Computer-Aided VLSI System Design

Chap.1-1 Fundamentals of Hardware Description Language

Lecturer: 徐以帆

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Verilog Course Overview

- Chapter 1
  - Fundamentals of HDL
  - Verilog language elements
- Chapter 2
  - Register transfer level modeling
  - 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

Hardware Description Language

From Wikipedia

- **H**ardware **D**escription **L**anguage (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?

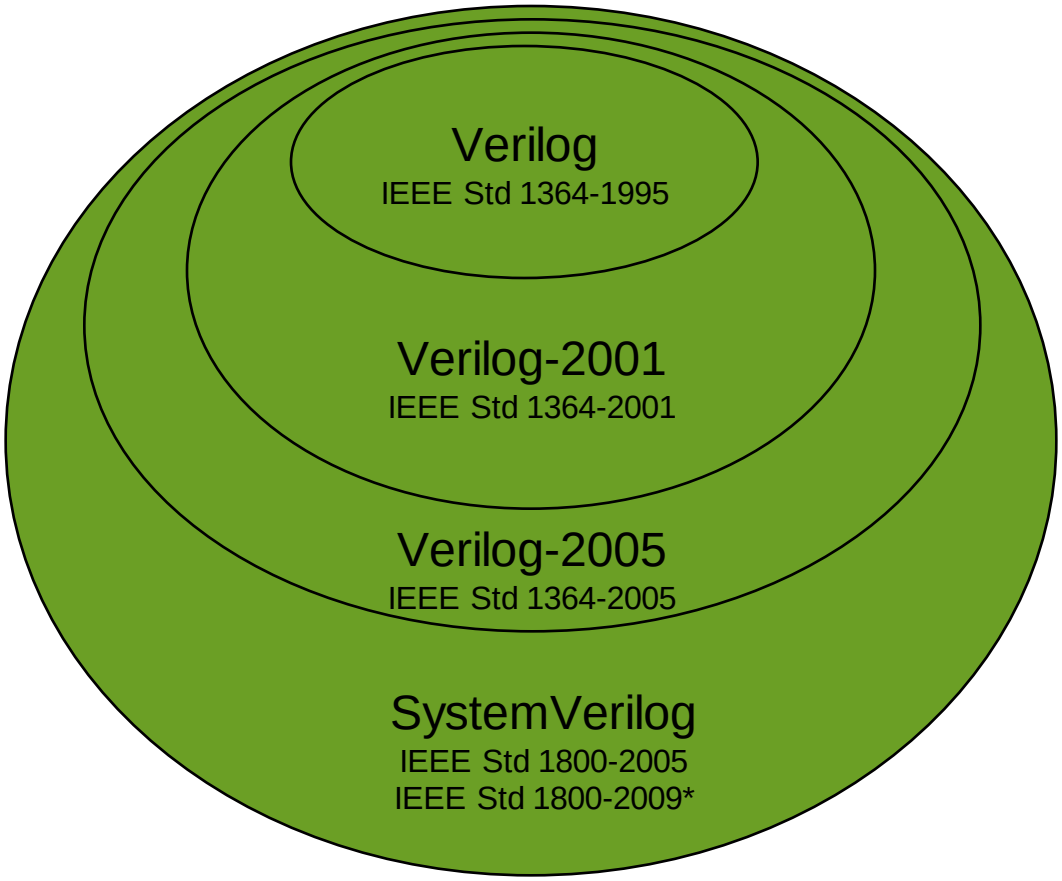
- The **efficiency** (to model/verify) does matter
  - Native support for concurrency
  - Native support for simulate the progress of time
  - Native support for simulate the model of system

- 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



Digital-signal HDL

* Verilog is merged into SystemVerilog in IEEE Std 1800-2009

Outline

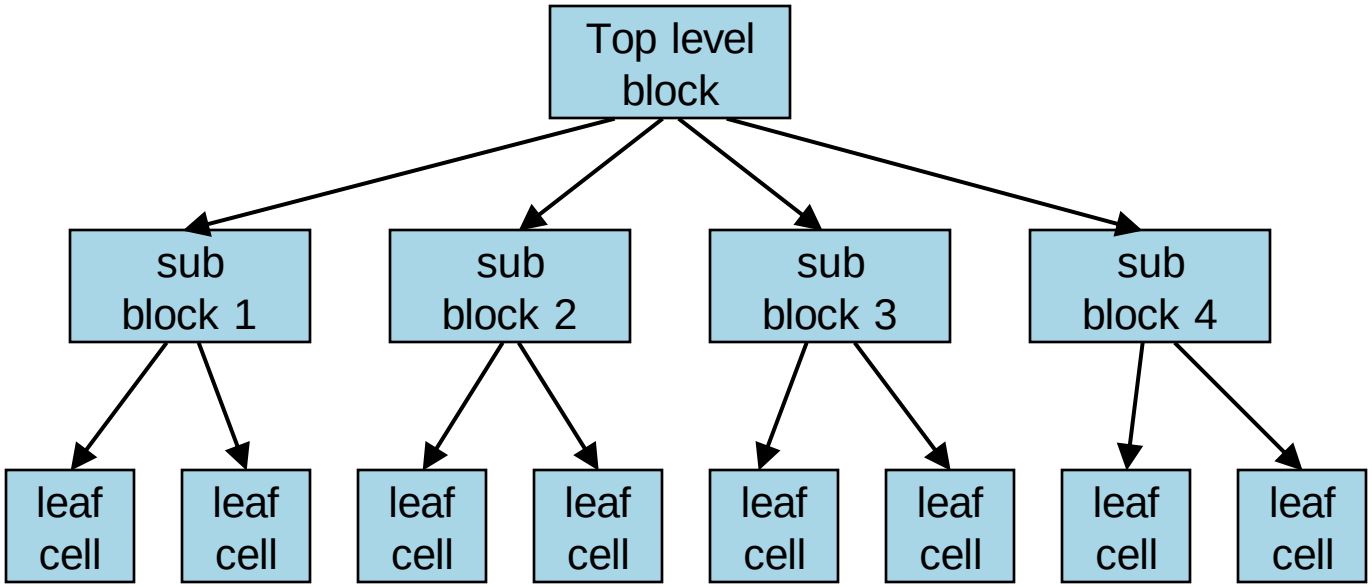
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling

