硬體描述語言

Chapter 01

Hardware Modeling

Ren-Der Chen (陳仁德)
Department of Computer Science and Information Engineering
National Changhua University of Education
E-mail: rdchen@cc.ncue.edu.tw

Fall, 2024

Hardware Description Language (HDL)

IC

數位　類比

- Computer-based programming language
- Model, represent, and simulate digital hardware
  - Hardware concurrency　並行:同時執行
  - Semantics for signal value and time　=> 波型圖

    信號
- Special constructs and semantics
  - 邊緣信號

    Edge transitions　0變為1的時間點 (or 1→0)
  - Propagation delays

    延遲

About Verilog

- Model hardware at different levels of abstraction 抽象化
- Mix different levels of abstraction in description and simulation 模擬
- Maintain clear relationships between language constructs and hardware ← 對應關係 →
- Support hierarchical decomposition and structured design methodology 階層式
- Provide medium for integration between design tools
- Link to ASIC (Application Specific Integrated Circuit) foundry support 用於特定功能的IC

Design Abstraction

大

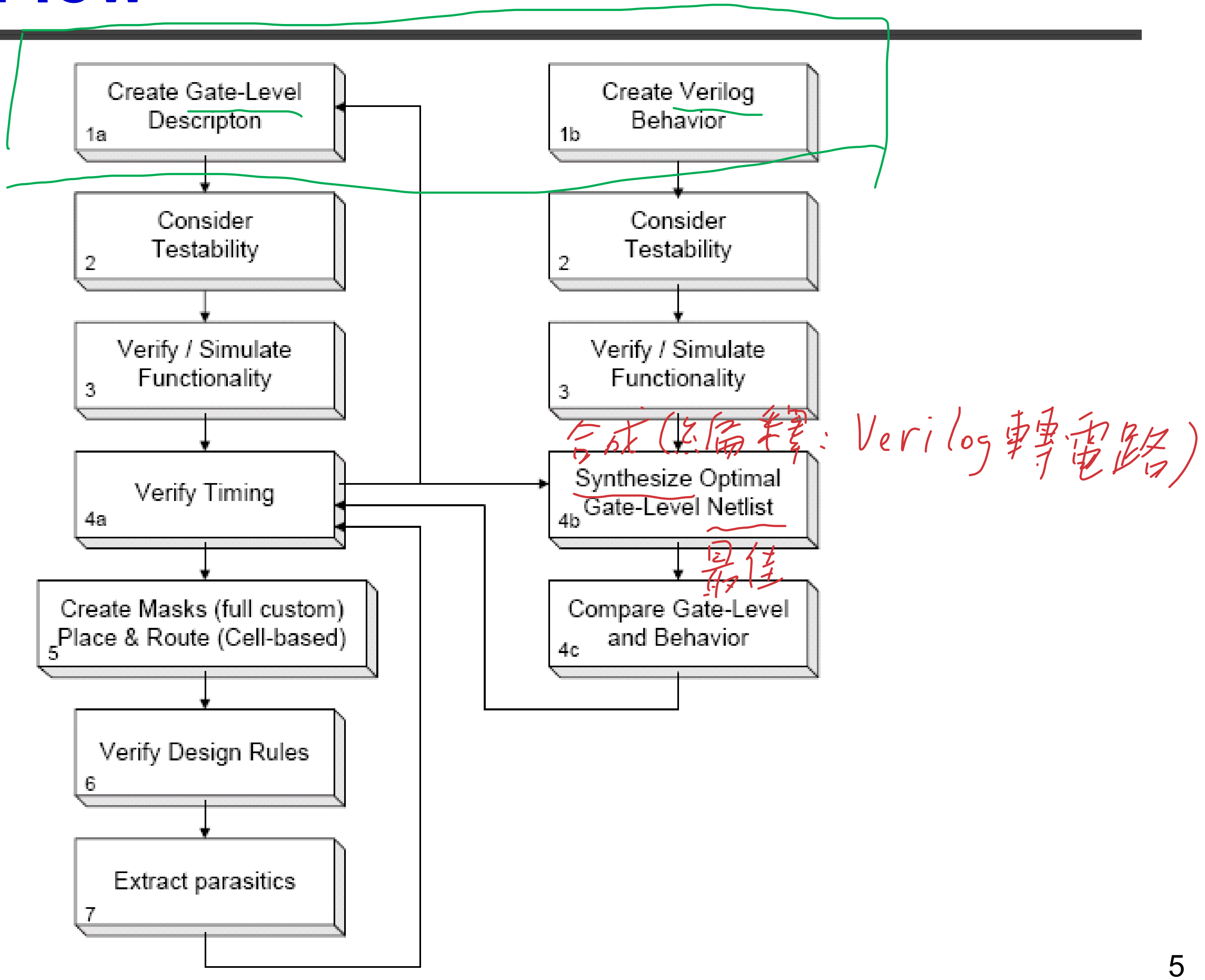
INCREASING ABSTRACTION

LEVEL	STRUCTURAL	BEHAVIORAL
PMS CHIP 晶片	CPUs, MEMORIES MICROPROCESSORS RAM, ROM, UART PARALLEL PORT	PERFORMANCE I/O RESPONSE ALGORITHMS OPERATIONS
暫存器 REGISTER	REGISTERS, ALUs COUNTERS, MUXES	TRUTH TABLES STATE TABLES OPERATIONS
邏輯閘 GATE	GATES, FLIP-FLOPS	BOOLEAN EQUATIONS
CIRCUIT 電晶體	TRANSISTORS, RLC	DIFFERENTIAL EQUATIONS
SILICON 半導體	GEOMETRICAL OBJECTS	---

INCREASING DETAIL

小

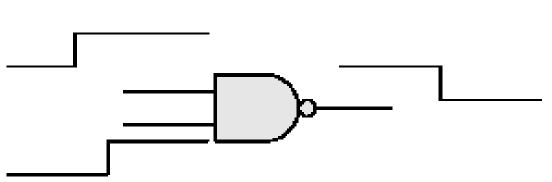
VLSI (Very Large Scale Integrated) Circuit Design Flow

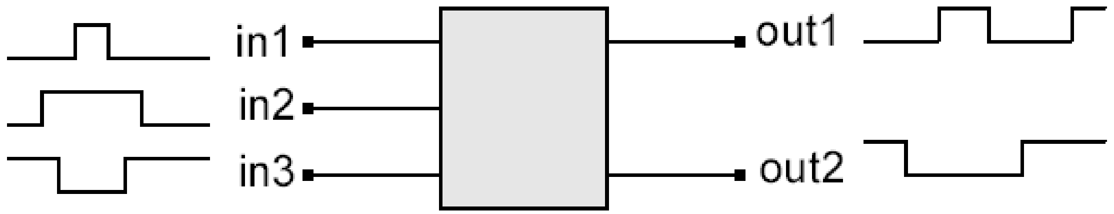


Create Gate-Level Descripton 1a
Consider Testability 2
Verify / Simulate Functionality 3
Verify Timing 4a
Create Masks (full custom) Place & Route (Cell-based) 5
Verify Design Rules 6
Extract parasitics 7
Create Verilog Behavior 1b
Consider Testability 2
Verify / Simulate Functionality 3
合成(編譯: Verilog轉電路)
Synthesize Optimal Gate-Level Netlist 4b
最佳
Compare Gate-Level and Behavior 4c

HDL Model

- Describe a relationship (event scheduling rule) between the input signals and output signals.



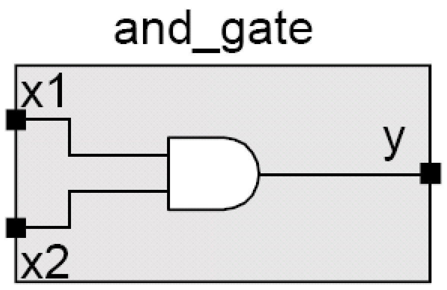


- For structural models, the relationship is apparent under simulation.
- For abstract models, the relationship may be obvious, e.g. “**assign** Y = A + B”.

Two-Input AND Gate Described by Two HDL Languages

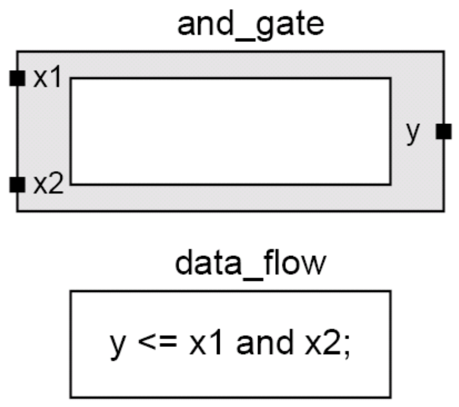
Verilog

```
module and_gate (x1, x2, y);
    input        x1, x2;
    output       y;

    and (y , x1, x2);
    // assign y = x1 & x2;
endmodule
```



VHDL

```
entity and_gate is
    port (x1, x2: in BIT; y: out BIT);
end and_gate;

architecture data_flow of and_gate is
begin
    y <= x1 and x2;
end data_flow;
```



HDL Design

- A HDL must
  - Model hardware reality and
  - Provide an efficient environment supporting a methodology that enhances the productivity of the designer.
- Traditional design based on gate-level schematic entry is being replaced by

電路圖

  - A methodology using HDL-based RTL (Register Transfer Level) descriptions and
  - A synthesis tool to produce an optimized gate-level realization of a desired functionality.

Schematic vs. HDL

- Visual orientation
- Functionality: implicit
- Focus: gate level
- Low level of abstraction
- Editing is sensitive to size

畫

對於複雜度較敏感

- Text orientation
- Functionality: explicit
- Focus: mixed
- High level of abstraction
- Editing is "insensitive " to size and complexity

語言

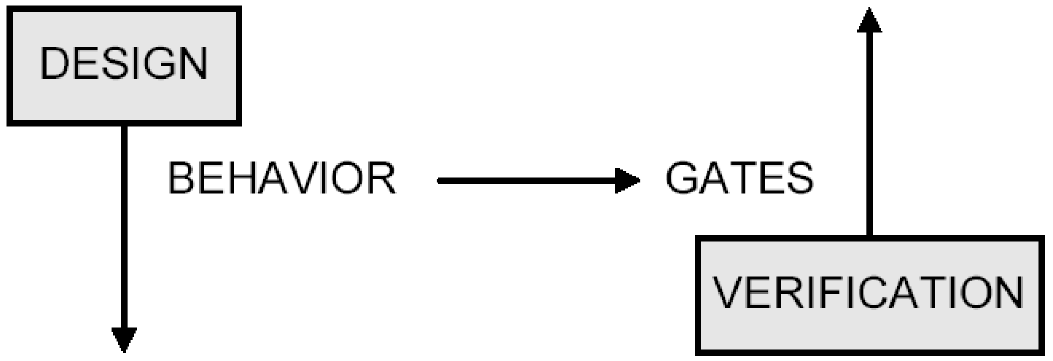
較不敏感

Benefits of HDL-Based Design

- Higher levels of abstraction
- Increase in productivity & quality
- Rapid prototyping with FPGAs
- Exploit synthesis tools and ASIC support
- Promote portable/parameterized/re-usable models
- Facilitate integration of third-party IP (Intellectual Property)

Structured Design Methodology

- Use top-down, hierarchical decomposition to partition a complex design into simpler, hierarchically organized, functional units.
- Nested module instantiation is the Verilog mechanism for hierarchical decomposition.

