Computer-Aided VLSI System Design

Chap.1 Fundamentals of Hardware Description Language

Lecturer: 張惇宥

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

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
- Simulation & Verification

Verilog Course Overview

- Chapter 1
  - Fundamentals of HDL
  - Language element
  - Structural level modeling
- Chapter 2
  - RTL (Dataflow) 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

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 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.

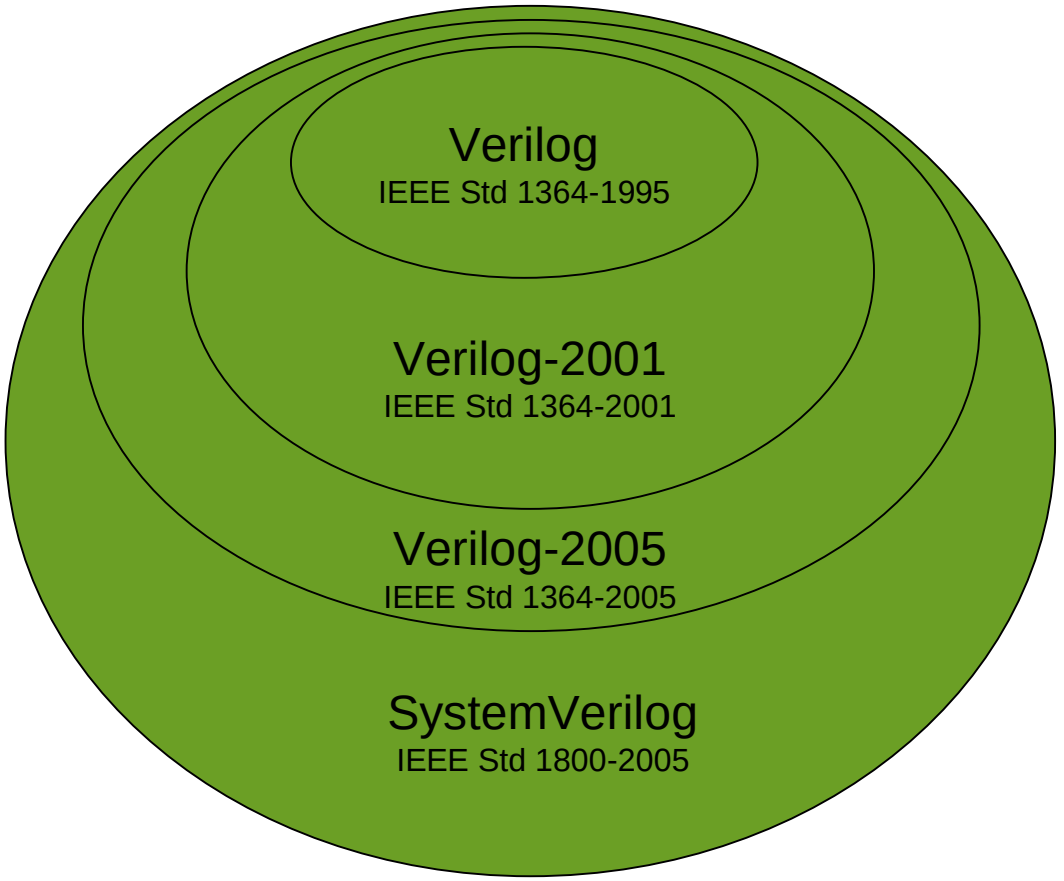
List of HDL for Digital Circuits

- **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.

About Verilog

- Introduction in 1984 by Phil Moorby and Prabhu Goel in Automated Integrated Design System
- Open and Standardize (IEEE 1364-1995) in 1995 by Cadence because of the increasing success of VHDL (standard in 1987)
- Become popular and makes tremendous improvement in productivity
  - Syntax similar to C programming language, though the design philosophy differs greatly
- Verilog standard: IEEE Standard for Verilog Hardware Description Language

History/Branch of Verilog



Digital-signal HDL

Outline

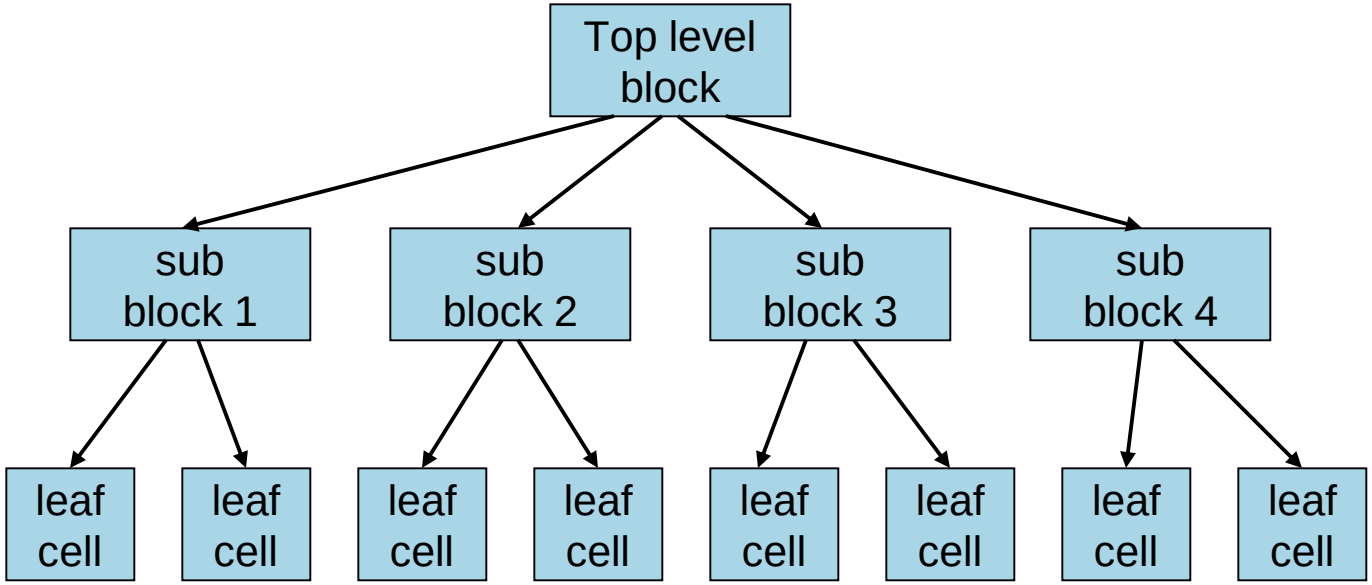
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Lexical Conventions
  - Data Type
  - Primitive
  - Timing and Delay
- Simulation & Verification