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

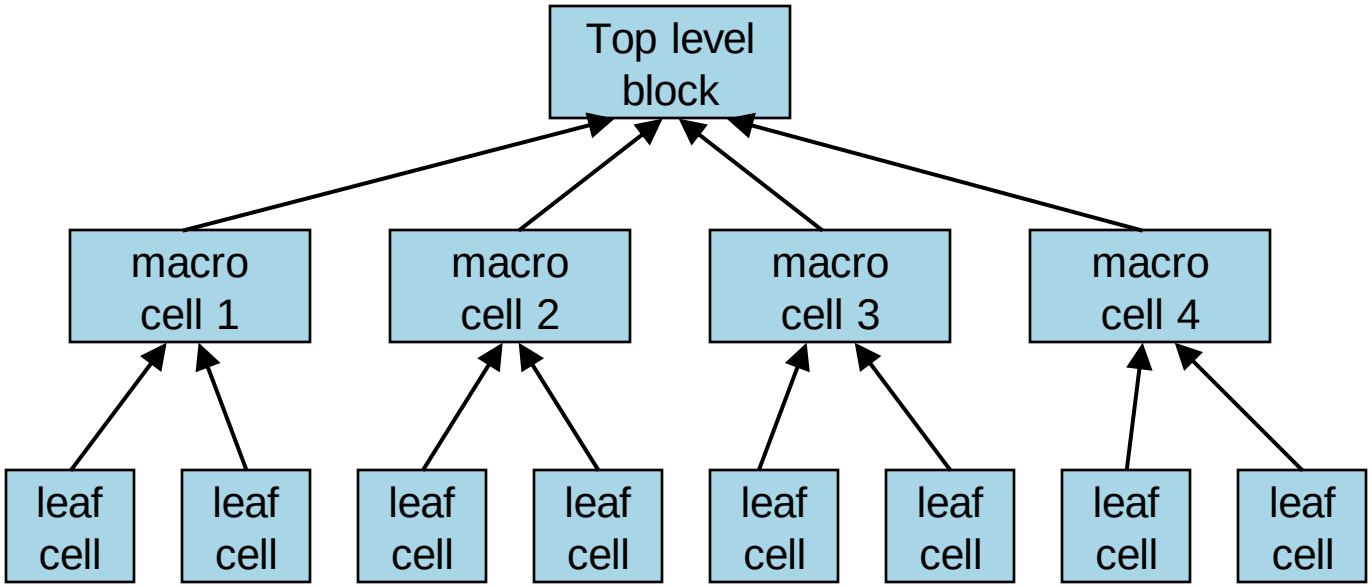
Top-down Design Methodology

- 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

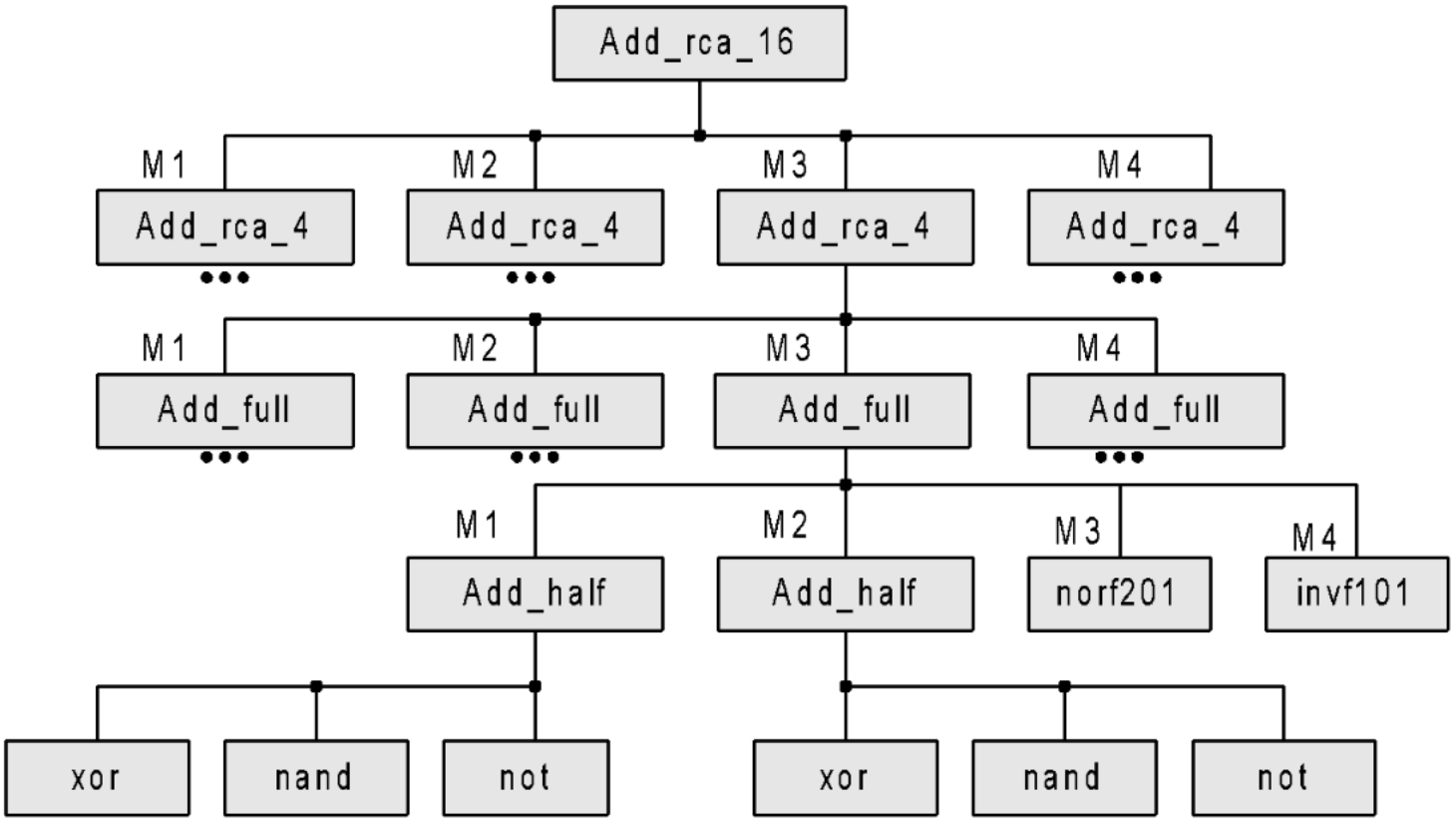
- We first identify the building blocks that are available to us.
- We build bigger cells, using these building blocks.
- These cells are then used for higher-level blocks until we build the top-level block in the design.



Example: 16-bit Adder

divide

conquer



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).

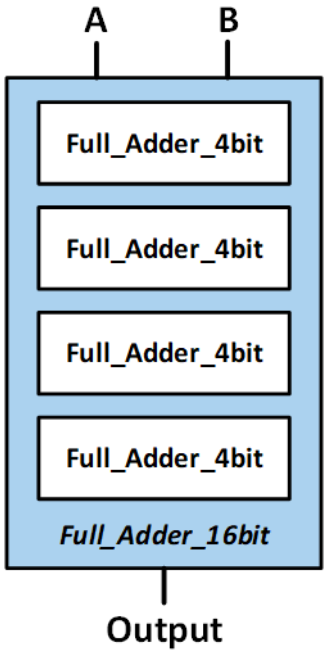
Modules

- 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 Variables>
        <SubModule>

    // Functional part
        <Timing Control Statements>
        <Parameter/Value Assignments>
        <Logic>

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 the ports' name in parentheses "()" after the module name, and declare ports in the module description
  2. List and declare ports in parentheses "()" after the module name at the same time

