Basic Logic Design with Verilog

TA: Chihhao Chao

chihhao@access.ee.ntu.edu.tw

Lecture note ver.1 by *Chen-han Tsai*
ver.2 revised by *Chih-hao Chao*

Outline

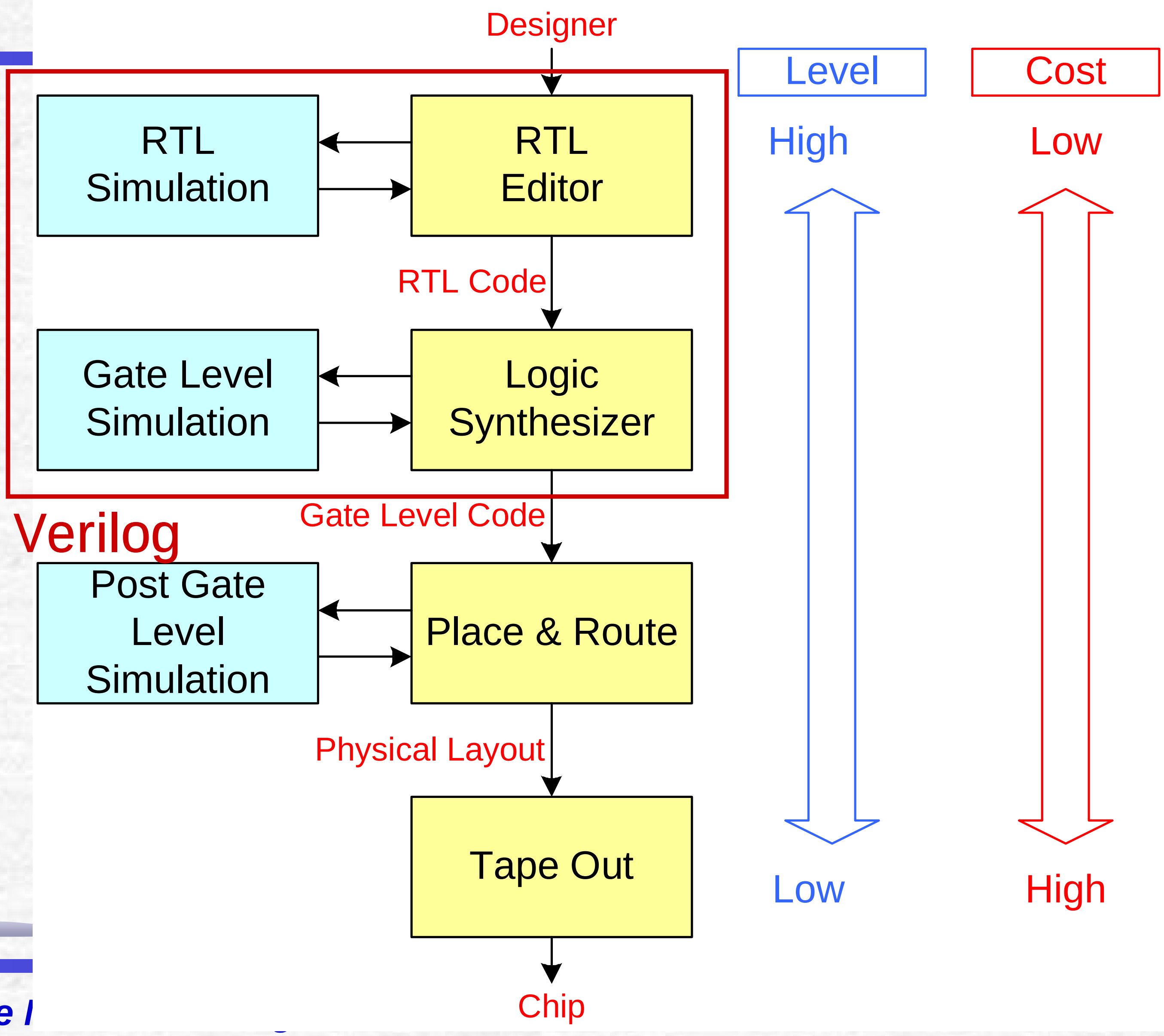
- Introduction to HDL/ Verilog
- Gate Level Modeling
- Behavioral Level Modeling
- Test bench
- Summary and Notes

Introduction to HDL/ Verilog

What is HDL/Verilog

- Why use HDL (Hardware Description Language)?
  - Design abstraction: HDL ⟷ layout by human
  - Hardware modeling
  - Reduce cost and time to design hardware
- Verilog is one of the most popular HDLs
  - VHDL (another popular HDL)
- Key features of Verilog
  - Supports various levels of abstraction
    - Behavior level
    - Register transfer level
    - Gate level
    - Switch level
  - Simulate design functions

Hardware Design Flow



Designer

RTL Simulation

RTL Editor

RTL Code

Gate Level Simulation

Logic Synthesizer

Verilog

Gate Level Code

Post Gate Level Simulation

Place & Route

Physical Layout

Tape Out

Chip

Level

High

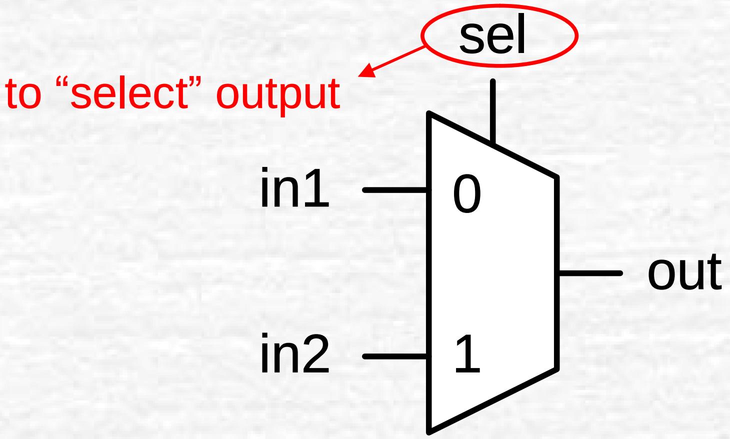
Low

Cost

Low

High

An Example
1-bit Multiplexer

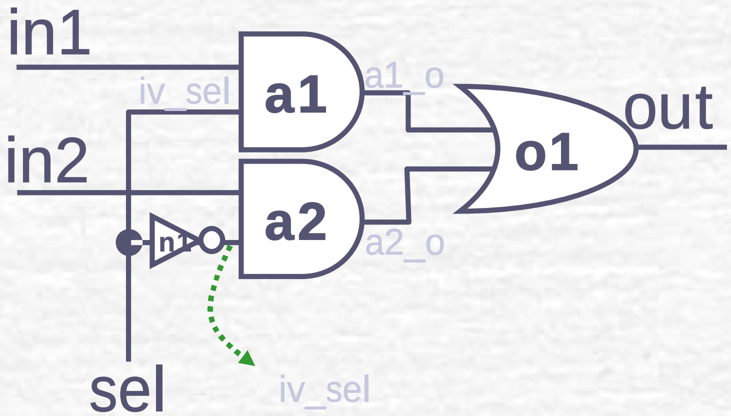

to “select” output

sel	in1	in2	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

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

$$out = (sel'\cdot in1) + (sel\cdot in2)$$

Gate Level Description



```
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
```

