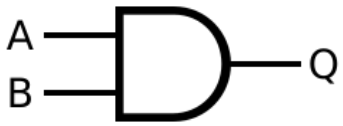
Verilog Tutorial

2024/11/14

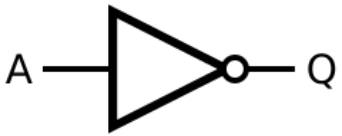
Digital Circuits

- 使用 0 和 1 表示訊號。而真實電路中，通常用低電位 0V 表示 0；高電位（可能為 5V, 3.3V … 等）表示 1。
- 數位電路由邏輯閘組成，有 AND Gate, OR Gate, XOR Gate, NOT Gate …



AND

XOR



NOT

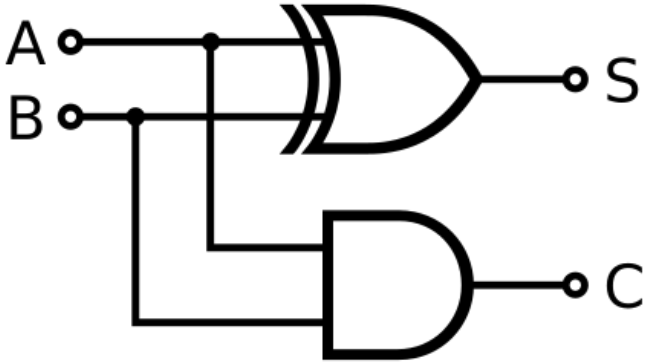
OR

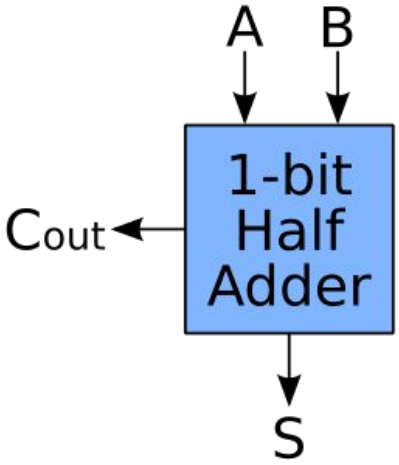
NAND

Gate 的真值表和其他的 Gate 可以看此 https://en.wikipedia.org/wiki/Logic_gate

數位電路範例: 半加器

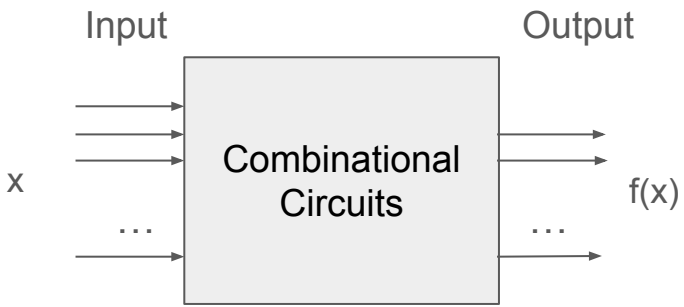