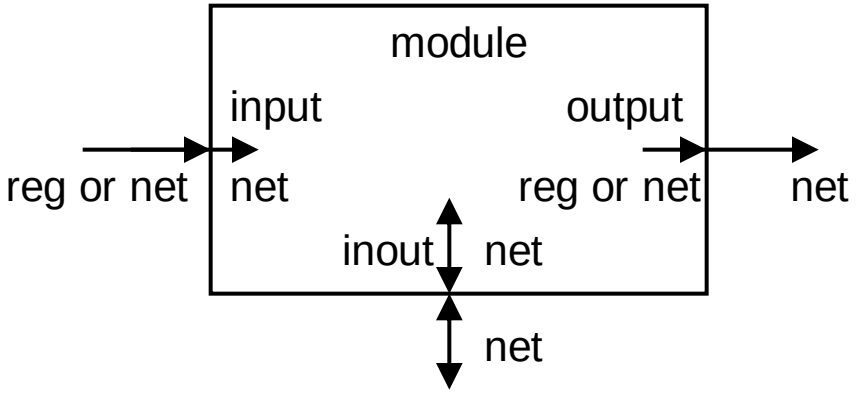
(recommended)

```
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-directional port
    - inout c;



Module Instantiation

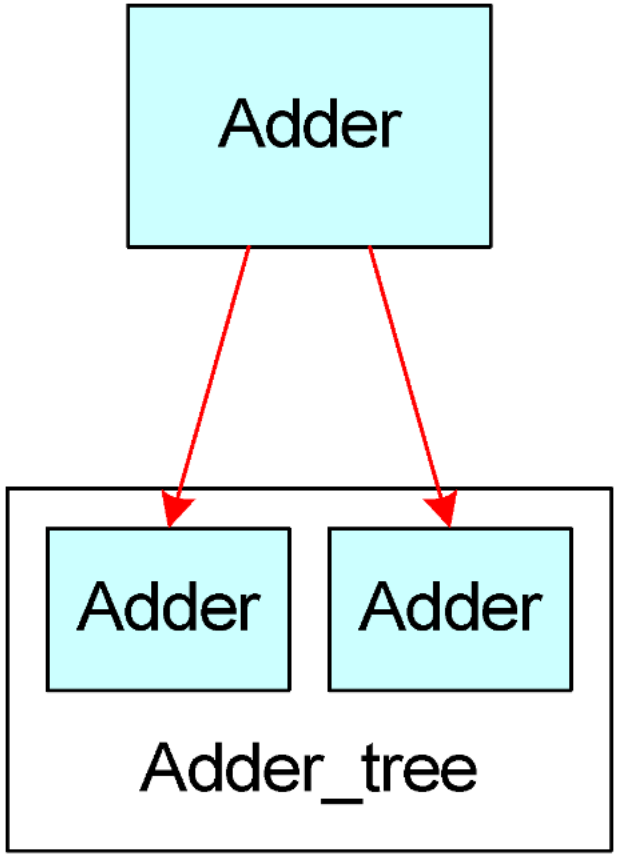
```
adder adder_0 ( out0 , in1 , in2 );
```

Module name Instance name

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

    assign  out=in1 + in2;
endmodule
```

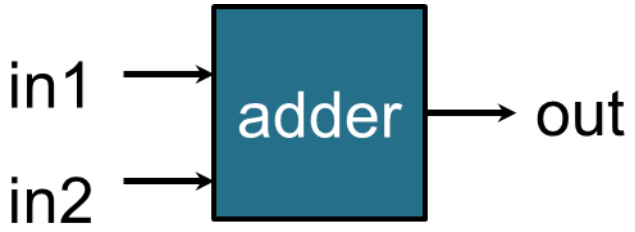
instance example

```
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

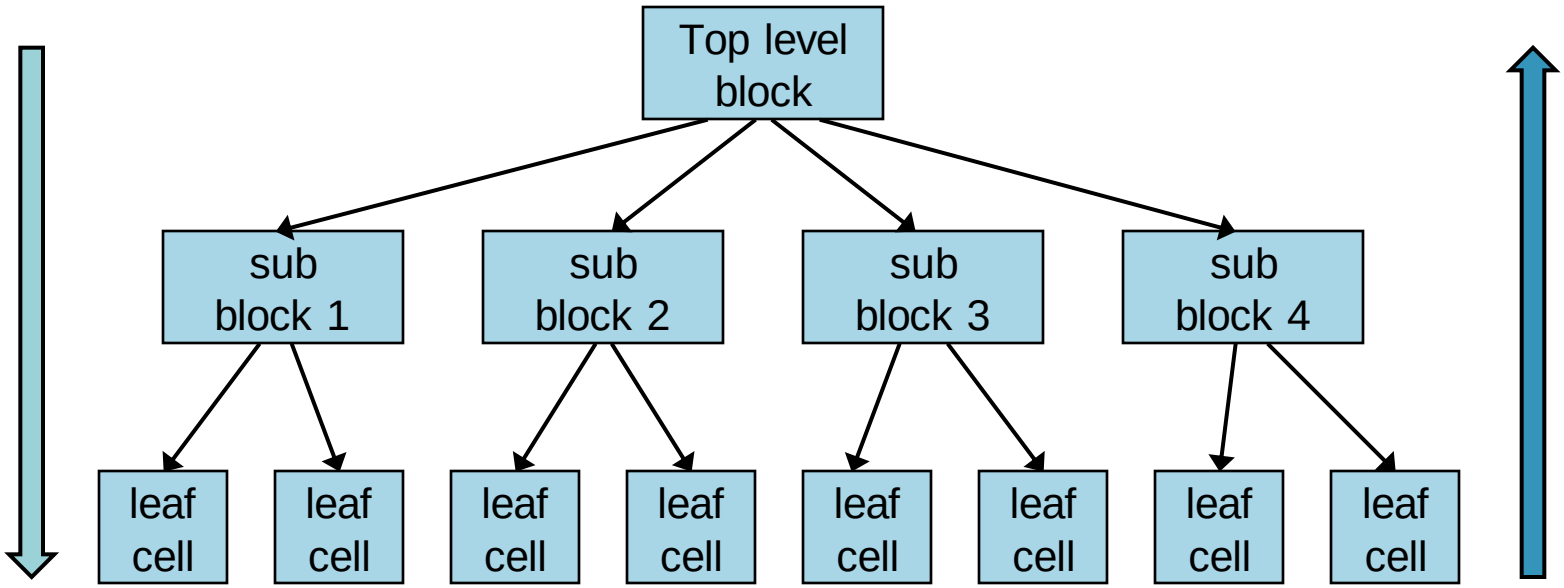
```
module adder (out,in1,in2);
  output out;
  input  in1 , in2;