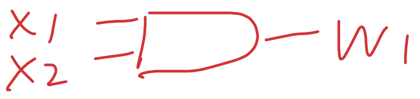


Alternative Styles of Design (1/2)

- **Structural modeling**
  - Interconnect objects to create a structure with a desired behavior.

    and (w1, x1, x2);

    and (w2, x3, x4);



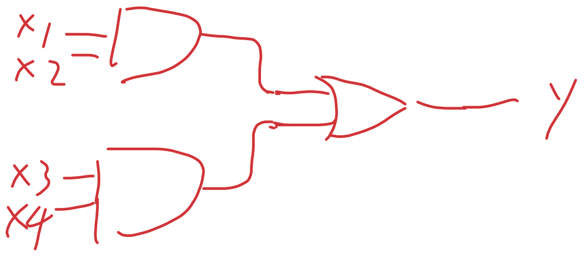
- **RTL modeling**
  - Use language operators to describe logic with implicit binding to hardware (implicit combinational logic).

    assign y = ~((x1& x2) | (x3 & x4));

→綁在一起

```
output:
3
7
```

```
int a=1,b=2;
int y=a+b;
cout<<y<<endl;
a=3,b=4;
cout<<y<<endl;
output:
 3
 3
```



Alternative Styles of Design (2/2)

- **Behavioral/algorithmic modeling**
  - Use HDL-based procedural code to describe the desired I/O behavior without explicit binding to hardware.

偵測

```
always @ ( x1 or x2 or x3 or x4)
  case ({x1, x2, x3, x4})
            0, 1, 2, 4, 5, 6, 8, 9, 10:        y = 1;
            default:                           y = 0;
endcase
```

Descriptive Styles for Modeling

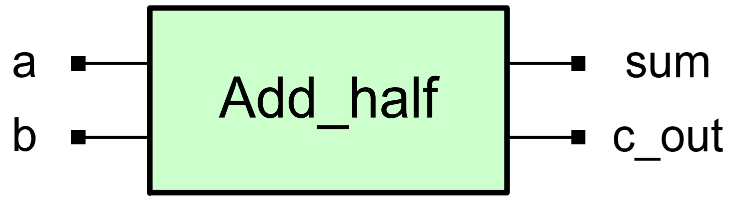
- **Structural**
  - Build a structural model by instantiating primitives and/or other modules within a module declaration, and interconnect with nets – explicit structural description.
- **Behavioral**
  - Continuous assignment statements – implicit structural description.
  - Verilog behaviors.
- **Primitives**
  - Built-in primitives, all combinational
  - User-defined primitives (UDPs)

Primitives 基本元件

可合成 不可合成

Combinational Logic	Three State	MOS Gates	CMOS Gates	Bi-Directional Gates	Pull Gates
and	bufif0	nmos	cmos	tran	pullup
nand	bufif1	pmos	rcmos	tranif0	pulldown
or	notif0	rnmos		tranif1	
nor	notif1	rpmos		rtran	
xor				rtranif0	
xnor				rtranif1	
buf					
not					

Design of a Half Adder

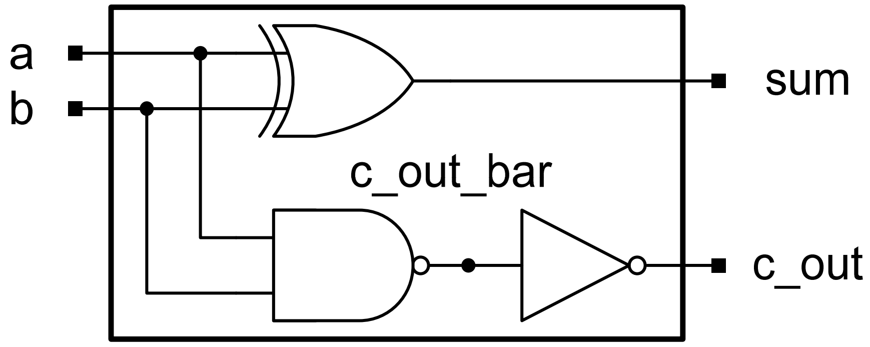


a\b	0	1
0	0	1
1	1	0

$sum = a \oplus b$

a\b	0	1
0	0	0
1	0	1

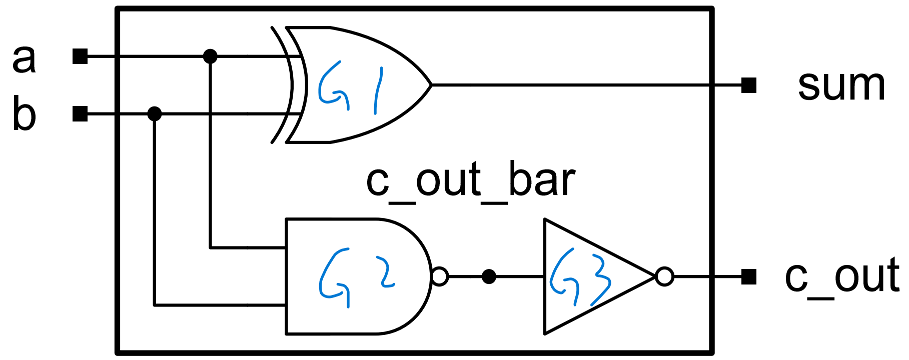
c_out = ab



Structural Description of a Half Adder

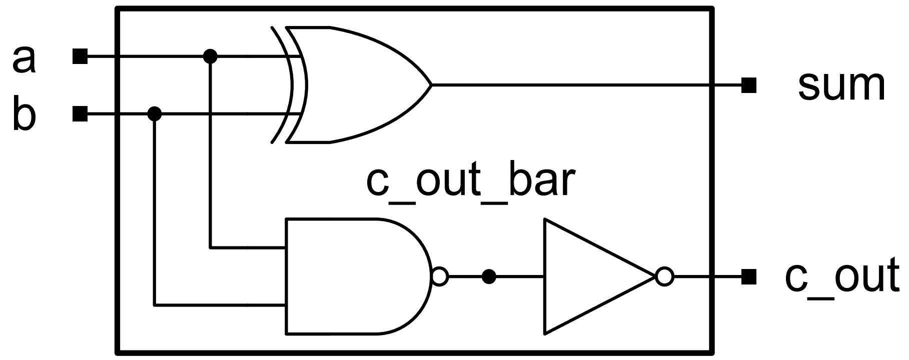
```
module add_half_struct (a, b, sum, c_out);
    input        a, b;
    output       sum, c_out;
    wire         c_out_bar;

    xor          G1 (sum, a, b);
    nand         G2 (c_out_bar, a, b);
    not          G3 (c_out, c_out_bar);
endmodule
```

不考慮
delay



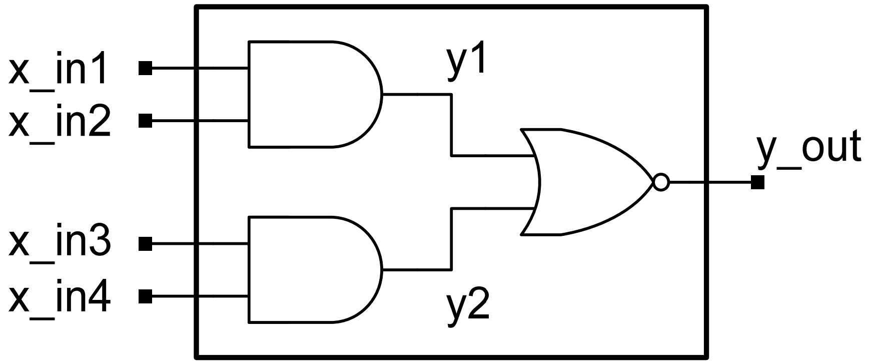
Cell-Based Implementation of a Half Adder

```
module add_half_cell (a, b, sum, c_out);
    input       a, b;
    output      sum, c_out;
    wire        c_out_bar;
引入真實的 gate
    xorf201     G1 (sum, a, b);
    nanf201     G2 (c_out_bar, a, b);
    invf101     G3 (c_out, c_out_bar);
endmodule
```



Structural Description of a 4-Input AOI

```
module AOI4_struct (x_in1, x_in2, x_in3, x_in4, y_out);
    input        x_in1, x_in2, x_in3, x_in4;
    output       y_out;
    wire         y1, y2;

    and          (y1, x_in1, x_in2);
    and          (y2, x_in3, x_in4);
    nor          (y_out, y1, y2);
endmodule
```

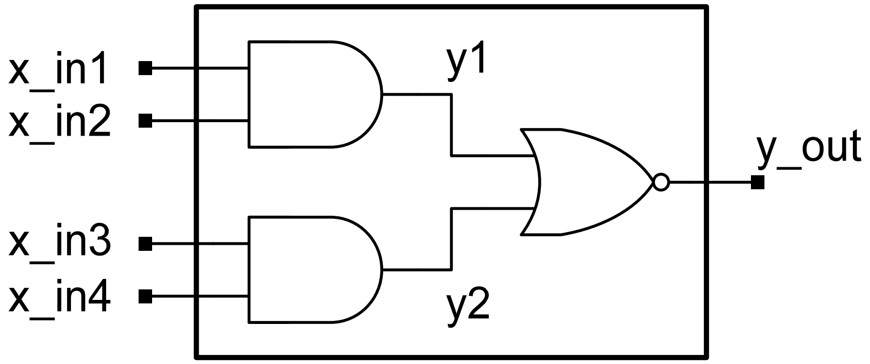


Structural Description of a 4-Input AOI with Unit Delay

```
module AOI4_struct_delay (x_in1, x_in2, x_in3, x_in4, y_out);
    input       x_in1, x_in2, x_in3, x_in4;
    output      y_out;
    wire        y1, y2;
```

加入 delay $(10^{-9}s)$

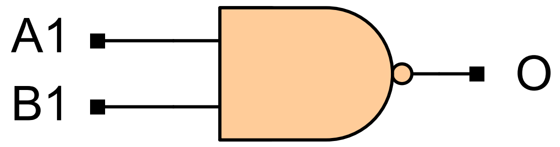
```
    and         #1 (y1, x_in1, x_in2);
    and         #1 (y2, x_in3, x_in4);
    nor         #1 (y_out, y1, y2);
endmodule
```



