HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL DESIGN

數位設計硬體描述語言

Verilog Basics

Materials partly adapted from "Digital System Designs and Practices Using Verilog HDL and FPGAs," M.B. Lin.

NCKU EE
LPHP Lab

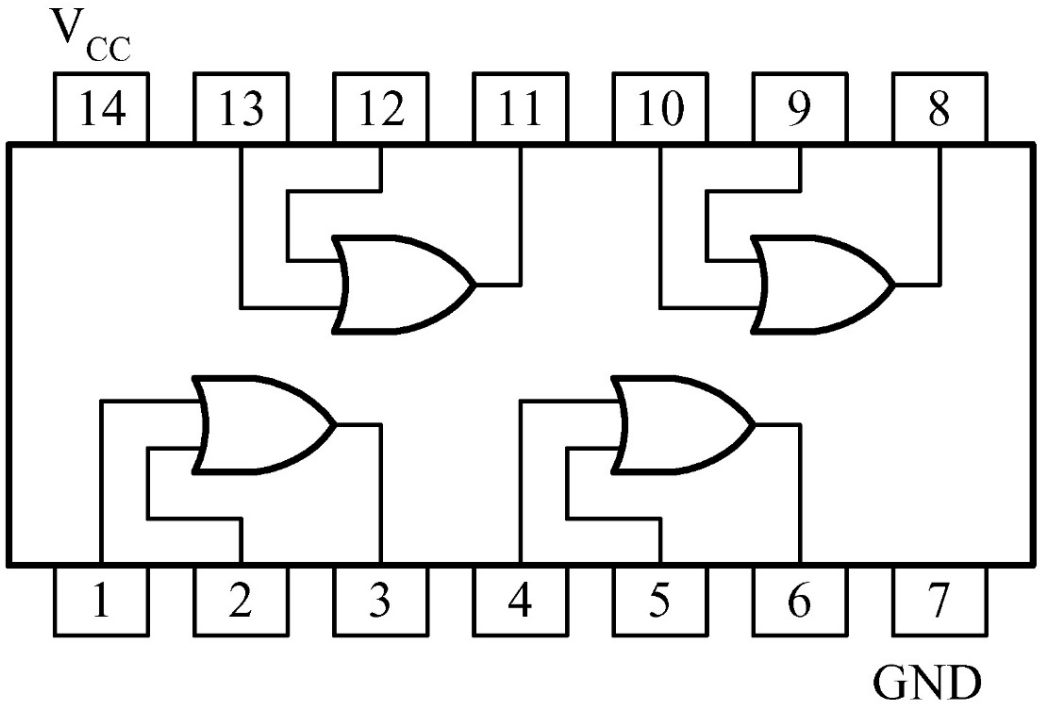
OUTLINE

- Verilog Basics
- Structural Modeling
- Dataflow Modeling

VERILOG BASICS

CONCEPT OF MODULES

- Module
  - A core circuit (called internal or body)
  - An interface (called ports)



MODULES – VERILOG HDL MODULES

- module --- The basic building block

module Module name

Port List, Port Declarations (if any)

Parameters (if any)

Declarations of *wires*, *regs*, and other variables

Instantiation of lower level modules or primitives

Dataflow statements (*assign*)

always and *initial* blocks. (all behavioral statements go into these blocks).

Tasks and functions.

endmodule statement

LEXICAL CONVENTIONS

- Almost the same lexical conventions as C language
- Identifiers: alphanumeric characters, _, and $
  - Verilog is a case-sensitive language
- White space: blank space (\b), tabs (\t), and new line (\n)

LEXICAL CONVENTIONS

- Comments
  - // --- the remaining of the line
  - /* ….*/ --- what in between them
- Sized number: <size>`<base format><number>
  - 4`b1001
  - 16`habcd

LEXICAL CONVENTIONS

- Unsized number:

  `<base format><number>

  - 2009
  - `habc

- x or z
  - x: an unknown value
  - z: a high impedance

LEXICAL CONVENTIONS

- Negative number:

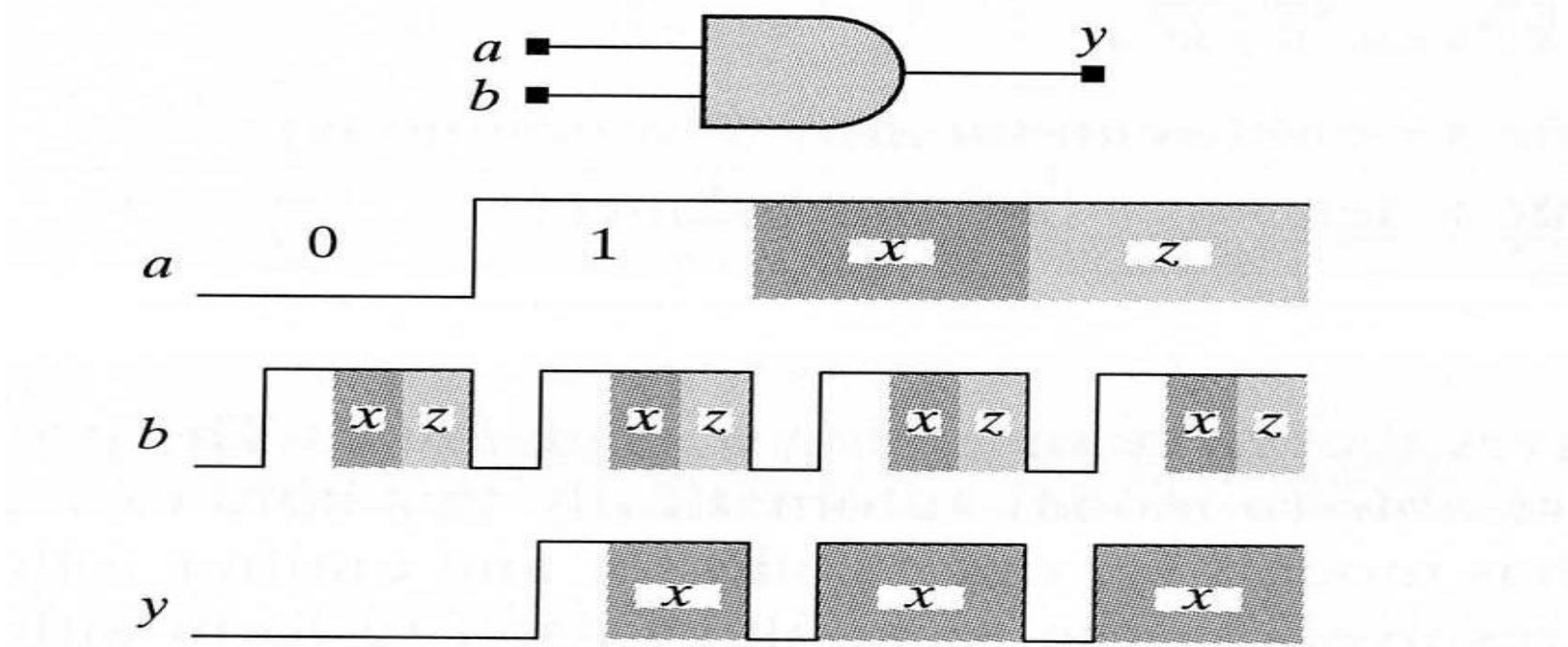
  -<size>`<base format><number>
  - -4`b1001
  - -16`habcd
- "_" and "?"
  - 16`b0101_1001_1110_0000
  - 8`b01??_11?? // = 8`b01zz_11zz
- String: "Have a lovely day"

CODING STYLE

- Lowercase letters
  - For all signal names, variable names, and port names
- Uppercase letters
  - For names of constants and user-defined types
- Meaningful names
  - For signals, ports, functions, and parameters

THE VALUE SET OF THE SIGNAL

- Four-value logic
  - 0 : logic 0, false condition
  - 1 : logic 1, true condition
  - z : high-impedance
  - x : unknown



DATA TYPES

- Nets: any hardware connection points
- Variables: any data storage elements

Nets		Variables
wire	supply0	reg
tri	supply1	integer
wand	tri0	real
wor	tri1	time
triand	trireg	realtime
trior	uwire	

NETS

- Driven by
  - Primitive
  - continuous assignment
  - force … release
  - module port

TYPES OF NETS

Net Types	Functionality
wire, tri	for standard interconnection wires (default)
wor, trior	for multiple drivers that are Wire-ORed
wand, triand	for multiple drivers that are Wire-ANDed
trireg	for nets with capacitive storage
tri1	for nets which pull up when not driven
tri0	for nets which pull down when not driven
supply1	for power rails
supply0	for ground rails

Nets that are defaulted to single bit nets of type wire. This can be overridden by using the following compiler directive.

