Digital System Design

Fundamentals of Hardware Description Language

Lecturer: 王景平

Date: 2025/02/27

Based on: Ch.1-3 of the textbook
Review: Logic Design (Concepts of Propagation Delay) – related to HW0

Outline

- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

Cell-Based IC Design Flow

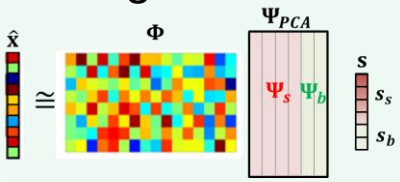
Algorithm

RTL

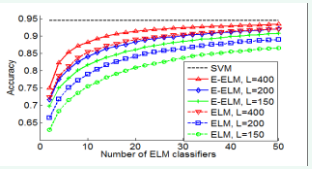
Synthesis

Layout

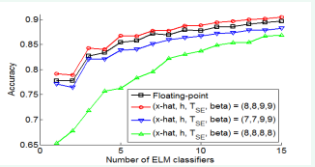
Algorithm



Floating-point Analysis



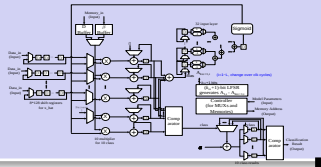
Fixed-point Analysis



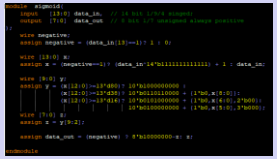
Spec

Parameter	Range	Descript ion
Dimension of input data (D_x)	128, 256, 512, 1024	MNIST Databas e
Number of hidden neurons (h_n)	1-256	

Architecture

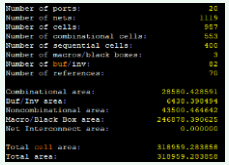


Verilog



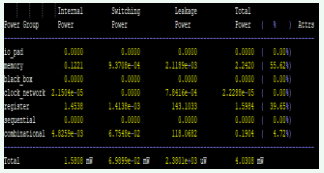
RTL Simulation

Gate-level Netlist

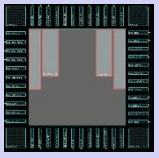


Gate-level Simulation

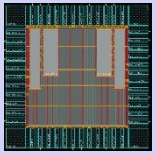
Power Optimization



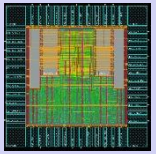
Floorplan



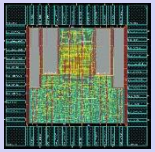
Powerplan



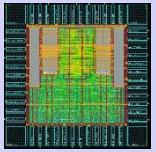
Placement



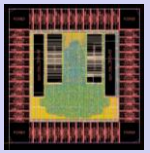
CTS



Route



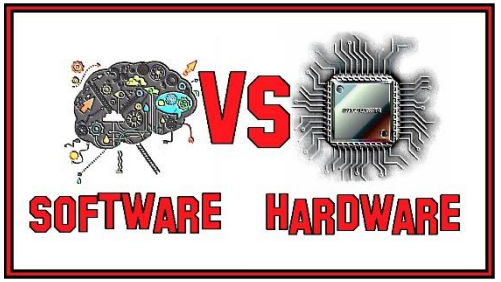
Post-layout Simulation



Hardware Description Language

- **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

- Difference between HDL and Software
  - Support concurrency
  - Support the simulation of the progress of time
  - Call functions vs. Instantiate hardware



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 on 1984 by Phil Moorby and Prabhu Goel in Automated Integrated Design System (renamed to Gateway Design Automation and bought by Cadence Design Systems)
- Open and Standardize (IEEE 1364-1995) on 1995 by Cadence because of the increasing success of VHDL (standard in 1987)
- Become popular and makes tremendous improvement on productivity
  - Syntax similar to C programming language, though the design philosophy differs greatly

History/Branch of Verilog

SystemVerilog

3.1
- test program blocks
- clocking domains
- mailboxes
- semaphores
- persistent events
- process control
- constrained random values
- direct C function calls

3.0
- assertions
- interfaces
- nested hierarchy
- unrestricted ports
- automatic port connect
- enhanced literals
- time values and units
- dynamic processes
- 2-state modeling
- packed arrays
- array assignments
- specialized procedures
- enhanced event control
- unique/priority case/if

from C / C++

- classes
- inheritance
- strings
- dynamic arrays
- associative arrays
- references
- int
- shortint
- longint
- shortreal
- double
- char
- void
- globals
- enum
- typedef
- structures
- unions
- casting
- const
- break
- continue
- return
- do–while
- ++ -- += -= *= /=
- >>= <<= >>>= <<<=
- &= |= ^= %=

Verilog-2001

- ANSI C style ports
- generate
- localparam
- constant functions
- standard file I/O
- $value$plusargs
- `ifndef `elsif `line
- @*
- (* attributes *)
- configurations
- memory part selects
- variable part select
- multi dimensional arrays
- signed types
- automatic
- ** (power operator)

Verilog-1995

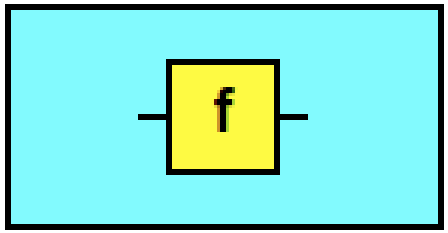
- modules
- parameters
- function/tasks
- always @
- assign
- $finish $fopen $fclose
- $display $write
- $monitor
- `define `ifdef `else
- `include `timescale
- initial
- disable
- events
- wait # @
- fork–join
- wire reg
- integer real
- time
- packed arrays
- 2D memory
- begin–end
- while
- for forever
- if–else
- repeat
- + = * /
- %
- >> <<

Outline

- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

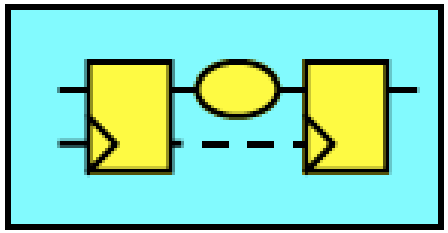
Levels of Modeling

Behavioral Level



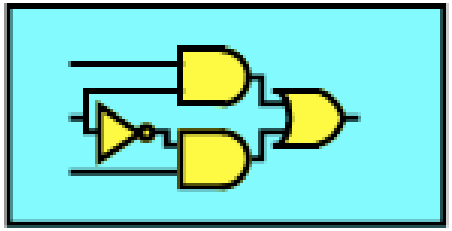

```
initial begin
    #(`CYCLE * `End_CYCLE );
    $display( "\n" );
    $display("-------------------------------------------------------\n");
    $display("Error!!! Something is wrong with your code ...!\n");
    $display("------------------------FAIL-------------------------\n");
    $display( "Terminated at: ", $time, " ns" );
    $display( "\n" );
    $stop;
end
```

Register Transfer Level (RTL)



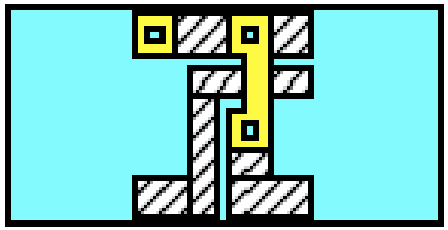
```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    assign  out=sel?in1:in2;
endmodule
```

Structural/Gate Level



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

    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
```

Transistor/Physical Level

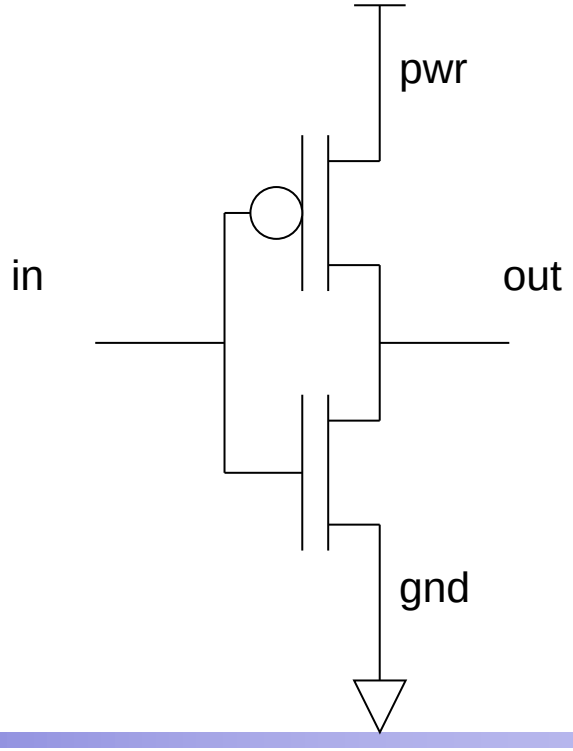


```
module inv(out, in);
// port declaration
  output  out;
  input   in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```