由 XOR 和 AND 組成的一個 1-bit 加法器。

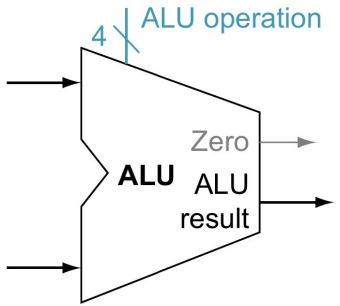




Combinational Logics / Circuits（組合邏輯）

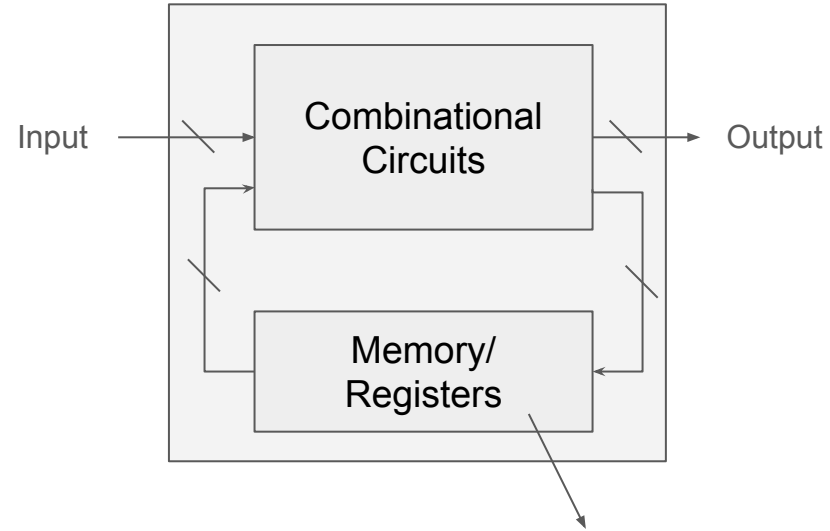
- 輸出完全由輸入決定
- 類似數學中的函數
- 例如：Adder, ALU, Multiplexer…

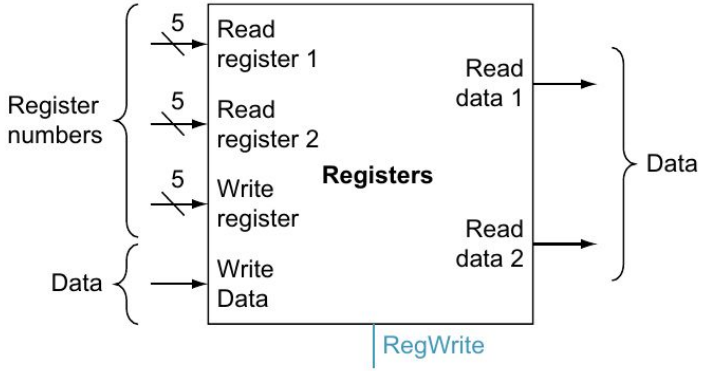


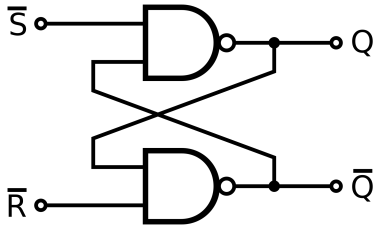


Sequential Logics / Circuits（序向邏輯）

- Outputs are determined by not only the inputs, but also the previous inputs/outputs or states.
- Example: Register File, Latches, Flip-flops