‘default_nettype <nettype>

VARIABLES

- Assigned value only within
  - Procedural statement
  - Task
  - Function
- Cannot be used as
  - input
  - inout

MODULE MODELING STYLES

MODULE MODELING STYLES

- Structural style
  - Gate level
  - Switch level
- Dataflow style
- Behavioral or algorithmic style
- Mixed style
  - RTL = synthesizable behavioral + dataflow constructs

PORT DECLARATION

Variable declaration

- output
- Input
- wire

Main body

```
module mux2x1(f, s, s_bar, a, b);
    output  f;
    input  s, s_bar;
    input  a, b;
    wire  nand1_out, nand2_out;
   // boolean function
    nand(nand1_out, s_bar, a);
    nand(nand2_out, s, b);
    nand(f, nand1_out, nand2_out);
endmodule
```

PORT CONNECTION RULES

- Named association
- Positional association

PORT DECLARATION

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional.

Connecting by using positional association

Connecting by using named association

Instance name is necessary.

STRUCTURAL MODELING

```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input x, y;
output s, c;
// half adder body
// instantiate primitive gates
xor (s,x,y);
and (c,x,y);
endmodule
```

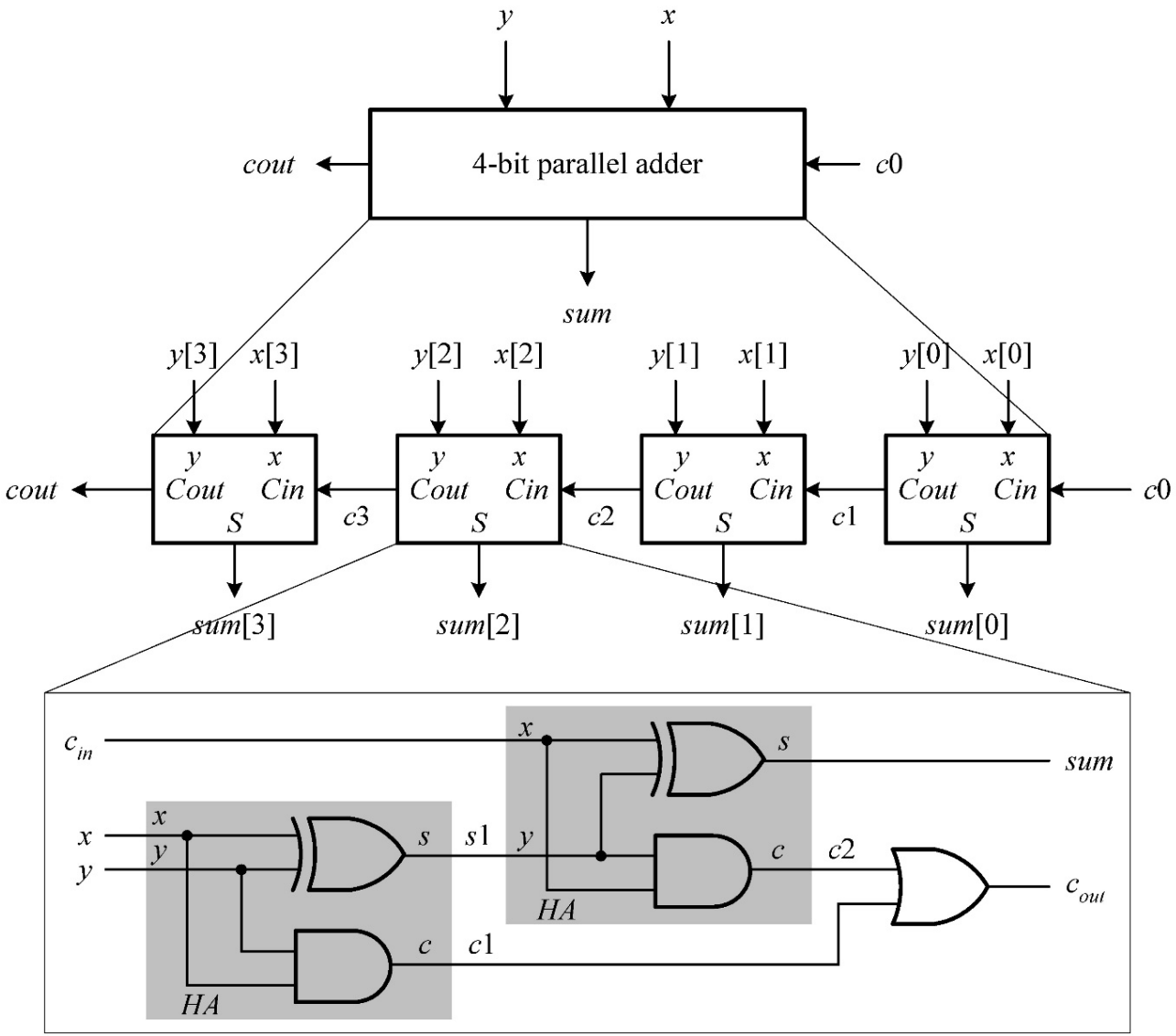

STRUCTURAL MODELING

```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
input x, y, cin;
output s, cout;
wire s1, c1, c2; // outputs of both half adders
// full adder body
// instantiate the half adder
half_adder ha_1 (x, y, s1, c1);
half_adder ha_2 (cin, s1, s, c2);
or (cout, c1, c2);
endmodule
```


STRUCTURAL MODELING

```
// gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
wire c1, c2, c3; // intermediate carries
// four_bit adder body
// instantiate the full adder
full_adder fa_1 (x[0], y[0], c_in, sum[0], c1);
full_adder fa_2 (x[1], y[1], c1, sum[1], c2);
full_adder fa_3 (x[2], y[2], c2, sum[2], c3);
full_adder fa_4 (x[3], y[3], c3, sum[3], c_out);
endmodule
```