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

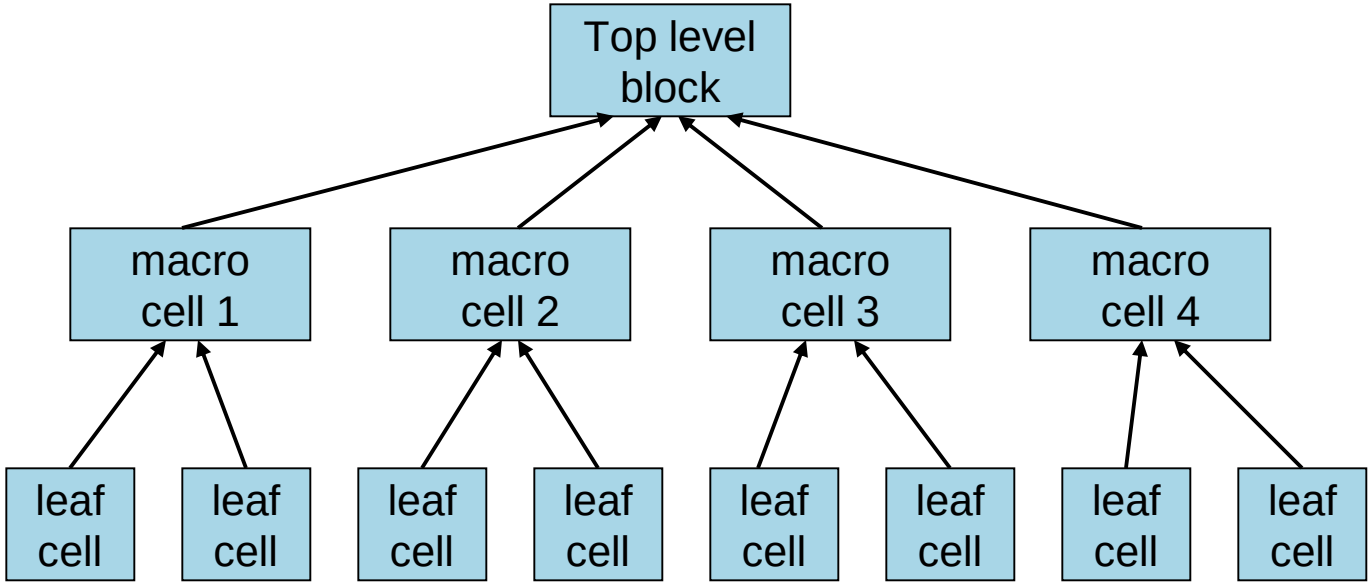
Top-down Design Methodology

- 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

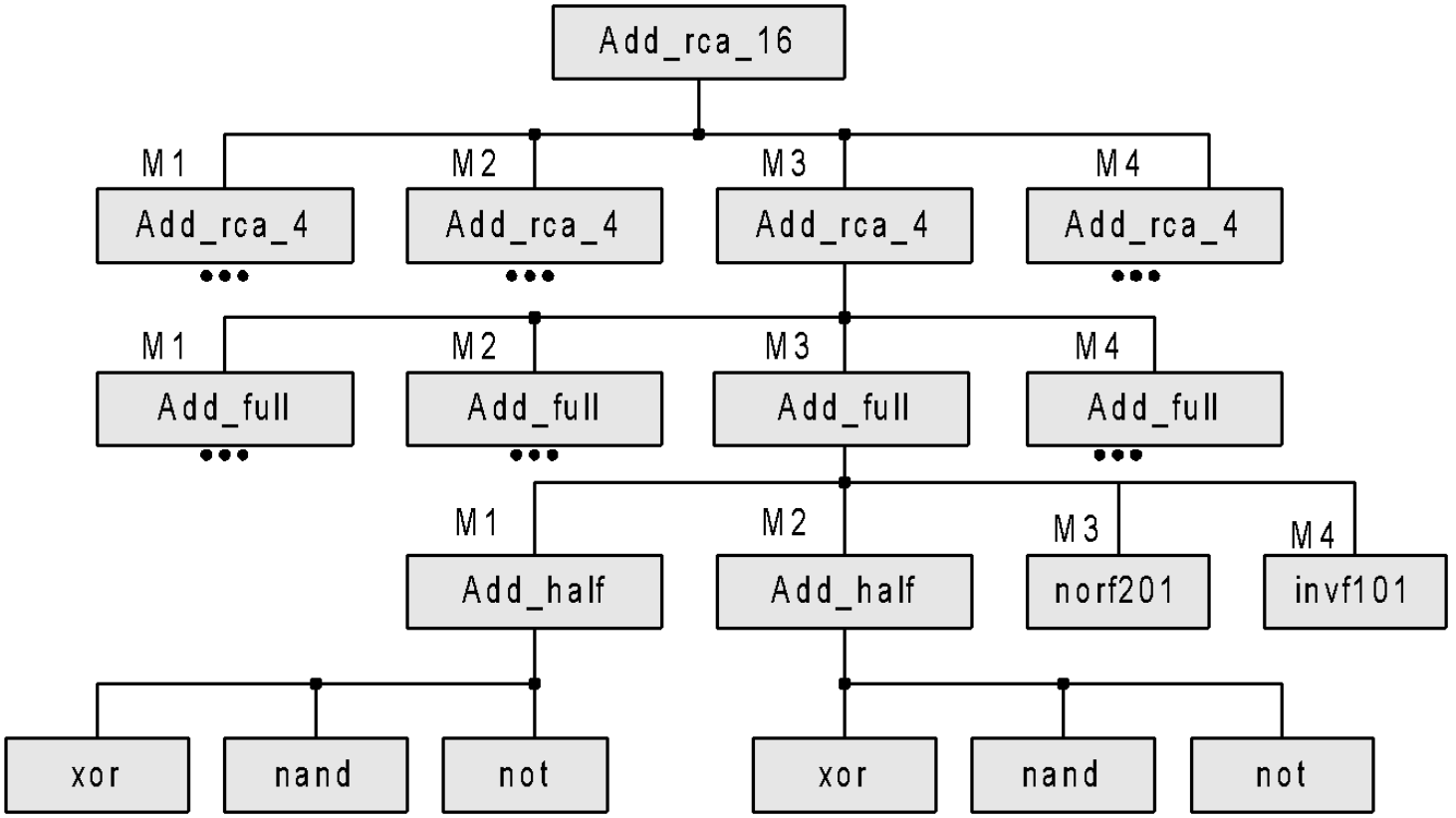
- We first identify the building block that are available to us.
- We build bigger cells, using these building blocks.
- These cells are then used for higher-level blocks until we build the top-level block in the design.



Example: 16-bit Adder

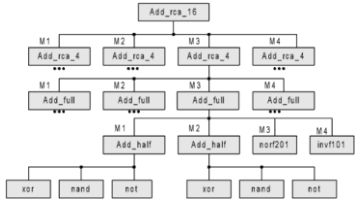
conquer

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).



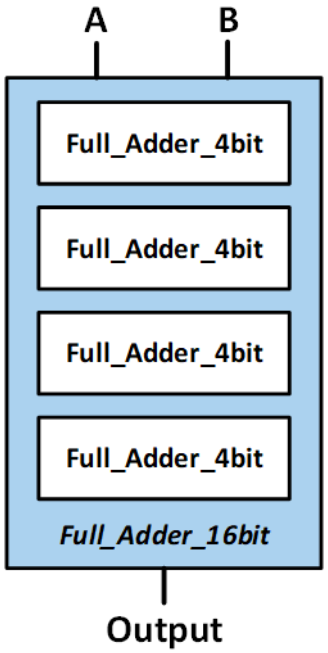
Module

- 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

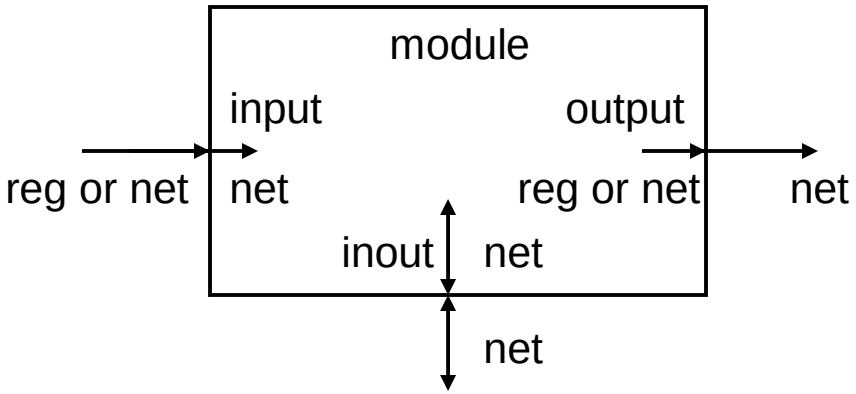
- Modules communicate with each other through **ports**
- There are three kinds of ports: input, output, and inout
- There are two ways to declare module ports
  1. List a module's ports in parentheses "()" after the module name, and declare ports in the module description
  2. List and declare a module's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

- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.

Module Instantiation (2/2)

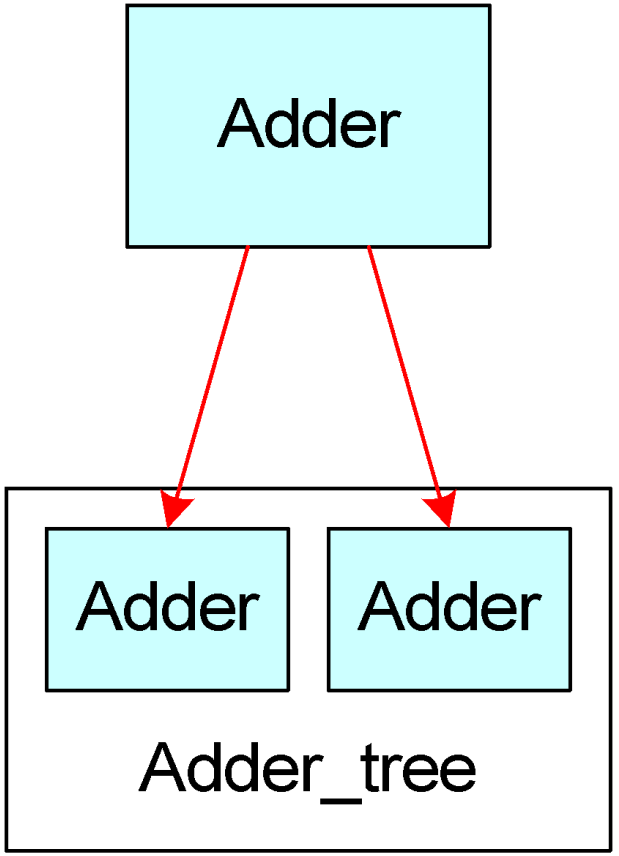
```
adder adder_0 ( out0 , in1 , in2 );
```

Module name Instance name

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

    assign  out=in1 + in2;
endmodule
```

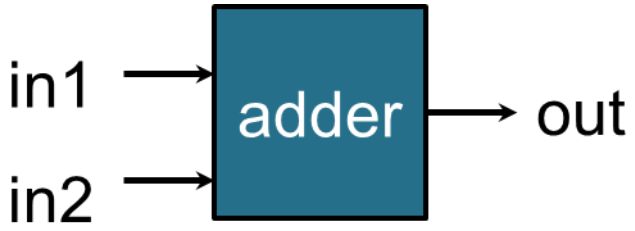
instance example

```
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() );`

Analogy: module ↔ class

As ***module*** is to ***Verilog HDL***, so ***class*** is to ***C++ programming language***.