An Example - 1-bit Inverter in Transistor Level

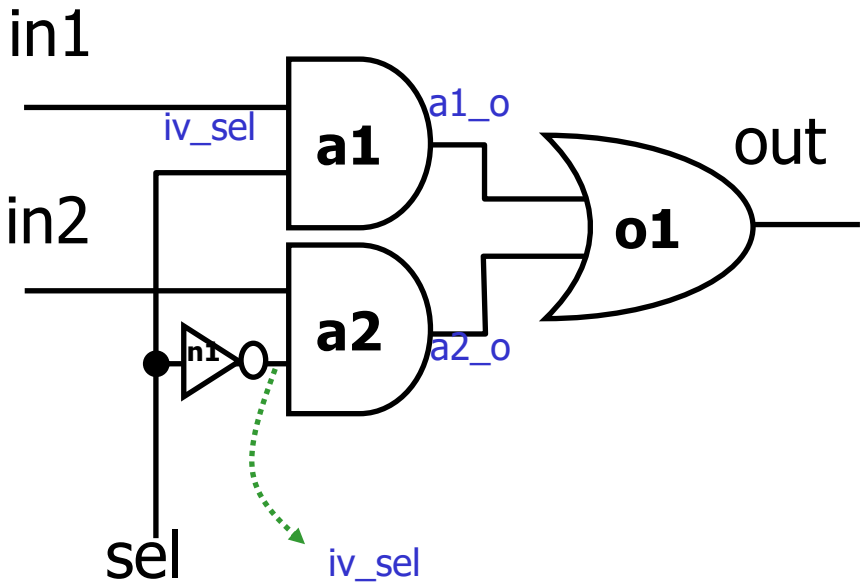
- Transistor level Verilog description
  - Using **NMOS and PMOS** to describe a circuit

```
module inv(out, in);
// port declaration
  output  out;
  input    in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```



An Example - 1-bit Multiplexer in Structure/Gate Level

- Structure/Gate Level
  - **Only netlist (gates and wires) in the code**
  - Synthesizable



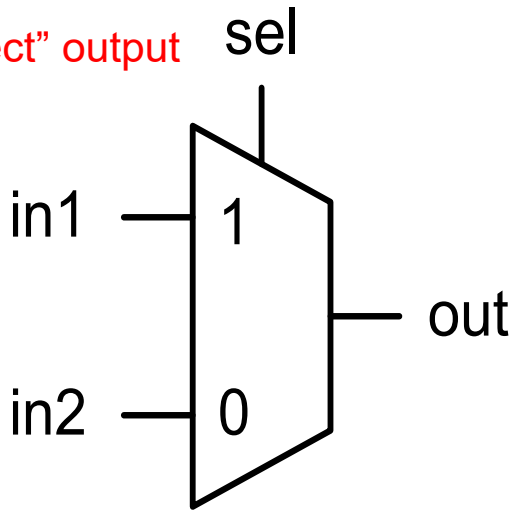

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    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
```

An Example - 1-bit Multiplexer in RTL Level

- RTL Level
  - Synthesizable

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

    assign  out=sel?in1:in2;
endmodule
```

To "select" output



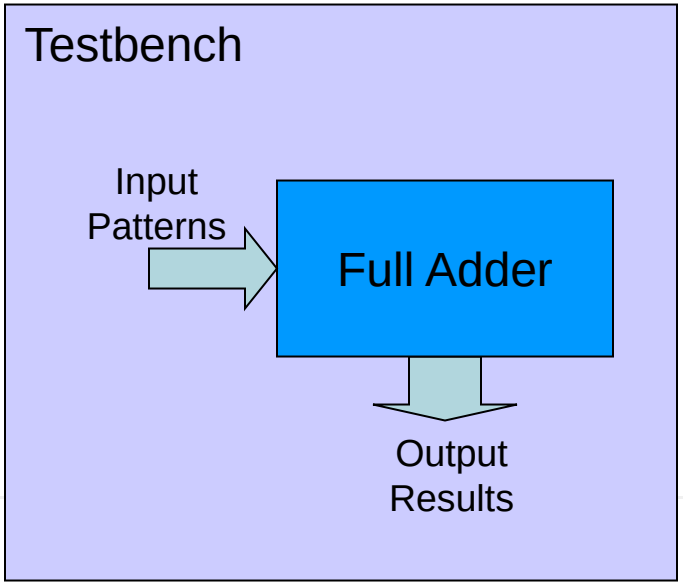
Continuous assignment

```
if (sel==0)
  out = in1;
else
  out = in2;
```

assign out = sel? in1 : in2 =

An Example – Testbench in Behavioral Level

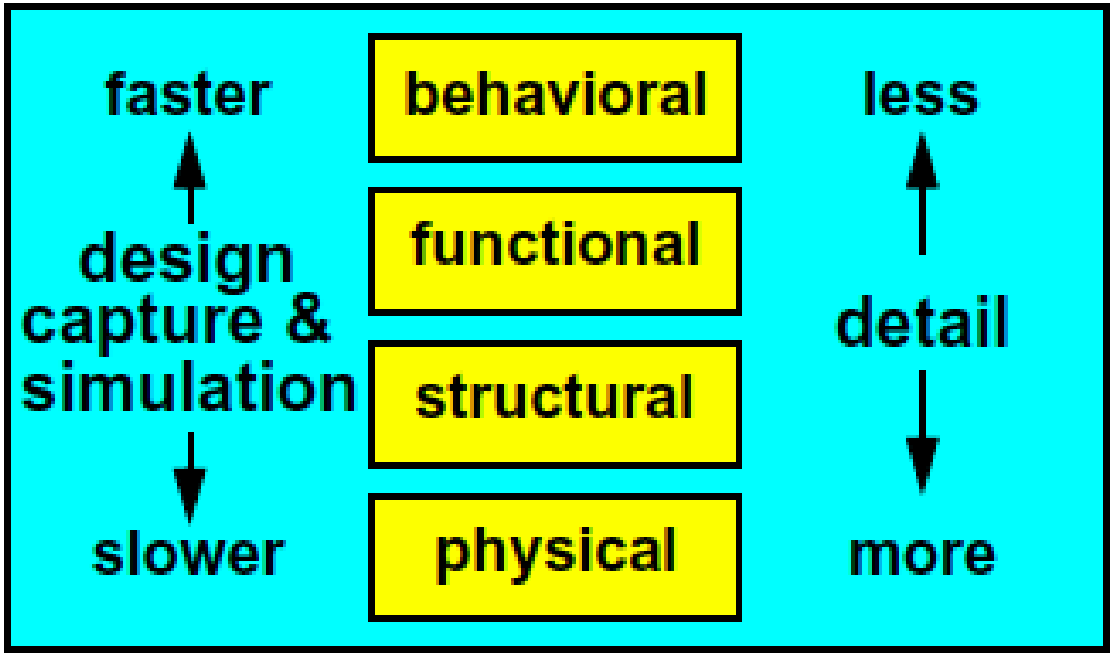
- Behavior Level
  - Testbench
  - Non-synthesizable




```
initial begin
    #(`CYCLE * `End_CYCLE );
    $display( "\n" );
    $display("------------------------------------------------------\n");
    $display("Error!!! Something is wrong with your code ...!\n");
    $display("------------------------FAIL----------------------\n");
    $display( "Terminated at: ", $time, " ns" );
    $display( "\n" );
    $stop;
end
```

Tradeoffs Among Modeling Levels

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



Outline

- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

Verilog Basic Components

- Verilog Basic Components
  - **module** instantiations
  - **wire or reg** declarations
  - **parameter** declarations
  - **gate** instantiation (primitive)
  - **continuous** assignments
  - **procedural** assignments (always block)
  - **function** definitions
  - **task** statements

Introduce in next week

Verilog Language Rules

- **Identifiers** (space-free sequence of symbols)
  - upper and lower case letters from the alphabet (case sensitive)
    - Verilog can distinguish them, but please don't use (APR)
  - digits (*0, 1, ..., 9*)
    - the first character must not be a digit, e.g. 0_and
  - underscore (_)
  - Max length of 1024 symbols
- 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 */

```
//============  Parameter ===============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;
```

```
/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
```

Data Types

- nets are further divided into several net types
  - **wire**, wand, wor, tri, triand, trior, supply0, supply1
- registers – variable 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

Wire Declaration

- **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**
  - Input, output, inout port declaration -- wire data type (default)
  - An un-driven wire defaults to a value of Z (high impedance).

Reg Declaration

- **reg**
  - Physical wires in a circuit
  - 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" ➔ always appear in "always" block

```
module test(a,b,c,d);
        input a,b;
        output c,d;
        reg d;
        assign c=a+b;
        always @(*) begin
                d=a-b;
        end
endmodule
```

wire type

reg type

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
    - No connections!
  - 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, BAD in pre-simulation

Vector

- wire and reg can be defined vector, default is 1bit
- vector is multi-bits element
- Format: [High#:Low#] or [Low#:High#]
  MSB LSB MSB LSB
- Using range specify part signals

```
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
```

Width Width

a[7:0] ⇔ a[0+:8] ⇔ a[7-:8]

LSB MSB

Array

- Arrays are allowed in Verilog for reg, integer, time, and vector register data types.
- Format: [High#:Low#] name [Low#:High#]

```
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
```

```
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
```

Memories

- In 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
```

