Verilog Implementation Of Arithmetic Logic Unit (ALU)

實驗目的

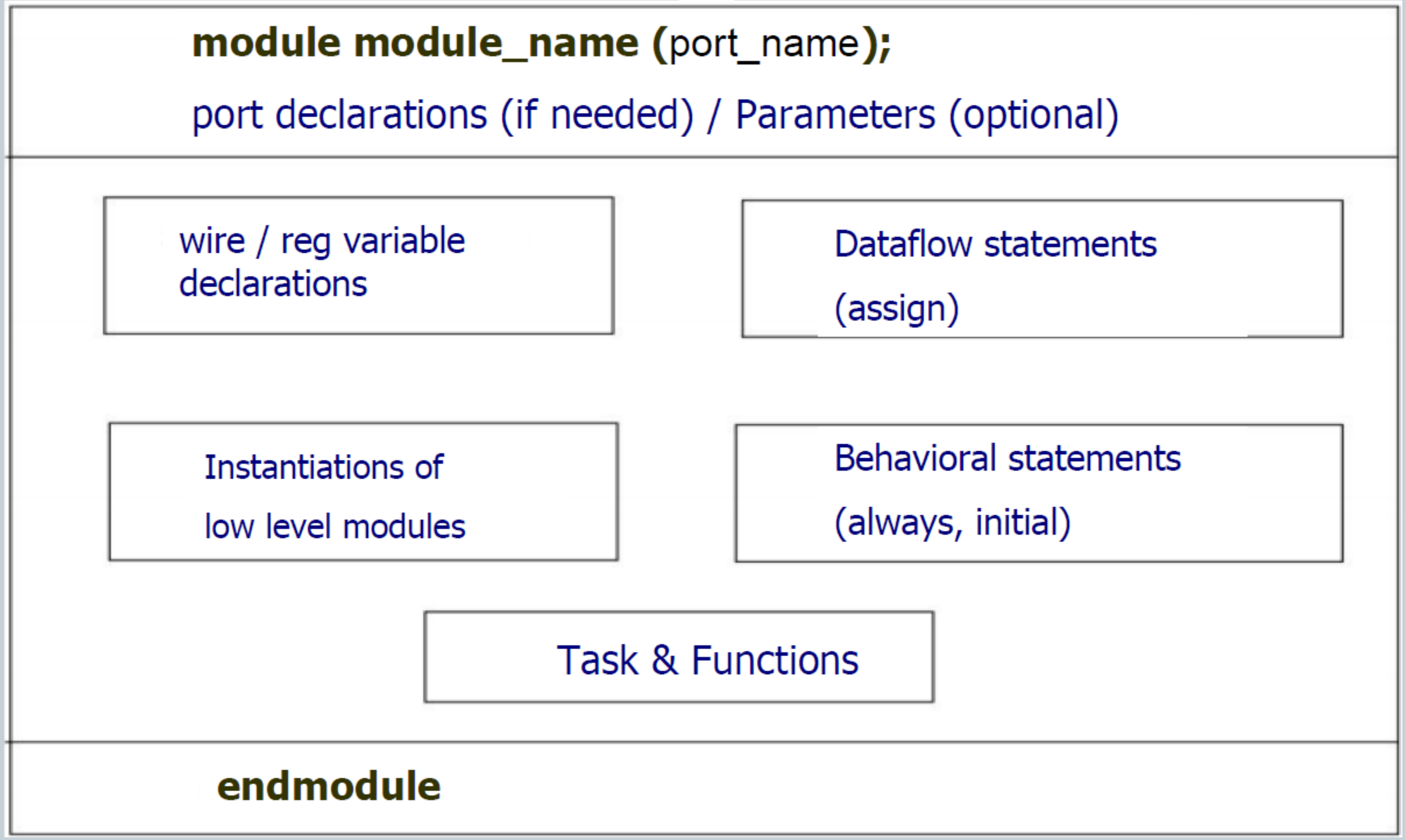
1. Verilog的基本認識
2. 認識MIPS processor的 control
3. Modelsim驗證硬體行為
4. 以Verilog實作 Arithmetic Logic Unit (ALU)