Syntax	module *m_Name*(IO list);
	...
	endmodule
		class *c_Name* {
		...
		};

Instantiation	*m_Name* **ins_name** (port connection list);	*c_Name* **obj_name**;
Member	**ins_name**.member_signal	**obj_name**.member_data
Hierarchy	instance.sub_instance.member_signal	object.sub_object.member_data

Analogy: module ↔ class

```
module m_AND_gate (in_a, in_b, out);
    input  in_a;
    input  in_b;
    output out;
    assign out = in_a & in_b;

endmodule
```

Model AND gate with Verilog HDL

```
class c_AND_gate {
    bool in_a;
    bool in_b;
    bool out;
    void evaluate()
         { out = in_a && in_b; }
};
```

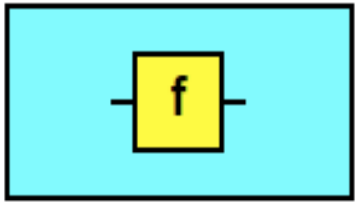
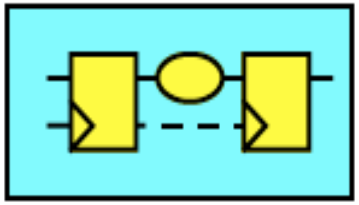
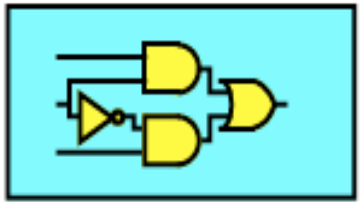
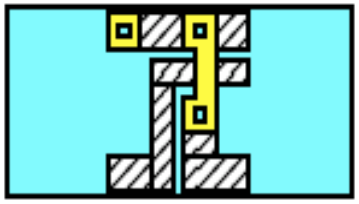
Model AND gate with C++

- assign and evaluate() is simulated/called at each $T_{i+1} = T_i + t_{resolution}$

Outline

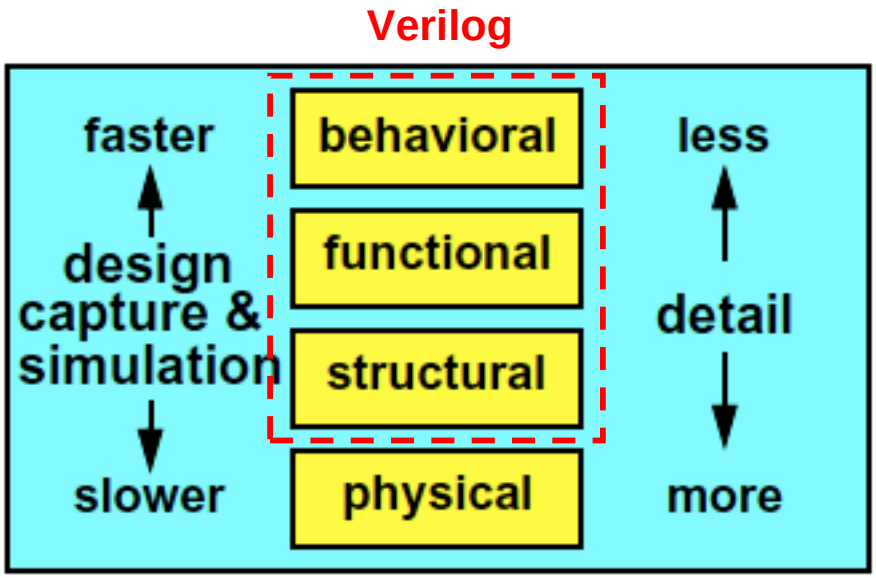
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Lexical Conventions
  - Data Type
  - Primitive
  - Timing and Delay
- Simulation & Verification

Cell-Based Design and Levels of Modeling

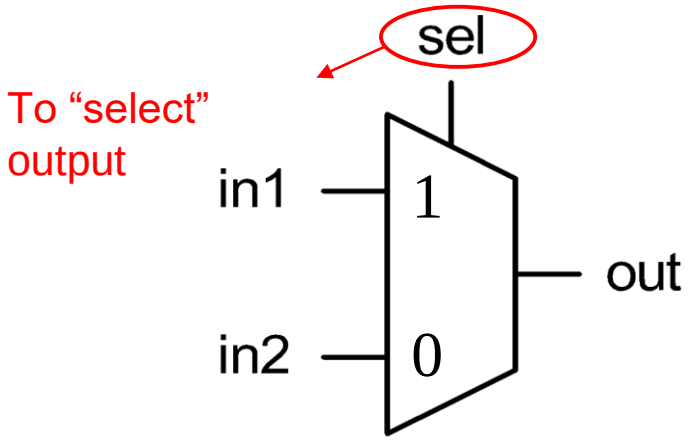
Behavioral Level		Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools
Register Transfer Level (RTL)		Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level
Structural/Gate Level		Usually generated by synthesis 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.



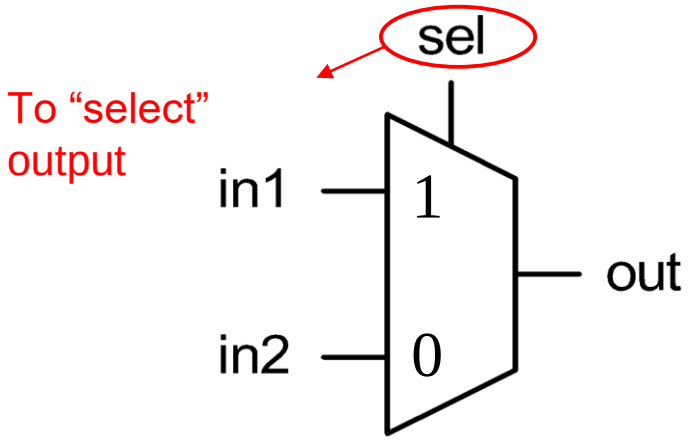
An Example - 1-bit Multiplexer in Behavioral Level




```
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



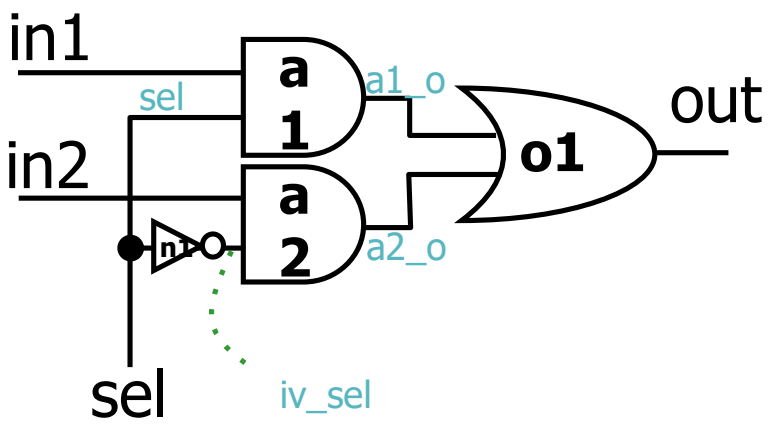

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    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



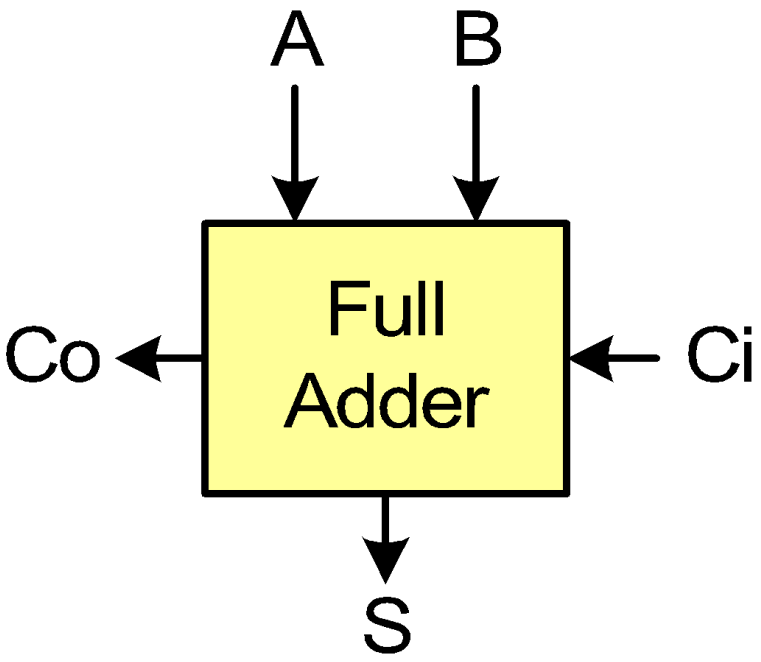

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    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)

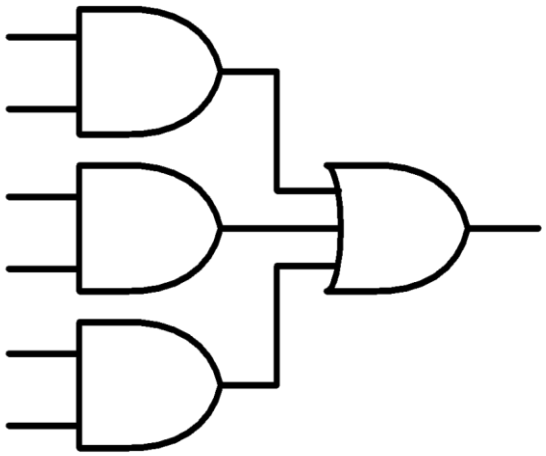
Case Study
1-bit Full Adder



Ci	A	B	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Case Study
1-bit Full Adder

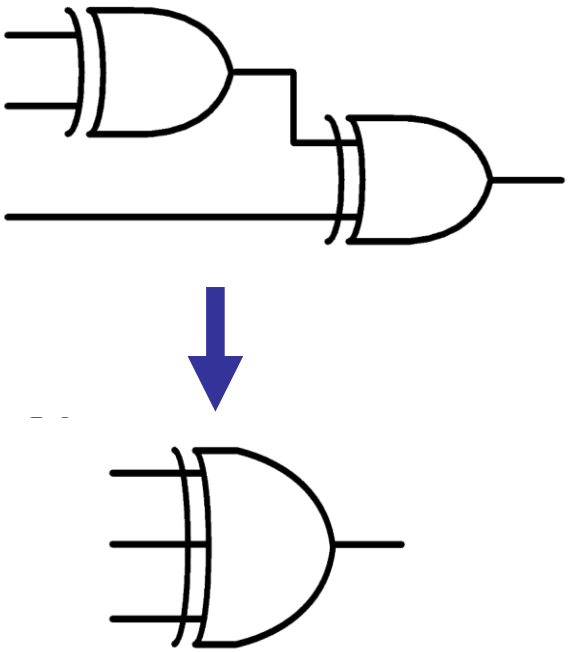
- co = (a • b) + (b • ci) + (ci • a);