Number 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

Number Representation

- Examples:
  - 6′b010_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 11_110
  - 16′Hx gives xxxxxxxxxxxxxxxx
  - 8′hz gives zzzzzzzz

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
wire [7:0] byte; **{{8{byte[7]}},byte};**	**Sign extension**

Signed Number System

- reg/wire can be declared signed/unsigned
  - reg signed [31:0] int_num;
  - wire signed [7:0] signed_byte;
- Signed/unsigned conversion
  - $signed(value)/$unsigned(value)
- Difference between signed/unsigned
  - Sign extension
  - Multiplication
  - …

Outline

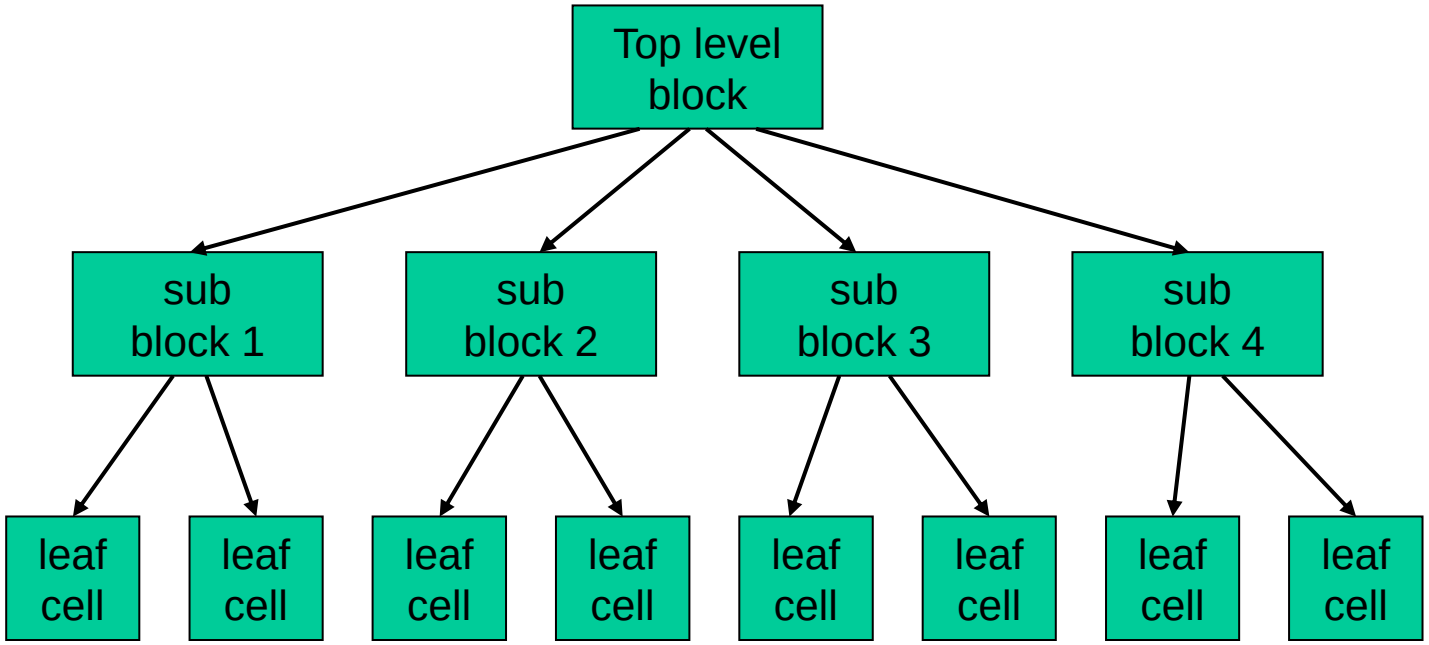
- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

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

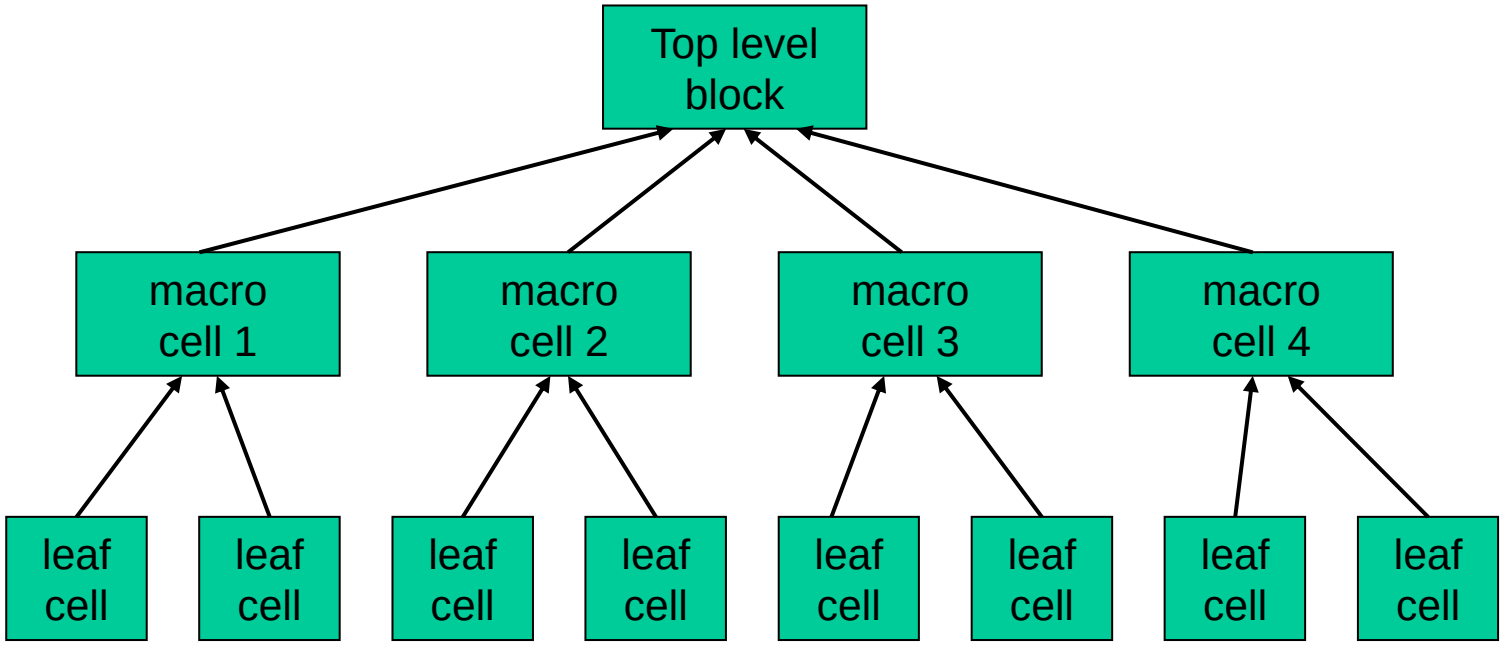
Top-down Design Methodology

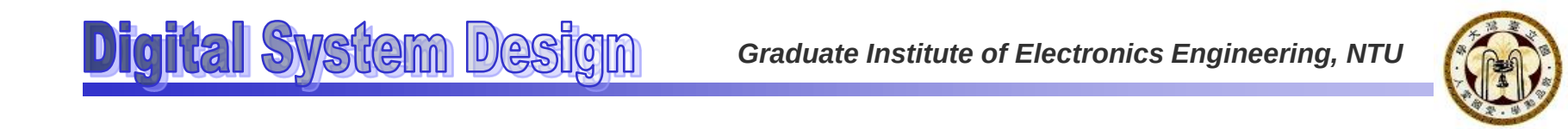
- We define the top-level block and identify the sub-blocks necessary to build the top-level block.
- We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.