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

Behavioral Level/RTL Description

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

    always@(in1 or in2 or sel)
    begin
        if(sel) out=in1;
        else    out=in2;
    end
endmodule
```

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

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

always block

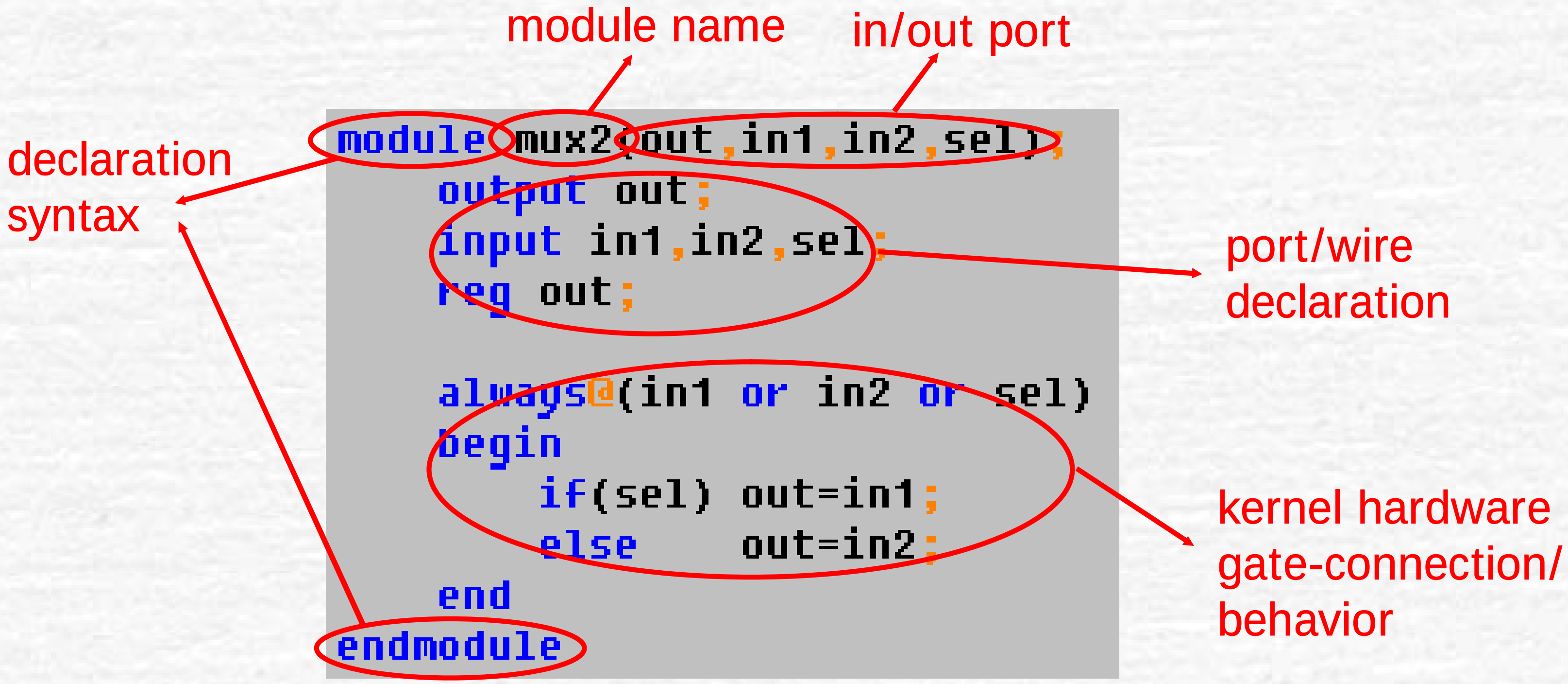
assign

RTL: you may see high level behavior in the code

Behavior: event-driven behavior description construct

Verilog HDL Syntax

A Simple Verilog Code



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

    always@(in1 or in2 or sel)
    begin
        if(sel) out=in1;
        else    out=in2;
    end
endmodule
```

module name

in/out port

declaration syntax

port/wire declaration

kernel hardware gate-connection/ behavior

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

Instances

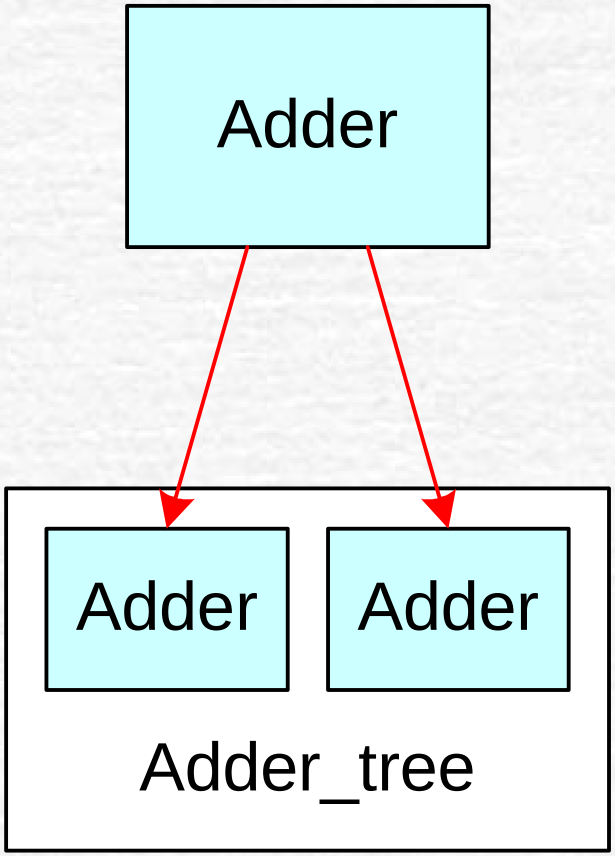
- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.

Module Instantiation

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

    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
```



Analogy: module ↔ class

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

Format	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
Hierachy	instance.sub_instance.member_signal	object.sub_object.member_data

Analogy: module ↔ class

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

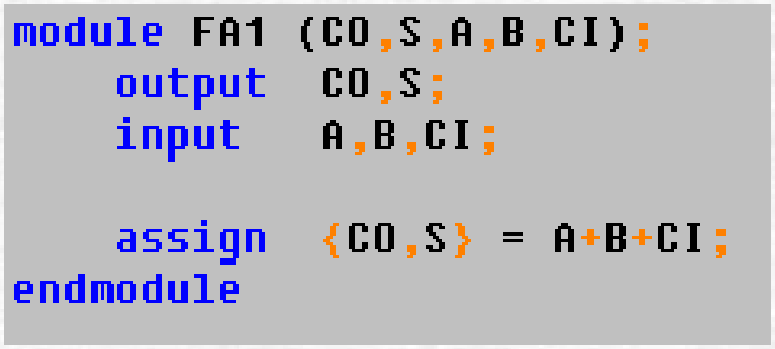
Model AND gate with C++