ASIC Cell Library Module of a NAND Gate

内部

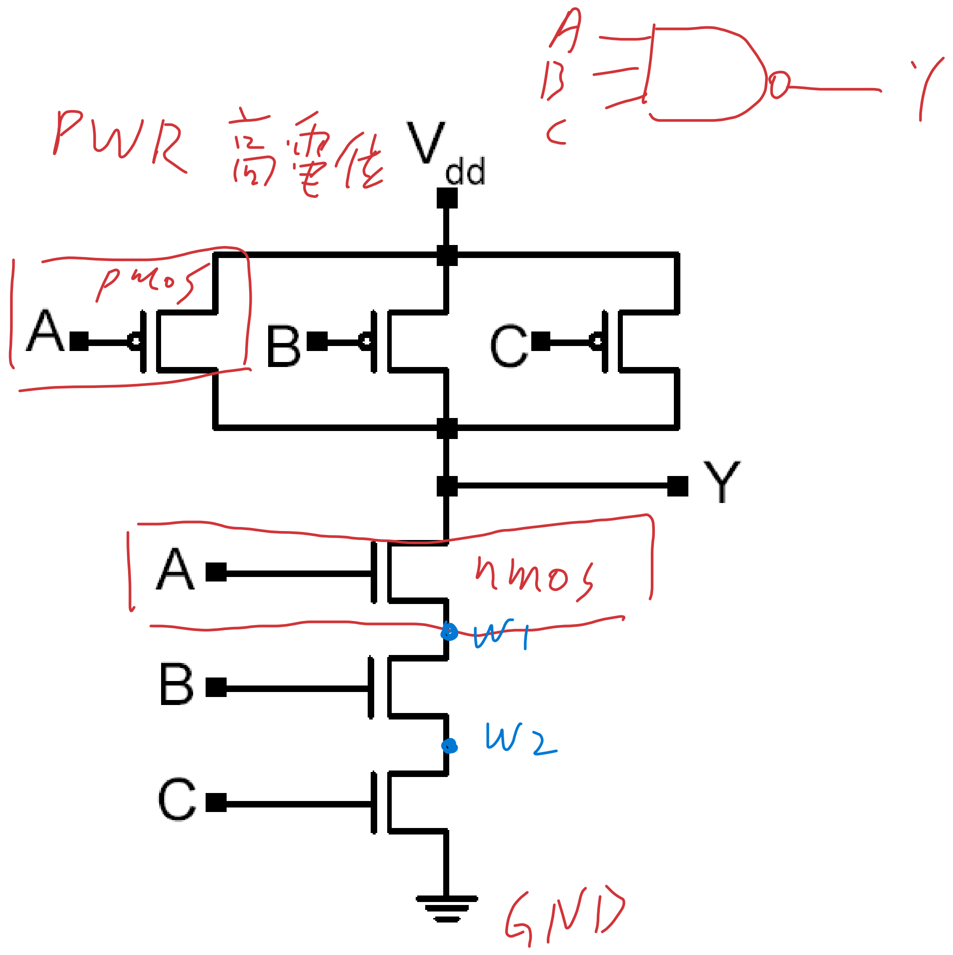
```
module nanf201 (A1, B1, O);
    input       A1, B1;
    output      O;
    nand        (O, A1, B1);

    specify
        specparam
        Tpd_0_1 = 1.13:3.09:7.75,   //  rising delay (min-typical-max)
        Tpd_1_0 = 0.93:2.50:7.34;   //  falling delay (min-typical-max)

        (A1 => O) = (Tpd_0_1, Tpd_1_0);
        (B1 => O) = (Tpd_0_1, Tpd_1_0);
    endspecify
endmodule
```



Switch-Level Description of a 3-Input NAND Gate

```
module nand3_cmos (Y, A, B, C);
    output      Y;
    input       A, B, C;
    supply0     GND;
    supply1     PWR;
    wire        w1, w2;
                                  控制
    pmos        (Y, PWR, A);
    pmos        (Y, PWR, B);
    pmos        (Y, PWR, C);
    nmos        (Y, w1, A);
    nmos        (w1, w2, B);
    nmos        (w2, GND, C);
endmodule
```

不能合成
只能模擬

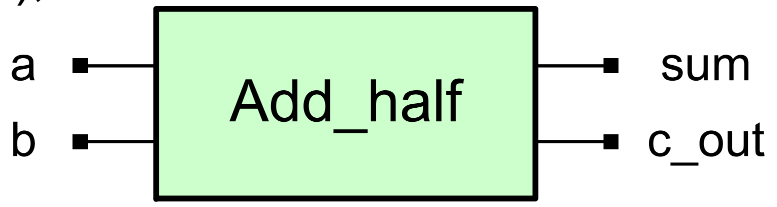


Implicit Structural Description of a Half Adder

半加器

```
module add_half_assign (a, b, sum, c_out);
    input        a, b;
    output       sum, c_out;

    assign       {c_out, sum} = a + b;   //  continuous assignment
endmodule
```

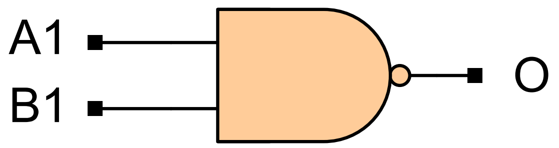
2 bit



- An implicit structural model uses pre-defined language operators and concurrent "continuous assignment statements" to implement logic with implicit binding to generic hardware.
- Implicit structural models can be translated easily by synthesis tools to create optimal combinational logic.

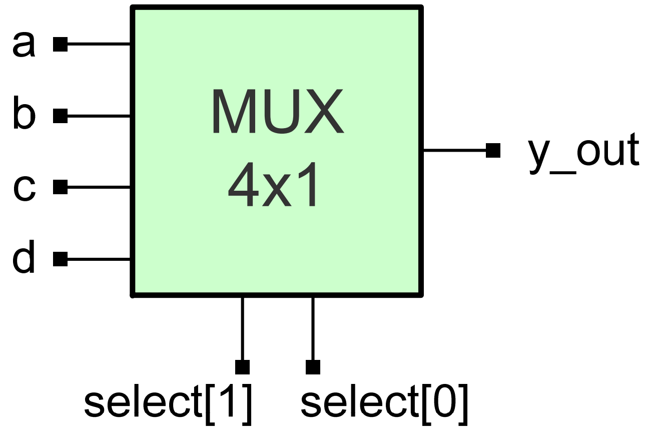
Implicit Structural Description of a NAND Gate

```
module nand2_assign (A1, B1, O);
    input       A1, B1;
    output      O;

    assign      O = ~(A1 & B1);   //  bitwise nand
endmodule
```



Implicit Structural Description of a 4x1 MUX

```
module mux4_assign (a, b, c, d, select, y_out);
    input       a, b, c, d;                       多功器
    input       [1:0] select;
    output      y_out;
    assign      y_out =  (a && !select [1] && !select [0]) ||     0 0
                         (b && !select [1] &&  select [0]) ||     0 1
                         (c &&   select [1] && !select [0]) ||    1 0
                         (d &&   select [1] &&  select [0]) ;     1 1
endmodule
```



Hierarchical Decomposition

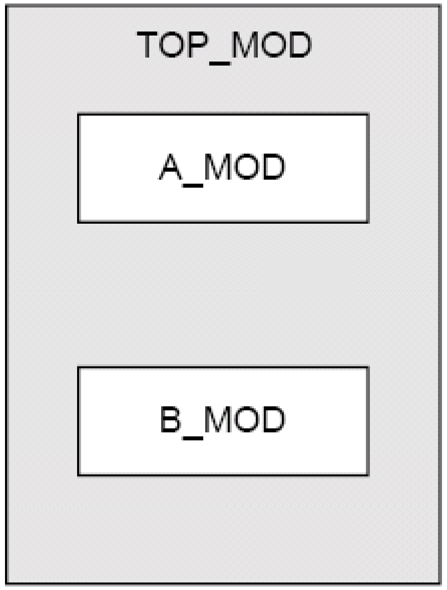
```
module TOP_MOD ();   // declaration
    A_MOD();         // instantiation
    B_MOD();         // instantiation
endmodule
```

呼叫

function

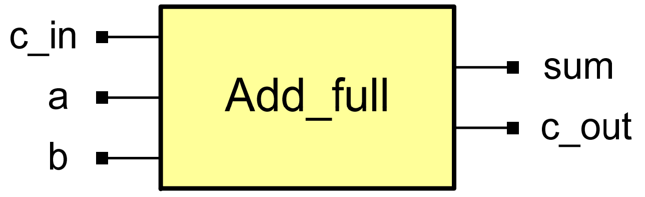
完全複製

```
module A_MOD();      // declaration
...
endmodule

module B_MOD();      // declaration
...
endmodule
```