  assign out = in1 + in2;
endmodule
```



- Connect module ports by *ordered 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() );`

Recommended Design Flow

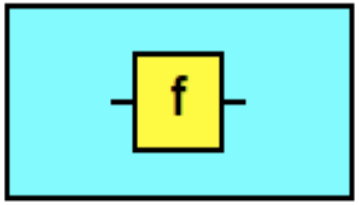
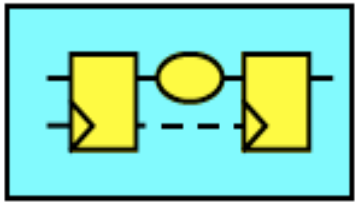
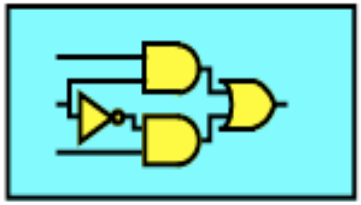
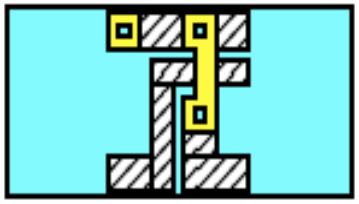
- Use top-down approach for architectural design
  - Functional decomposition, datapath planning, etc.
- Use bottom-up approach for design implementation
  - Detailed logic, handshake signal, timing, etc.
- You may have to iterate many times



Outline

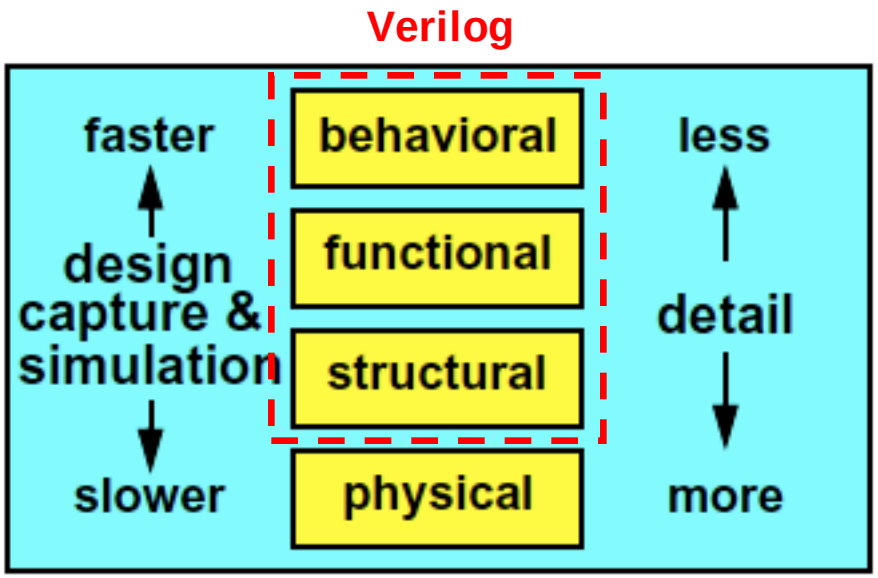
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling

Cell-Based Design and Levels of Modeling

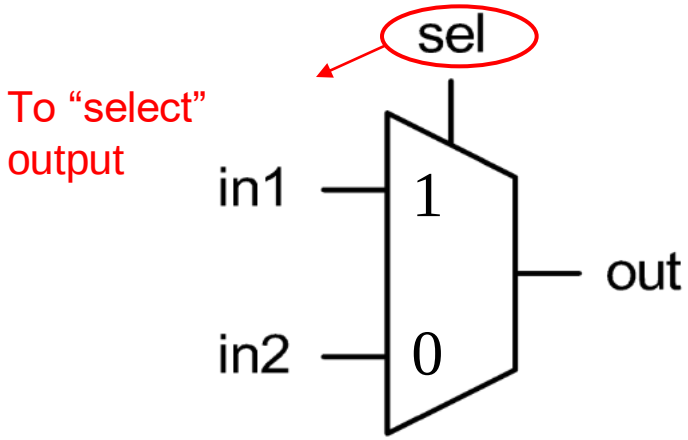
Behavioral Level		Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools
Register Transfer Level (RTL)		Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level
Structural/Gate Level		Usually generated by synthesis 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 with a different level of detail. More detail means more effort for designers and the simulator.



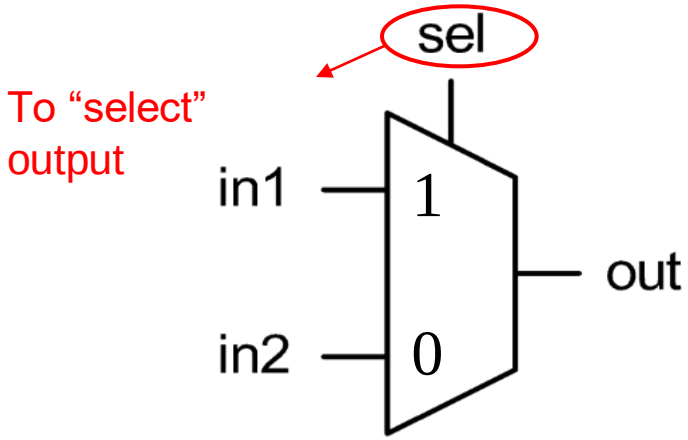
An Example - 1-bit Multiplexer in Behavioral Level




```
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



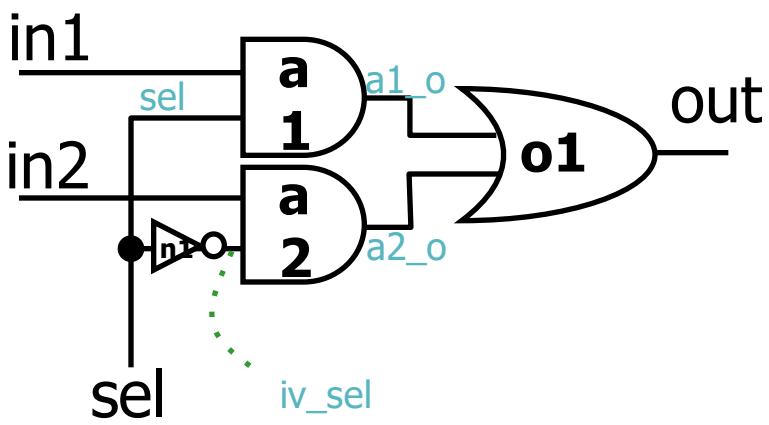

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    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 nets (usually wire)

Summary: Verilog Basic Cell

- Verilog Basic Components
- **Declarations**
  - port declarations
  - wire and 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 (RTL modeling)
    assign out = out_r;

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

endmodule
```

Computer-Aided VLSI System Design

Chap.1-2 Verilog Language Elements

Lecturer: 徐以帆

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

Verilog Language Rules

- Verilog is a **case sensitive** language (with a few exceptions)
  - **Avoid to use**
- Terminate statements with semicolon **;**
- Single line comments
  - // A single-line comment goes here
- Multi-line comments
  - /* Multi-line comments like this
    Multi-line comments like this */
- Statements can be grouped into **`begin-end`** blocks or **`fork-join`** blocks
  - Similar to { } in C/C++

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` — correct
- `8bit_result, a+b, $n321` — wrong

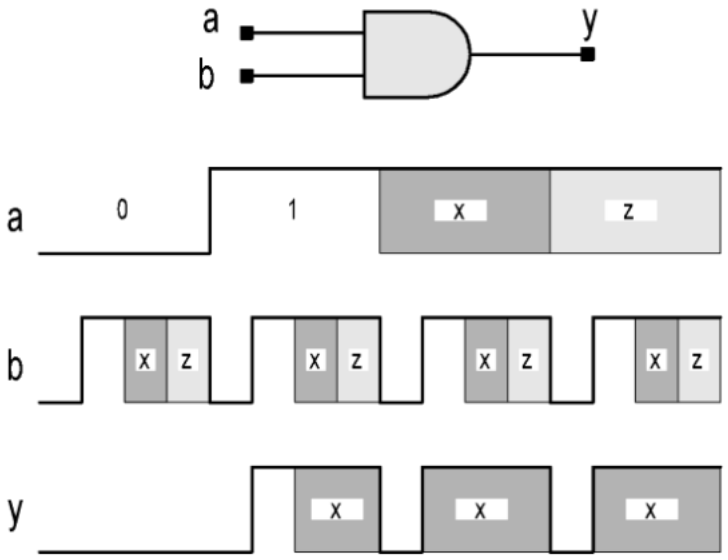
Four-Valued Logic System

- Each data in Verilog may hold one of the four values
  - 0 represent a logic **low** or **false** condition
  - 1 represent a logic **high** or **true** condition
  - z represent a **high-impedance** value
    - When nets are unconnected or tri-state drivers are undriven
  - x represent an **unknown** logic value
    - When the simulator can't decide the 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 (with some exception), and make following stages 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 evaluate 4-valued logic



	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Constants Specification Methods

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

1. parameter
2. `define compiler directive
3. localparam

```
parameter BUS_WIDTH = 8;
`define BUS_WIDTH 8
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

- Declare run-time constants
- Can only be used in the module it is defined
- Parameter definition syntax:

  parameter <par> [= <val>] {, [parameter] <par> [= <val>]}

- Can be declared separately or in the module declaration

(recommended)

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

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

```
module  var_mux #(parameter width = 2,
flag = 1'b1, file = "a.dat") (out,
v0, v1, sel);
  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 (1/2)

- Use **defparam** to override all parameter values in any submodule.
- Not recommended anymore

```
module  top;
  ……
  wire  [1:0]  a_out, a0, a1;
  wire  [3:0]  b_out, b0, b1;

  var_mux  U0(a_out, a0, a1, sel);
  var_mux  U1(b_out, b0, b1, sel);

  defparam top.U0.width = 2,
           top.U0.delay = 1,
           top.U1.width = 4,
           top.U1.delay = 2;
  ……
endmodule
```

Overriding the Values of Parameters (2/2)

- Override the parameters of the submodule when instantiating
- Pass the parameter down the hierarchy if needed