```
30
31 module FA_co ( co, a, b, ci );
32
33   input   a, b, ci;
34   output  co;
35   wire    ab, bc, ca;
36
37   and g0( ab, a, b );
38   and g1( bc, b, c );
39   and g2( ca, c, a );
40   or  g3( co, ab, bc, ca );
41
42 endmodule
43
```

Case Study
1-bit Full Adder

- sum = a ⊕ b ⊕ ci

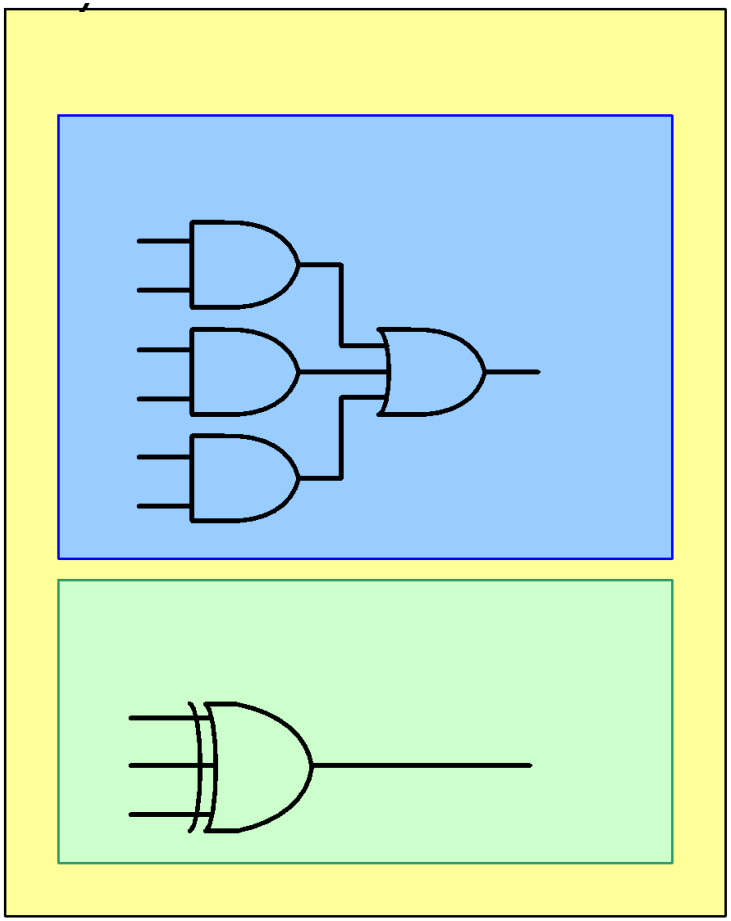


```
44 module FA_sum ( sum, a, b, ci );
45
46   input   a, b, ci;
47   output  sum, co;
48
49   xor g1( sum, a, b, ci );
50
51 endmodule
52
```

Case Study
1-bit Full Adder

- Full Adder Connection
  - Instance *ins_c* from FA_co
  - Instance *ins_s* from FA_sum

```
20
21 module FA_gatelevel( sum, co, a, b, ci);
22
23   input    a, b, ci;
24   output   sum, co;
25
26   FA_co    ins_c( co, a, b, ci );
27   FA_sum   ins_s( sum, a, b, ci );
28
29 endmodule
30
```



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
- Simulation & Verification

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

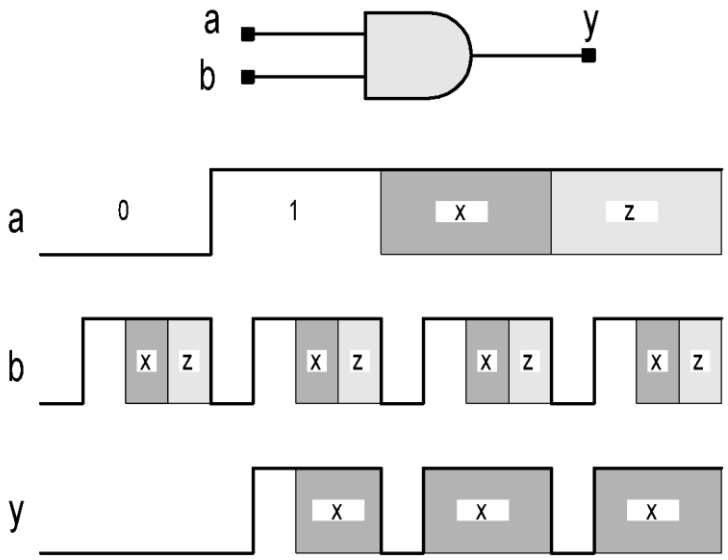
Four-Valued Logic System

- 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 (see Appendix A in text)



	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 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
- Simulation & Verification

Data Types

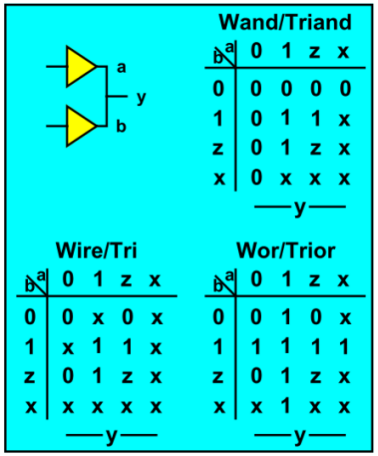
- nets are further divided into several net types
  - wire, wand, wor, tri, triand, trior, supply0, supply1
- registers – variable to store a logic value for event-driven simulation - reg
- integer - supports computation 32-bits signed
- time - stores time 64-bit unsigned
- real - stores values as real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

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*



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" ⇨ usually use "**assign**" and "**assign**" **does not** appear in "**always**" block
  - When use "reg" ⇨ 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 ⇨ 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
reg r_1bit;

// vector
reg [3:0] r_4bit_vec;

// 1D array: memory: 32bit×8
reg [31:0] r_memory [7:0];
```

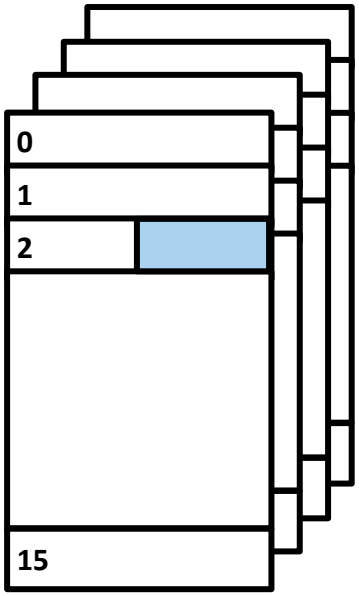
```
// 1D array: 8bit×8
wire [7:0] w_net [7:0];

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

// 3D array: 100×16×4 float variables
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 ⇨ 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'bxxxxx11
7'bz11 // x padding to fill the MSB, 7'bzzzzz11

reg [1:0]  a;
reg [82:0] b;
a = 'd5 // gives {80{1'b0}, 3'b101}
b = 'd5 // gives 01
```