Design of a Full Adder

全加器

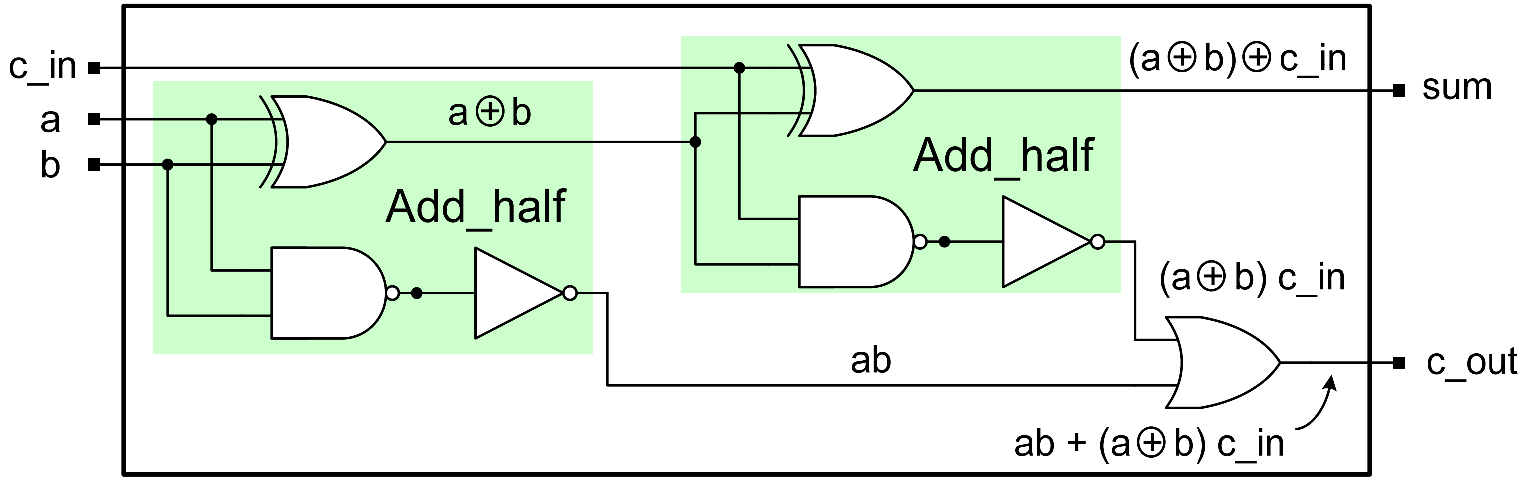


ab\c_in	0	1
00	0	1
01	1	0
11	0	1
10	1	0

sum = $(a \oplus b) \oplus c\_in$

ab\c_in	0	1
00	0	0
01	0	1
11	1	1
10	0	1

c_out = $ab + (a \oplus b)\, c\_in$

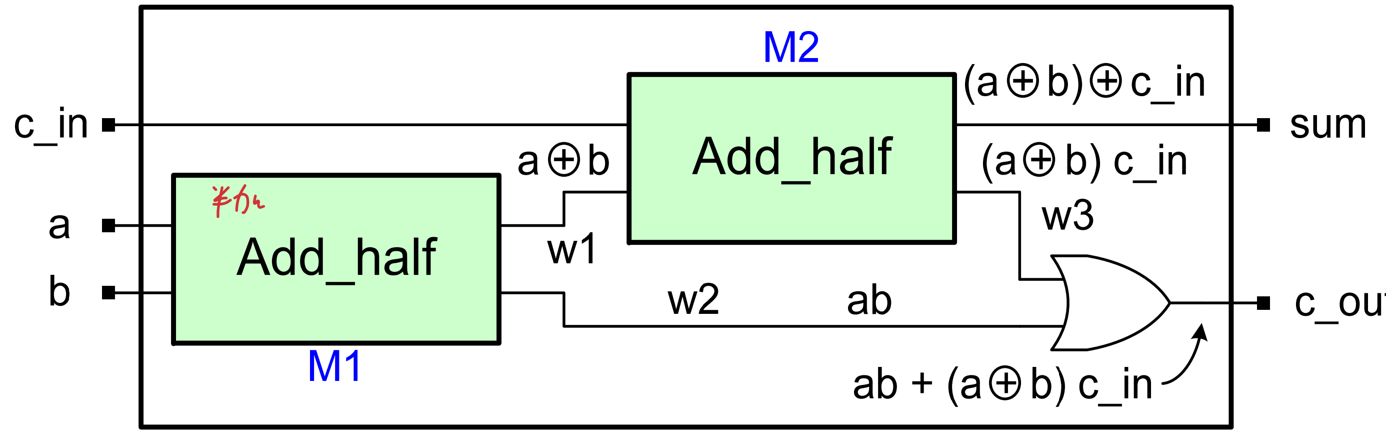


Structural Description of a Full Adder

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

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

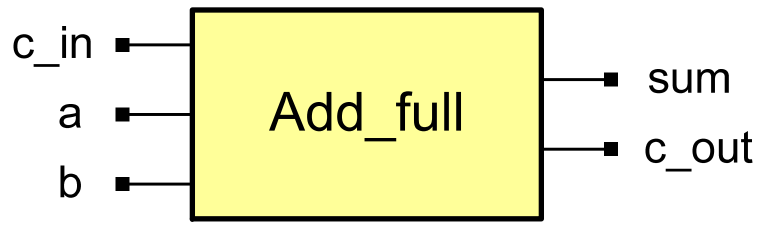
Instance name

1. The use of an instance name with a primitive is optional.
2. A module instance must always have a name.



Implicit Structural Description of a Full Adder

```
module add_full_assign (sum, c_out, a, b, c_in);
    output    sum, c_out;
    input     a, b, c_in;

    assign    sum = a ^ b ^ c_in;          //  bitwise exclusive-or
    assign    c_out = (a & b) | (b & c_in ) | (a & c_in);
endmodule
```



ab\c_in	0	1
00	0	1
01	1	0
11	0	1
10	1	0

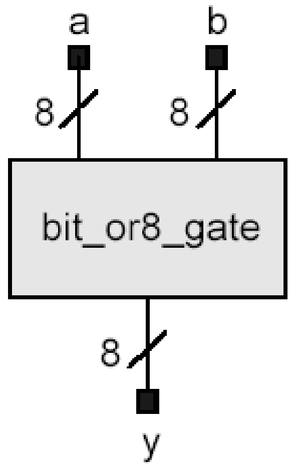
$\text{sum} = a \oplus b \oplus c\_in$

ab\c_in	0	1
00	0	0
01	0	1
11	1	1
10	0	1

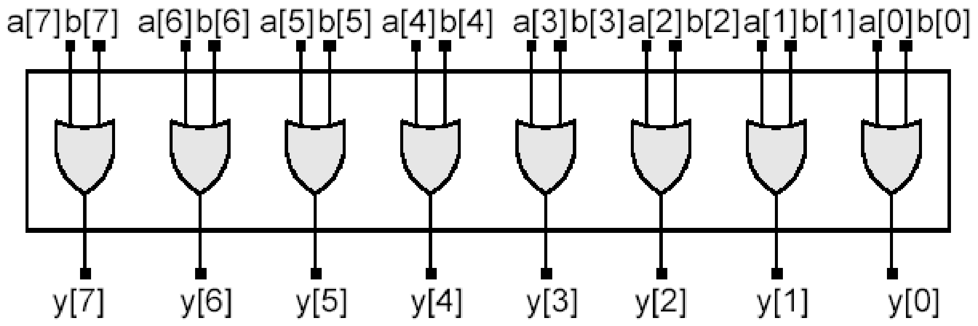
$c\_out = ab + bc + ca$

Two Alternatives for Continuous Assignment

```
module bit_or8_gate1 (y, a, b);
    input       [7:0] a, b;    array
    output      [7:0] y;
    assign      y = a | b;   //  bitwise or
endmodule
```



```
module bit_or8_gate2 (y, a, b);
     input      [7:0] a, b;
    output      [7:0] y;
    wire        [7:0] y = a | b;
endmodule
```

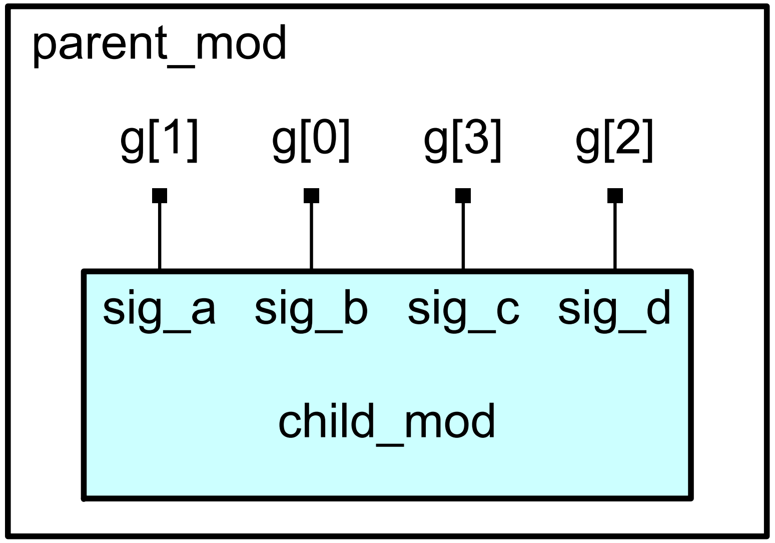

a[7]b[7] a[6]b[6] a[5]b[5] a[4]b[4] a[3]b[3] a[2]b[2] a[1]b[1] a[0]b[0]
y[7] y[6] y[5] y[4] y[3] y[2] y[1] y[0]

Multiple Instantiations and Assignments

```
module multiple_gates (a1, a2, a3, a4, y1, y2, y3);
    input       a1, a2, a3, a4;
    output      y1, y2, y3;

    nand  #1    G1(y1, a1, a2, a3), (y2, a2, a3, a4), (y3, a1, a4);
endmodule

module multiple_assigns (a1, a2, a3, a4, y1, y2, y3);
    input       a1, a2, a3, a4;
    output      y1, y2, y3;

    assign  #1  y1 = a1 ^ a2, y2 = a2 | a3, y3 = a1 + a2;
endmodule
```

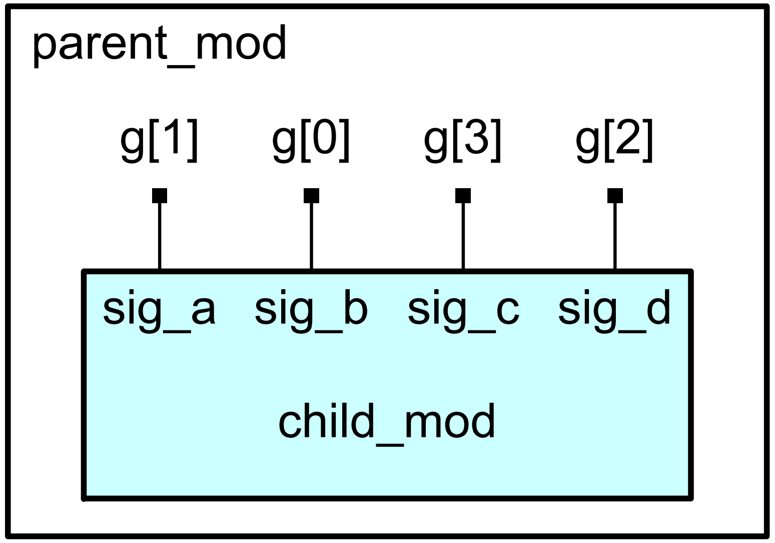
Port Connection by Position Association

```
module parent_mod;
    wire [3:0] g;
    child_mod    G1 (g[1], g[0], g[3], g[2]);
endmodule

module child_mod (sig_a, sig_b, sig_c, sig_d);
    output        sig_a, sig_b;
    input         sig_c, sig_d;
    //  module description goes here
endmodule
```



Port Connection by Name Association

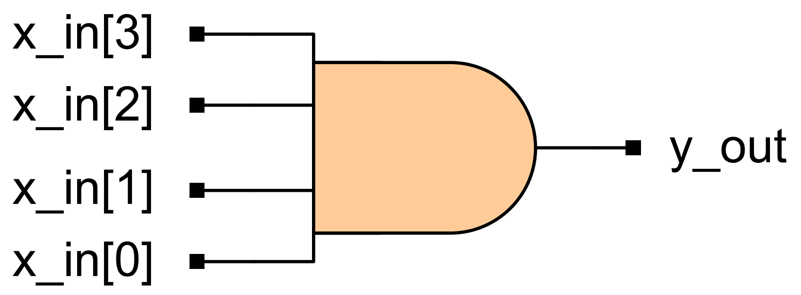
```
module parent_mod;
    wire [3:0] g;
    child_mod    G1 (.sig_c(g[3]), .sig_d(g[2]), .sig_b(g[0]), .sig_a(g[1]));
endmodule

module child_mod (sig_a, sig_b, sig_c, sig_d);
    output          sig_a, sig_b;
    input           sig_c, sig_d;
    //  module description goes here
endmodule
```



Two Alternatives for a 4-Input AND Gate

```
module and4_assign_1 (x_in1, x_in2, x_in3, x_in4, y_out);
    input       x_in1, x_in2, x_in3, x_in4;
    output      y_out;
    assign      y_out = x_in1 & x_in2 & x_in3 & x_in4;
endmodule


module and4_assign_2 (x_in, y_out);
    input       [3:0] x_in;
    output      y_out;
    assign      y_out = & x_in;
endmodule
```