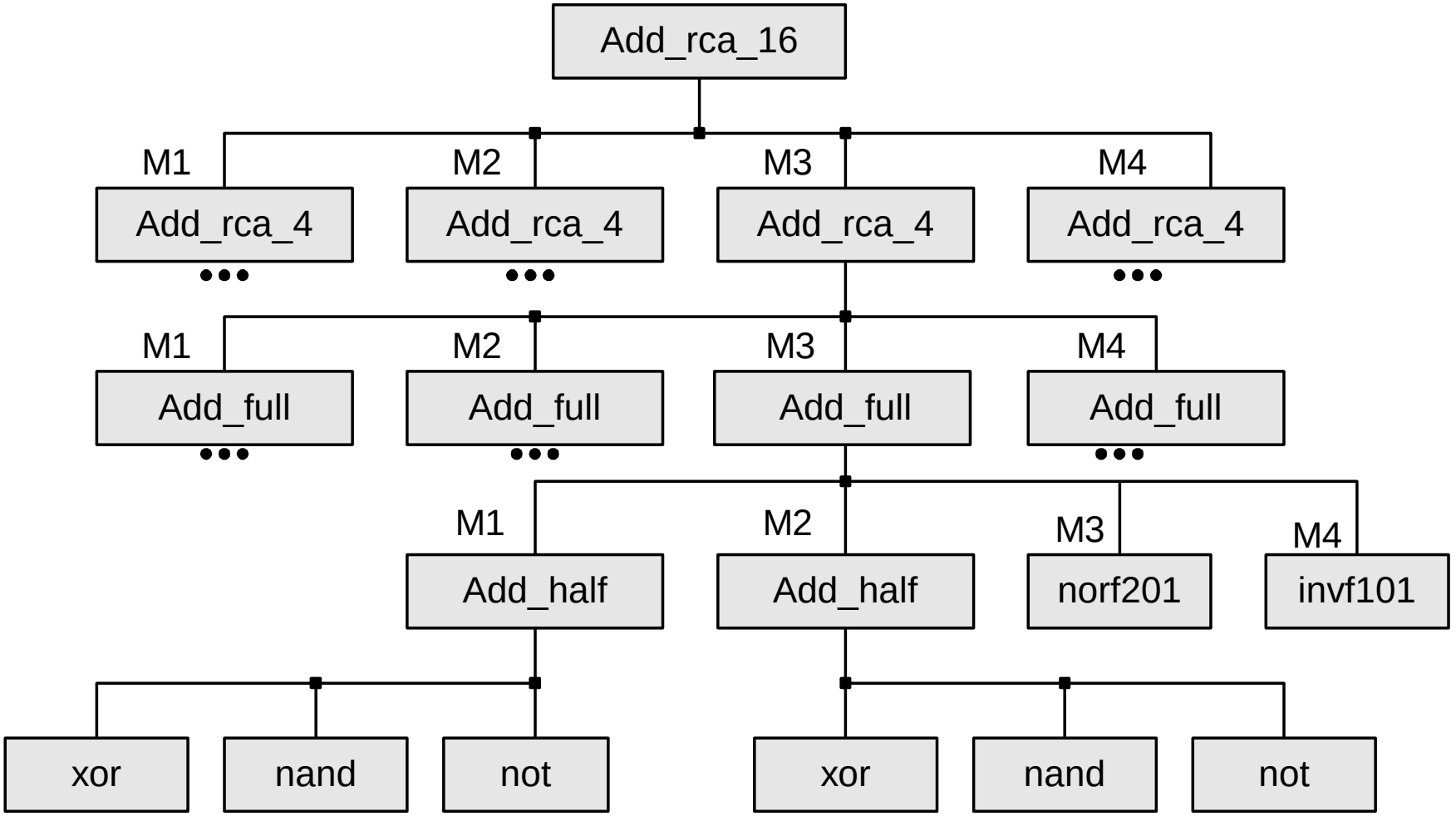
Bottom-up Design Methodology

- 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.



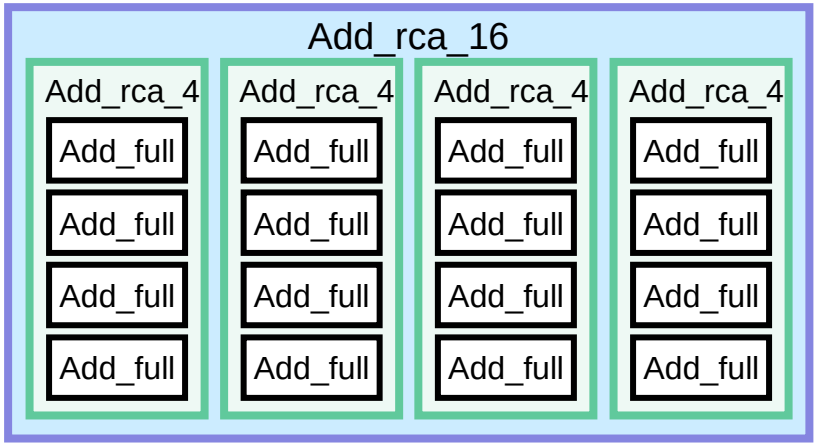


Example: 16-bit Adder



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**.
- Module
  - Basic building block in Verilog
  - Created by "declaration"
  - Used by "instantiation"
  - Interface is defined by ports
  - May contain instances of other modules
  - All modules run concurrently



Design Encapsulation - Module

- Encapsulate structural and functional details in a module

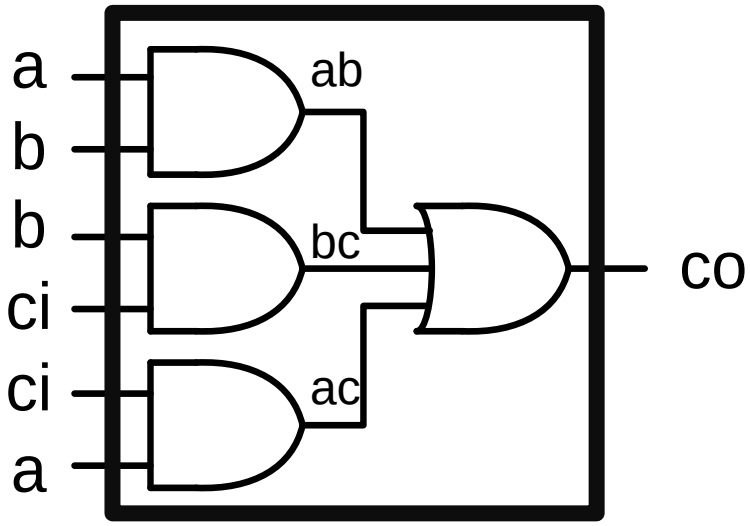
```
module <Module Name> (<PortName List>);

 // Structural part
      <List of Ports>
      <Lists of Nets and Registers>
      <SubModule List>  <SubModule Connections>

// Behavior part
      <Timing Control Statements>
            <Parameter/Value Assignments>
      <Stimuli>
      <System Task>
endmodule
```

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

    assign  out=in1 + in2;
endmodule
```

- Encapsulation makes the model available for instantiation in other modules

Case Study: Full Adder – Carry Out

- Module definition
- Port declaration
- Module logic




```
module FA_co(co, a, b, ci);

    output co;
    input a, b, ci;

    wire ab, bc, ac;
    and g0(ab, a, b);
    and g1(bc, b, ci);
    and g2(ac, ci, a);
    or  g3(co, ab, bc, ac);

endmodule
```

Port Declaration

- Port directions
  - Input port: **`input a;`**
  - Output port: **`output b;`**
  - Bidirectional port: **`inout c;`**
- Ports are declared as wires
  - **`input w;`** → w is a wire
  - Equivalent: **`input wire w;`**
- Output ports can be regs
  - **`output reg out;`**

```
module FA_co(co, a, b, ci);
    output co;
    input  a, b, ci;
    wire ab, bc, ac;
    and g0(ab, a, b);
    and g1(bc, b, ci);
    and g2(ac, ci, a);
    or  g3(co, ab, bc, ac);
endmodule
```

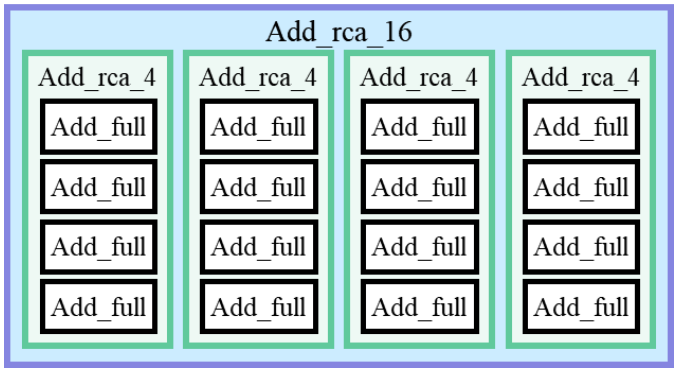
```
module FA_co(co, a, b, ci);
    output co;
    input  a, b, ci;
    …
endmodule
```

=

```
module FA_co(
    output co,
    input  a,
    input  b,
    input  ci);
    …
endmodule
```

Module Instantiation

- 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.



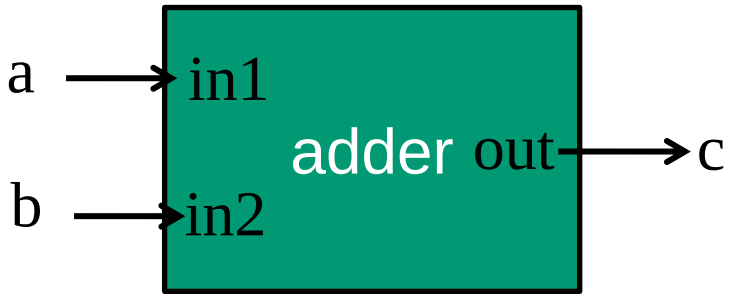

```
module adder(out,in1,in2);
    output  out;
    input   in1,in2,sel;

    assign  out=in1 + in2;
endmodule
```

```
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
```

instance name IO interface

Ports Connection

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

  assign out = in1 + in2;
endmodule
```



- Connect module ports by *name* **(Recommended!!!!!)**
  - Usage: .PortName (NetName)
  - `adder adder_0 ( .out(C) , .in1(A) , .in2(B) );`
- Connect module ports by order list
  - `adder adder_1 ( C , A , B );    // C = A + B`
- Not fully connected
  - `adder adder_2 ( .out(C) , .in1(A) , .in2() );`
- Output ports can only be connected to **wire**