```
module  top;
  ……
  wire  [1:0]  a_out, a0, a1;
  wire  [3:0]  b_out, b0, b1;

  var_mux #(2, 1)
    U0(a_out, a0, a1, sel);
  var_mux #(4, 2)
    U1(b_out, b0, b1, sel);

  ……
endmodule
```

same order as declaration

(recommended)

```
module  top;
  ……
  wire  [1:0]  a_out, a0, a1;
  wire  [3:0]  b_out, b0, b1;

  var_mux #(.width(2), .delay(1))
    U0(a_out, a0, a1, sel);
  var_mux #(.width(4), .delay(2))
    U1(b_out, b0, b1, sel);

  ……
endmodule
```

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;
```

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>**
- Use `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)

- Use **`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
```

- Use **`define** to aggregate all the global constants 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

- Use 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
…
```
define.vh

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

Simulation: (if *define.vh* and *test.v* are not in the same directory)
vcs … +incdir+<dir_of_included_file>

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

Data Types

- nets - physical line connecting each component
  - wire, wand, wor, tri, triand, trior, supply0, supply1
- variables - abstract data storage for event-driven simulation
  - reg, integer, time, real, realtime

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 changes 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-AND and wired-OR
  - **tri0** and **tri1**
    - pull down and pull up

Variable Types

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

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;
- **realtime** sim_rtime;
  - initial sim_rtime = $realtime;

Wire & Reg

- **wire**
  - Physical wires in a circuit
  - A **wire** does not store its value, it must be driven by
    - Connecting to the output of a gate or submodule
    - Assigning a value in a continuous assignment (**assign**)
  - Cannot be assigned within a procedural assignment (always blocks, initial blocks, functions, etc.)
  - An un-driven wire defaults to high impedance (Z)
  - By default, **input**, **output** and **inout** port declarations are wire

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

Wire & Reg

- **reg**
  - Abstract data storage element
  - A **reg** stores its value until modified by
    - Assigning a new value in a procedural assignment
  - Cannot be assigned within a continuous assignment or connected to the output port of a gate/submodule
  - An un-initialized reg defaults to unknown (X)
  - **reg** is used to model flip-flops and latches, but does NOT necessarily implies a real register

Wire & Reg

- Use of **wire** & **reg**
  - "**assign**" statement ➔ **wire**
  - "**always**" block ➔ **reg**
  - "**input**" port in module ➔ **wire**
  - "**output**" port in module ➔ **wire/reg**
  - "**input**" port to gate/submodule ➔ **wire/reg**
  - "**output**" port from gate/submodule ➔ **wire**

* Module port connection is implicitly a continuous assignment

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

```
module  test;
  reg  A;
  wire B, C, D;

  sub_test(.a(A), .b(B),
           .c(C), .d(D));
endmodule
```

Data Type after SystemVerilog*

- **logic** (synthesizable)
  - **reg** that can also be used as **wire**
  - Unified data type simplifying the use of **wire** and **reg**
- Use of **logic**
  - Can be used with "**assign**" or in an "**always**" block

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

```
module  test;
  logic A, B, C, D;

  sub_test(.a(A), .b(B),
           .c(C), .d(D));
endmodule
```

- Cannot be used in midterm exam!

Vector

- **wire** and **reg** defaults to 1-bit
- 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 *_expr must be constant, parameter, or loop index

```
wire a;           // scalar net variable, default
wire [7:0] bus;   // 8-bit bus, bus[7] is MSB
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

- Use base and width to perform part select
- Base shall be an integer value, and width shall be a positive constant

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

big_vect   [lsb_base_expr +: width_expr]
little_vect[lsb_base_expr -: width_expr]
big_vect   [msb_base_expr -: width_expr]
little_vect[msb_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 can be multi-dimensional.
- Can only access one element at a time.

```
integer   count[0:7];       // Array of 8 integer variables
wire      bool[31:0];       // Array of 32 one-bit wire nets
time      chk_pt[1:100];    // Array of 100 time variables
reg [4:0] port_id[0:7];     // Array of 8 5-bit register variables, 1D memory
real      matrix[4:0][4:0]; // Two dimensional array of real variables
reg [3:0] rc_addr[0:5][0:7];// 2D array of 4-bit register variables, 2D memory
```

```
count[5]           // 5th element of the count array
chk_ptr[100]       // 100th element of the check point array
port_id[3]         // 3rd element of port_id array. This is a 5-bit value
port_id[3][2:0]    // Part select of the lower 3-bit of the 3rd element
port_id[3:0][2]    // Illegal, cannot part select multiple elements
rc_addr[4][6][2]   // 2nd bit of the 4th element of the 6th bank
Rc_addr[4][0:3]    // Illegal, can not access multiple elements at a time
```

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
```

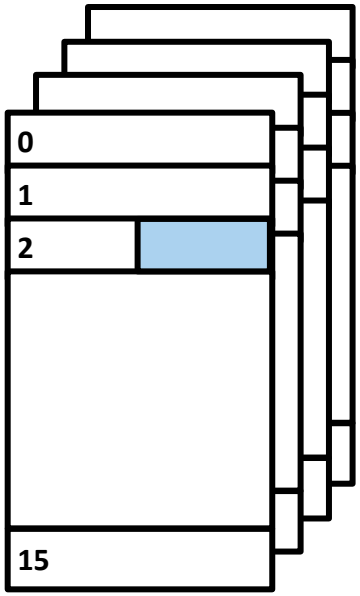
```
reg [1:n] rega; // An n-bit register is not the same
reg mema [1:n]; // as a memory of n 1-bit registers
```

Multibank Memory

- An 4-bank 16x8 bit 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
```

Arrays Extended after SystemVerilog*

- Until Verilog-2005, there can be n-D array of only 1D vector.
- **Packed array** is introduced in SystemVerilog.

```
reg [7:0][15:0] arr; // 2D packed array
arr[5][12]           // Access a bit
arr[5:3]             // Access 3 16-bit elements
```

- Packed array can also be the building block of an array.

```
reg [7:0][15:0] arr [0:3][0:6];
// 2D (unpacked) array of 2D packed array
arr[1][4][5][12]    // Access a bit
arr[1:2]            // Access 2 7-element array
```

- This topic will not be in the midterm exam, but is useful!

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 (1/2)

- Format: <size>'<sign><base_format><number>
  - <size> - number of bits (in decimal)
    - default is unsized and implementation-dependent but at least 32 bits
  - <sign> - **s** or **S** if signed, empty if unsinged (default)
    - Only affects the interpretation, not the bit pattern
  - <base format> - arithmetic base of number
    - <d> <D> - decimal
    - <h> <H> - hexadecimal
    - <o> <O> - octal
    - <b> <B> - binary

Integer Constant Representation (2/2)

- Format: <size>'<sign><base_format><number>
  - <number> - value given in base <base_format>
    - _ can be used for reading clarity
    - If number has fewer bits than specified, the higher bits will be filled by 0, x or z (refer to next slide)
  - Negative number (2's complement)
    - Negative sign is placed before <size>
    - Change the bit pattern of the parsed integer to its 2's complement
    - Only affects the bit pattern, not the interpretation

Truncation and Extension

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

  `2'b0110` ⇨ `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'b00000_010
8’sb110 // 0 padding to fill the MSB, 8'b00000_110
7'bx11  // x padding to fill the MSB, 7'bxxxx_x11
7’sbz11 // z padding to fill the MSB, 7'bzzzz_z11

reg [1:0]  a;
reg [82:0] b;
a = ‘d5; // gives 01
b = ‘d5; // gives 0…(80)…0_101
```