HIERARCHICAL DESIGN

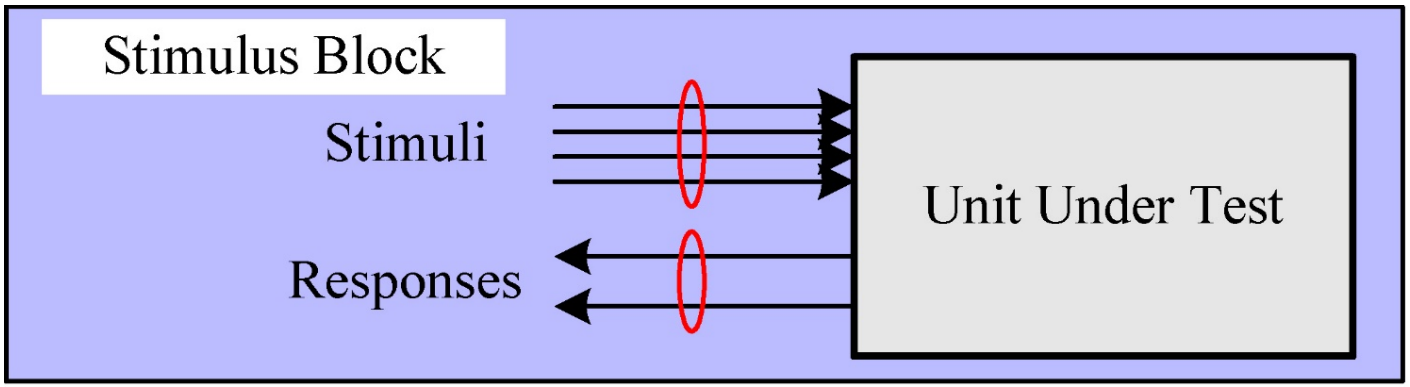


BEHAVIORAL MODELING

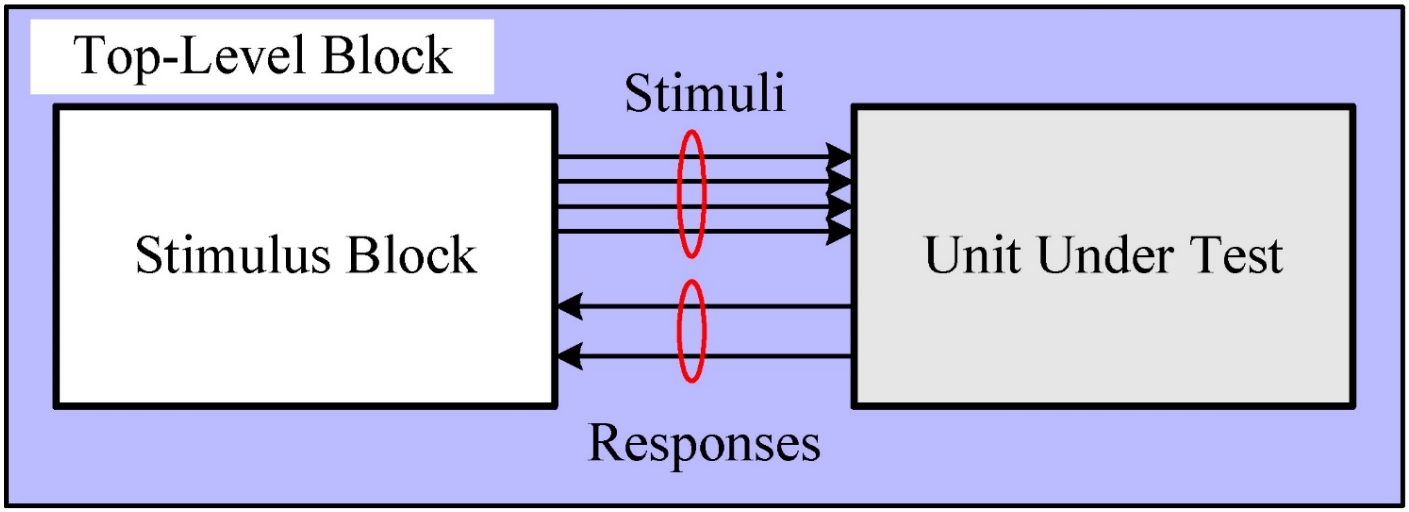
```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out;  // need to be declared as reg types
// specify the function of a full adder
always @(x, y, c_in) // always @(*) or always@(x or y or c_in)
#5 {c_out, sum} = x + y + c_in;
endmodule
```


SIMULATION

BASIC SIMULATION CONSTRUCTS



(a) Stimulus block at the top-level module.



(b) Stimulus block is considered as a separate module.

SYSTEM TASKS FOR SIMULATION

- $display
  - $display(*ep*1, *ep*2, …, *epn*);