```
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

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

Port Connection

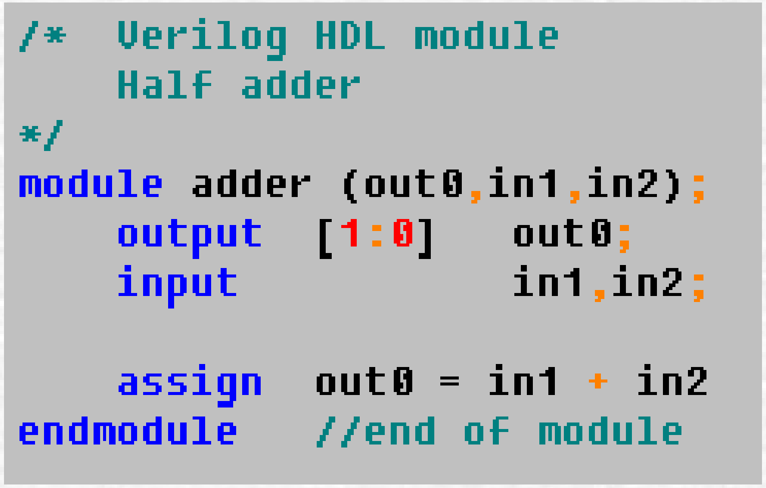


```
module FA1 (CO,S,A,B,CI);
    output  CO,S;
    input   A,B,CI;

    assign  {CO,S} = A+B+CI;
endmodule
```

- Connect module port by order list
  - FA1 fa1(c_o, sum, a, b, c_i);
- Not fully connected
  - FA1 fa3(c_o,, a, b, c_i);
- Connect module port by name *.PortName(NetName)*
  - FA1 fa2(.A(a), .B(b), .CO(c_o),.CI(c_i), .S(sum));
  - Recommended

Verilog Language Rule

- Case sensitive
- Identifiers:
  - Digits **0…9**
  - Underscore _
  - Upper and lower case letters from the alphabet



```
/*  Verilog HDL module
    Half adder
*/
module adder (out0,in1,in2);
    output  [1:0]   out0;
    input           in1,in2;

    assign  out0 = in1 + in2
endmodule   //end of module
```

- Terminate statement/declaration with semicolon “**;**”
- **Comments:**
  - Single line: // it’s a single line comment example
  - Multi-line: /* when the comment exceeds single line,
    multiline comment is necessary*/

Register and Net

- Registers
  - Keyword : **reg, integer, time, real**
  - Event-driven modeling
  - Storage element (modeling sequential circuit)
  - Assignment in “always” block
- Nets
  - Keyword : **wire, wand, wor, tri**
    triand, trior, supply0, supply1
  - Doesn’t store value, just a connection
  - input, output, inout are default “wire”
  - Can’t appear in “always” block assignment

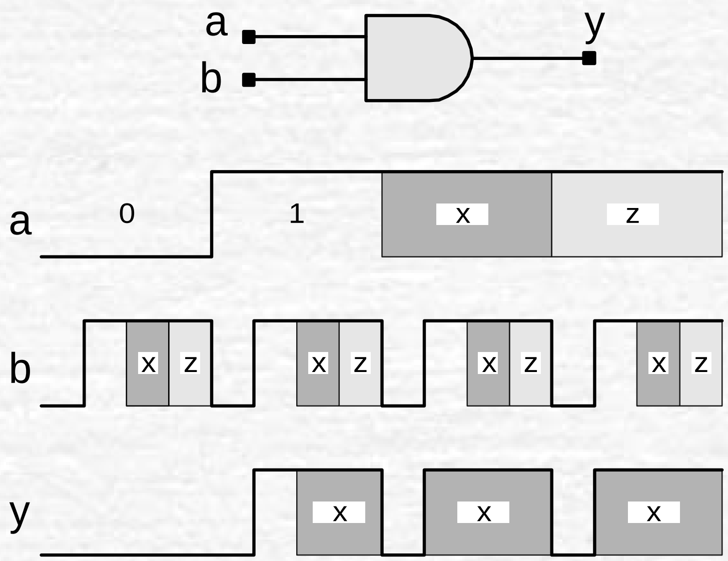
Four-valued Logic

- Verilog's nets and registers hold four-valued data
  - 0 represent a logic zero or false condition
  - 1 represent a logic zero or false 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
      - Output of a gate with z inputs

Logic System

- **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 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



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 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
    - x, z is automatically extented

Number Representation

- Examples:

6′b010_111	gives 010111
8′b0110	gives 00000110
4′bx01	gives xx01
16′H3AB	gives 0000001110101011
24	gives 0...0011000
5′O36	gives 11110
16′Hx	gives xxxxxxxxxxxxxxxx
8′hz	gives zzzzzzzz

Value and Number Expressions : Examples

```
659          // unsized decimal
‘h 837ff     // unsized hexadecimal
‘o7460       // unsized octal
4af          // illegal syntax
4’b1001      // 4-bit binary
5’D 3        // 5-bit decimal
3’b01x       // 3-bit number with
                unknown LSB
12’hx        // 12-bit unknown
8’d -6       // illegal syntax
-8’d 6       // phrase as - (8’d6)
```

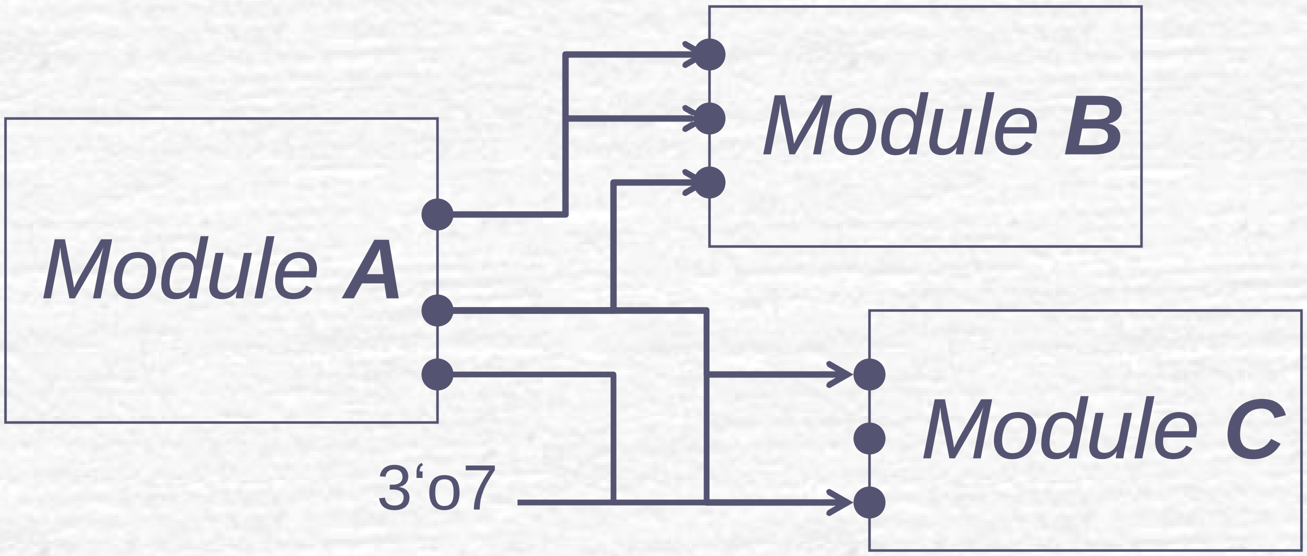
// underline usage

```
27_195_000
16’b0001_0101_0001_1111
32’h12ab_f001
```