Integer Constant Example

- Examples:
  - 6'b1010_111 gives 01_0111
  - 8'b0110 gives 0000_0110
  - 4'bx01 gives xx01
  - 16'H3AB gives 0000_0011_1010_1011
  - 24 gives 0...001_1000 **(default as signed unsized decimal)**
  - 5'O36 gives 1_1110
  - 16'Hx gives xxxx_xxxx_xxxx_xxxx
  - 8'hz gives zzzz_zzzz
  - 'hff gives 0...0_1111_1111
  - 8'd-6 illegal
  - -8'd6 gives 1111_1010 (2's complement of 6) = $(250)_{10}$
  - 4'shf gives 1111 = $(-1)_{10}$
  - -4'shf gives -(-4'd 1) = 0001 = $(1)_{10}$

Real Constant Representation

- In Verilog, **real** constant can be represented in
  - Decimal format `<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 1.2×10^{10}
  - 236.1_51e-2 gives 235.151×10^{-2}
  - .12 illegal
  - .3E4 illegal

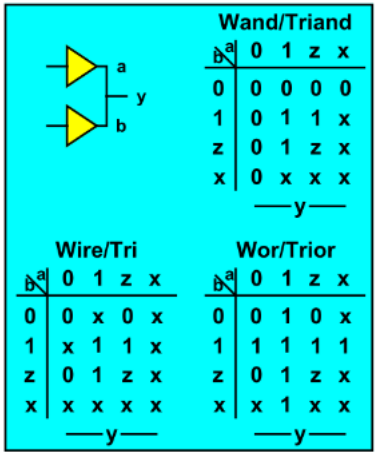
Vector Concatenations

- An 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)

APPENDIX: Nets-Wired Logic

- The family of nets includes the types **wand** and **wor**
  - A **wand** net type resolves multiple driver as wired-and logic, e.g. open collector technology
  - A **wor** net type resolves multiple drivers as wired-or logic, e.g. emitter-coupled technology
- The family of nets includes **supply0** and **supply1**
  - **supply0** has a fixed logic value of 0 to model a ground connection
  - **supply1** has a fixed logic value of 1 to model a power connection
- Used when model at *transistor-level*



Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

Primitives

- Primitives are modules ready to be instantiated
- **Smallest modeling block for simulator**
  - Behave like 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, ...);
  - Building blocks for structural level modeling
- User defined primitive (UDP)
  - building block defined by designer
  - Similar to defining a truth table

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

Operands

- **wire** & **reg** are the two main data types used for operands in RTL/Behavioral Level description
- Since all metal wires in actual circuits only represent 0 or 1, we should consider all the variables in binary
  - Unsigned: Ordinary binary
  - Signed: 2's complement binary
  - Whether the variable is unsigned or signed is up to your interpretation

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

Operators

Concatenation and replications	{ }
Negation	! , ~
Unary reduction	& , | , ^ , ^~ , …
Arithmetic	+ , - , * , / , ** , %
Shift	>> , << , >>> , <<<
Relational	< , <= , > , >=
Equality	== , != , **===** , **!==**
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 toward zero
  - 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'hfffffffd / 2 ); // 7ffffffe
      $displayb ( 2 * 1'bx); // xx...
    end
endmodule
```

How to deal with fractional 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 (correct sign extension for signed signal)
  - "**<<<**" arithmetic shift left
- The right operand is treated as unsigned, and does not affect the sign and size of the expression

```
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 = 1…110 (2's compl.)
→ Interpreted as unsigned
→ 1…110 = large number

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 and X.
  - **“===”** , **“!==”** do perform a definitive match for Z and X.

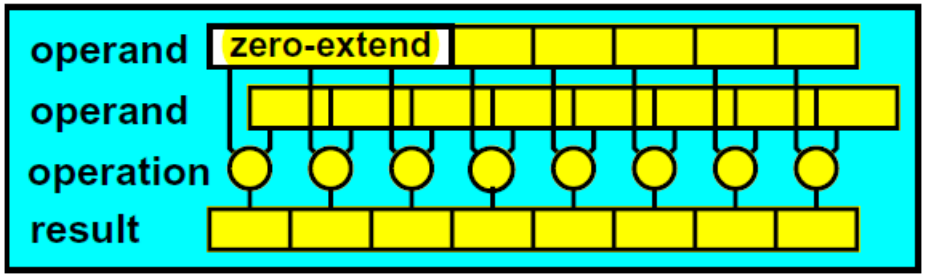
```
module equality;
  initial begin
      $displayb ( 4'b0011 == 4'b1010 ); // 0
      $displayb ( 4'b0011 != 4'b1x10 ); // 1
      $displayb ( 4'b1010 == 4'b1x10 ); // x
      $displayb ( 4'b1x10 == 4'b1x10 ); // x
      $displayb ( 4'b1z10 == 4'b1z10 ); // x
    end
endmodule
```

NON-SYNTHESIZABLE

```
module identity();
  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
```

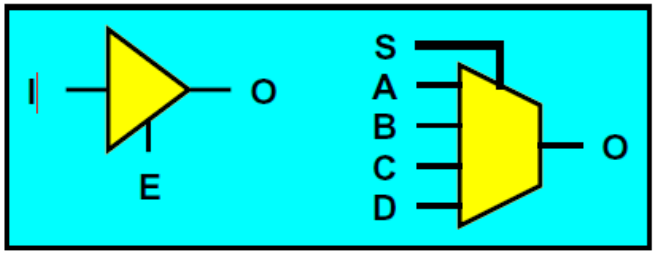
Usually connects two Boolean value (e.g. (a > b) && (a > 0))

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
Concatenation & Replications	{ }
Unary	+ , - , ! , ~ , & , | , ^ , ^~ , …
Arithmetic	**
	* , / , %
	+ , -
Shift	>> , << , >>> , <<<
Relational	< , <= , > , >=
Equality	== , != , **===** , **!==**
Bitwise	&
	^ , ^~
	|
Logical	&&
	||
Conditional	? :

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

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

Signing and Sizing

- What are the results of the following expressions

```
reg [31:0] A, B, C, D, E;

A = -12 / 3;                    // -4
B = -'d12 / 3;                  // 1431655761
C = -'sd12 / 3;                 // -4
D = -'sd12 / $signed('d3);      // -4
E = -4'sd12 / 3'sd3;            // 1
```

```
reg [7:0] o;
reg [3:0] a, b;
reg signed [3:0] c;

a = 4'd15;  b = 4'd8;  c = -4'd1;
o = (a + b) + c;                // 38
```

Signing

- Signals are **"regarded as"** signed in Verilog
  - In signed operations, the bit pattern is viewed as 2's complement
  - The bit pattern itself does not change

$(1111)_2$ ⇨ signed data type ⇨ $(-1)_{10}$
$(1111)_2$ ⇨ unsigned data type ⇨$(15)_{10}$

- The number without size and base format specification is a signed decimal
- The number with a base format should be viewed as unsigned unless the ***s*** designator is included

$12 = (00\ldots001100)_2$
$-12 = (11\ldots110100)_2 = -12$ (signed)
$-'d12 = (11\ldots110100)_2 = 4291967296$ (unsigned)
$-'sd12 = (11\ldots110100)_2 = -12$ (signed)

Sign Conversion

- A built-in system functions to handle type casting
  - **$signed():** Return value is signed
  - **$unsigned():** Return value is unsigned

- Example

```
reg [7:0] a, b, c;

a = $unsigned(-4)      // a = 8'b11111100
b = $unsigned(-4'sd4) // b = 8'b00001100
c = $signed(4'b1100)  // c = 8'b11111100
```

Signed Signal (1/2)