Case Study: Adder Tree

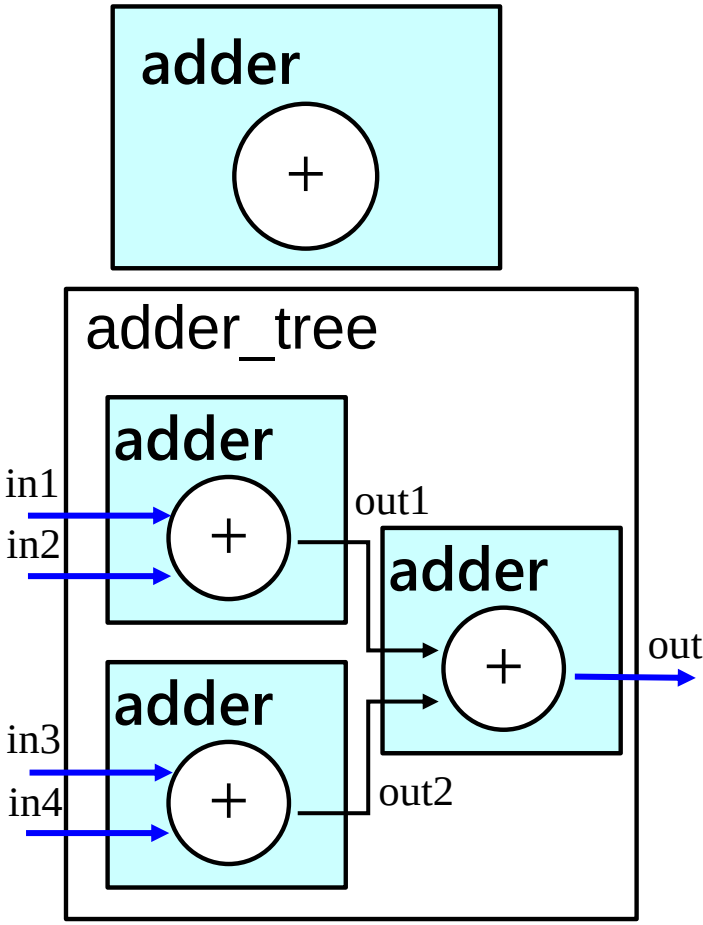
```
module adder(out, in1,in2);
        output out;
        input in1, in2;

        assign out = in1 + in2;
endmodule
```

instance example

```
module adder_tree(out, in1,in2,in3,in4);
        output out;
        input in1, in2, in3, in4;
        wire out1, out2;

        adder add_1 (.out(out1), .in1(in1), .in2(in2));
        adder add_2 (.out(out2), .in1(in3), .in2(in4));
        adder add_3 (.out(out), .in1(out1), .in2(out2));
```

instance name IO interface

```
endmodule
```



Parameter Declaration

- Parameters are not variables, they are constants. (hardware design view)
- Can be defined as a bit or a vector
- Typically parameters are used to specify conditions, states, width of vector, entry number of array, and delay

```
module var_mux #(
  parameter width = 2
)(out, i0, i1, sel);
  localparam flag = 1'b0;
  output [width-1:0] out;
  input  [width-1:0] v0, v1;
  input  sel;

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

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

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.U1.width = 4;
    top.U2.width = 3;
```

- Alternatively, parameters can be set on initialization

```
var_mux U0 #(.width(2)) (a_out, a0, a1, sel);
```

Primitives (HW0)

- 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,....);
    - Output port is defaulted in the first port
    - The number of input ports is adjustable (except buf/not)
    - and AND3 (output, in0, in1, in2)

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	

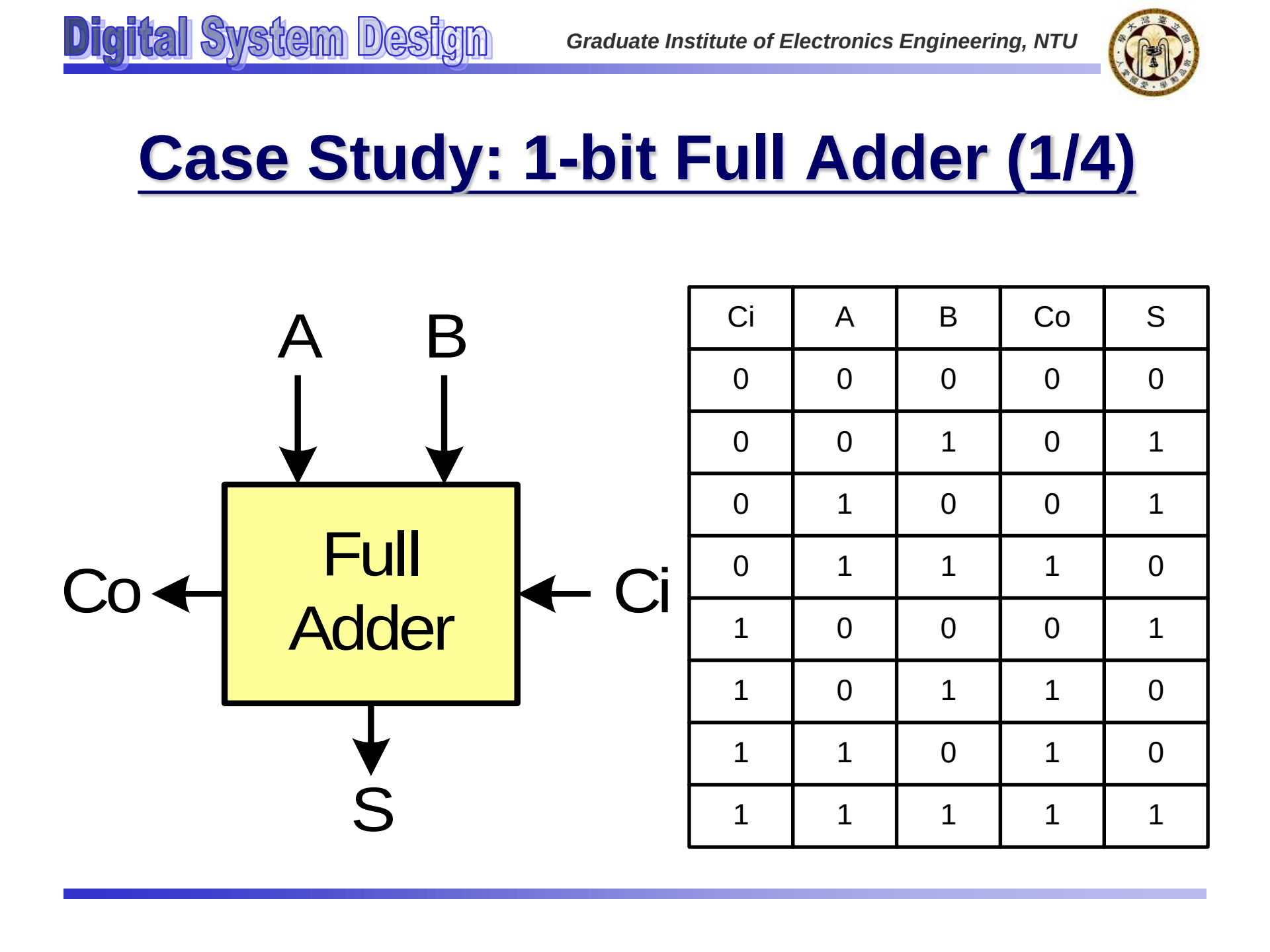
User-Defined Primitive(UDP)

- **1** output/multiple inputs
- Output should be the **first** port
- Ports **cannot** be vector
- Combinational: output is **wire**
- Sequential: output is **reg**

```
1   primitive dff(q, clk, d);
2       output reg  q;
3       input  wire clk, d;
4
5       table
6       //  clk     d : q : q'
7           (01)    0 : ? : 0;
8           (01)    1 : ? : 1;
9           (0?)    1 : 1 : 1;
10          (0?)    0 : 0 : 0;
11
12          (?0)    ? : ? : -;
13          ?     (??): ? : -;
14      endtable
15  endprimitive
```

Gate Level Modeling (HW1)

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

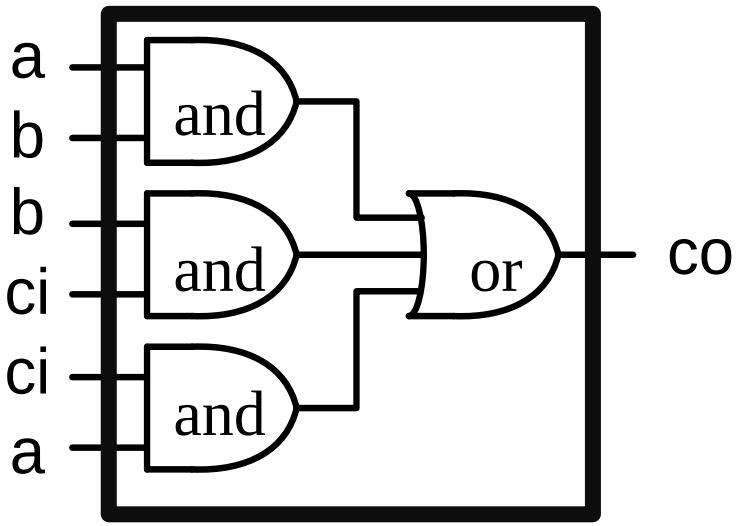


Case Study: 1-bit Full Adder (1/4)