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 32-bit unsigned decimal)**
  - 5'O36 gives 1_1110
  - 16'Hx gives xxxxxxxxxxxxxxxx
  - 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 1.2×10^{10}
  - 236.1_51e-2 gives 235.151×10^{-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
- Simulation & Verification

Primitives

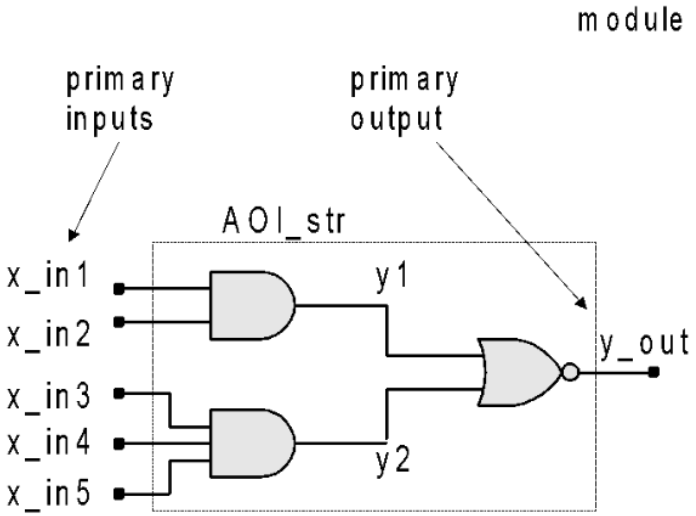
- Primitives are modules ready to be instanced
- **Smallest modeling block for simulator**
  - Behavior as software execution in simulator, not hardware description
- Verilog build-in primitive gate
  - *and, or, not, buf, xor, nand, nor, xnor*
  - prim_name inst_name(output, in0, in1,....);
- User defined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

Verilog Built-in Primitives

		Ideal MOS switch	Resistive gates	
and	buf	nmos	rnmos	pullup
nand	not	pmos	rpmos	pulldown
or	bufif0	cmos	rcmos	
nor	bufif1	tran	rtran	
xor	notif0	tranif0	rtranif0	
xnor	notif1	tranif1	rtranif1	

Structural Models

- Verilog primitives encapsulate pre-defined functionality of common logic gates
- The counterpart of a schematic is a structural model composed of Verilog primitives
- Model structural detail by instantiating and connecting primitives



module name

module ports

```
module AOI_str (y_out, x_in1, x_in2, x_in3, x_in4, x_in5);
  output y_out;
  input  x_in1, x_in2, x_in3, x_in4, x_in5;

  wire   y1, y2;

  nor    nor1(y_out, y1, y2);
  and    and1(y1, x_in1, x_in2);
  and    and2(y2, x_in3, x_in4, x_in5);
endmodule
```

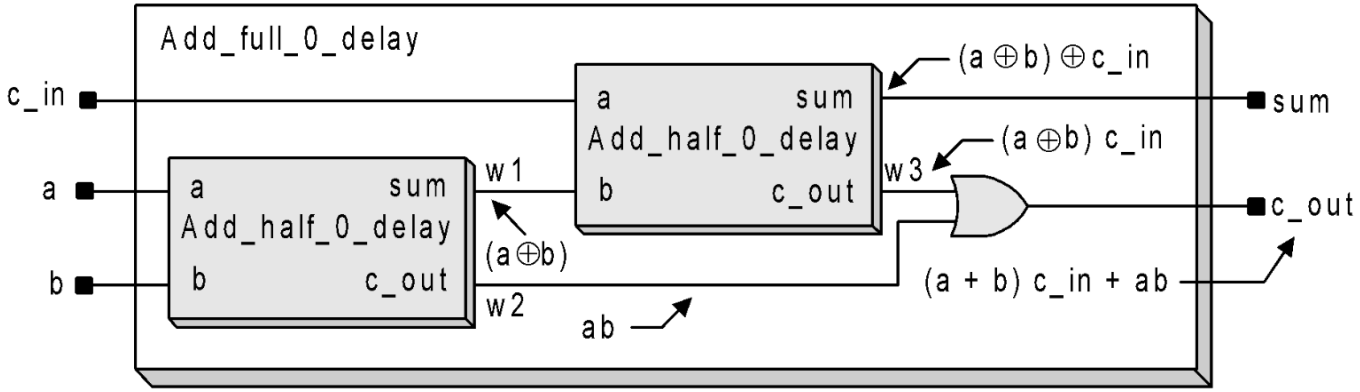
port modes

internal wires establish connectivity

instantiated primitives

Hierarchical Design Example

- Model complex structural detail by instantiating modules within modules




```
module Add_full_0_delay (sum, c_out, a, b, c_in);
  input  a, b, c_in;
  output c_out, sum;
  wire   w1, w2, w3;

  Add_half_0_delay M1 (w1, w2, a, b);
  Add_half_0_delay M2 (sum, w3, c_in, w1);
  or (c_out, w2, w3);
endmodule
```

module instance name

MODELING TIP

Use nested module instantiations to create a top-down design hierarchy.

MODELING TIP

The ports of a module may be listed in any order.
The instance name of a module is required.

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
- Simulation & Verification

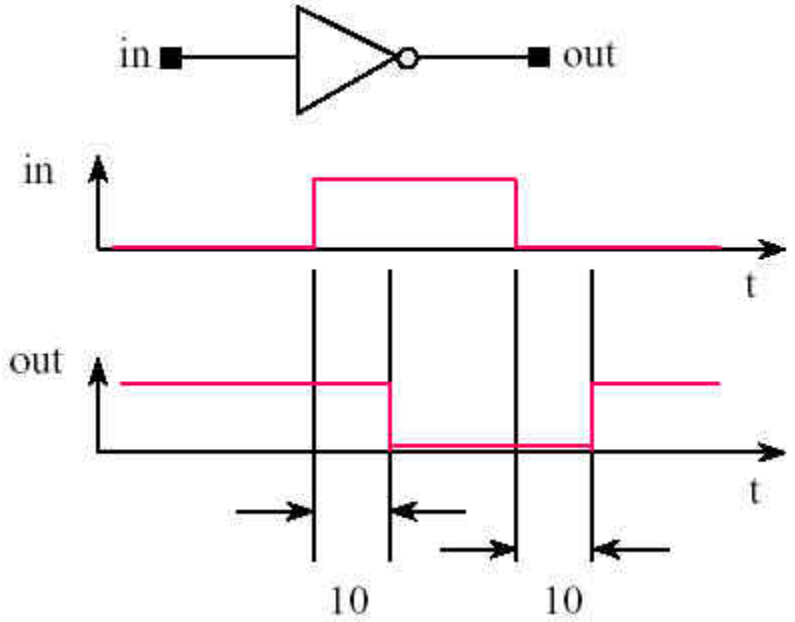
Timing and Delay (for verification)

- 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

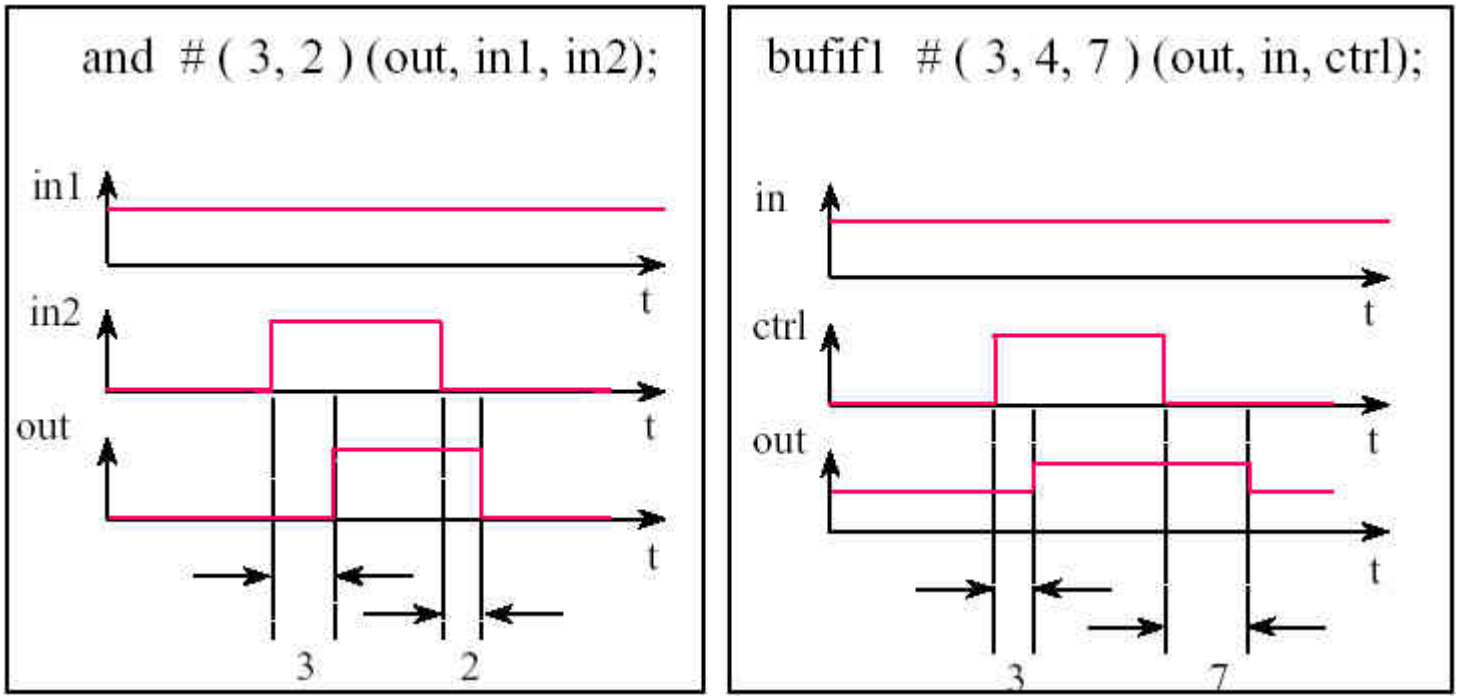
- Delay specification defines the propagation delay of that primitive gate.

```
not  #10 (out,in);
```