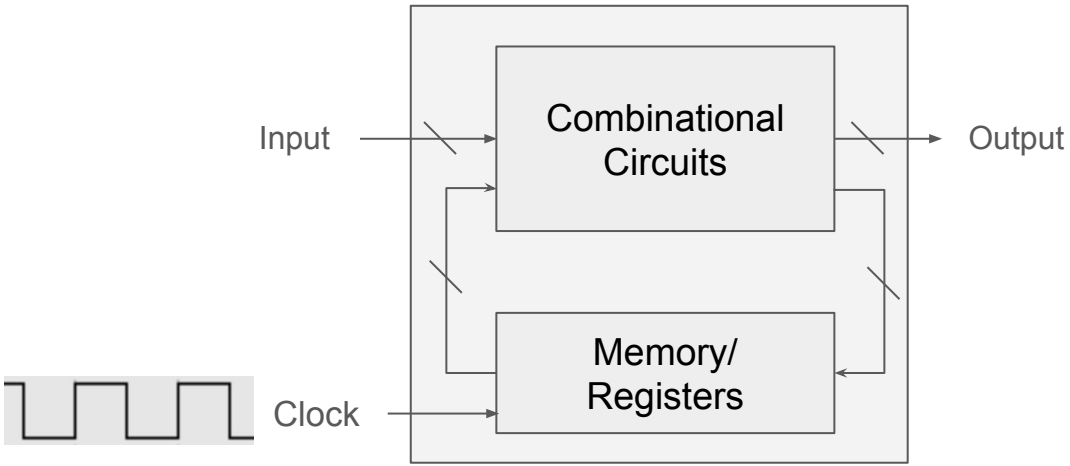




Implemented with latches, or flip-flops

Timing: The Clock

- 序向邏輯中的狀態什麼時候改變呢？ Level-Triggered, Edge-Triggered
- 我們只會用到 Edge-Triggered。可以指定在 Rising Edge, Falling Edge 才做寫入。



邏輯設計參考閱讀

- Computer Organization and Design RISC-V Edition **Appendix A**

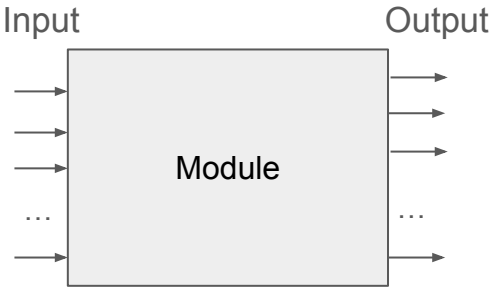
Verilog 簡介

- 硬體描述語言 Hardware Description Language
- 一部份的語句無法被合成（Non-synthesizable），這些語句只能用在 Testbench
- 將硬體透過類似 C 的高階語言的語法，描述暫存器資料的移轉的方式來描述硬體，我們稱為 Register Transfer Level 。故 Verilog 也是一種 RTL Language.
- 透過 Synthesis Tool 可把這樣的硬體轉為較低階的 Gate Level (即由 Gate 和 Netlist 組成）

Modules - The Building Blocks

Verilog 描述的硬體由一個或多個 Module 組成。下面定義一個名為 <Name> 的 Module 的設計。

```
module <Name> (<Port List>);
    <Port Declaration>
    <Variable Declaration>
    <Module Functionality or Structures>
    …
endmodule
```

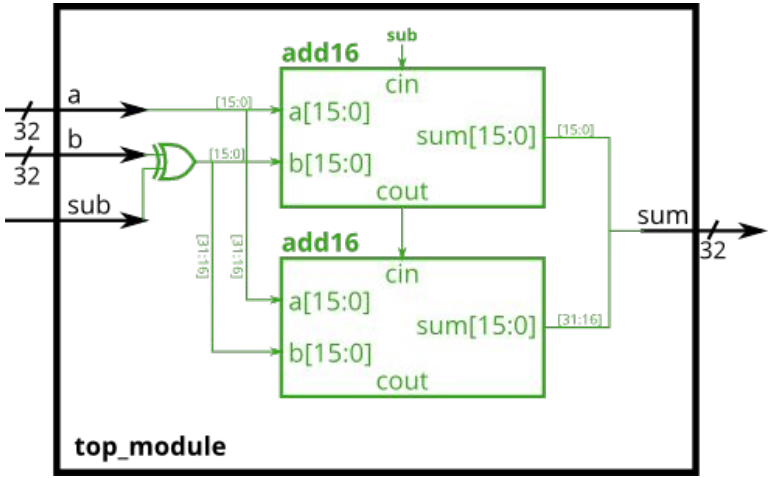


Module Hierarchy

一個 module 內可以有一個或多個 module instances。

```
module top_module(a, b, sub, sum);
    ...
    ...
    add16  add16_1(...);
    add16  add16_2(...);

    ...
endmodule
```



最外層的 module 叫做 top module

Port Declarations (二種方式)

將 Port 型別放在 Port List 中

```
1  module top_module(
2      input a, b, cin,
3      output cout, sum );
4
5      assign {cout, sum} = a + b + cin;
6
7  endmodule
8
```

