ,  $^$ ,  $^~$ )
  - operate on each individual bit of a vector



```
module bit_wise;
    initial begin
        $displayb ( 4'b01zx & 4'b0000 ) ; // 0000
        $displayb ( 4'b01zx & 4'b1100 ) ; // 0100
        $displayb ( 4'b01zx & 4'b1111 ) ; // 01xx
        $displayb ( 4'b01zx | 4'b1111 ) ; // 1111
        $displayb ( 4'b01zx | 4'b0011 ) ; // 0111
        $displayb ( 4'b01zx | 4'b0000 ) ; // 01xx
        $displayb ( 4'b01zx ^ 4'b1111 ) ; // 10xx
        $displayb ( 4'b01zx ^~ 4'b0000 ) ; // 10xx
    end
endmodule
```



# Logical Operators

## ❖ Logical operators (&&, ||)

- An operand is logically false if **all** of its bits are 0
- An operand is logically true if **any** of its bits are 1

```
module logical;
    initial begin
        $displayb ( 2'b00 && 2'b10 ) ; // 0
        $displayb ( 2'b01 && 2'b10 ) ; // 1
        $displayb ( 2'b0z && 2'b10 ) ; // x
        $displayb ( 2'b0x && 2'b10 ) ; // x
        $displayb ( 2'b1x && 2'b1z ) ; // 1
        $displayb ( 2'b00 || 2'b00 ) ; // 0
        $displayb ( 2'b01 || 2'b00 ) ; // 1
        $displayb ( 2'b0z || 2'b00 ) ; // x
        $displayb ( 2'b0x || 2'b00 ) ; // x
        $displayb ( 2'b0x || 2'b0z ) ; // x
    end
endmodule
```

Usually connect two Boolean value  
(e.g. (a > b) && (a > 0))



# Conditional Operators

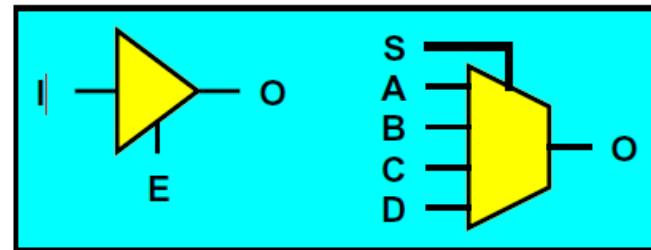
## ❖ Conditional Operator

- Usage:

*conditional\_expression ? true\_expression: false\_expression;*

```
module driver(O,I,E);
output O; input I,E;
  assign O = E ? I : 'bz;
endmodule

module mux41(O,S,A,B,C,D);
output O;
input A,B,C,D; input [1:0] S;
  assign O = (S == 2'h0) ? A :
              (S == 2'h1) ? B :
              (S == 2'h2) ? C : D;
endmodule
```





# Operator Precedence

| Type of Operators       | Symbols  |             |                |              |              |  |
|-------------------------|----------|-------------|----------------|--------------|--------------|--|
| Concatenate & replicate | { }      | $\{{\ }}$   |                |              |              |  |
| Unary                   | !        | $\sim$      | &              | $\wedge$     | $\wedge\sim$ |  |
| Arithmetic              | *        | /           | %              |              |              |  |
|                         | +        | -           |                |              |              |  |
| Logical shift           | $<<$     | $>>$        |                |              |              |  |
| Relational              | <        | $\leq$      | >              | $\geq$       |              |  |
| Equality                | $\equiv$ | $\neq$      | $\equiv\equiv$ | $\neq\equiv$ |              |  |
| Binary bit-wise         | &        | $\wedge$    | $\wedge\sim$   |              |              |  |
| Binary logical          | $\&\&$   | $\parallel$ |                |              |              |  |
| Conditional             | ? :      |             |                |              |              |  |

Use parentheses (i.e. “()”) to explicitly define the operation order



# Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Verilog Syntax of RTL
  - Operands
  - Operators
- ❖ Continuous Assignments for Combinational Circuits



# Procedures

- ❖ **Procedure block:** a group of statements that appear between a begin and an end
  - **initial, always, task, function**
  - considered as a statement – can be nested
- ❖ Procedures execute **concurrently** with other procedures