// X and Z is sign-extended

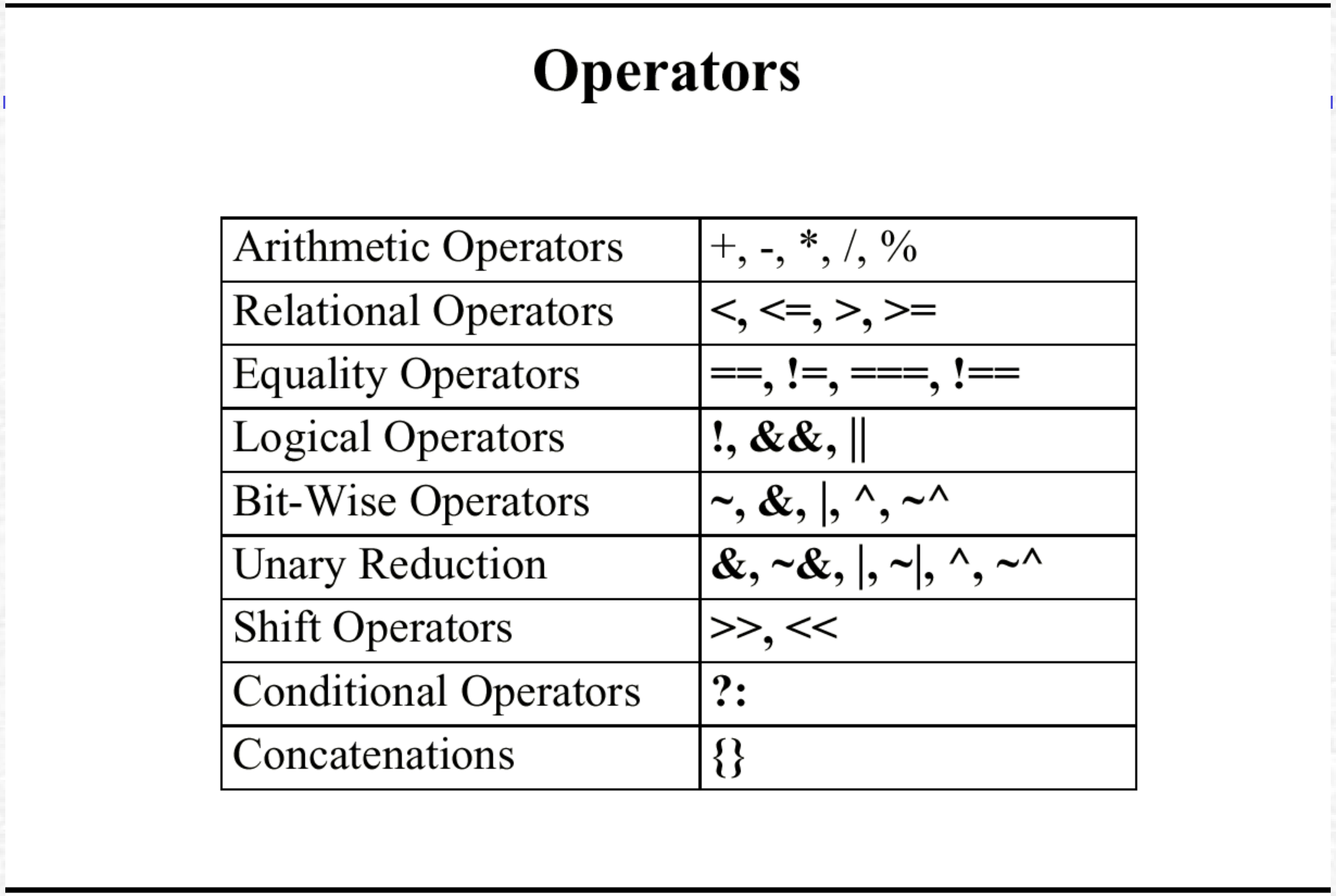
```
reg [11:0] a;
initial
begin
    a = ‘hx;        // yields xxx
    a = ‘h3x;       // yields 03x
    a = ‘h0x;       // yields 00x
end
```

Net Concatenations : An Easy Way to Group Nets



Representations	Meanings
{b[3:0],c[2:0]}	{b[3] ,b[2] ,b[1] ,b[0], c[2] ,c[1] ,c[0]}
{a,b[3:0],w,3’b101}	{a,b[3] ,b[2] ,b[1] ,b[0],w,1’b1,1’b0,1’b1}
{4{w}}	{w,w,w,w}
{b,{3{a,b}}}	{b,a,b,a,b,a,b}

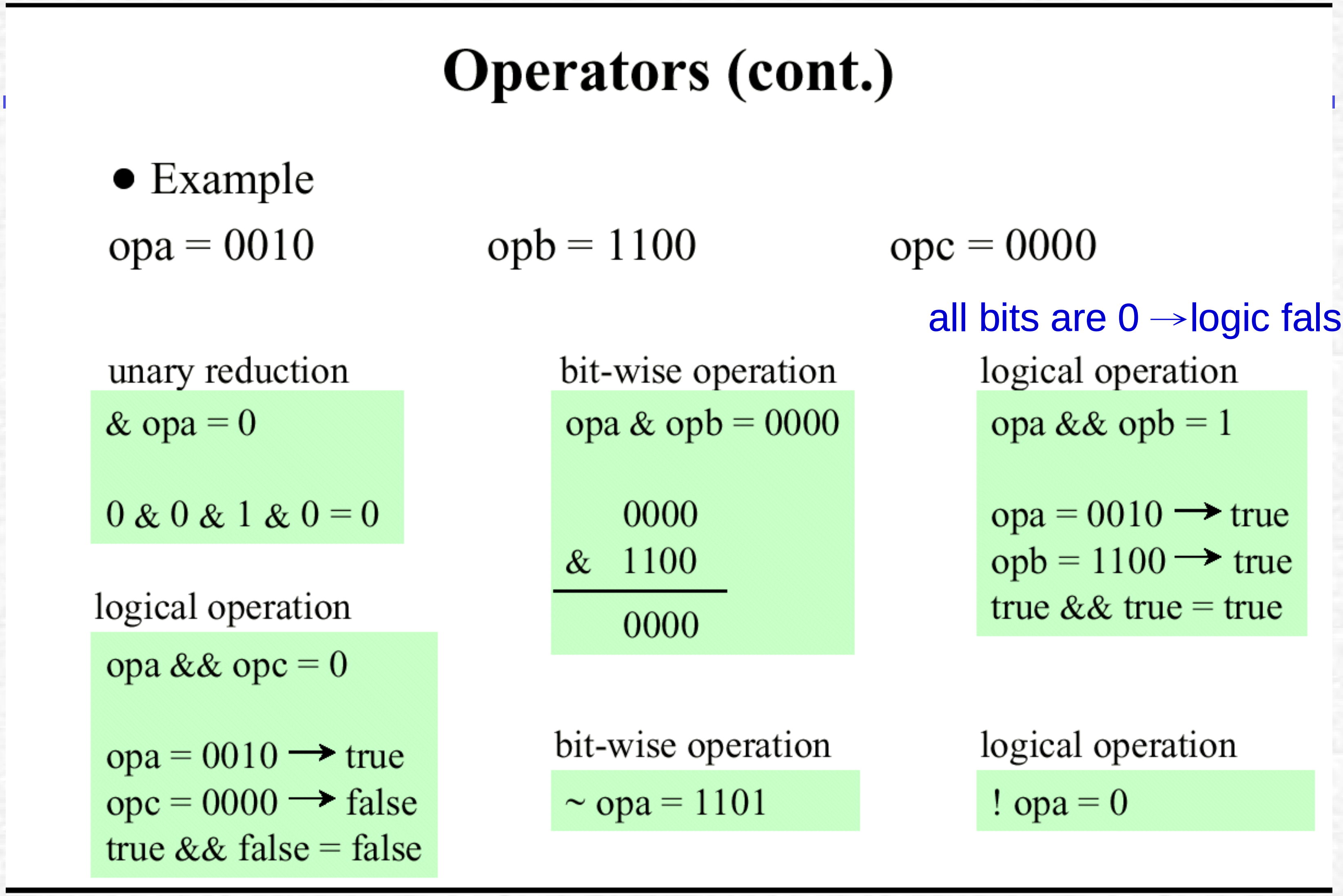
(excerpts from CIC training course： Verilog_9807.pdf)



Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Equality Operators	==, !=, ===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^, ~^
Unary Reduction	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	?:
Concatenations	{}

(excerpts from CIC training course： Verilog_9807.pdf)



Operators (cont.)

- Example

opa = 0010

opb = 1100

opc = 0000

all bits are 0 →logic false

unary reduction

& opa = 0

0 & 0 & 1 & 0 = 0

logical operation

opa && opc = 0

opa = 0010 → true

opc = 0000 → false

true && false = false

bit-wise operation

opa & opb = 0000

```
  0000
& 1100
------
  0000
```

bit-wise operation

~ opa = 1101

logical operation

opa && opb = 1

opa = 0010 → true

opb = 1100 → true

true && true = true

logical operation

! opa = 0

Compiler Directives

- *`define*
  - `define RAM_SIZE 16
  - Defining a name and gives a constant value to it.
- *`include*
  - `include adder.v
  - Including the entire contents of other verilog source file.
- *`timescale*
  - `timescale 100ns/1ns
  - Setting the reference time unit and time precision of your simulation.

System Tasks

- *$monitor*
  - $monitor ($time,"%d %d %d",address,sinout,cosout);
  - Displays the values of the argument list whenever any of the arguments change except $time.
- *$display*
  - $display ("%d %d %d",address,sinout,cosout);
  - Prints out the current values of the signals in the argument list
- *$finish*
  - $finish
  - Terminate the simulation

Gate Level Modeling

- Gate Level Modeling
- Case Study

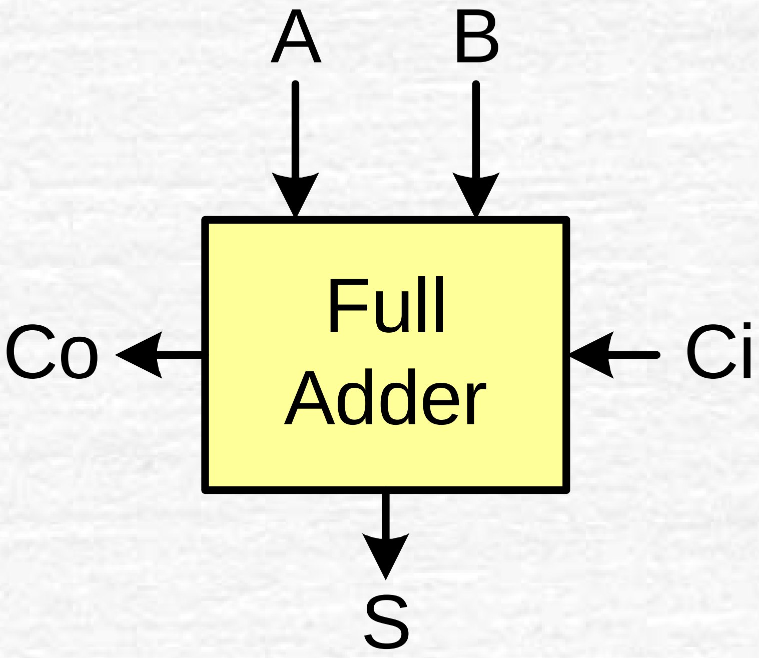
Gate Level Modeling

- Steps
  - Develope the boolean function of output
  - Draw the circuit with logic gates/primitives
  - Connect gates/primitives with net (usually wire)
- HDL: Hardware **Description** Language
  - Figure out architecture first, then write code.

Primitives

- Primitives are modules ready to be instanced
- Smallest modeling block for simulator
- 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

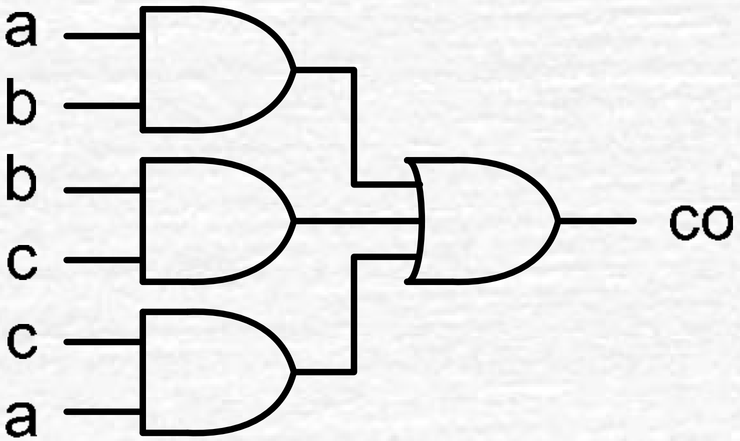
Case Study
1-bit Full Adder

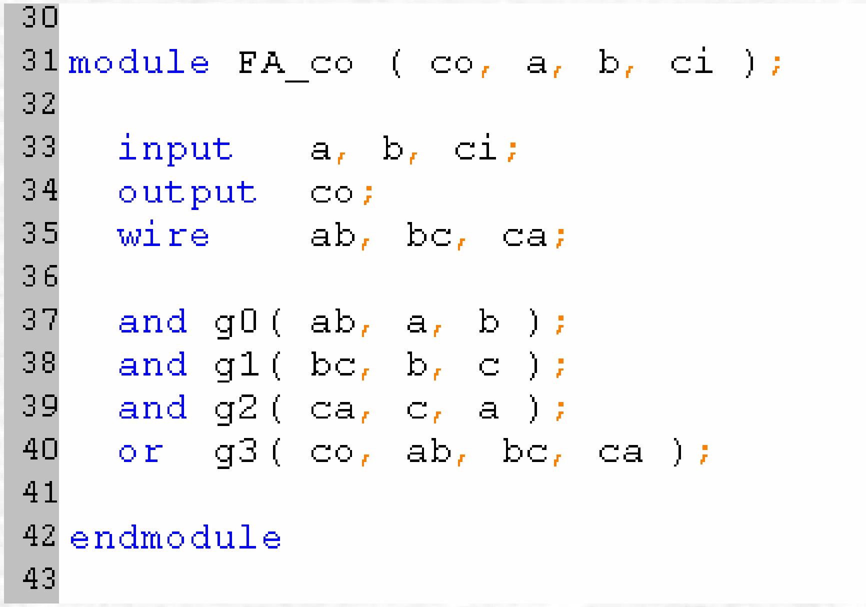


Ci	A	B	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Case Study
1-bit Full Adder