Port List 只放 Port Name

Port 的型別放在 Module Body

```
1  module top_module(a, b, cin, cout, sum );
2      input a, b, cin;
3      output cout, sum;
4
5      assign {cout, sum} = a + b + cin;
6
7  endmodule
8
```

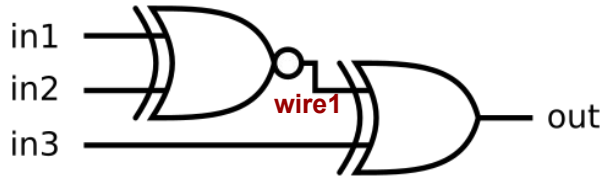
Variable Declaration

```
module top_module (
    input in1,
    input in2,
    input in3,
    output out);

    wire wire1;

    xor xor1(out, wire1, in3);
    xnor xnor1(wire1, in1, in2);

endmodule
```



https://hdlbits.01xz.net/wiki/Exams/m2014_q4g

描述電路三種層次

- Gate Level Modeling
- Dataflow Level Modeling
- Behavior Level Modeling

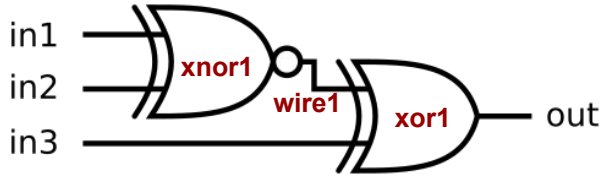
Gate Level

```
module top_module (
    input in1,
    input in2,
    input in3,
    output out);

    wire wire1;

    xor xor1(out, wire1, in3);
    xnor xnor1(wire1, in1, in2);

endmodule
```



直接使用**邏輯閘**來描述電路

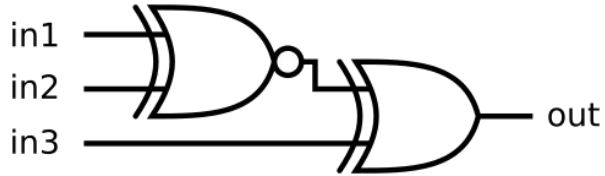
https://hdlbits.01xz.net/wiki/Exams/m2014_q4g

Dataflow Level

```
1  module top_module (
2      input in1,
3      input in2,
4      input in3,
5      output out);
6
7      assign out = in3 ^ ~(in1 ^ in2);
8  endmodule
9
```

使用 **assign** 來指派數值
等號右邊的值會被放到等號左邊的變數。

每次等號右邊的值有變動時，等號左邊的值就會重新被計算。故這樣的敘述稱為 **continuous assignment** 。



https://hdlbits.01xz.net/wiki/Exams/m2014_q4g

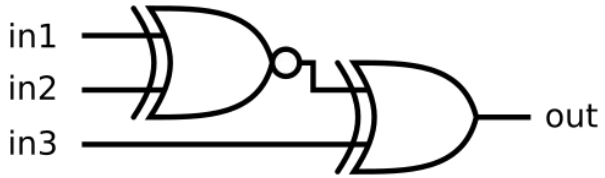
Behavior Level

```
1  module top_module (
2      input in1,
3      input in2,
4      input in3,
5      output reg out);
6
7      always @(*)
8          out = in3 ^ ~(in2 ^ in1);
9
10 endmodule
11
```

```
1  module top_module (
2      input in1,
3      input in2,
4      input in3,
5      output reg out);
6
7      always @(*) begin
8          reg tmp;
9          tmp = in1 ^ in2;
10         out = in3 ^ ~tmp;
11     end
12
13 endmodule
14
```

reg 才能在 always block 內被 assign

@( ) 表示裡面變數有更動時，執行此always block 。* 表示 always block 內所有用到變數。



- 即使用 procedure block 來描述電路。Procedure blocks 包含 always block 和 initial block 。其中 initial block 無法合成電路，只會用在 testbench 。
- 裡面可用 if-else, case… 等語法...
- 若有多行敘述，可用 begin … end 包起來。類似 C 的 { … }