- Signed addition and subtraction
  - Equivalent to unsigned signals
- Signed multiplication, division, and comparison
  1. Use the **`signed`** declaration of regs and wires

```
reg cmp;  reg signed [7:0] a, b;

cmp = a < b;
```

  2. Use **`$signed()`** function to cast unsigned variables (recommended)

```
reg cmp;  reg [7:0] a, b;

cmp = $signed(a) < $signed(b);
```

  3. For constant, use the ***`s`*** designator

```
reg cmp;  reg [7:0] a;

cmp = $signed(a) < -8'sd2;
```

Signed Signal (2/2)

- Some expressions will convert signed signals to unsigned
  - Bit-selection
  - Part-selection, even if the selection specifies the entire vector
  - Concatenation
  - Comparison results (1, 0)

```
reg signed [ 7:0] a, b;
reg signed [16:0] c;

c = a[7:0]; // {{8{1'b0}}, a}
// b[7:0] is unsigned and therefore zero-extended

c = {a, b}; // {1'b0, {a, b}}
// {a, b} is unsigned and therefore zero-extended
```

Signing and Sizing Rules

- Verilog evaluates an expression by the following steps

1. **Determine the sign of the right-hand side (RHS), then cast all RHS to the result type**
   - If all operands are signed, the result is signed
   - If any operand is unsigned, the result is unsigned
   - The LHS should not be considered
2. **Determine the expression size by choosing the largest operand size, including the LHS**
3. **Resize all RHS operands to the expression size**
   - Signed number is sign extended
   - Unsigned number is zero extended
4. **Evaluate the RHS expression, producing a result of type and size found in step1-2**
5. **Assign the result value to LHS, truncate the higher bits if size too large**

Signing and Sizing Examples (1/6)

```
wire        [3:0] a;
wire signed [3:0] b;
wire signed [7:0] out;
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)
> a = 5 (0000_0101), b = -2 (0000_1110)
> out = 19 (0001_0011)
```

- a is unsigned, b is signed ⇨ result is unsigned ⇨ **zero** extension

```
wire signed [3:0] a;
wire signed [3:0] b;
wire signed [7:0] out;
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)
> a = 5 (0000_0101), b = -2 (1111_1110)
> out = 3 (0000_0011)
```

- a is signed, b is signed ⇨ result is signed ⇨ **sign** extension

```
wire signed [3:0] a;
wire signed [3:0] b;
wire        [7:0] out;
assign out = a + b;
```

```
> a = -3 (1101), b = -2 (1110)
> a = -3 (1111_1101), b = -2 (1111_1110)
> out = 251 (1111_1011)
> $signed(out) = -5 (1111_1011)
```

Signing and Sizing Examples (2/6)

- Relational and equality operator
- **The result of part-select is unsigned**

```
reg signed [3:0] a;
Reg        [3:0] b;
reg signed [2:0] c;
reg cmp;

a = -1; b = 15; c = 7;

cmp = a > 3;                    // 0, signed -1 > 3
cmp = a > 'd3;                  // 1, unsigned 1111 > 0011
cmp = a > 4'sd3;                // 0, signed -1 > 3
cmp = a[3:0] > 4'sd3;           // 1, unsigned 1111 > 0011
cmp = (a == b);                 // 1, unsigned 1111 == 1111
cmp = (a == $signed(b));        // 1, signed -1 == -1
cmp = (a == c);                 // 1, signed extension
```

Signing and Sizing Examples (3/6)

- Average of two number

```
reg [3:0] a, b, avg;          // a = 14, b = 10
reg [4:0] avg_good;

// will not work properly, avg = (8) >> 1 = 4
avg = (a + b) >> 1;

// will work correctly (“0” is unsized), avg = 12
avg = (a + b + 0) >> 1;

// will work correctly, avg = 12
avg_good = (a + b) >> 1;
```

Signing and Sizing Examples (4/6)

- Using sign extension improperly

```
reg signed [15:0] o, o_bad;
reg signed [ 3:0] a, b;       // a = -4, b = 5

// will not work properly
// o_bad = (0…0_11111100)2 * (0…0_0101)2 = 1260
o_bad = { {4{a[3]}}, a } * { {4{b[3]}}, b };

// will work correctly, o = -20
o = a * b;
```

Signing and Sizing Examples (5/6)

- Signed 16-bit multiplier

```
reg         [ 7:0] a, b;       // a = 100, b = -100
reg signed [15:0] c, c_bad;

// will not work properly
// c_bad = (01100100)2 * (10011100)2 = (11110000)2 = -16
c_bad = $signed( $signed(a) * $signed(b) );

// will work correctly, c = -10000
c = $signed(a) * $signed(b);
```

- The expression in **`$signed()`** should be evaluate first
  - The width of `$signed(a) * $signed(b)` in the c_bad expression is 8 bits, so this expression gives an incorrect result

Signing and Sizing Examples (6/6)

- Conditional operator

```
reg [ 7:0] a, b;        // a = 100, b = -100
reg [15:0] c, d;        // d = 66
reg condition;          // condition = 1

// unsigned, c = 15600
c = (condition) ? $signed(a) * $signed(b) : d;

// signed, c = -10000
c = $signed(condition) ? $signed(a) * $signed(b) : $signed(d);

// signed, c = -10000
c = (condition) ? $signed(a) * $signed(b) : $signed(d);
```

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

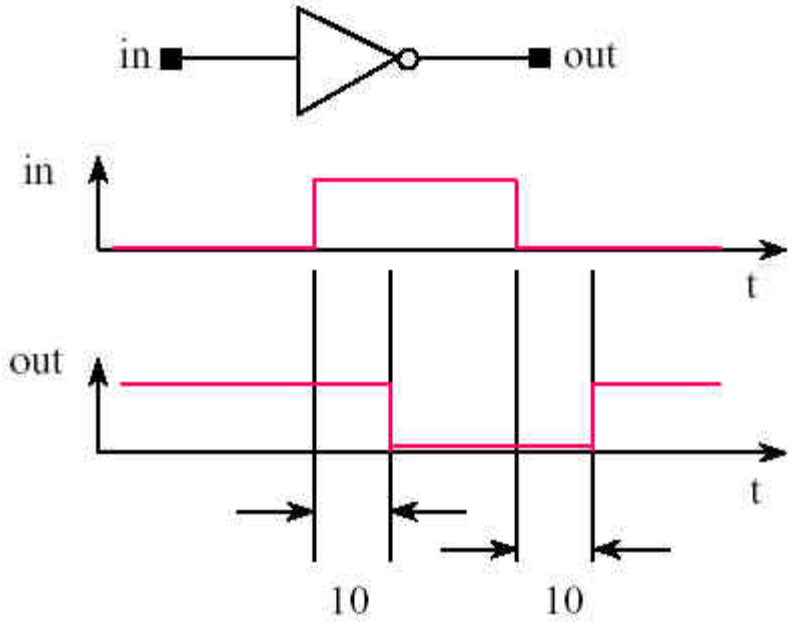
Timing and Delay (for verification)

- 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 between them.
- Use timing / delay description to model the delay: **#**
- Along with the delay specifications of the blocks, we can perform timing check

Delay Specification in Primitives

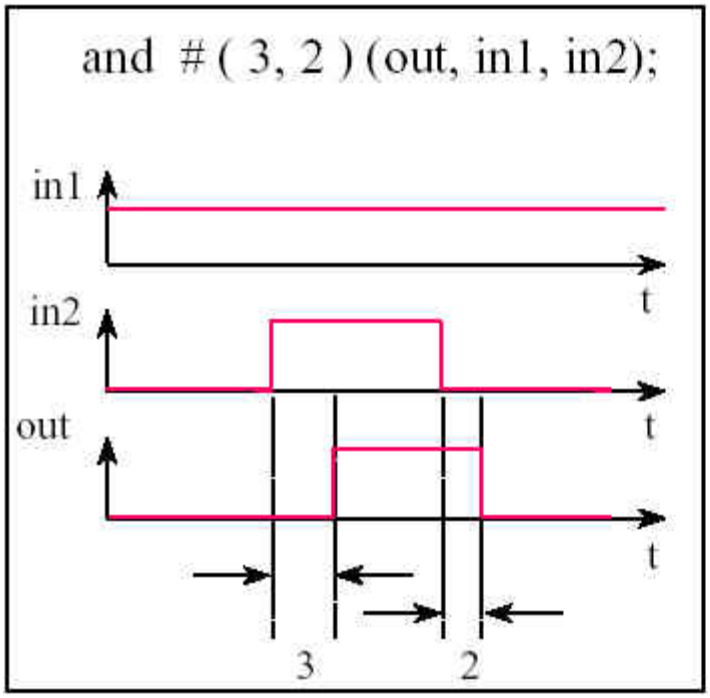
- Delay specification defines the propagation delay of that primitive gate.