    *ep*1, *ep*2, …, *epn*: quoted strings, variables, expressions
- $monitor
  - $monitor(*ep*1, *ep*2, …, *epn*);
- $monitoton
- $monitotoff
- $stop
- $finish

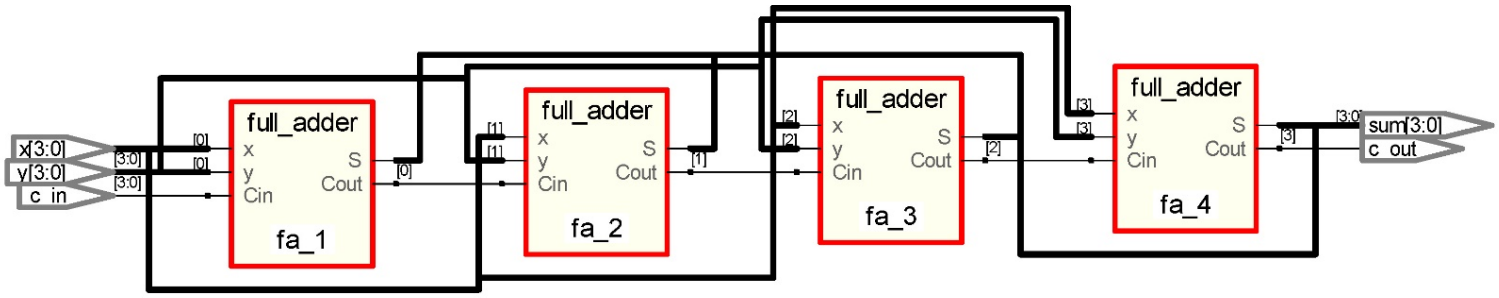
TIME SCALE FOR SIMULATIONS

- Reference time unit
  - `timescale <reference_time_unit>/<time_precision>
    - <reference_time_unit>: unit of measurement for times and delays
    - <time_precision>: the precision to which the delays are rounded off during simulation
  - Only 1, 10, 100 are valid integers
- Examples
  - `timescale 1ns/10ps
  - `timescale 100ns/1ns