Module Instances

<模組名> <實例名>(<Port 連接>);

```
module Full_Adder(input a, b, cin, output sum, cout);
    wire c1, c2;
    wire s1;

    Half_Adder hadd1(.a(a), .b(b), .sum(s1), .cout(c1));
    Half_Adder hadd2(s1, cin, sum, c2);

    assign cout = c1 | c2;
endmodule
```

使用 Port 名稱來連接

按模組定義 Port 順序連接

模組名

實例名

連接輸入輸出埠

```
module Half_Adder(input a, b, output reg sum, cout);
    always @(*) begin
        sum = a ^ b;
        cout = a & b;
    end
endmodule
```

Data Values

每個 bit 有四種狀態

- 0: 即表示 0 、低電位 0V
- 1: 即表示 1 、高電位 5.5V, 3.3V, …
- x: don’t care, unknown value
- z: High Impedance, Floating (即沒有輸入)

數字表示

4'b1101

4	表示此數值有幾個 bit 。若沒有寫則按所在情境決定或為 32 bits
b	b 表示後面接的數字為二進位。這裡也可以是 h (十六進位), o(八進位), d (十進位).
1101	為數字本身。

50

單純十進位數字. bits 數按所在情境決定, 或為 32 bits

連接運算子 (Concatenation)

{a0, a1, …, an**}**，即在 { } 把要連結的表達式放進去，並用 , (逗號分隔) 。

```
{2'b10, 4'b1001}  // = 101001

{j, k}            // 即把 j, k 二變數連起來
```

若 { ... } 內均為變數名稱，可用在等號左邊

```
assign {c, sum} = a + b;
```

```
always @(*)
    {c, sum} = a + b;
```

複製運算子 (Replication Operator)

`{n{m}}` 即 m 重複 n 次。其中 n 必須是常數。

```
wire [63:0] out;
wire [9:0]  tmp;

assign tmp = {5{2'b01}};            //   0101010101
assign out = {{32{in[31]}}, in};    // Sign-extension
```

其他運算子

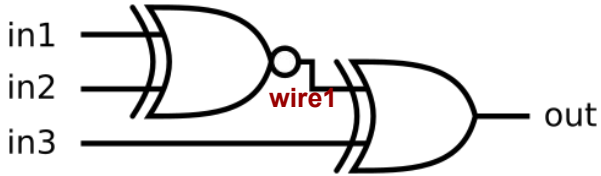
其他 Verilog 運算子類似於 C，包含 `==`, `!=`, `+`, `-`, `*`, `/`, `&`, `|`, `^`, `~`, `>>`, `<<`, `? :`

Data Type: Net

- 連接線
- 分為三種 **wire**, **wand**, **wor**。可使用 wire, wand, wor 宣告。
- input, output 若沒有特別宣告，則為 wire
- 不會儲存狀態。若未有任何元件驅動 wire，其值為 z (浮接狀態)。可透過 **assign** 或連接至 module instance 、gate 的輸出 port 來驅動。但不能在 procedure block 內指定其值。
- 一條 wire 不能被同時驅動 (連接 2 個以上 output 到一個 wire)。
- wand 在同時被驅動時，會把所有接到它的值做 and 運算作為其值。
- wor 在同時被驅動時，會把所有接到它的值做 or 運算作為其值。

Example 1

```
1  module top_module (
2      input in1,
3      input in2,
4      input in3,
5      output out);
6
7      wire wire1;
8      xnor xnor1(in1, in2, wire1);
9      xor  xor1(wire1, in3, out);
10 endmodule
```



Example 2

```
wire out;              // Declare the wire
assign out = a + b;    // Continuous assignment
```

Data Type: Reg

- 暫存器，類似變數，可儲存狀態
- **只能**在 procedure block (always block 或 initial block) 指定其值，預設值為 x
- module 的 output 可以被宣告為 reg

```
reg out;

always @(*) begin
    out = a + b;
end
```

```
module sample (input a, output reg out);
    always @(*) begin
        out = ~a;
    end
endmodule
```