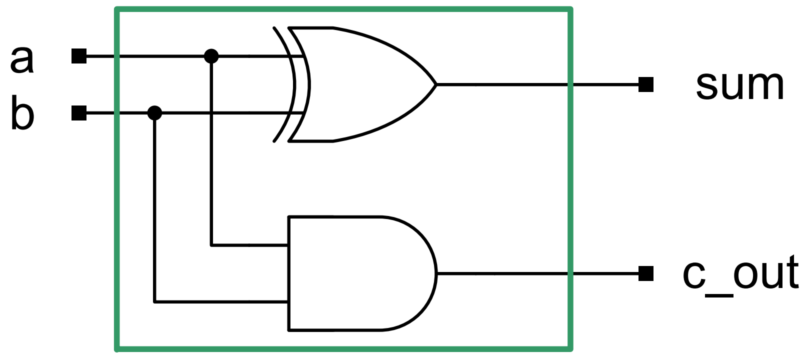
x_in 彼此做 &



RTL Description of a Half Adder

```
module add_half_rtl (sum, c_out, a, b);
    input       a, b;
    output      sum, c_out;
    reg         sum, c_out;

    always @ (a or b)
    begin
            sum = a ^ b;
            c_out = a & b;
    end
endmodule
```

[illegible] assign



D Flip-Flop 正反器

RTL Description of a Flip-Flop

clk → set
clk → rst

設定方向

設定來源 →

```
module flip_flop_rtl (data_in, clk, set, rst, q);
  input          data_in, clk, set, rst;
  output         q;
  reg            q;

  // synchronous set and reset
  always @ (posedge clk)
  begin
      if (rst == 1)
          q = 0;
      else if (set == 1)
          q = 1;
      else
          q = data_in;
  end
endmodule
```

same

同步 from clk

正

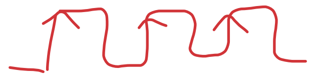
negative 負緣觸發

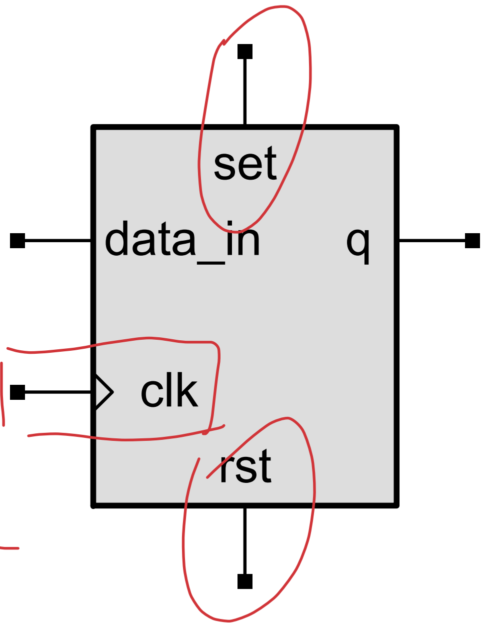
內部變數

在過程中被給值

不綁定

時脈



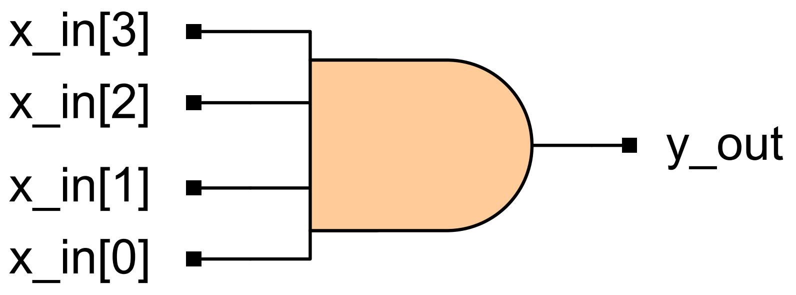


rst > set 優先

Algorithm-Based Description of a 4-Input AND Gate

```
module and4_alg (y_out, x_in);
    input        [3:0] x_in;
    output       y_out;
    reg          y_out;
    integer      k;

    always @ (x_in)   //  x_in[3] or x_in[2] or x_in[1] or x_in[0]
    begin: and_loop
          y_out = 1;
          for (k=0; k <= 3; k = k+1)
               if (x_in[k] == 0)
               begin
                     y_out = 0;
                     disable and_loop;
               end
    end
endmodule
```

break



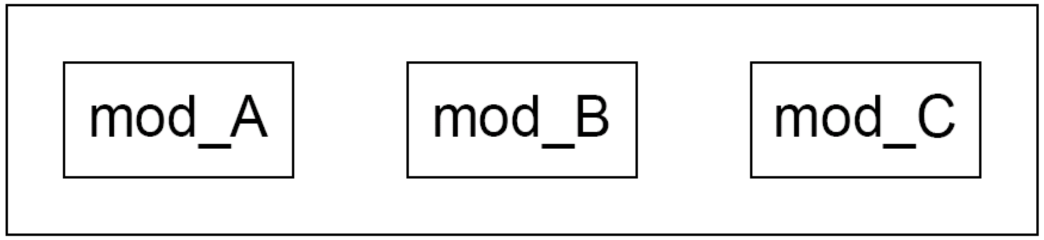
Module Interconnection

- Modules may contain instantiations of other modules.
- A hierarchical structure of modules is implicitly created by nested module instantiations.
- Modules are interconnected by ports.
- Modules may connect to other modules and primitives.
- Models having different levels of abstraction can be mixed in a design hierarchy, e.g. gate-level and behavioral.

Mixed Levels of Abstraction - Ex1

```
module mod_A();          // structural
    nand(); ...
endmodule

module mod_B();          // behavior/algorithm
    always ...
endmodule

module mod_C();          // data flow
    assign X = ...
endmodule
```

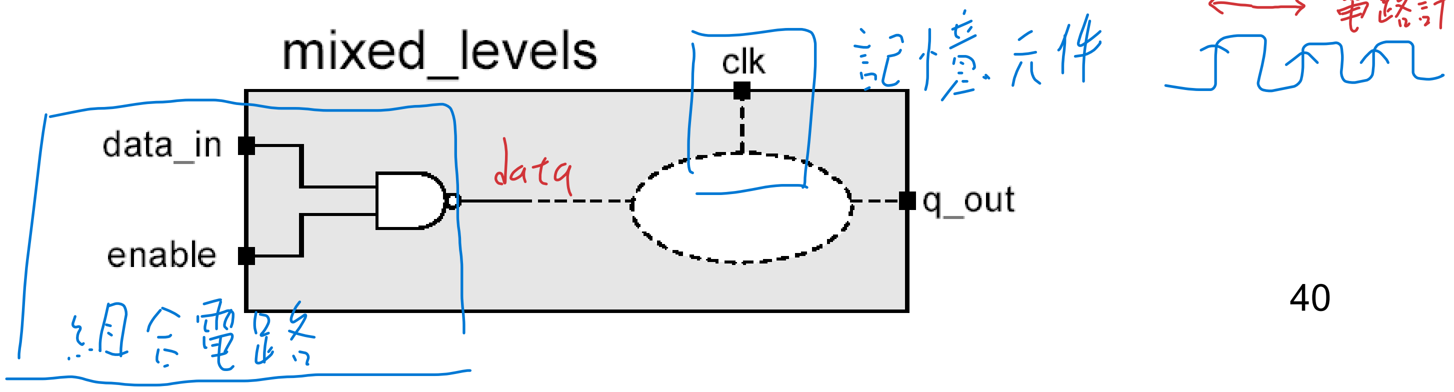
module top_of_design



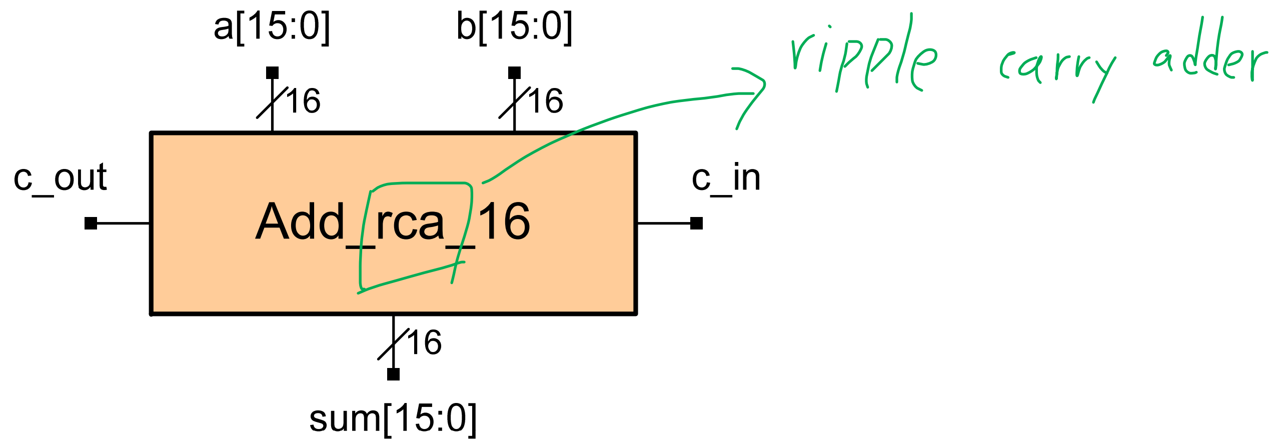
Mixed Levels of Abstraction - Ex2

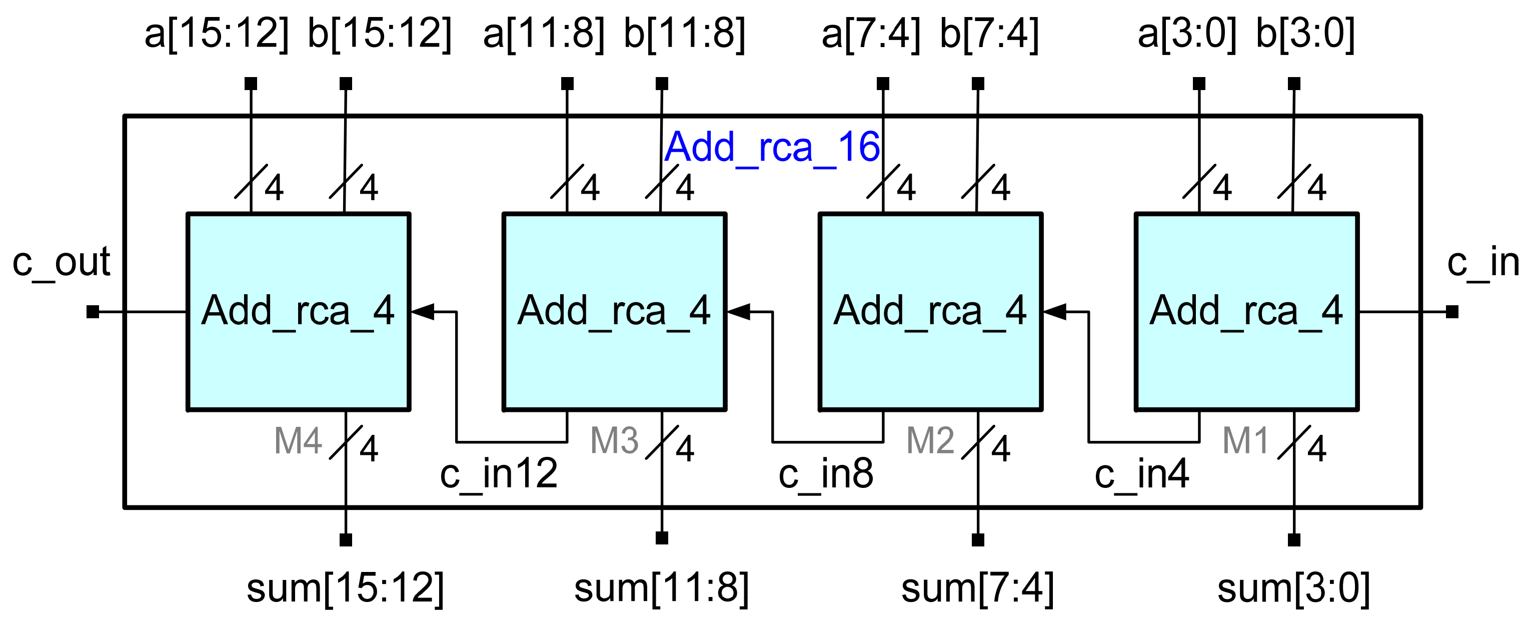
```
module mixed_levels (data_in, enable, clk, q_out);
        input           data_in, enable, clk;
        output          q_out;
不綁定  reg             q_out;
綁定    wire (預設)     data, data_in, enable, clk;

        always @ (posedge clk)            //  behavioral statement
              q_out = data;               //  procedural assignment

        nand (data, data_in, enable);     //  primitive instantiation
endmodule
```