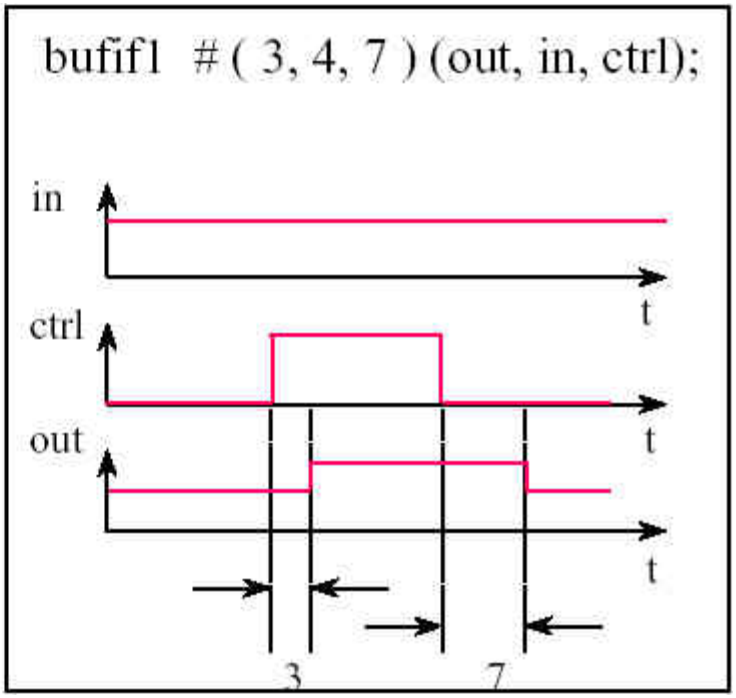
not #10 (out,in);



Delay Specification in Primitives

- Verilog supports (rise, fall, turn-off) delay specification.
  - Fall delay: transition to 0
  - Tun-delay: transition to z
  - Rise delay: otherwise



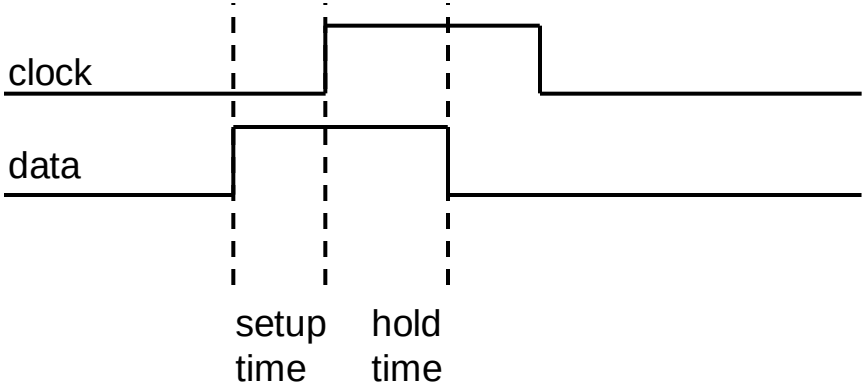


Delay Specification in Primitives

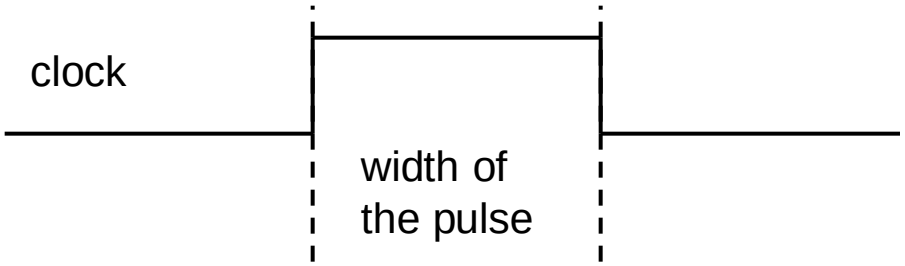
- All delay specification in Verilog can be specified as *(minimum : typical : maximum)* delay
- Examples
  - *(min:typ:max)* delay specification of all transition
    - or #(3.2:4.0:6.3) U0(out, in1, in2);
  - *(min:typ:max)* delay specification of RISE transition and FALL transition
    - nand #(1.0:1.2:1.5, 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



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




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

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

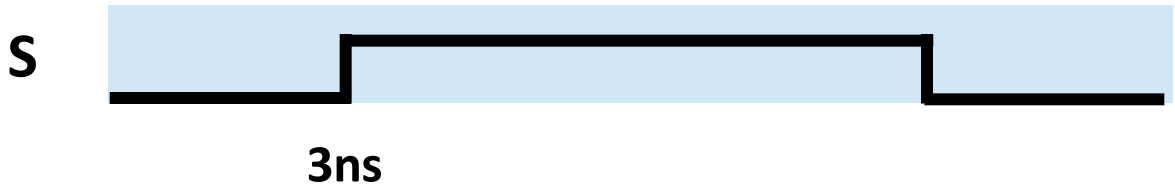
```
specify
    $width(posedge clock, 6);
endspecify
```

Outline

- Lexical Conventions
- Data Types
- Primitives
- Operands and Operators
- Signing and Sizing
- Timing and Delay
- Events

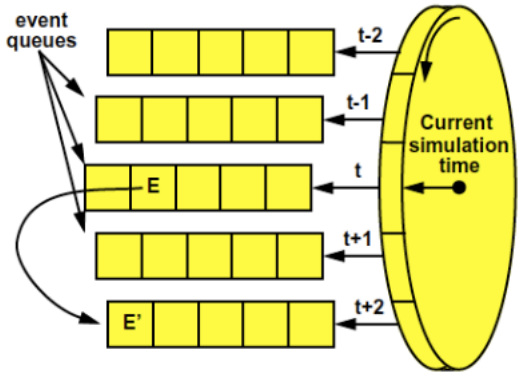
Events

- A special data type holding state and timing description
  - State: level change (1→0, 0→1, 1→X, 0→Z, etc.), edge (logic-level transition)
  - Timing: virtual time in simulator
- Example: signal ***s*** changed from 0 to 1 at 3ns



Event-Based Simulation

- Execute statement (evaluate expression and update variable) when defined event occurs
  - input transition cause an event on circuit
  - simulation is on the OR-ed occurrence of sensitive events
- Benefits
  - Accelerate simulation speed: Only evaluate expression when variables on RHS change
  - Allow high-level description (behavior) of a constructor



A current event scheduling another event

APPENDIX: Coding Environment

- VS Code is an easy-to-use IDE with many extensions
- Verilog extension supports syntax highlighting and linting
- Keep in mind, the extension can help with your homework, but not your exam