Vector (向量)

- 表示多位元的資料型態。net, reg 均可被定義為 vector 。
- 宣告變數時，在 data type 與 <變數名稱> 之間加入 `[msb:lsb]` ，建議 msb bit 數減 1，而 lsb 為 0。

```
wire [2:0] a;    // 3-bit wire
wire [3:0] b;    // 4-bit wire
reg  [3:0] c;    // 4-bit register

assign a = 3'b110;
assign b[3] = 1'b1;
assign b[1:0] = a[2:1];
```

用 `[n]` 來選擇第 n 個 bit 或用 `[n:m]` 來選擇 n ~ m 連續的 bits 。

Array (陣列)

- 宣告變數時，在 reg, wire 右邊加入 `[a:b]` 用來表示陣列。其中 a 是第一元素的 index ， b 是最後元素的 index 。

```
reg [31:0] registers [0:31];

assign out = registers[read_index];
assign out5 = registers[5];

always @(posedge clk)
    if (write)
        registers[write_index] <= write_data;
```

if-else 敘述

只能用在 Procedure Block

```
if (<條件式>)
    <if body>
else
    <else block>
```

```
if (a == 2'b01) begin
    b = c;
    d = e;
end else begin
    b = 3 + a;
    d = k;
end
```

case 敘述

```
case (<sel>)
    <a>: < <a> == <sel> 時執行 >
    <b>: < <b> == <sel> 時執行 >
    …
    default: < 其他情形執行 >
endcase
```

```
case (sel)
    2'b00: a = b + c;
    2'b01: a = b - c;
    default: a = b;
endcase
```

```
casez (sel)
    2'b00: a = b + c;
    2'b1?: a = b - c;
    default: a = b;
endcase
```

常用在 Multiplexer , ALU

Assignment in Procedure Block

- Blocking assignment: 使用 = 來寫入值
  - 在執行當下即完成寫入
  - 通常**組合邏輯** 時用
- Non-blocking assignment: 使用 <= 來寫入值
  - 在執行當下，只先算好 <= 右邊的值。等到模擬完此 time slot 前再進行寫入
  - 通常是**序向邏輯** 時用

```
always @(*) begin
    if (sel)
        c = a + b;
    else
        c = 0;
end
```

```
always @(posedge clk) begin
    a <= b;
    b <= a;
end
```

組合邏輯

可用 continuous assignment 或 always block 表示組合邏輯

```
module Adder (input [31:0] a, b,
              output [31:0] sum);

    assign sum = a + b;

endmodule
```

```
module Adder (input [31:0] a, b,
              output reg [31:0] sum);

    always @(*)
        sum = a + b;

endmodule
```

blocking assignment

序向邏輯 (Edge-Triggered)

需用 always block 。通常使用 non-blocking assignment 指定暫存器的值

```
module D_FlipFlop (input clk, d, rst,
                   output reg q);

    always @(posedge clk or posedge rst) begin
        if (rst == 1)
            q <= 0;
        else
            q <= d;
    end

endmodule
```

clk 從 0 –> 1 時執行此 always block 。若希望 clk 從 1 –> 0 時執行，則把 **posedge** 改成 **negedge**

non-blocking assignment

Parameters

可定義在 module 內作為常數使用。

- **`parameter`**：可在宣告 module instance 時被修改
- **`localparam`**：不可被修改

```
module Adder(a, b, sum);
    parameter width=32;
    input [width-1:0] a, b;
    output [width-1:0] sum;

    assign sum = a + b;
endmodule
```

實際用 HDLBits 操作示範

https://hdlbits.01xz.net/wiki/Main_Page

Testbench

- Testbench 也是一個 module
- 通常會有 initial block
- 在 Testbench module 裡會把要測試的 module 放進來測試