- co = (a • b) + (b • ci) + (ci • a);

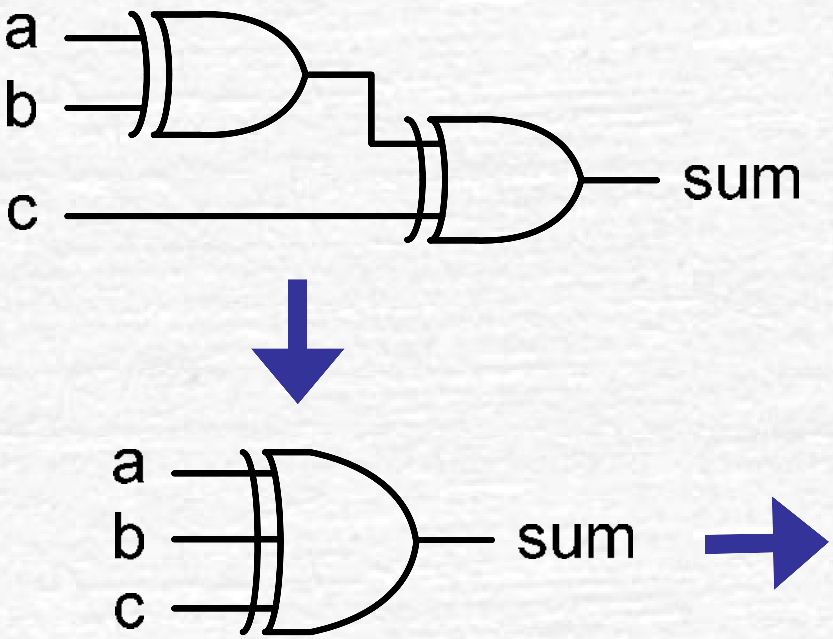


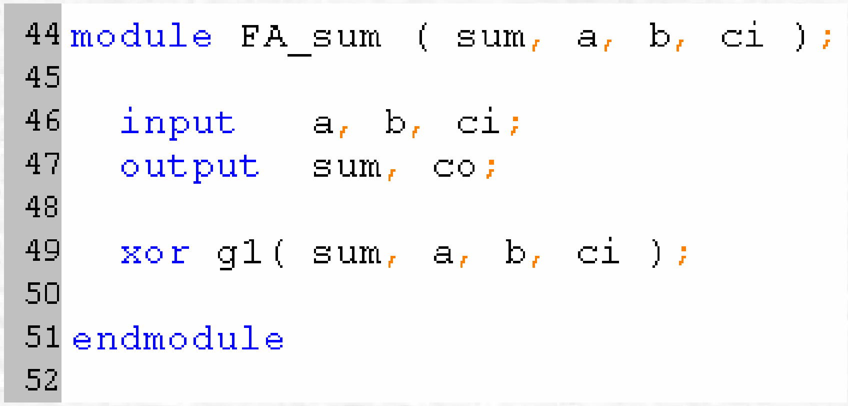


```
module FA_co ( co, a, b, ci );

  input   a, b, ci;
  output  co;
  wire    ab, bc, ca;

  and g0( ab, a, b );
  and g1( bc, b, c );
  and g2( ca, c, a );
  or  g3( co, ab, bc, ca );

endmodule
```

Case Study
1-bit Full Adder

- sum = a ⊕ b ⊕ ci

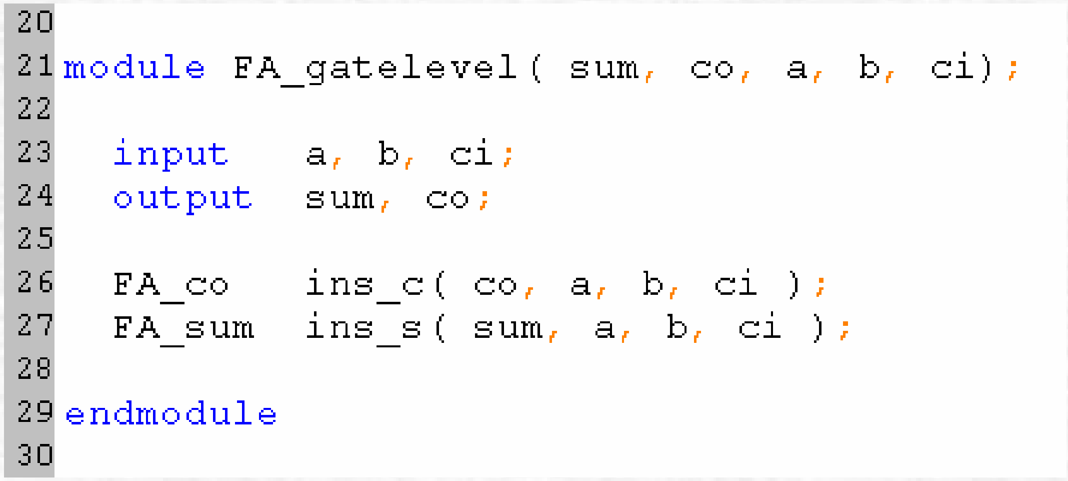




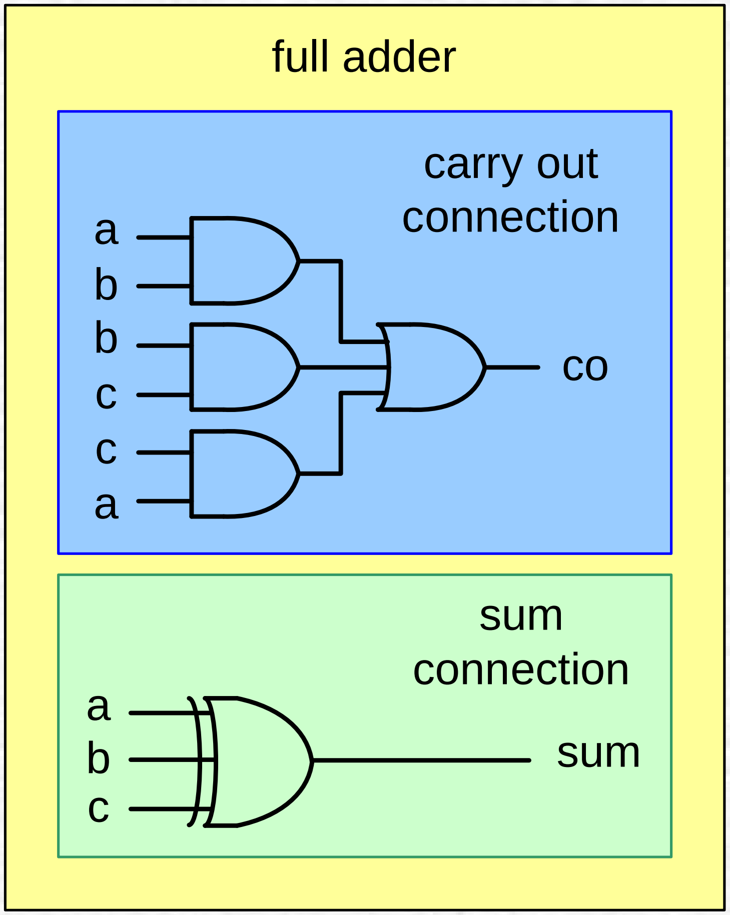
```
module FA_sum ( sum, a, b, ci );

  input    a, b, ci;
  output   sum, co;

  xor g1( sum, a, b, ci );

endmodule