時脈周期讓組合電路計算

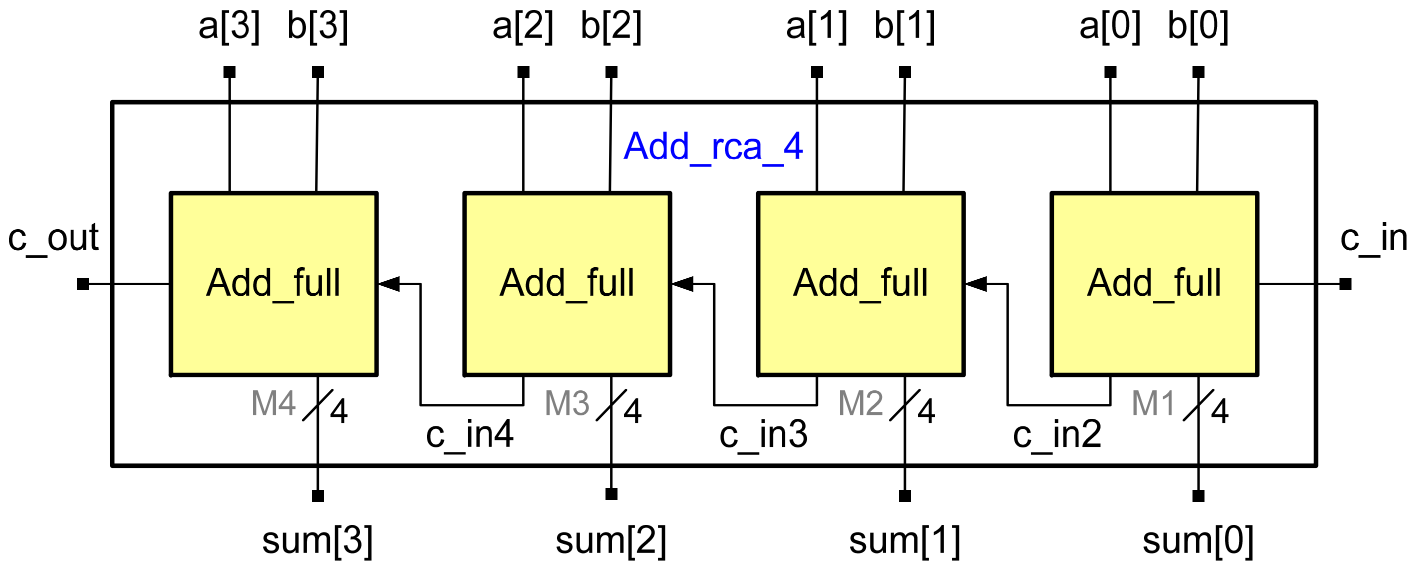


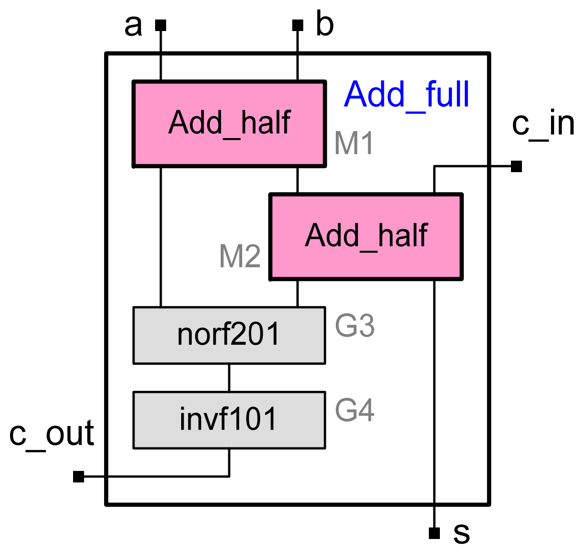
Structured Top-Down Design Methodology (1/2)

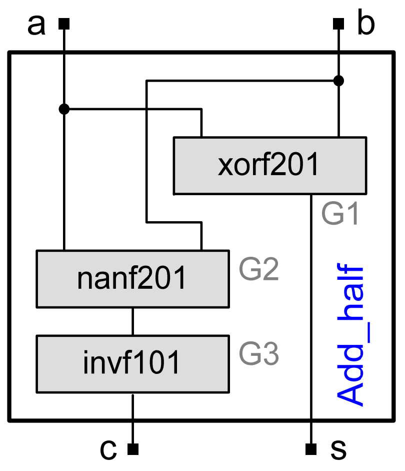




Structured Top-Down Design Methodology (2/2)







Add_half & Add_full

```
module Add_half (s, c, a, b);
    output      s, c;
    input       a, b;
    wire        c_bar;

    xorf201     G1 (s, a, b);
    nanf201     G2 (c_bar, a, b);
    invf101     G3 (c, c_bar);
endmodule
```

```
module Add_full (s, c_out, a, b, c_in);
    output      s, c_out;
    input       a, b, c_in;
    wire        s1, c1, c2, c_out_bar;

    Add_half    M1 (s1, c1, a, b);
    Add_half    M2 (s, c2, s1, c_in);
    norf201     G3 (c_out_bar, c1, c2);
    invf101     G4 (c_out, c_out_bar);
endmodule
```

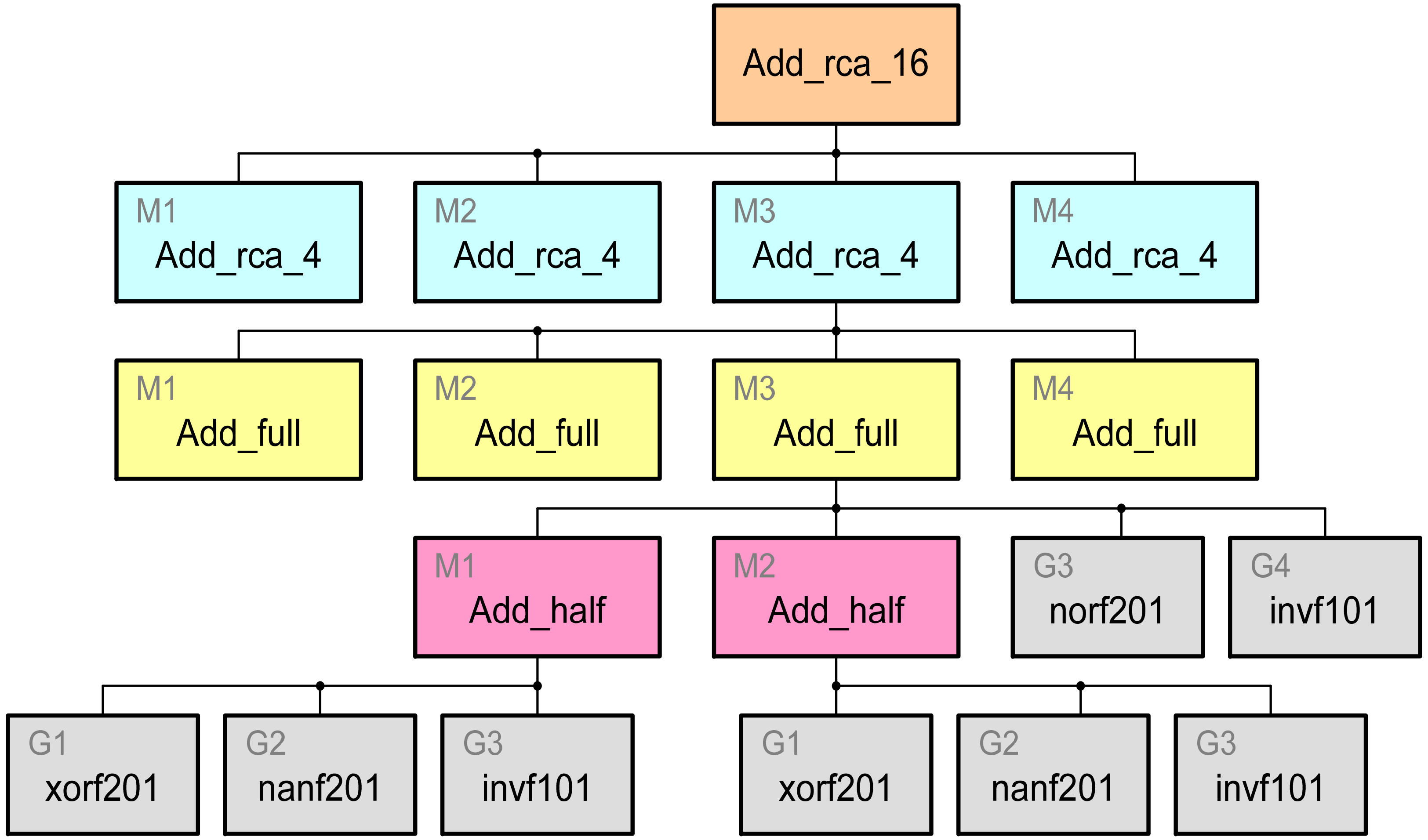
Add_rca_4

```
module Add_rca_4 (sum, c_out, a, b, c_in);
    input       [3:0]   a, b;
    input               c_in;
    output      [3:0]   sum;
    output              c_out;
    wire                c_in2, c_in3, c_in4;

    Add_full    M1  (sum[0], c_in2, a[0], b[0], c_in);
    Add_full    M2  (sum[1], c_in3, a[1], b[1], c_in2);
    Add_full    M3  (sum[2], c_in4, a[2], b[2], c_in3);
    Add_full    M4  (sum[3], c_out, a[3], b[3], c_in4);
endmodule
```