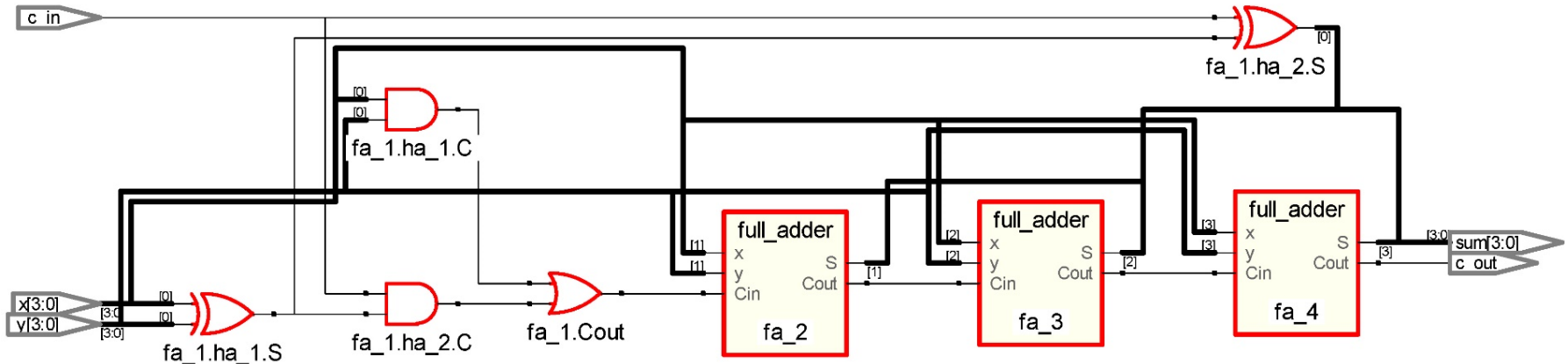
AN EXAMPLE --- A 4-BIT ADDER

```
// Gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input   [3:0] x, y;
input   c_in;
output [3:0] sum;
output c_out;
wire   C1,C2,C3;  // Intermediate carries
// -- four_bit adder body--
// Instantiate the full adder
   full_adder fa_1 (x[0],y[0],c_in,sum[0],C1);
   full_adder fa_2 (x[1],y[1],C1,sum[1],C2);
   full_adder fa_3 (x[2],y[2],C2,sum[2],C3);
   full_adder fa_4 (x[3],y[3],C3,sum[3],c_out);
endmodule
```


AN EXAMPLE --- A 4-BIT ADDER



(a) The synthesized result of the four-bit addder module



(b) The result after dissovling the first full_adder module

AN EXAMPLE --- A TEST BENCH

```
`timescale 1 ns / 100 ps  // time unit is in ns.
module four_bit_adder_tb;
//Internal signals declarations:
reg [3:0] x;
reg [3:0] y;
reg c_in;
wire [3:0] sum;
wire c_out;
// Unit Under Test port map
    four_bit_adder UUT
   (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
reg [7:0] i;
initial begin // for use in post-map and post-par simulations.
// $sdf_annotate ("four_bit_adder_map.sdf", four_bit_adder);
// $sdf_annotate ("four_bit_adder_timesim.sdf", four_bit_adder);
end
```


AN EXAMPLE --- A TEST BENCH

```
initial
  for (i = 0; i <= 255; i = i + 1) begin
      x[3:0] = i[7:4]; y[3:0] = i[3:0]; c_in =1'b0;
  #20 ;   end
initial #6000 $finish;
initial
  $monitor($realtime,“ns %h %h %h %h", x, y, c_in, {c_out, sum});
endmodule
```


AN EXAMPLE --- SIMULATION RESULTS

```
     0ns 0 0 0 00
#   20ns 0 1 0 01
#   40ns 0 2 0 02
#   60ns 0 3 0 03
#   80ns 0 4 0 04
#  100ns 0 5 0 05
#  120ns 0 6 0 06
#  140ns 0 7 0 07
#  160ns 0 8 0 08
#  180ns 0 9 0 09
#  200ns 0 a 0 0a
#  220ns 0 b 0 0b
#  240ns 0 c 0 0c
#  260ns 0 d 0 0d
# 280ns 0 e 0 0e
# 300ns 0 f 0 0f
# 320ns 1 0 0 01
# 340ns 1 1 0 02
# 360ns 1 2 0 03
# 380ns 1 3 0 04
# 400ns 1 4 0 05
# 420ns 1 5 0 06
# 440ns 1 6 0 07
# 460ns 1 7 0 08
# 480ns 1 8 0 09
# 500ns 1 9 0 0a
# 520ns 1 a 0 0b
# 540ns 1 b 0 0c
```


AN EXAMPLE --- SIMULATION RESULTS