```

Case Study
1-bit Full Adder

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



```verilog

module FA_gatelevel( sum, co, a, b, ci);

  input     a, b, ci;
  output    sum, co;

  FA_co     ins_c( co, a, b, ci );
  FA_sum    ins_s( sum, a, b, ci );

endmodule

```



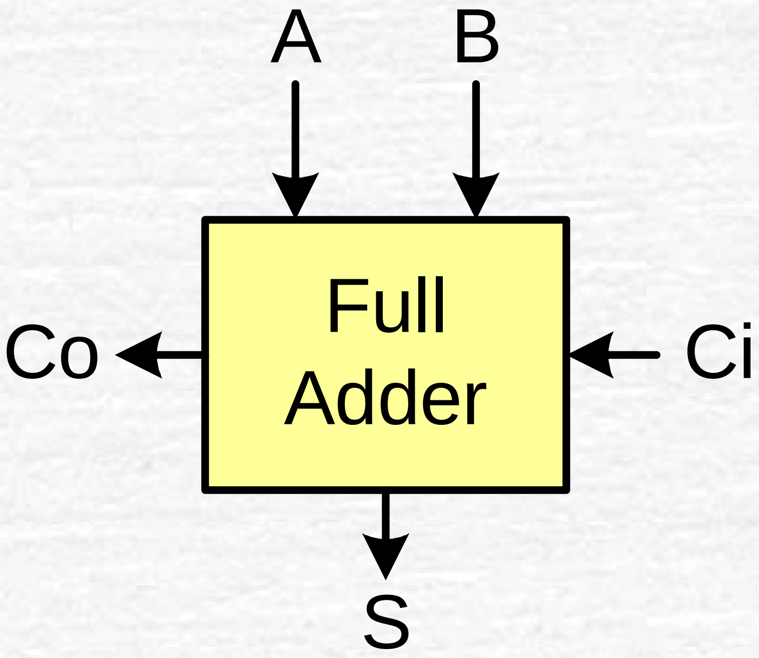
RT-Level & Behavioral Level Modeling

- RT-Level & Behavioral Level Modeling
- Case Study

RT-Level & Behavioral Level Modeling

- High level description
  - User friendly
  - Concise code
  - Faster simulation speed (event driven)
- Widely used for some common operations
  - +,-,*
  - &,|,~
- Two main formats
  - always block (for behavior level)
  - assign (for RT level)

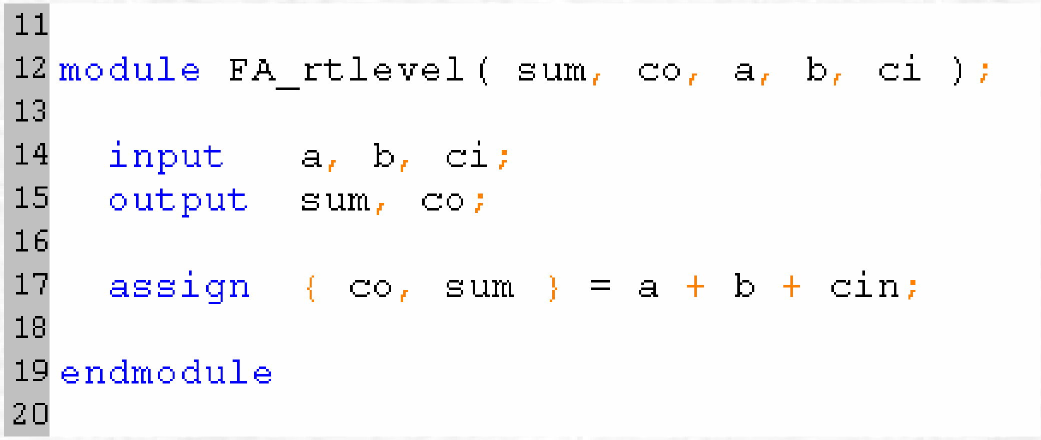
Case Study
1-bit Full Adder



{Co,S} = A + B + Ci

Case Study
1-bit Full Adder

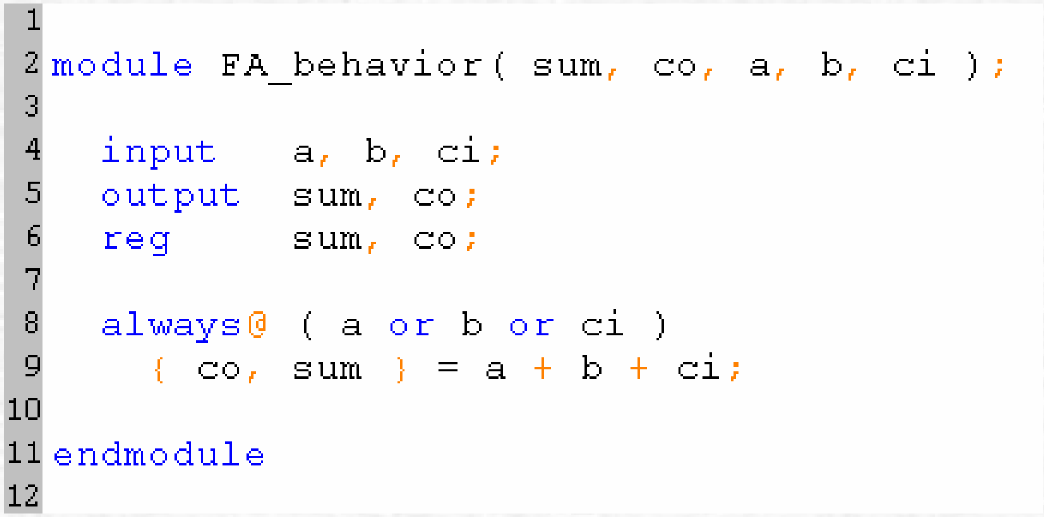
- RT-level modeling of combinational circuit
    - Describe boolean function with *operators* and use continuous assignment **assign**



```verilog

module FA_rtlevel( sum, co, a, b, ci );

  input   a, b, ci;
  output  sum, co;

  assign  { co, sum } = a + b + cin;

endmodule

```

Case Study
1-bit Full Adder

- Behavior-level modeling of combinational circuit:
  - Use event-driven construct: **always** block
  - Event: ***@(*** *sensitive_list* ***)***