Add_rca_16

```
module Add_rca_16 (sum, c_out, a, b, c_in);
    input        [15:0]   a, b;
    input                 c_in;
    output       [15:0]   sum;
    output                c_out;
    wire                  c_in4, c_in8, c_in12;

    Add_rca_4   M1  (sum[3:0],    c_in4,   a[3:0],    b[3:0],    c_in);
    Add_rca_4   M2  (sum[7:4],    c_in8,   a[7:4],    b[7:4],    c_in4);
    Add_rca_4   M3  (sum[11:8],   c_in12,  a[11:8],   b[11:8],   c_in8);
    Add_rca_4   M4  (sum[15:12],  c_out,   a[15:12],  b[15:12],  c_in12);
endmodule
```

Design Hierarchy



Add_4_assign

```
module Add_4_assign (sum, c_out, a, b, c_in);
    input       [3:0]   a, b;
    input               c_in;
    output      [3:0]   sum;
    output              c_out;

                1bit           4bits
    assign      {c_out, sum} = a + b+ c_in;
endmodule                4bits        1bit
```

Add_16_mix

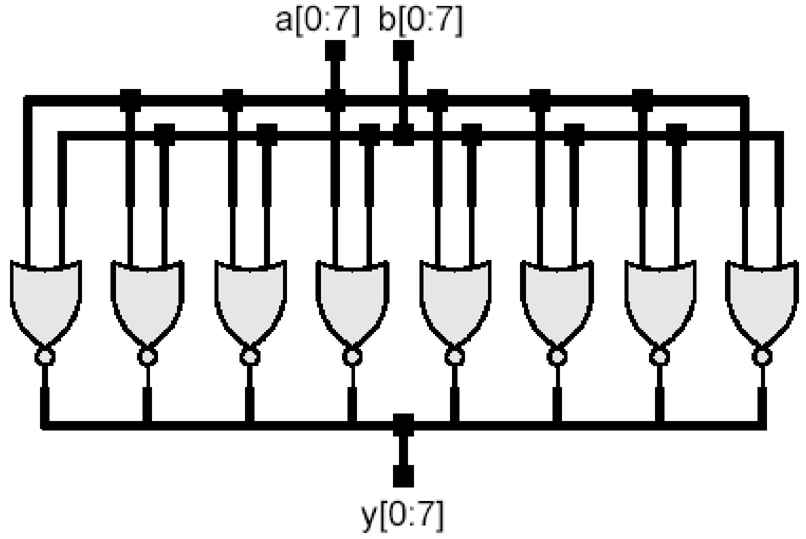
```
module Add_16_mix (sum, c_out, a, b, c_in);
    input         [15:0]  a, b;
    input                 c_in;
    output        [15:0]  sum;
    output                c_out;
    wire                  c_in4, c_in8, c_in12;

    Add_rca_4      M1  (sum[3:0],    c_in4,  a[3:0],    b[3:0],    c_in);
    Add_rca_4      M2  (sum[7:4],    c_in8,  a[7:4],    b[7:4],    c_in4);
    Add_4_assign   M3  (sum[11:8],   c_in12, a[11:8],   b[11:8],   c_in8);
    Add_4_assign   M4  (sum[15:12], c_out,   a[15:12], b[15:12], c_in12);
endmodule
```

[0:7] 左0→右7 給予編號
[7:0] 7→0

Array of Instances of NOR Gates

```
module array_of_nor_1 (y, a, b);
    output      [0:7]    y;
    input       [0:7]    a, b;

    nor  (y[0], a[0], b[0]);
    nor  (y[1], a[1], b[1]);
    nor  (y[2], a[2], b[2]);
    nor  (y[3], a[3], b[3]);
    nor  (y[4], a[4], b[4]);
    nor  (y[5], a[5], b[5]);
    nor  (y[6], a[6], b[6]);
    nor  (y[7], a[7], b[7]);
endmodule
```

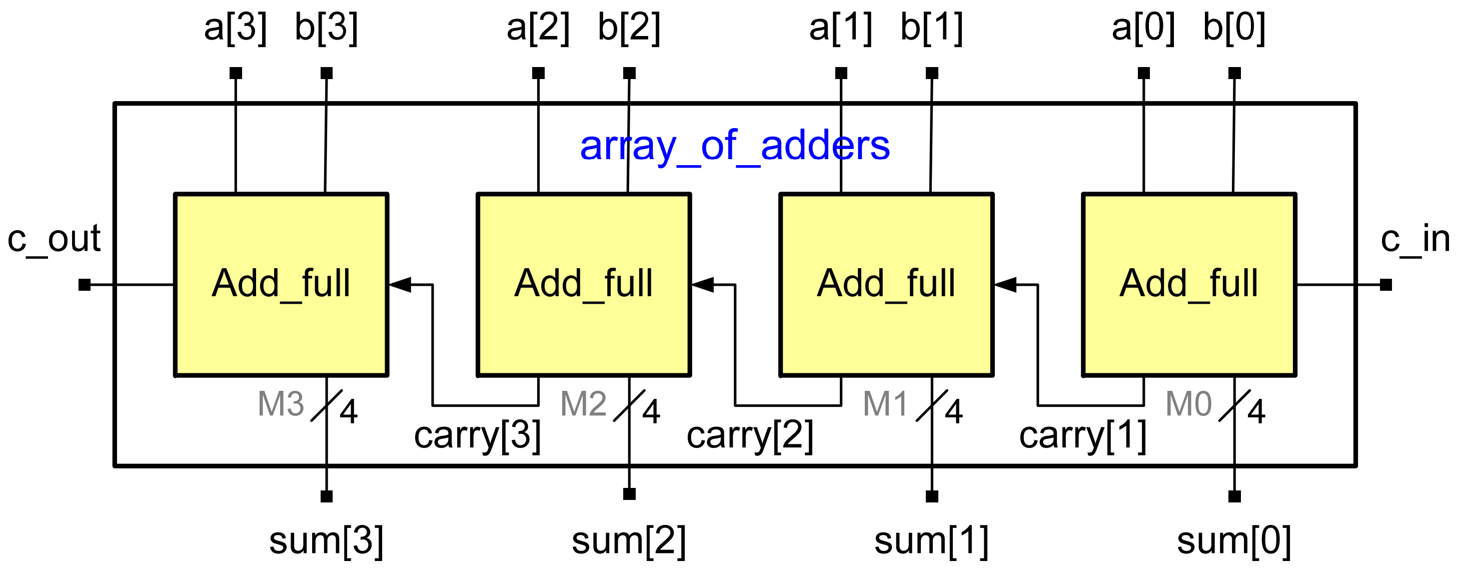
```
module array_of_nor_2 (y, a, b);
    output      [0:7]    y;
    input       [0:7]    a, b;

    nor    M1    [0:7]    (y, a, b);
endmodule
```



A 4-Bit Slice Ripple-Carry Adder

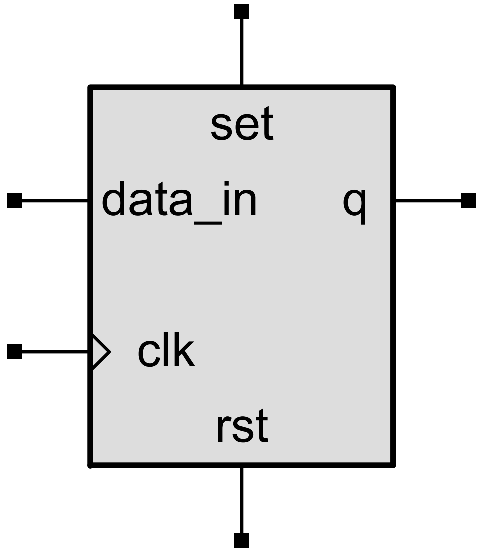
```
Add_full   M0  (sum[0], carry[1], a[0], b[0], c_in);
Add_full   M1  (sum[1], carry[2], a[1], b[1], carry[1]);
Add_full   M2  (sum[2], carry[3], a[2], b[2], carry[2]);
Add_full   M3  (sum[3], c_out,    a[3], b[3], carry[3]);
```

```
Add_full  M [3:0]  (sum, {c_out, carry[3:1]}, a, b, {carry[3:1], c_in});
```



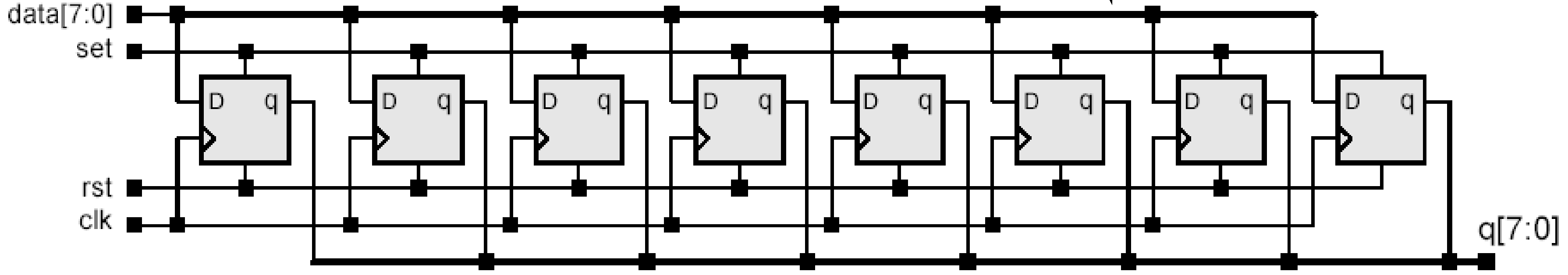
Eight-Bit Register

```
flip_flop      M0  (q[0], data_in[0], clk, set, rst);
flip_flop      M1  (q[1], data_in[1], clk, set, rst);
flip_flop      M2  (q[2], data_in[2], clk, set, rst);
flip_flop      M3  (q[3], data_in[3], clk, set, rst);
flip_flop      M4  (q[4], data_in[4], clk, set, rst);
flip_flop      M5  (q[5], data_in[5], clk, set, rst);
flip_flop      M6  (q[6], data_in[6], clk, set, rst);
flip_flop      M7  (q[7], data_in[7], clk, set, rst);
```



