Verilog 筆記

結構

```
.
|-Module & Instance
|-Ports & type
|-Simple type
    |-types
    |-assign
    |-always block
|-Sequential & Combinational
    |-blocking
    |-non-blocking
|-Condition & Branch
    |-case
    |-if-else
|-FSM
    |-One-pulse
        |-Moore
        |-Mealy
    |-Example: rrab
|-Test bench
    |-dump infos
    |-test bench sample
|-Tips
```

Module & Instance

Module 宣告方式

```
module moduleName (
    //ports
);
//Code
endmodule
```

Instance 宣告方式

```
moduleName instanceName(
    //ports
);
```

Ports & type

- input, output
- reg
- vector

```
module moduleName(
    output  reg     o_R0,   // output & reg
    output          o_R1,   // output
    output  [3:0]   o_R2,   // output 4-bit vector
    input   [3:0]   i_S0,   // input 4-bit vector
    input           i_S1    // input
);
//code
endmodule
```

Simple types

Warning: reg 不代表真的會合成個 reg 出來

- wire, reg
- or, and, not, nor, nand...
- 直接寫上常數，沒有給定 bit 時，預設會變成 32 bits

assign

當 RHS 有變化時，就會更新 LHS

範例一、單純地給值

```
wire [3:0] const_12;
assign const_12 = 4'd12;
```

範例二、基本運算

```
output [1:0] o_sum;
input i_a, i_b;
assign o_sum = i_a + i_b;
```

範例三、控制條件

```
output [1:0] o_sum;
input reset_n, i_a, i_b;
assign o_sum = (en) ? (i_a + i_b) : (2'b0);
```

範例四、串接

```
output o_carry_out, o_sum;
input i_a, i_b;
assign {o_carry_out, o_sum} = i_a + i_b;
```

always block

以上的 assign 都可被換成 always block

範例一、單純地給值

```
reg [3:0] const_12;
always @(*) begin
    const_12 = 4'd12;
end
```

範例二、基本運算

```
output reg [1:0] o_sum;
input i_a, i_b;
always @(*) begin
    o_sum = i_a + i_b;
end
```

範例三、控制條件

```
output reg [1:0] o_sum;
input reset_n, i_a, i_b;
always @(*) begin
    if (en) begin
        o_sum = i_a + i_b;
    end else begin
        o_sum = 2'b0;
    end
end
```

範例四、串接

```
output reg o_carry_out, o_sum;
input i_a, i_b;
always @(*) begin
    {o_carry_out, o_sum} = i_a + i_b;
end
```

Sequential & Combinational

Sequential

```
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        //Do something
    end else begin
        //Do something
    end
end
```

Combinational

```
always @(*) begin
    //Do something
end
```

在 sequential 的部分會放入 `clk` ，裡頭會使用 `<=` 在 combinational 的部分會放入需要注意的線路，不過可以用 `*` 取代，它會自動幫你處理好，裡頭使用 `=`

Condition & Branch

在寫判斷的時候，如果只利用 `assign` 來做的話，有可能在條件複雜的時候會很容易出錯。這時候可以利用 `always block` 來完成。

```
assign next = (i_s == 2'b00) ? (2'd0) :
                      (i_s == 2'b01) ? (2'd1) :
                      (i_s == 2'b10) ? (2'd2) :
                      (2'd3);
```

變成

```
always @(*) begin
    case (i_s)
        2'b00: begin
            next = 2'd0;
        end
        2'b01: begin
            next = 2'd1;
        end
        2'b10: begin
            next = 2'd2;
        end
        2'b11: begin
            next = 2'd3;
        end
    endcase
end
```

如果有條件判斷，記得要寫滿或補上預設的狀態，所以的接線都應該出現在每個 `begin-end` 裡頭，如下：

```
always @(*) begin
    if (i_s == 2'b00) begin
        a = 2'b00;
        b = 2'b01;
    end else if (i_s == 2'b10) begin
        a = 2'b10;
        b = 2'b11;
    end else begin
        a = 2'b01;
        b = 2'b11;
    end
end
```