Initial Block

- 為 Procedure Block 的一種
- 模擬開始時被執行一次
- 用在 Testbench

```
module Testbench;    // Testbench 本身不需要有 I/O port
    reg clk;
    reg rst;
    integer cycles = 0;

    // 用來連接 counter 的 output
    wire [3:0] count;

    // 把 counter 放進來
    Counter counter(.clk(clk), .rst(rst), .count(count));

    initial begin
        $dumpfile("counter.vcd");
        $dumpvars;
        $monitor("cycles: %3d, counter: %b", cycles, count);

        rst = 1; clk = 0;
        #(`CYCLE_TIME) rst = 0; clk = 1;  // begin here
    end

    always @(clk)
        #(`CYCLE_TIME/2)  clk <= ~clk;

    always @(posedge clk) begin
        cycles <= cycles + 1;
        if (cycles == 32)
            $finish;
    end
endmodule
```

Compiler Directives

- **`define**: 定義 Macro，類似 C 的 `#define`
  例: `` `define CLOCK_CYCLE 8 `` 。定義後可用 `` `CLOCK_CYCLE `` 來使用此 macro
- **`undef**: 取消定義 macro
- **`timescale**: 設定時間單位
  例: `` `timescale 1ns/1ps `` 一個時間單位為 1ns，最小精度為 1ps 。
- **`include**: 將外部檔案引入，類似 C 的 `#include`

System Tasks

- 可放在 Procedure Block 內。不可合成，故只用在 Testbench
- 儲存波型圖
  - `$dumpfile` 指定要儲存波型圖的檔案名稱
  - `$dumpvars` 要把哪些變數波型儲存下來
- 印出訊息
  - `$display` 印出訊息至 console，在最後加入 “\n” 換行符號
  - `$write` 印出訊息至 console，不加入換行符號
  - `$monitor` 監測變數變化，當有改變時印出訊息至 console
- 檔案處理: `$fopen, $fclose, $fdisplay, $fwrite`
- 其他
  - `$finish`: 終止模擬

https://www.asic-world.com/verilog/sys_task_func1.html

Delays

#<要 Delay 幾個時間單位>

- Delay 可放在 Procedure Block 內，通常用於 Testbench
- 不可合成，或在合成時被忽略

```
initial begin
    rst = 1;          // 一開始先把 rst 設為 1
    #8  rst = 0;      // 等 8 個時間單位後，再把 rst 設為 0
end
```

實際示範寫 Testbench

```
module Counter(input clk, rst, output reg [3:0] count);
    always @(posedge clk, posedge rst) begin
        if (rst)
            count <= 0;
        else begin
            count <= count + 1;
        end
    end
endmodule;
```

```
module Testbench;    // Testbench 本身不需要有 I/O port
    reg clk;
    reg rst;
    integer cycles = 0;

    // 用來連接 counter 的 output
    wire [3:0] count;

    // 把 counter 放進來
    Counter counter(.clk(clk), .rst(rst), .count(count));

    initial begin
        $dumpfile("counter.vcd");
        $dumpvars;
        $monitor("cycles: %3d, counter: %b", cycles, count);

        rst = 1; clk = 0;
        #(`CYCLE_TIME) rst = 0; clk = 1;  // begin here
    end

    always @(clk)
        #(`CYCLE_TIME/2)  clk <= ~clk;

    always @(posedge clk) begin
        cycles <= cycles + 1;
        if (cycles == 32)
            $finish;
    end
endmodule
```

Verilog 參考網站

- ASIC-World Verilog https://www.asic-world.com/verilog/index.html
- VLSI Verify Verilog Tutorial https://vlsiverify.com/verilog/
- HDLBits https://hdlbits.01xz.net/wiki/Main_Page
  類似 Verilog 版 Leetcode
- EDA Playground https://hdlbits.01xz.net/wiki/Main_Page
  線上提供各樣 EDA Tools 操作環境