Verilog的基本認識

Verilog的介紹

- Module
- Module Level
- Four Value Logic
- Number representation
- Operators
- Verification

Module



module module_name (port_name**);**

port declarations (if needed) / Parameters (optional)

wire / reg variable declarations

Dataflow statements (assign)

Instantiations of low level modules

Behavioral statements (always, initial)

Task & Functions

endmodule

Module

- Port (input、output、inout)
  - 在Verilog中預設的port其宣告型態為wire，因此假若在port的宣告中只有宣告output、input或是inout，則皆將其視為wire，然而假如需要將訊號的值儲存起來，就要將port的種類宣告為reg。
  - input 與inout型態的port不能被宣告為reg。因為reg會儲存值，而輸入訊號只是表現出外來訊號的改變情況，所以不能儲存其值。

Module

- Data Type
  - nets
    - wire sum;
  - register
    - reg sum;
  - vectors
    - wire [7:0] bus;
    - reg [31:0] busA;
  - integers
    - integer counter = 1;
  - real
  - parameters
    - parameter width = 32;
    - reg [width-1:0] busB;
  - arrays
    - reg [7:0] mem [0:1023]

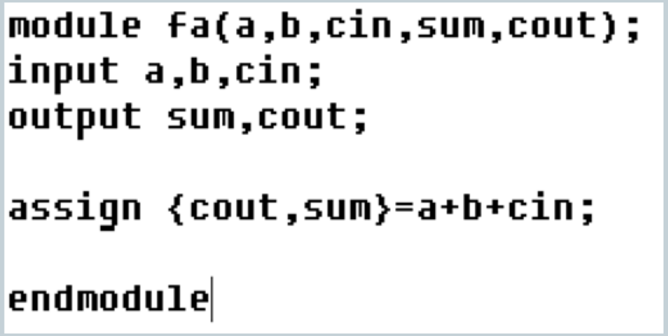
（長度1K，寬度8位元的記憶體）

Module

- Port Mapping
  - Implicit Positional Mapping

    fa fa0(A[0], B[0], cin, SUM[0], c1);

    Ordering matters!



```
module fa(a,b,cin,sum,cout);
input a,b,cin;
output sum,cout;

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

endmodule
```

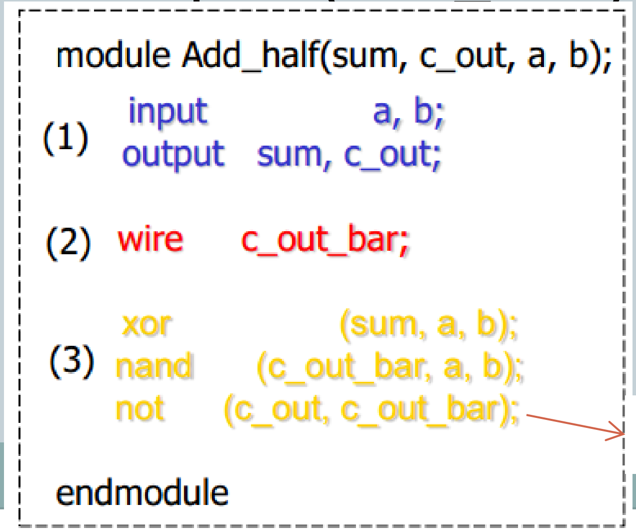
  - Explicit Named Mapping

    fa fa0(.a(A[0]), .b(B[0]), .cin(cin), .cout(c1), .sum(SUM[0]));

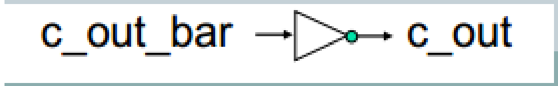
    Ordering doesn’t matter!