# Assignments (1/2)

- ❖ Assignment: Drive value onto nets and registers
- ❖ There are two basic forms of assignment
  - **continuous assignment**, which assigns values to nets
  - **procedural assignment**, which assigns values to registers
- ❖ Basic form

<left-hand side> = <right-hand side>

| Statement type        | Left-hand side (LHS)   |
|-----------------------|--|
| Continuous assignment | Net*<br><code>wire</code> , <code>tri</code>                             |
| Procedural assignment | Register*<br><code>reg</code> , <code>integer</code> , <code>real</code> |

- \* Include scalar, vector, concatenation, and bit/part-select
- May synthesizable



# Assignments (2/2)

## ❖ Continuous assignment

```
module holiday_1(sat, sun, weekend);
    input sat, sun; output weekend;
    assign weekend = sat | sun;      // outside a procedure
endmodule
```

## ❖ Procedural assignment

```
module holiday_2(sat, sun, weekend);
    input sat, sun; output weekend; reg weekend;
    always #1 weekend = sat | sun; // inside a procedure
endmodule
```

```
module assignments
    // continuous assignments go here
    always begin
        // procedural assignments go here
    end
endmodule
```



# Continuous Assignments (1/3)

## ❖ Syntax of continuous assignment

```
assign [#delay] <net name> = <expression>
```

## ❖ The continuous assignment is **always active**

- Any changes in the RHS of the continuous assignment are evaluated and the LHS is updated

## ❖ The LHS shall be

- **Scalar or vector net**
- Bit/part-select of vector net
- Concatenation of any of the above type

```
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
```

## ❖ The RHS can be registers or nets or function call



# Continuous Assignments (2/3)

## ❖ Syntax of continuous assignment

```
assign [#delay] <net name> = <expression>
```

## ❖ The LHS is updated at any change in the RHS expression after a specified delay

- The delay is optional
- The continuous assignment **with delay is non-synthesizable**
- Continuous assignments can only contain simple, left-hand side delay (i.e., limited to a # delay ); because of their continuous nature, @ timing control is unnecessary

## ❖ Example

```
assign #10 o = in1 + in2
```



# Continuous Assignments (3/3)

- ❖ The continuous assignment is **always active**

```
module assignment_1();
wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire;

assign pwr_stable = Ok & (!Fire);
assign pwr_on = 1;
assign pwr_good = pwr_on & pwr_stable;

initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end
initial begin $monitor("TIME=%0d",$time," ON=",pwr_on, " STABLE=",
    pwr_stable," OK=",Ok," FIRE=",Fire," GOOD=",pwr_good);
#10 $finish; end
endmodule
```

```
> TIME=0 ON=1 STABLE=0 OK=0 FIRE=0 GOOD=0
> TIME=1 ON=1 STABLE=1 OK=1 FIRE=0 GOOD=1
> TIME=6 ON=1 STABLE=0 OK=1 FIRE=1 GOOD=0
```



# Examples of continuous Assignment

## ❖ Examples of continuous Assignment

```
wire [15:0] addr;
wire out, cout;

wire [2:0] a_low = a[3:0];
// net declaration assignment, a_low[2:0] = a[2:0]
assign out = i1 & i2;
// i1 and i2 are nets
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
// Continuous assign for vector nets addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
// LHS is a concatenation of a scalar net and vector net
```



# Assignment Delays

## ❖ Regular Assignment Delay

```
wire out;  
assign #10 out = in1 & in2;  
// delay in a continuous assign
```

## ❖ Implicit Continuous Assignment Delay

```
wire #10 out = in1 & in2;
```

## ❖ Net Declaration Delay

```
wire # 10 out; // net delays  
assign out = in1 & in2;
```

- Remind again, **non-synthesizable!** For Testbench/Behavior model usage



# Avoiding Combinational Loops

- ❖ Avoid **combinational loops** (or logic loops)
  - HDL Compiler and Design Compiler will automatically open up asynchronous combinational loops
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

```
wire [3:0] a;  
wire [3:0] b;  
  
assign a = b + a;
```