Delay Specification in Primitives

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

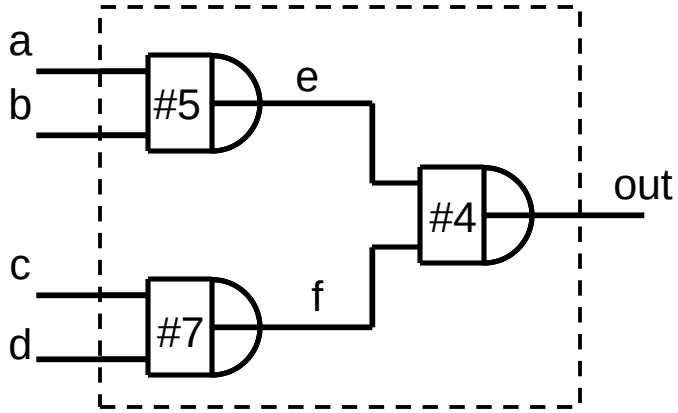


Delay Specification in Primitives

- 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);

Types of Delay Models

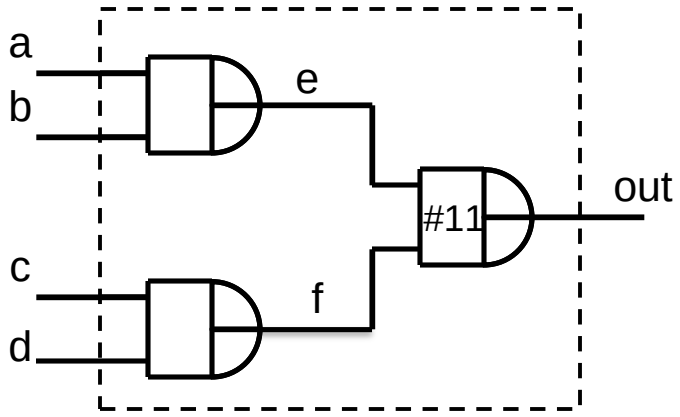
- Distributed Delay
  - Specified on a per element basic
  - Delay value are assigned to individual in the circuit




```
module  and4(out, a, b, c, d);
  … …
  and  #5  a1(e, a, b);
  and  #7  a2(f, c, d);
  and  #4  a3(out, e, f);
endmodule
```

Types of Delay Models

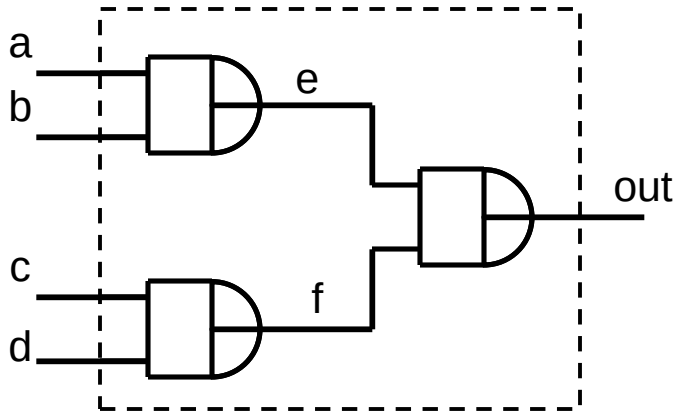
- Lumped Delay
  - They can be specified as a single delay on the output gate of the module
  - The cumulative delay of all paths is lumped at one location




```
module  and4(out, a, b, c, d);
  … …
  and       a1(e, a, b);
  and       a2(f, c, d);
  and  #11   a3(out, e, f);
endmodule
```

Types of Delay Models

- Pin-to-Pin Delay
  - Delays are assigned individually to paths from each input to each output.
  - Delays can be separately specified for each input/output path.



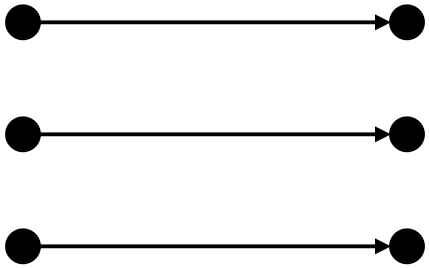
Path a-e-out, delay = 9
Path b-e-out, delay =9
Path c-f-out, delay = 11
Path d-f-out, delay = 11

Path Delay Modeling

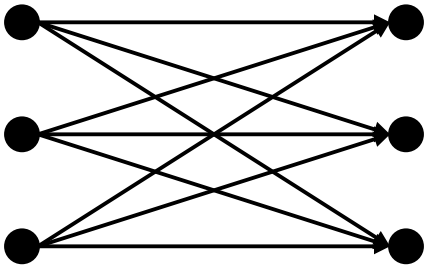
- Specify blocks
  - Assign pin-to-pin timing delay across module path
  - Set up timing checks in the circuits
  - Define ***specparam*** constants

```
module  and4(out, a, b, c, d);
  … …
  // specify block with path delay
    statements
  specify
      (a => out) = 9;
      (b => out) = 9;
      (c => out) = 11;
      (d => out) = 11;
  endspecify

  // gate instantiations
  and     a1(e, a, b);
  and     a2(f, c, d);
  and     a3(out, e, f);
endmodule
```

Parallel/Full Connection



```
(a[0] => out[0]) = 9;
(a[1] => out[1]) = 9;
(a[2] => out[2]) = 9;
(a[3] => out[3]) = 9;


(a => out) = 9;
```



```
(a => out) = 9;
(b => out) = 9;
(c => out) = 11;
(d => out) = 11;


(a,b *> out) = 9;
(c,d *> out) = 11;
```

Specparam Statement

- Special parameters can be declared for use inside a ***specify*** block.
- Instead of using hardcoded delay numbers to specify pin-to-pin delays

```
module  and4(out, a, b, c, d);
  … …
  // specify block with path delay
    statements
  specify
    specparam  delay1 = 9;
    specparam  delay2 = 11;

      (a,b *> out) = delay1;
      (c,d *> out) = delay2;
  endspecify
  … …
endmodule
```

Rise, Fall, and Turn-off Delays

- Pin-to-pin timing can also be expressed in more detail by specifying rise, fall, and turn-off delay values

```
// specify one delay statements
   specparam  t_delay = 9;
     (clk => q) = t_delay;
```

```
// specify two delay statements
   specparam  t_rise = 9;
   specparam  t_fall = 13;
      (clk => q) = (t_rise, t_fall);
```

```
// specify three delay statements
   specparam  t_rise = 9;
   specparam  t_fall = 13;
   specparam  t_turnoff = 11;
      (clk => q) = (t_rise, t_fall, t_turnoff);
```

```
// specify six delay statements
   specparam  t_01= 9, t_10 = 13;
   specparam  t_0z = 11, t_z1 = 9;
   specparam  t_1z = 11, t_z0 = 13;
      (clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);
```

```
// specify twelve delay statements
   specparam  t_01= 9, t_10 = 13;
   specparam  t_0z = 11, t_z1 = 9;
   specparam  t_1z = 11, t_z0 = 13;
   specparam  t_0x= 9, t_x1 = 13;
   specparam  t_1x = 11, t_x0 = 9;
   specparam  t_xz = 11, t_zx = 13;
      (clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0,
                    t_0x, t_x1, t_1x, t_x0, t_xz, t_zx);
```

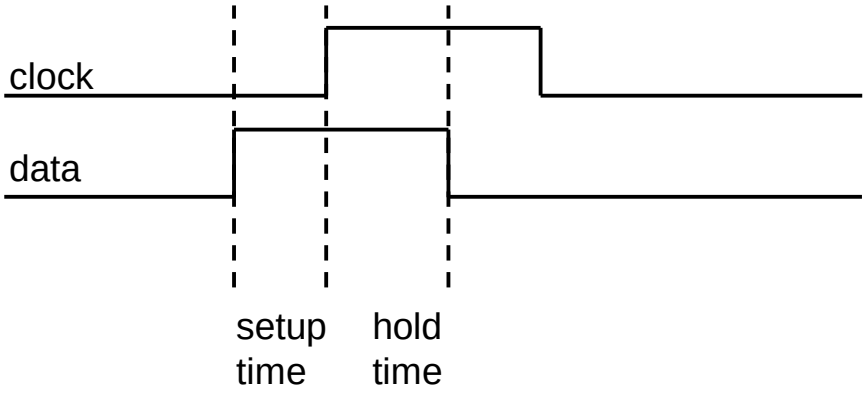
Min, Max, and Typical Delays

- Min, max, and typical delay value were discussed earlier for gates
- Can also be specified for pin-to-pin delays.

```
// specify two delay statements
   specparam  t_rise = 8:9:10;
   specparam  t_fall = 12:13:14;
   specparam  t_turnoff = 10:11:12
      (clk => q) = (t_rise, t_fall, t_turnoff);
```