Module

- module module_name (port_name);
  - **port declaration**
  - **data type declaration**
  - **module functionality or structure**
- endmodule
- Example: (add_half)



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

(2)    wire      c_out_bar;

       xor               (sum, a, b);
(3)    nand      (c_out_bar, a, b);
       not       (c_out, c_out_bar);

     endmodule
```



Module Level

- Behavior level
  - 此為Verilog中的最高層次，在這個層次中我們只需要考慮模組的功能或函數，不用考慮硬體方面詳細的電路是如何設計。在這個層次的設計工作就類似寫C語言一樣。
- Dataflow level
  - 在這個層次中必須要指明資料處理的方式。設計的人要說明資料如何在暫存器中儲存與傳送，在設計裡資料處理的方式。
- Gate level
  - 在這層次中模組是由邏輯閘連接而成，在這層次的設計工作就好像以前用描繪邏輯閘來設計線路一樣。

Behavioral level

- always敘述
  - always敘述的觀念有如監督程式一般，隨時監看著輸出入埠訊號的變化，然後告知模組內部進行相關的處理。
  - 語法

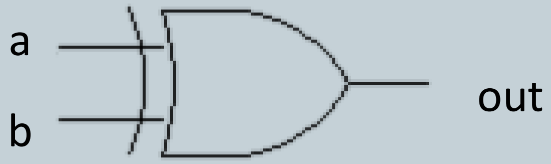
```
always@(event_expression)
   statement
```

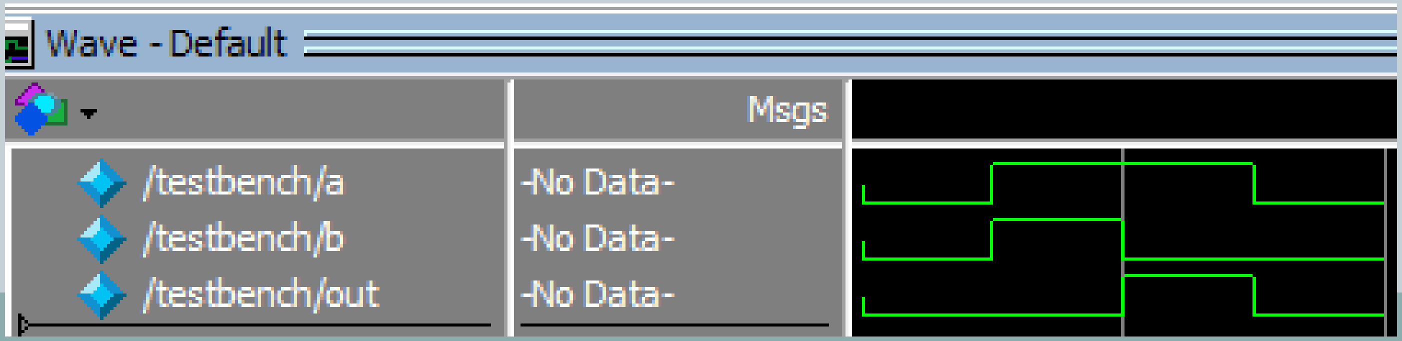
```
always@(event_expression)
begin
   statements
end
```

Behavioral level

- always@ 敘述
  - 範例: 互斥閘