只是這樣寫有時候會多寫很多行，比如說 `b = 2'b11` 各自出現兩次，可以換成下面的寫法，他們是等價的：

```
always @(*) begin
    b = 2'b11;
    if (i_s == 2'b00) begin
        a = 2'b00;
        b = 2'b01;
    end else if (i_s == 2'b10) begin
        a = 2'b10;
    end else begin
        a = 2'b01;
    end
end
```

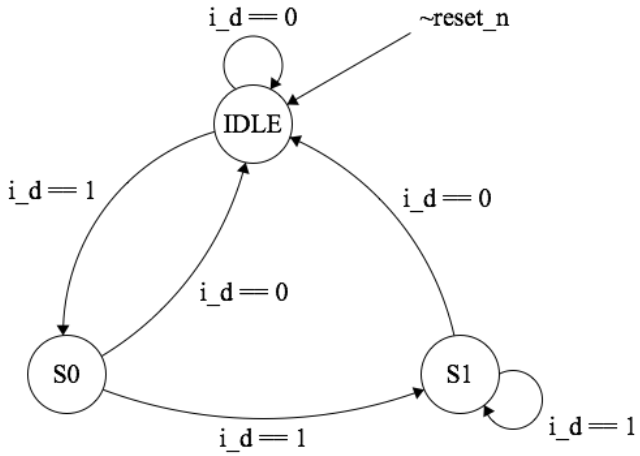
FSM

One-pulse

Moore

以下圖為例，因為要把 `output` 畫進去小圈圈有點塞不下，所以直接指明：

```
IDLE: 1'b0;
S0: 1'b1;
S1: 1'b0;
```

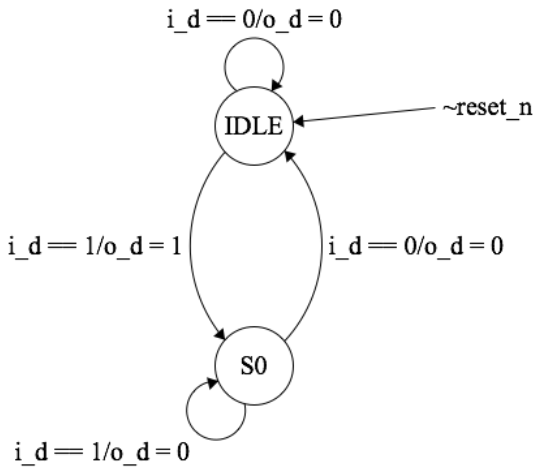



```
module mooreFSM (
    output  reg o_d,
    input       i_d,
    input       clk,
    input       reset_n
);
parameter   IDLE    =   2'b00;
parameter   S0      =   2'b01;
parameter   S1      =   2'b10;
// inner state/next_state
reg [1:0] state, next_state;
//state
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        state   <=  IDLE;
    end else begin
        state   <=  next_state;
    end
end
//next_state
always @(*) begin
    case (state)
```

```
        IDLE: begin
            if (i_d == 1'b1) begin
                next_state = S0;
            end else begin
                next_state = IDLE;
            end
        end
        S0: begin
            if (i_d == 1'b1) begin
                next_state = S1;
            end else begin
                next_state = IDLE;
            end
        end
        S1: begin
            if (i_d == 1'b1) begin
                next_state = S1;
            end else begin
                next_state = IDLE;
            end
        end
        default:    begin
            next_state = IDLE;
        end
    endcase
end
//output
always @(*) begin
    o_d = 1'b0;
    case (state)
        S0: begin
            o_d = 1'b1;
        end
    endcase
end
endmodule
```

Mealy

以下圖為例：




```
module mealyFSM (
    output  reg o_d,
    input       i_d,
    input       clk,
    input       reset_n
);
parameter   IDLE    =   2'b00;
parameter   S0      =   2'b01;
// inner state/next_state
reg [1:0] state, next_state;
reg o_d_next;
//state
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        state   <=  IDLE;
    end else begin
        state   <=  next_state;
    end
end
//output
always @(*) begin
    o_d_next = 1'b0;
```

```
    case (state)
        IDLE: begin
            if (i_d == 1'b1) begin
                o_d_next = 1'b1;
            end
        end
    endcase
end
//next_state
always @(*) begin
    next_state = IDLE;
    case (state)
        IDLE: begin
            if (i_d == 1'b1) begin
                next_state = S0;
            end
        end
        S0: begin
            if (i_d == 1'b1) begin
                next_state = S0;
            end
        end
    endcase
end
// D-FF stores output
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        o_d <= 1'b0;
    end else begin
        o_d <= o_d_next;
    end
end
endmodule
```