Timing Checks (For Testbench)

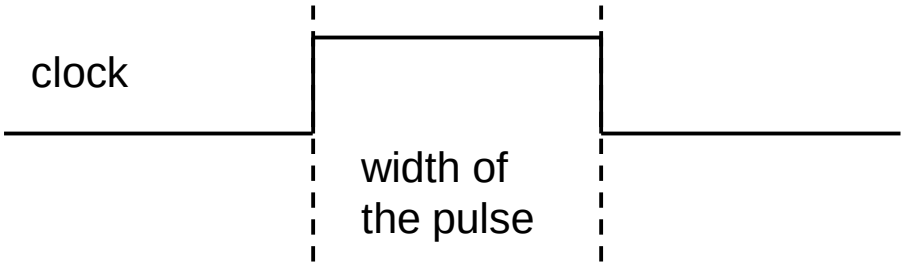
- setup time and hold time checks




```
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.



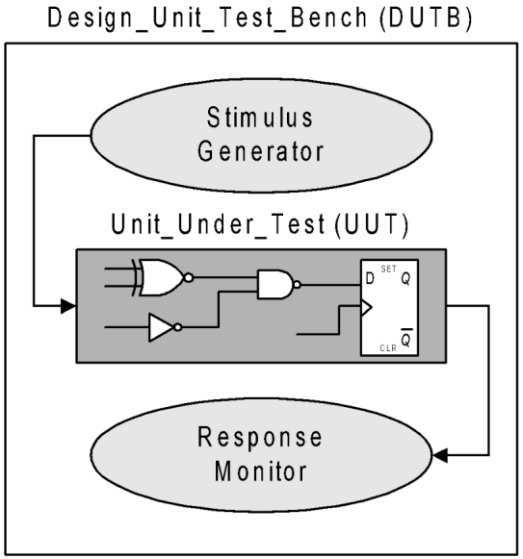

```
specify
    $width(posedge clock, 6);
endspecify
```

Outline

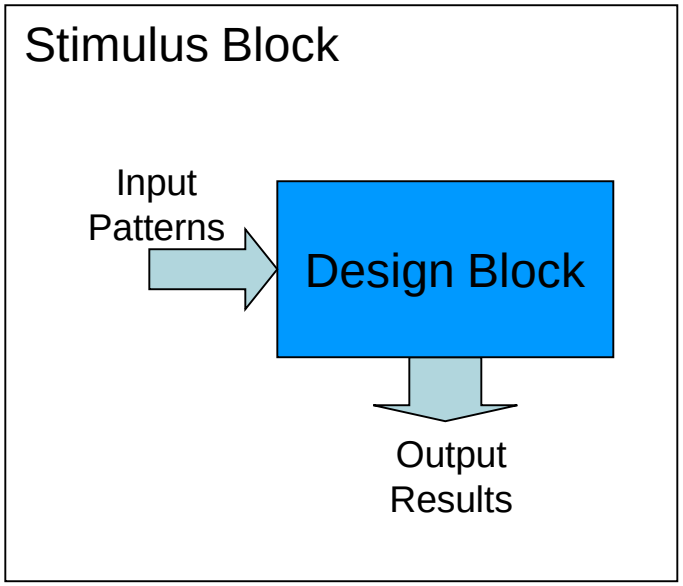
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Lexical Conventions
  - Data Type
  - Primitive
  - Timing and Delay
- Simulation & Verification

Verification Methodology

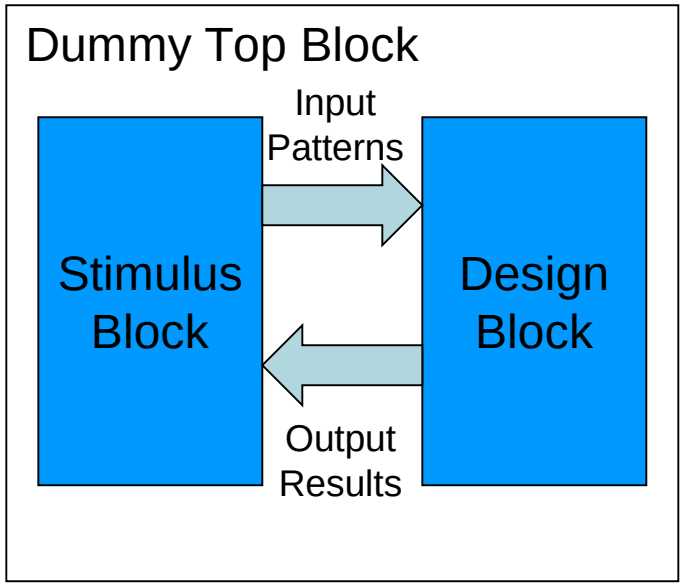
- Task: systematically verify the functionality of a model.
- Approaches: Simulation and/or formal verification
- Simulation:
  (1) detect syntax violations in source code
  (2) simulate behavior
  (3) monitor results



Components of a Simulation

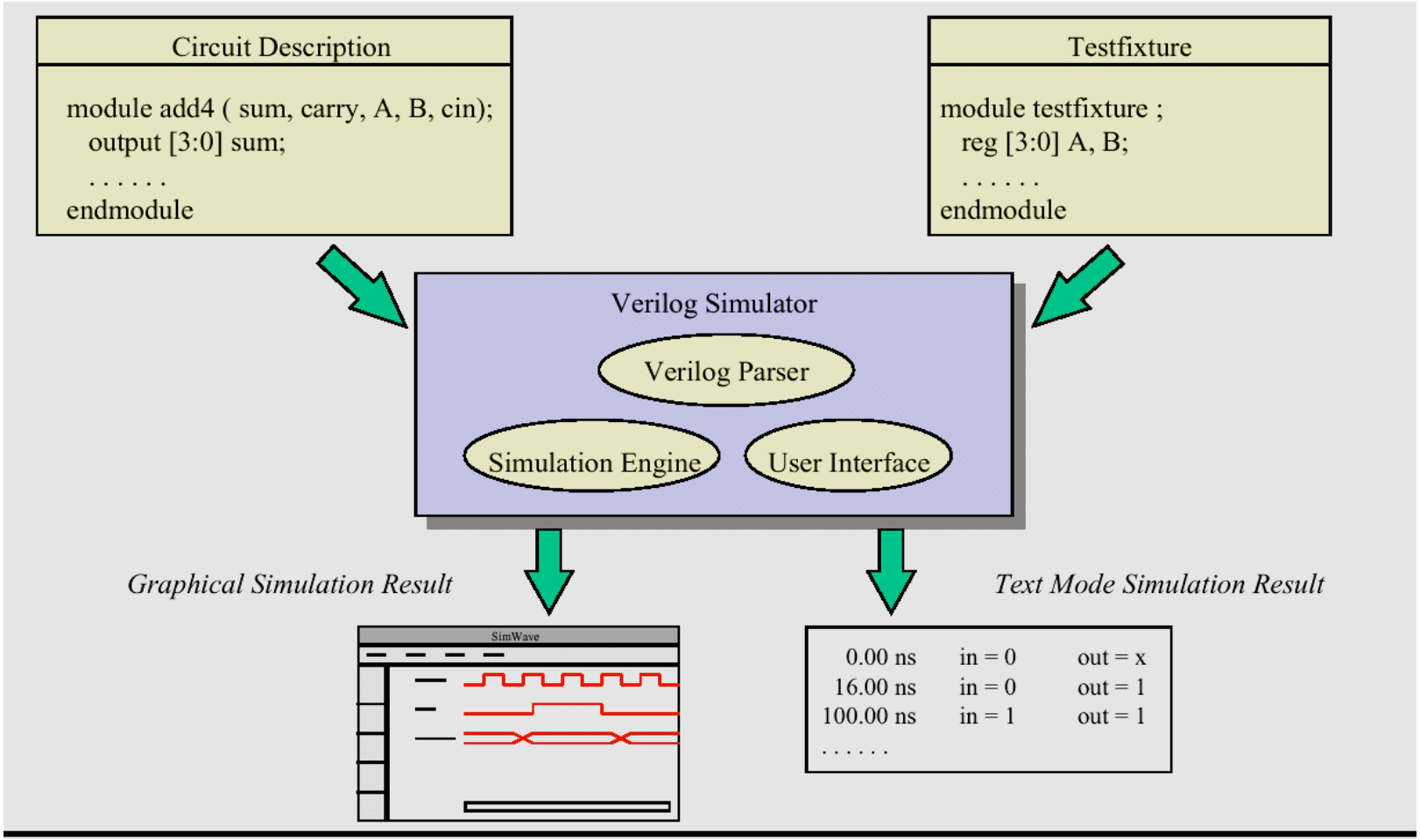


The output results are verified by console/waveform viewer



The output results are verified by testbench or stimulus block

Verilog Simulator



Testbench Template

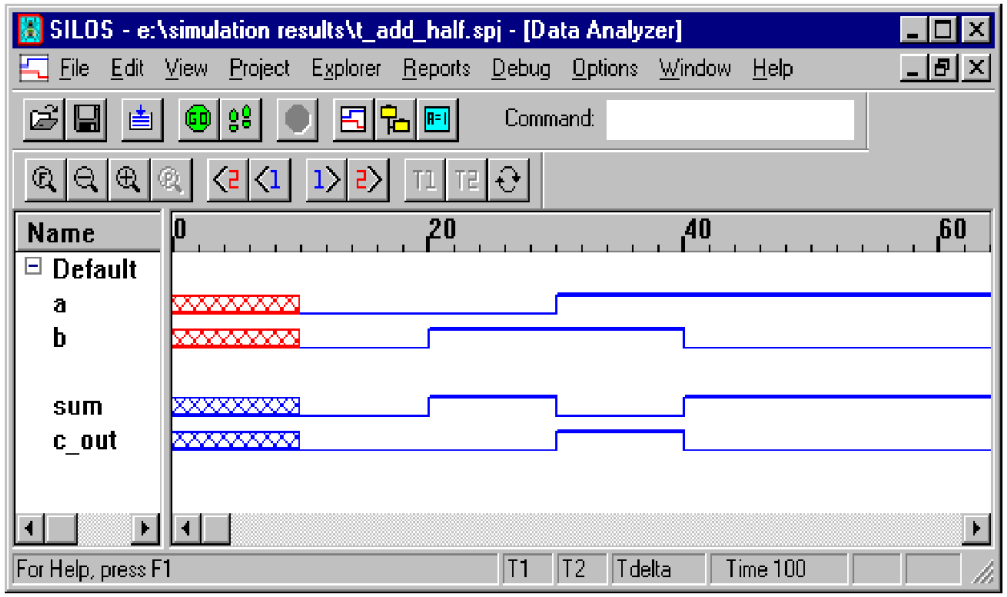
- Consider the following template as a guide for simple testbenches:

```
module t_DUTB_name ();      // substitute the name of the UUT
  reg …;                    // Declaration of register variables for primary inputs of the UUT
  wire …;                   // Declaration of primary outputs of the UUT
  parameter     time_out = // Provide a value

  UUT_name M1_instance_name ( UUT ports go here);

  initial $monitor (  );    // Specification of signals to be monitored and displayed as text

  initial #time_out $stop; // (Also $finish) Stopwatch to assure termination of simulation

  initial                   // Develop one or more behaviors for pattern generation and/or
                            // error detection
    begin
                            // Behavioral statements generating waveforms
                            // to the input ports, and comments documenting the test.
                            // Use the full repertoire of behavioral
                            // constructs for loops and conditionals.
    end
endmodule
```

Example: Testbench

```
`timescale 1ns/10ps

module t_Add_half();
  wire          sum, c_out;
  reg           a, b;                       // Variable for waveforms

  Add_half_0_delay M1 (sum, c_out, a, b); // UUT

  initial begin                             // Time Out
    #100 $finish;                           // Stopwatch
  end

  initial begin                             // Stimulus patterns
    #10 a = 0; b = 0;                       // Statements execute in sequence
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
  end
endmodule
```

Simulation Results



MODELING TIP

A Verilog simulator assigns an *initial* value of **x** to all variables.

Propagation Delay

- Gate propagation delay specifies the time between an input change and the resulting output change
- Transport delay describes the time-of-flight of a signal transition
- Verilog uses an inertial delay model for gates and transport delay for nets

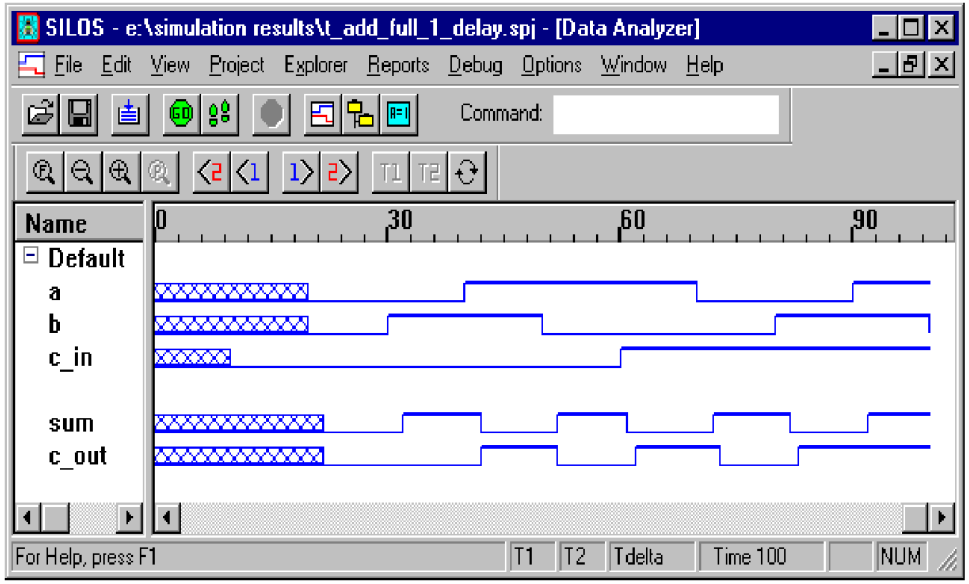
MODELING TIP

All primitives and nets have a default propagation delay of 0.

Example: Propagation Delay

- Unit-delay simulation reveals the chain of events

```
module Add_full (sum, c_out, a, b, c_in);
  output          sum, c_out;
  input           a, b, c_in;
  wire            w1, w2, w3;

  Add_half        M1 (w1, w2, a, b);
  Add_half        M2 (sum, w3, w1, c_in);
  or              #1 M3 (c_out, w2, w3);
endmodule

module Add_half (sum, c_out, a, b);
  output          sum, c_out;
  input           a, b;

  xor             #1 M1 (sum, a, b);
  and             #1 M2 (c_out, a, b);

endmodule
```



System Tasks and Function: $ (1/2)

- Displaying information
  - *$display*("ID of the port is %b", port_id);
    - ID of the port is 00101
- Monitoring information
  - *$monitor*($time, "Value of signals clk = %b rst = %b", clk, rst);

```
0 Value of signals clk = 0 rst = 1
5 Value of signals clk = 1 rst = 1
10 Value of signals clk = 0 rst = 0
```

- Stopping and finishing in a simulation
  - *$stop*; // provided to stop during a simulation
  - *$finish*; // terminates the simulator

System Tasks and Function: $ (2/2)

- Math functions
  - *$clog2*(<arg>);
    - the ceiling of the log base 2 of the argument

```
module ram_model (address, write, chip_select, data);
parameter ram_depth = 256;
localparam addr_width = $clog2(ram_depth);
input [addr_width - 1:0] address;
input write, chip_select;
```

- Probabilistic distribution functions
  - *$random*;
    - 32-bit random signed integer
- Conversion functions
  - *$rtoi*(<real_value>); // convert real value to integer
  - *$itor*(<integer>); // convert integer to real value

Compiler Directives: `

- ***`define***
  - `define RAM_SIZE 16
  - Defining a name and gives a constant value to it.
  - the identifier `RAM_SIZE will be replaced by 16
- ***`include***
  - `include adder.v
  - Including the entire contents of another Verilog source file.
- ***`timescale***
  - `timescale 1ns/10ps
  - `timescale *<reference_time_unit>* / *<time_precision>*
  - Setting the reference time unit and time precision of your simulation.

Example: Error Calculation in TB

- Calculate the percentage error of an L-dim 16-bit array

```
`define diff_abs(a, b) (a-b)>0 ? a-b : b-a
module test;
parameter array_len = L;
integer i;
real total_error = 0;
real error_percentage;

reg [15:0] mem [0:array_len-1];
real golden [0:array_len-1];

initial begin
    for (i=0; i<array_len; i=i+1) begin
        total_error = total_error
           + (`diff_abs($itor(mem[i]), golden[i]))/golden[i];
    end
    error_percentage = total_error*100/$itor(array_len);
    $display("error percentage: %f %%", error_percentage );
end
endmodule
```

Simulation Schemes

- There are 3 categories of simulation schemes
  - Time-based: Simulation on real time scale, used by SPICE simulators
  - **Event-based**: Simulation on events of signal transition, used by Verilog simulators. Note each event must occurs on discrete time specified by the testbench.
  - Cycle-based: Used by system/platform level verification, less used in cell-based IC designing.

Reference

- TSRI Verilog 上課講義
- IEEE Standard for Verilog Hardware Description Language
- Verilog 教學網站