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 (2/4)

- co = (a • b) + (b • ci) + (ci • a);

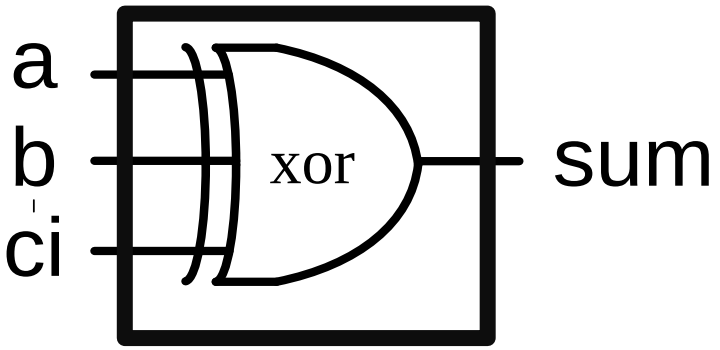



```
30
31 module FA_co ( co, a, b, ci );
32
33   input    a, b, ci;
34   output   co;
35   wire     ab, bc, ca;
36
37   and g0( ab, a, b );
38   and g1( bc, b, c );
39   and g2( ca, c, a );
40   or  g3( co, ab, bc, ca );
41
42 endmodule
43
```

instance name IO interface

Case Study: 1-bit Full Adder (3/4)

- 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
```

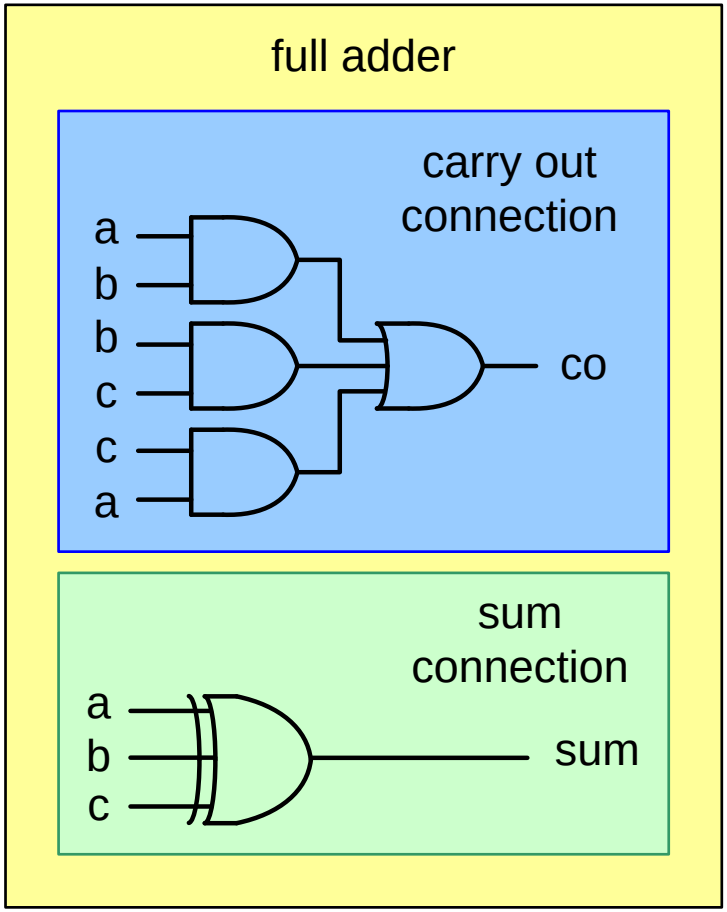
instance name IO interface

Case Study: 1-bit Full Adder (4/4)

- 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
```

instance name IO interface



Outline

- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

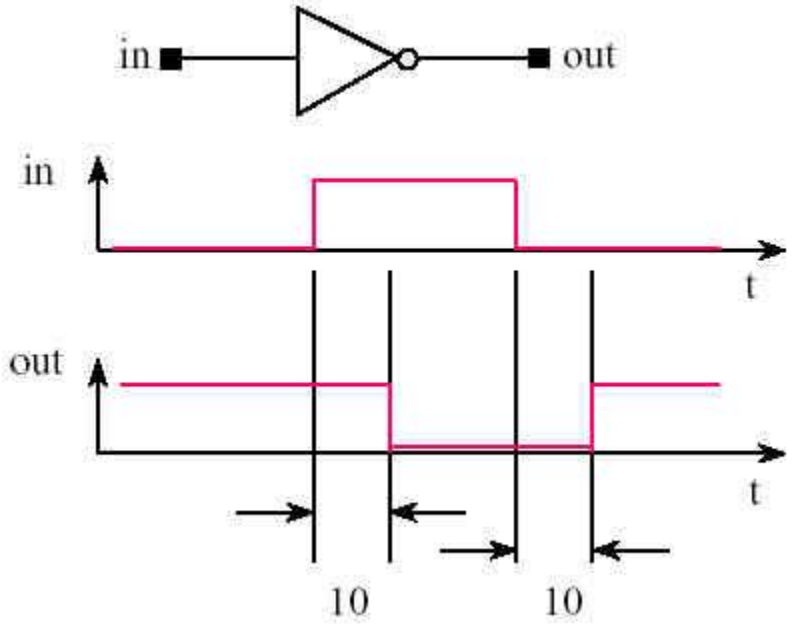
Timing and Delay
(for HW0 verification)

- 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 (HW0)

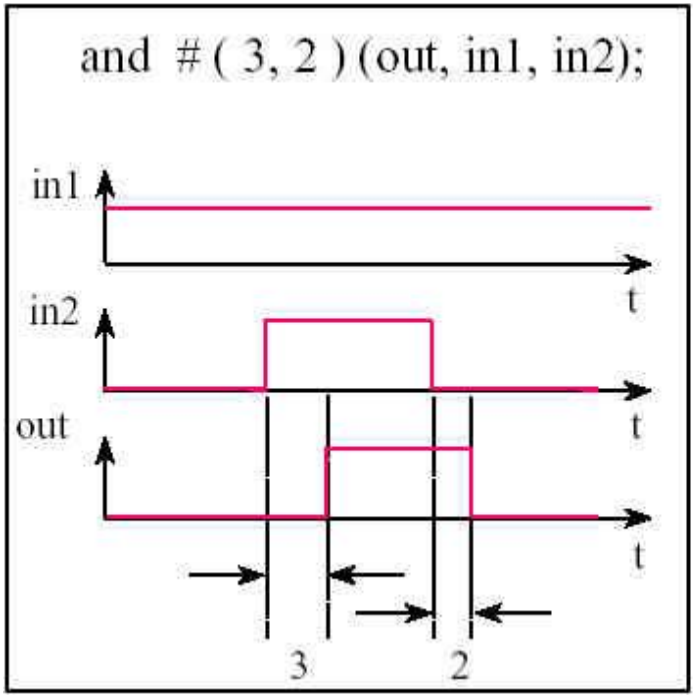
- Delay specification defines the propagation delay of that primitive gate.