rrab

```
/*
request 給定 2'b00, 2'b01, 2'b10 的時候會丟回 2'b00, 2'b01, 2'
b10

如果 request 給的是 2'b11 則會根據之前的狀態做改變
如果之前是 2'b00, 2'b01 -> 2'b10
```

```
如果之前是 2'b10->2'b01

假設 request 的那兩個 bit 代表的是工作的話
1st bit -> job1
2ed bit -> job2
2'b00->沒有任何工作
2'b01->job1 來了
2'b10->job2 來了
2'b11->job1, job2 同時來了
所以兩個人輪流做就是 2'b01<->2'b10 這樣
*/
module rrab (
    output reg [1:0] grant,
    input [1:0] request,
    input clk,
    input reset_n
);
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;


    reg [1:0] grant_i;
    reg last_winner, winner;

    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            last_winner <= 1'b0;
            grant       <= S0;
        end else begin
            last_winner <= winner;
            grant       <= grant_i;
        end
    end

    always @(*) begin
        case(request)
            S0: begin
                grant_i = S0;
            end
            S1: begin
                grant_i = S1;
```

```
            end
            S2: begin
                grant_i = S2;
            end
            S3: begin
                if (last_winner == 1'b0) begin
                    grant_i = S2;
                end else begin
                    grant_i = S1;
                end
            end
        endcase
    end

    always @(*) begin
        if (grant_i == S0) begin
            winner = last_winner;
        end else begin
            winner = grant_i[1];
        end
    end
endmodule
```

Test bench

dump infos

```
$monitor(...);
//當 ... 裏面的值有變動時就會印出
$display(...);
//不管 ... 裡面的值有沒有變動都會印出
//用法和 printf 很類似
```

Test bench 大概寫法

```
module test;
// 因為 input 的值在這裡要給，所以用 reg
reg i_data, clk, reset_n;
// output 則是由 instance 傳出來，用 wire 即可
wire o_data;

initial begin
    //do something
    $finish;
end

always begin
    #5 clk = ~clk;
    // delay 5 units 以後 toggle
end
endmodule
```

Tips

就是以前踩過的雷ＯＴＺ

ARS

`>>>` 是 `arithmetic right shift` 在 Verilog 裡的運算符號，功能如下：

```
4-bit: 0111 >>> 1 = 0011
4-bit: 1100 >>> 1 = 1110
```

所以很直覺的就拿來用的後果卻是變成這樣：

```
4-bit: 0111 >>> 1 = 0011
4-bit: 1100 >>> 1 = 0110
```

因為要被算的數沒有指定為 `signed` ，所以被當成 `unsigned` 處理

對於 `unsigned` 的數就直接用單純的 logic shift（ `>>` ）來算 因此答案就不會是我們原本想的那樣了。

如果想要讓他以原本的想法動的話，要改成

```
$signed(1100) >>> 1 = 1110;
```

number???

有時候總會需要給一些常數，或是加減某些常數的時候，在這樣的情況下有機會寫成：

```
a = b + 1;
```

因為編譯器不知道你的 `1` 的長度是多少，預設會是 `32-bit` 的數字。所以在某些時候會因為這個位數的問題發生悲劇。

latch...

只要不小心， latch 會死命的追著人跑ＯＴＺ

`case` 或是 `if-else` 沒寫完整，因為編譯器不知道你的用途，也沒辦法就這樣空白著，只好幫你弄一個 `latch` 出來。

另外，把自己接到自己身上也會發生 `latch` 。

bits???

bit 的數目一定要對好，一定要對好，一定要對好。

該開到 4 bits 的，就應該要確定真的有寫出 4 bits 。

個人慘痛經驗是宣告變數的時候把 3 bits 的變數宣告成 4 bits 這樣其實還好，更可怕的是同時也把 4 bits 宣告成 3bits 了.....

如果不夠注意的話，就很容易變成潛在的 bug（這東西花了三個小時的抓蟲時間OTZ）

最後更新：hydai@6/7