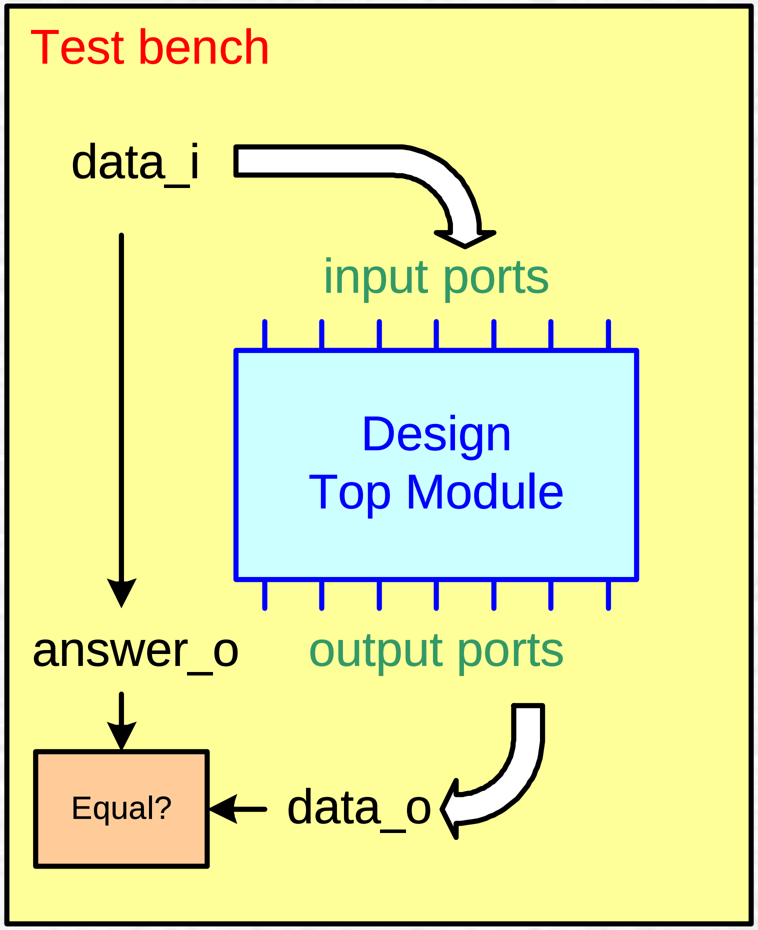


```
module FA_behavior( sum, co, a, b, ci );

  input   a, b, ci;
  output  sum, co;
  reg     sum, co;

  always@ ( a or b or ci )
    { co, sum } = a + b + ci;

endmodule
```

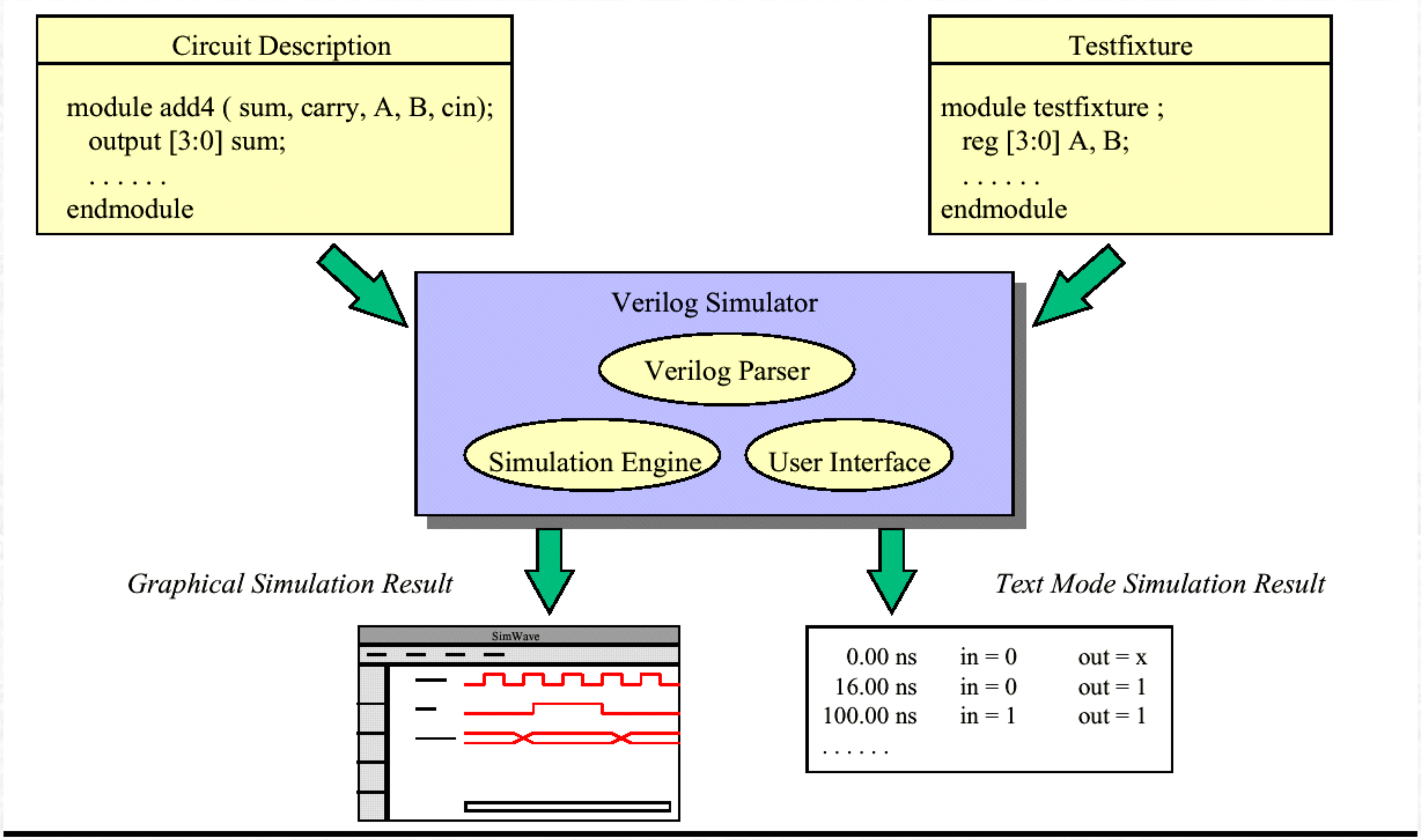
Test bench

Test Methodology

- Systematically verify the functionality of a model.
- Simulation:
  - (1) detect syntax violations in source code
  - (2) simulate behavior
  - (3) monitor results



Verilog Simulator



Circuit Description

```
module add4 ( sum, carry, A, B, cin);
  output [3:0] sum;
  . . . . . .
endmodule
```

Testfixture

```
module testfixture ;
  reg [3:0] A, B;
  . . . . . .
endmodule
```

Verilog Simulator

Verilog Parser

Simulation Engine

User Interface

Graphical Simulation Result

Text Mode Simulation Result

```
  0.00 ns    in = 0    out = x
 16.00 ns    in = 0    out = 1
100.00 ns    in = 1    out = 1
. . . . . .
```

Testbench for Full Adder

```
module t_full_add();
reg    a, b, cin;          // for stimulus waveforms
wire   sum, c_out;

full_add M1 (sum, c_out, a, b, cin); //DUT

initial #200 $finish;      // Stopwatch

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

Summary

- Design module
  - Gate-level or RT-level
  - Real hardware
    - Instance of modules exist all the time
  - Each module has architecture figure
    - Plot architecture figures before you write verilog codes
- Test bench
  - Feed input data and compare output values versus time
  - Usually behavior level
  - Not real hardware, just like C/C++

Note

- Verilog is a platform
  - Support hardware design (design module)
  - Also support C/C++ like coding (test bench)
- How to write verilog well
  - Know basic concepts and syntax
  - Get a good reference (a person or some code files)
  - Form a good coding habit
    - Naming rule, comments, format partition (assign or always block)
- Hardware
  - Combinational circuits (today's topic)
    - 畫圖(architecture), then 連連看(coding)
  - Sequential circuits (we won't model them in this course)
    - register: element to store data