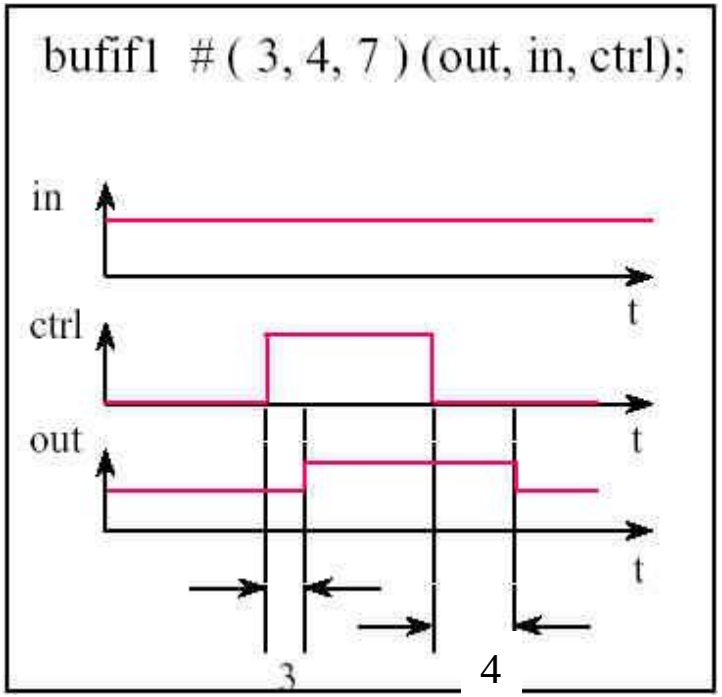
```
not  #10 (out,in);
```

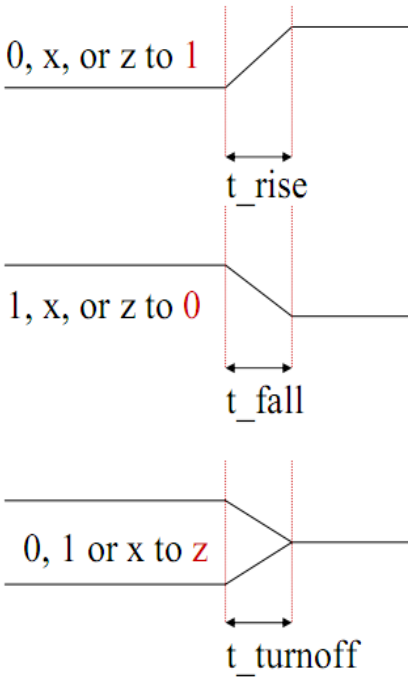


Delay Specification in Primitives

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





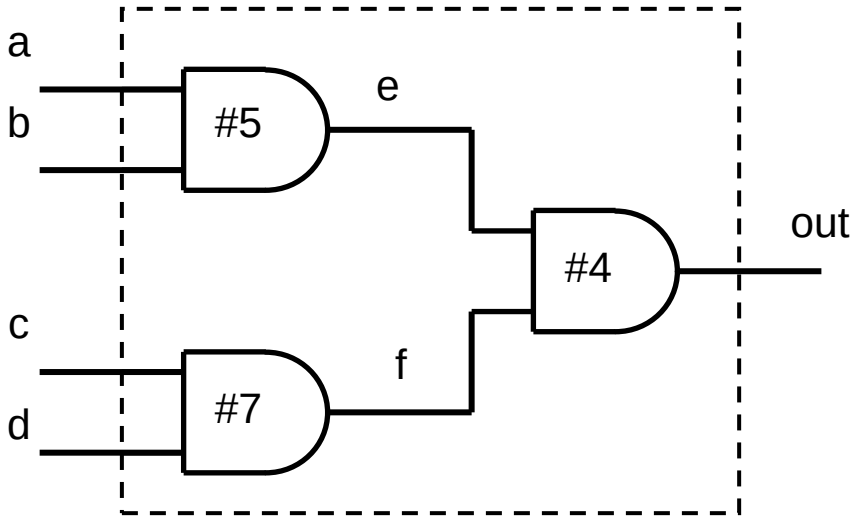


Delay Specification in Primitives

- 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 (HW0)

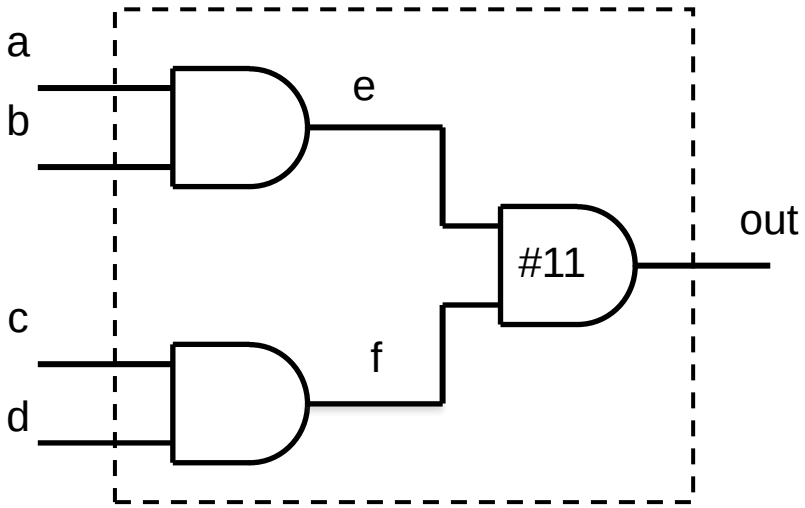
- **Distributed Delay**
  - Specified on a per element basic
  - Delay value are assigned to individual in the circuit




```
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

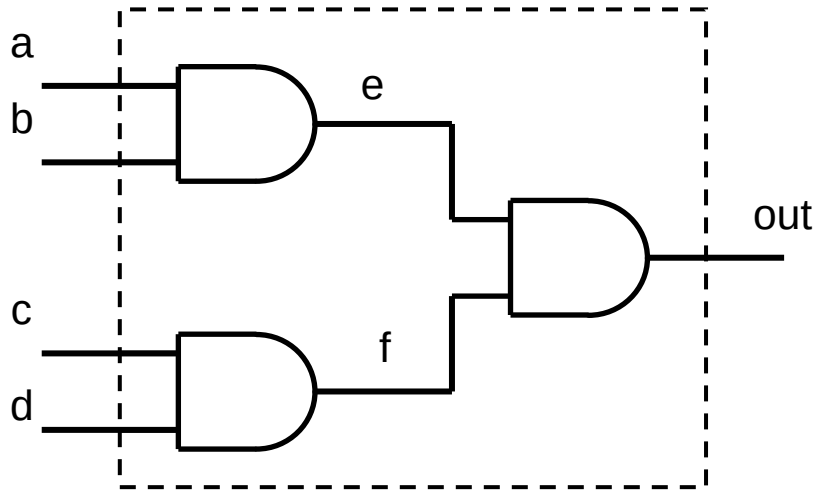
- **Lumped Delay**
  - They can be specified as a single delay on the output gate of the module
  - The cumulative delay of all paths is lumped at one location




```
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

```
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
```

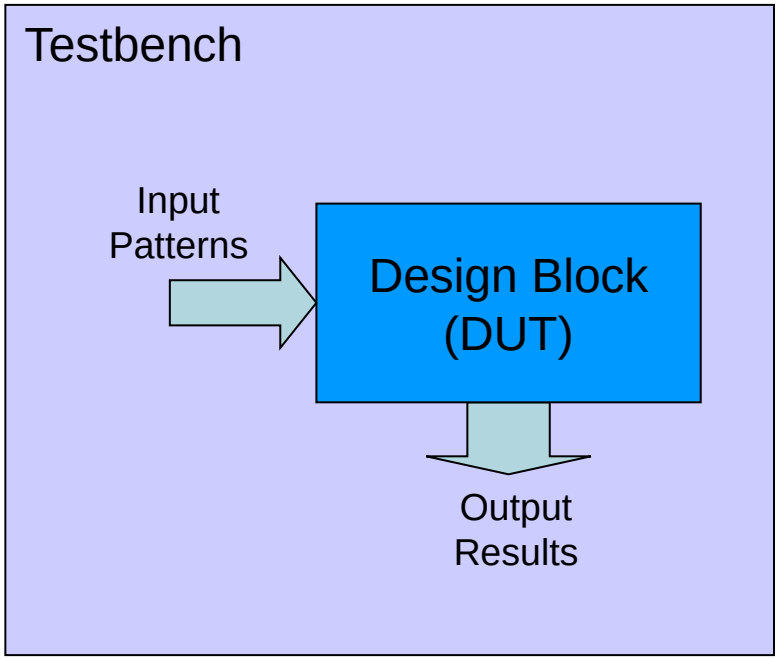
Outline

- Overview and History
- Levels of Modeling in Verilog
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
  - Transistor Level Modeling
- Language Elements in Verilog
- Hierarchical Design Methodology
- Timing & Delay
- Simulation and Verification