flip_flop

```
flip_flop  M [7:0]  (q, data_in, clk, set, rst);
```

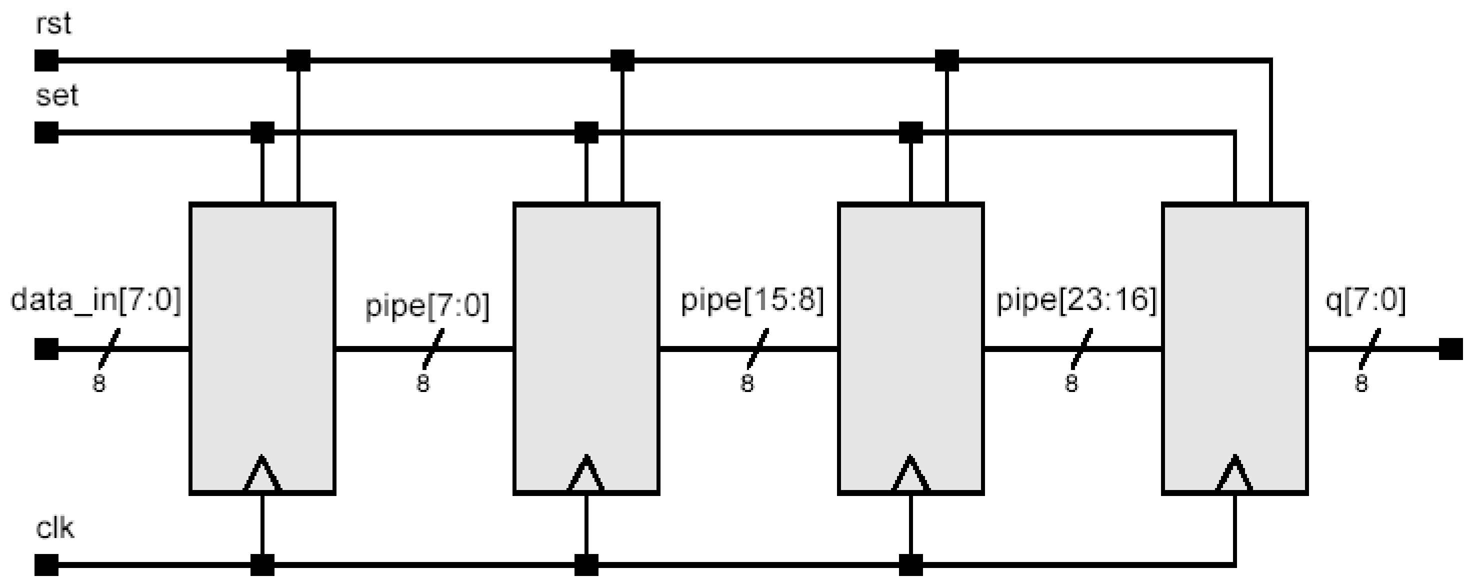


array_of_flops

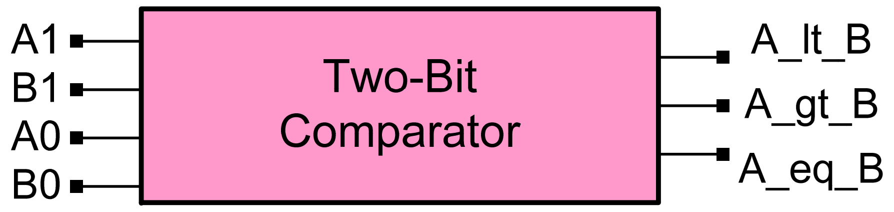
Four-Stage Pipeline of Registers

```
array_of_flops     M0 (pipe[7:0],     data_in,       clk, set, rst);
array_of_flops     M1 (pipe[15:8],    pipe[7:0],     clk, set, rst);
array_of_flops     M2 (pipe[23:16],   pipe[15:8],    clk, set, rst);
array_of_flops     M3 (q[7:0],        pipe[23:16],   clk, set, rst);
```

```
array_of_flops    M [3:0]  ({q, pipe}, {pipe, data_in}, clk, set, rst);
```



Two-Bit Comparator



B_1B_0 \ A_1A_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

A_lt_B

B_1B_0 \ A_1A_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

A_gt_B

B_1B_0 \ A_1A_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

A_eq_B

A_lt_B = A1’B1 + A1’A0’B0 + A0’B1B0

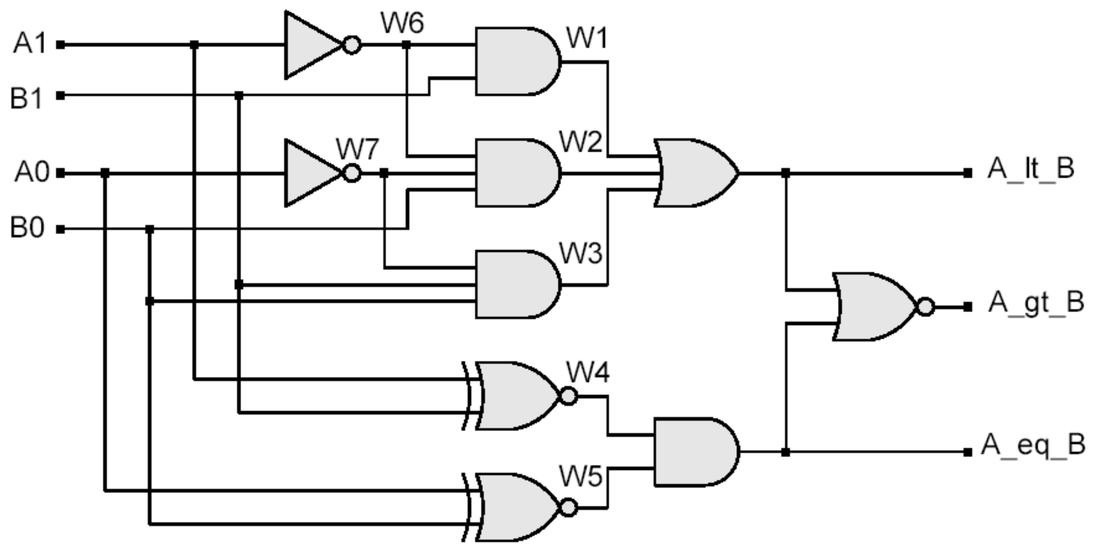
A_gt_B = A1B1’ + A0B1’B0’ + A1A0B0’

A_eq_B = A1’A0’B1’B0’ + A1’A0B1’B0 + A1A0B1B0 + A1A0’B1B0’ = (A1 $\odot$ B1)(A0 $\odot$ B0)

2. X

Structural Description of the Comparator

```
module compare_struct (A1, A0, B1, B0, A_lt_B, A_gt_B, A_eq_B)
    input        A1, A0, B1, B0;
    output       A_lt_B, A_gt_B, A_eq_B;
    wire         w1, w2, w3, w4, w5, w6, w7;

    or   (A_lt_B, w1, w2, w3);
    nor  (A_gt_B, A_lt_B, A_eq_B);
    and  (A_eq_B, w4, w5);
    and  (w1, w6, B1);
    and  (w2, w6, w7, B0);
    and  (w3, w7, B1, B0);
    not  (w6, A1);
    not  (w7, A0);
    xnor(w4, A1, B1);
    xnor(W5, b0, a0);
endmodule
```



“assign” Description of the Comparator (A)

```
module compare_assign_a (A1, A0, B1, B0, A_lt_B, A_gt_B, A_eq_B)
    input       A1, A0, B1, B0;
    output      A_lt_B, A_gt_B, A_eq_B;

    assign      A_lt_B   =
                (~A1) & B1 | (~A1) & (~A0) & B0 | (~A0) & B1 & B0;
    assign      A_gt_B  =
                A1 & (~B1) | A0 & (~B1) & (~B0) | A1 & A0 & (~B0);
    assign      A_eq_B =
                (~A1) & (~A0) & (~B1) & (~B0) | (~A1) & A0 & (~B1) &
    B0 |  A1 & A0 & B1 & B0 | A1 & (~A0) & B1 & (~B0);
endmodule
```

4.

“assign” Description of the Comparator (B & C)

```
module compare_assign_b (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0)
    …
    assign      A_lt_B  = ({A1, A0}  <  {B1, B0});
    assign      A_gt_B  = ({A1, A0}  >  {B1, B0});
    assign      A_eq_B = ({A1, A0} == {B1, B0});
endmodule
```

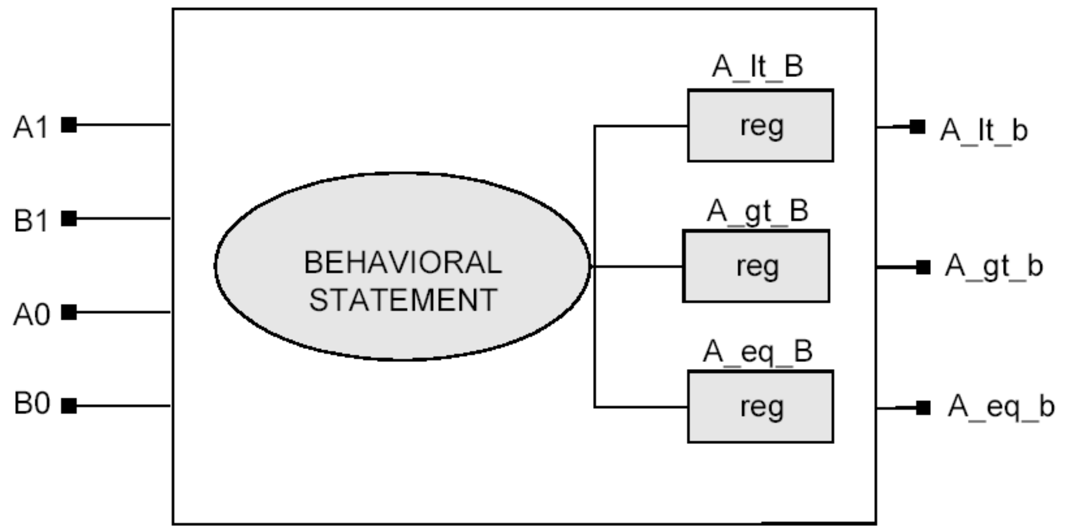
2bits

```
module compare_assign_c (A_lt_B, A_gt_B, A_eq_A, A, B)
    …
    assign      A_lt_B  = (A < B);
    assign      A_gt_B  = (A > B);
    assign      A_eq_B = (A == B);
endmodule
```

ξ.

Algorithmic Description of the Comparator

```
module compare_alg (A_lt_B, A_gt_B, A_eq_B, A, B)
    input          [1:0] A, B;
    output         A_lt_B, A_gt_B, A_eq_B;
    reg            A_lt_B, A_gt_B, A_eq_B;

    always @ (A or B)
    begin
        A_lt_B = 0; A_gt_B = 0; A_eq_B = 0;
        if (A == B)
            A_eq_B = 1;
        else if (A > B)
            A_gt_B = 1;
        else
            A_lt_B = 1;
    end
endmodule
```



Conclusions

- All the $\boxed{\text{five}}$ descriptions of the comparator 運行 are equivalent. 一樣的
- A synthesis tool will create *the same optimal physical realization* for each description.
- The descriptions 差異 vary only in their complexity 複雜度 and readability. 可讀性
- Good descriptions: *compare_assign_c* and *compare_alg*.

0, 1, X, Z
未知 斷開

Representation of Numbers

Number	#Bits	Base	Dec Equiv	Stored
2'b10	2	Binary 2	2	10
3'd5	3	Decimal 8	5	101
3'o5	3	Octal 10	5	101
8'o5	8	Octal	5	00000101
8'ha	8	Hex 16	10	00001010
3'b5 error 2進制沒有5	Not Valid!			
3'b01x	3	Binary	-	01x
12'hx 填滿	12	Hex	-	xxxxxxxxxxxx
8'hz	8	Hex	-	zzzzzzzz
8'b0000_0001	8	Binary	1	00000001
8'b001	8	Binary	1	00000001
8'bx01	8	Binary	-	xxxxxx01
'bz	unsized	Binary	-	z .... z (32 bits)
8'HAD	8	Hex	173	10101101