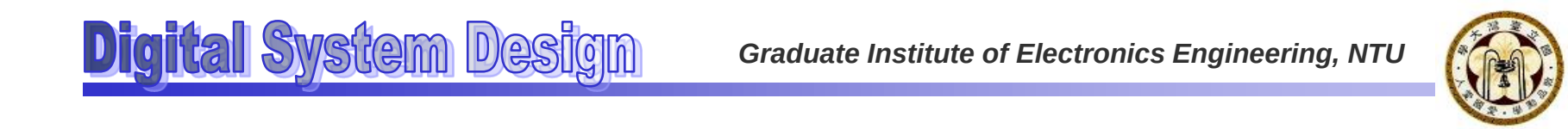
Verification Methodology

- 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

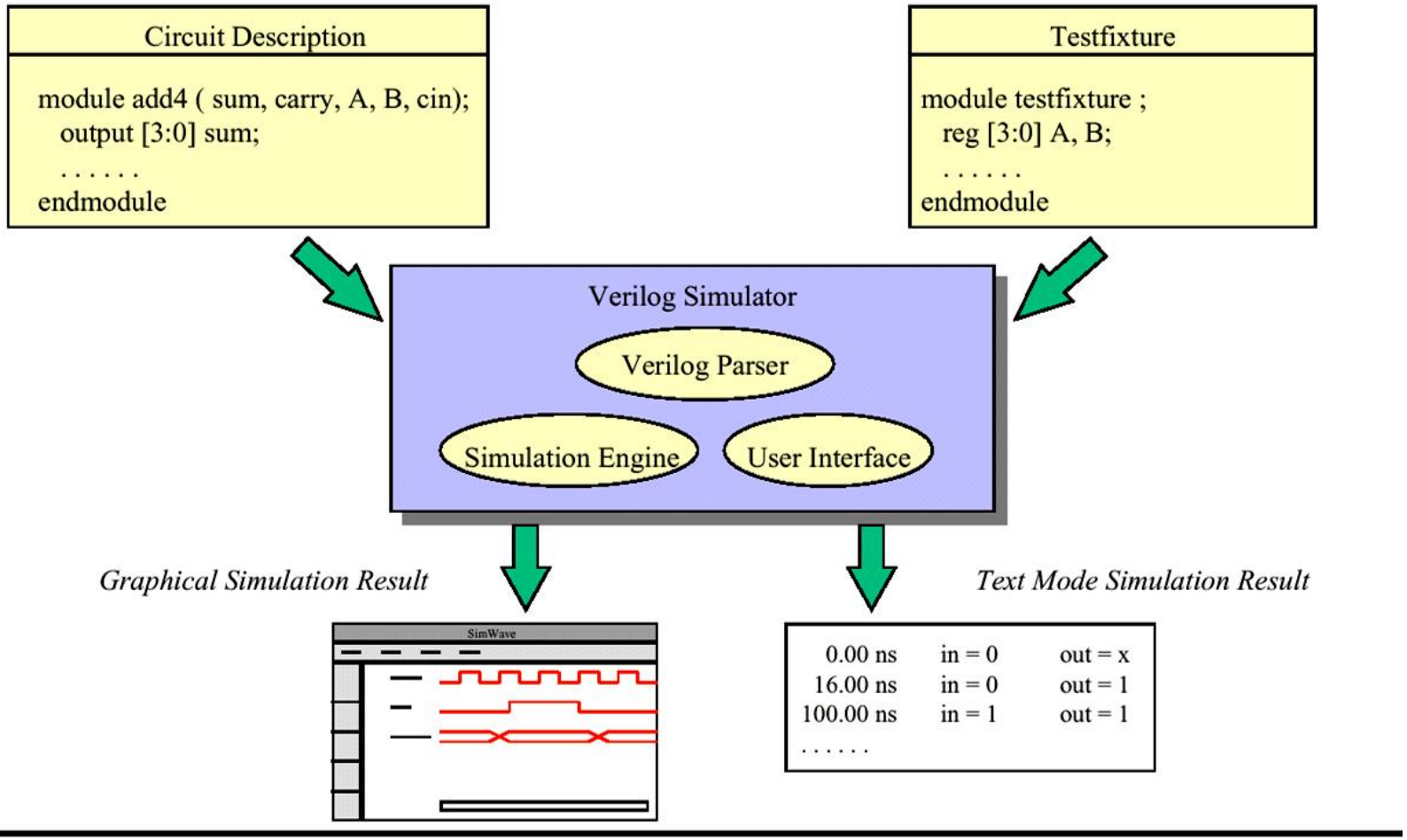
Components of a Simulation



The output results are verified by testbench



Verilog Simulator



Testbench Template

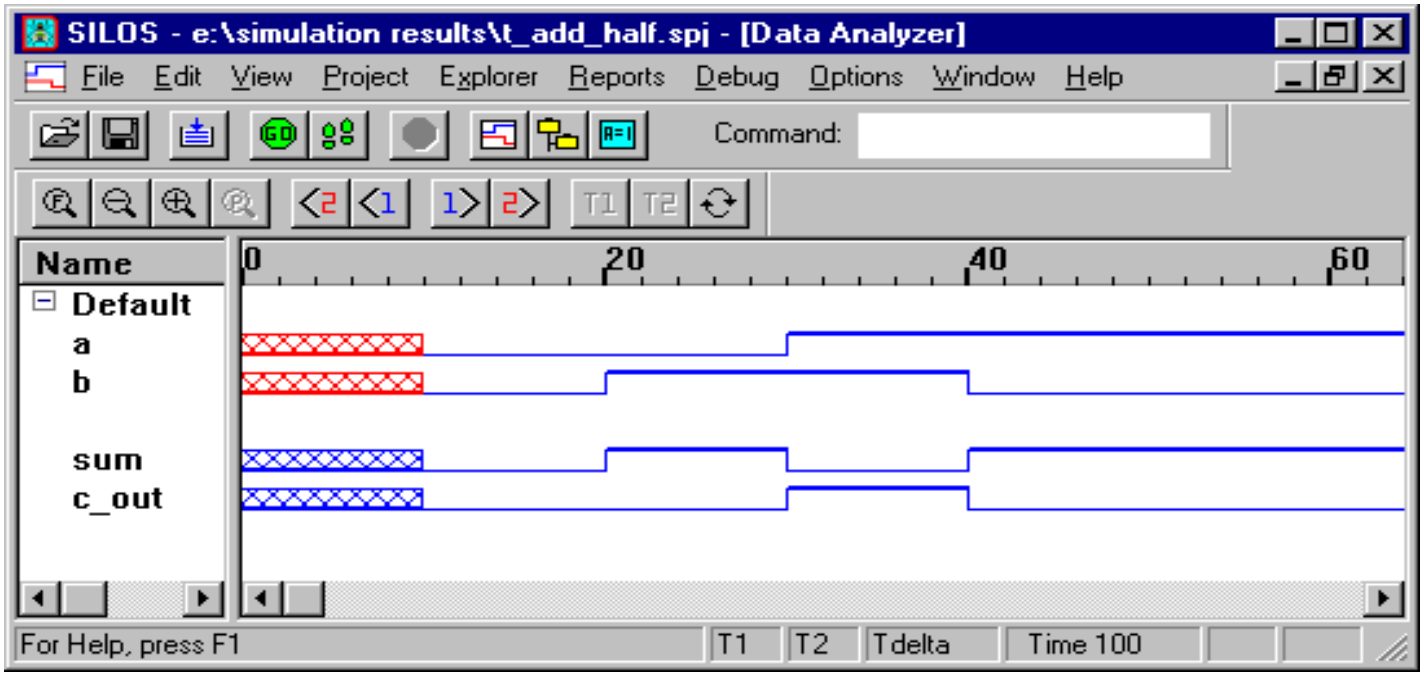
- Consider the following template as a guide for simple testbenches:

```
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

```
module t_Add_half;
  wire          sum, c_out;
  reg           a, b;          // Variable for stimulus waveforms

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

  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

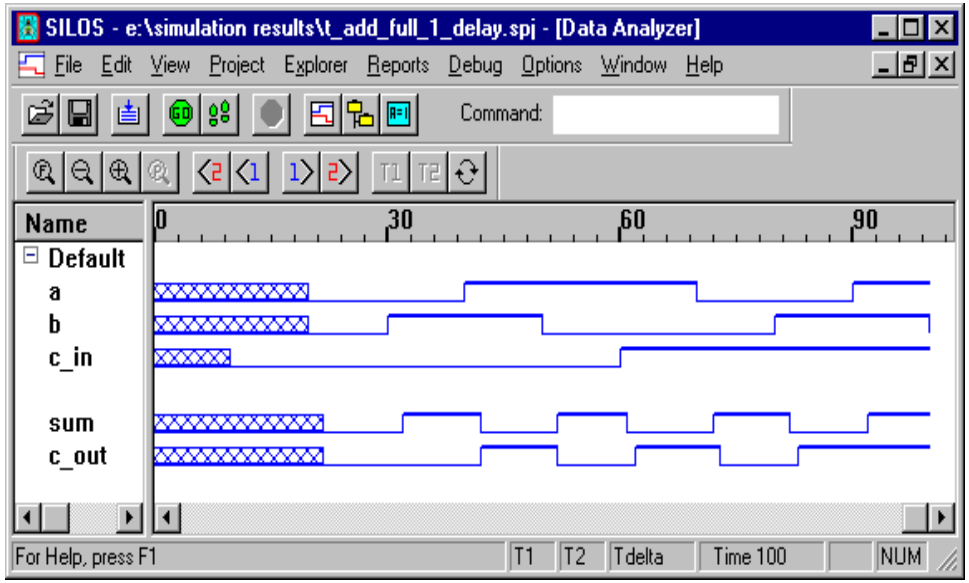
MODELING TIP

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

Example: Propagation Delay (HW0)

- 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
```



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 other verilog source file.
  - Can be replaced by specifying files to simulator in console
- *`timescale*
  - `timescale 1ns/10ps
  - `timescale <*reference_time_unit*> / <*time_precision*>
  - Setting the reference time unit and time precision of your simulation.

System Tasks

- 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
  - *$stop*; // stop during a simulation
  - *$finish*; // terminates the simulator

System Tasks

- Open file
  - *f_in = $fopen("filepath", "mode");*
    - Integer f_in: file pointer
    - Mode: w/wb/r/rb/a/ab/…
- Read from file
  - *ret = $fscanf(f_in, "fmt", …);*
    - Integer ret: Normal(1)/Error(0)/EOF(-1)
    - Similar to **fscanf** in C
- Write to file
  - *$fwrite(f_in, "fmt", …);*
    - Similar to **fprintf** in C
- Read memory data
  - $readmem[b/h]("filepath", mem, [start], [end]);
    - Load data to mem[start:end]

System Tasks

- Dump waveform
  - $fsdbDumpfile("waveform_filepath", size_limit(in MB));
    - Specify waveform file to dump
  - $fsdbDumpvars(depth, instance, "option");
    - depth: all signal(0)/signal in current scope(1)/n-1 levels below(n)
    - instance: instance name(e.g., DUT)
    - Option:
      - "+all"
      - "+mda"
      - …