```
always@(a , b)
begin
            out =a ^ b;
end
```



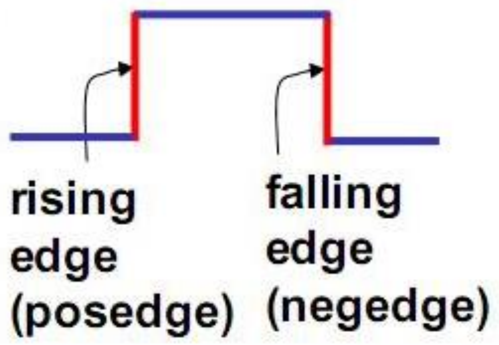


	Msgs
/testbench/a	-No Data-
/testbench/b	-No Data-
/testbench/out	-No Data-

Behavioral level

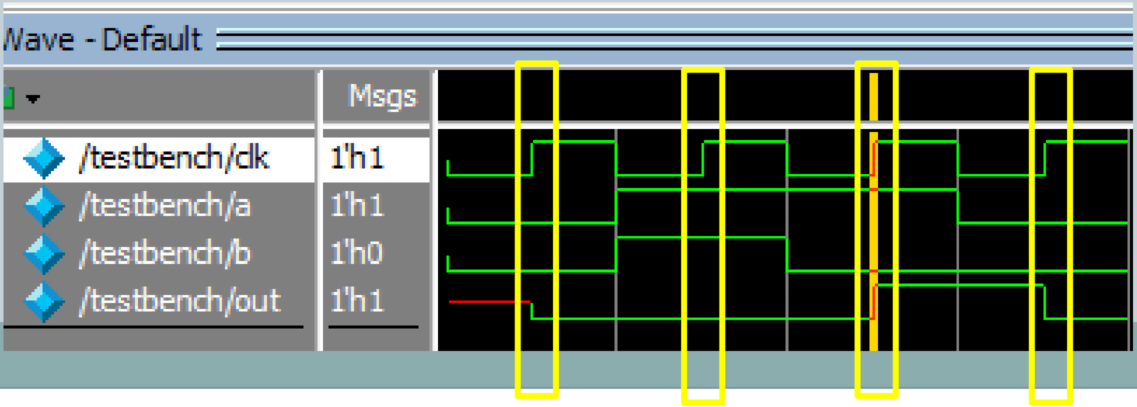
- always@ 敘述（邊緣觸發）

```
always @  ( [posedge or negedge] event)
begin
  . . . statements . . .
end (sequential circuit)
```



Rising edge or positive edge (posedge)
Falling edge or negative edge (negedge)

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



```
/testbench/clk   1'h1
/testbench/a     1'h1
/testbench/b     1'h0
/testbench/out   1'h1
```

Behavioral level

- if-else敘述
  - 可用來進行訊號值的判斷，後根據判斷結果執行相關處理
  - 語法

```
if (expression)
        statement
else    statement
```

```
if (expression)
begin
        statements
end
else
begin
        statements
end
```

Behavioral level

- if-else敘述
  - 範例：優先權編碼器(priority encoder)

```
always(x , y , z)
begin
        if (z)                              // highest
                out = result4;
        else if (y)
                out = result3;
        else if (x)
                out = result2;
        else
                out = result1;         //lowest
end
```

Behavioral level

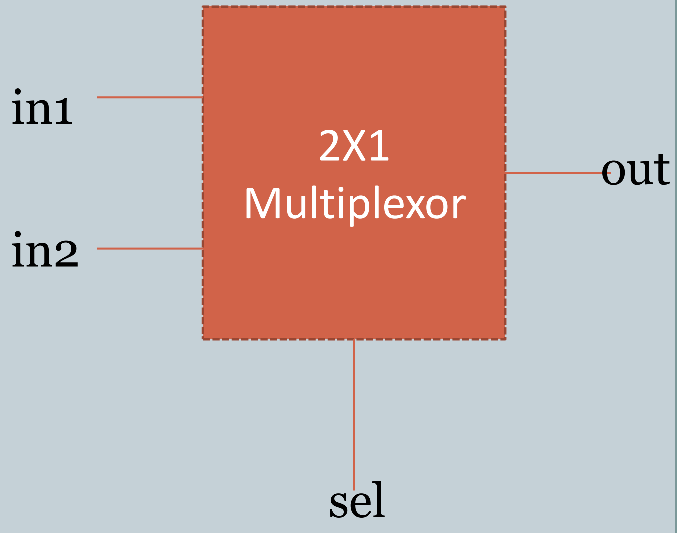
- case 敘述
  - Case敘述唯一多路分支選擇的敘述。
  - 語法

```
case (expression)
        alter_1, alter_2: stm_1;
        alter_3: stm_2;
        …
        default: default_stm;
endcase
```

Behavioral level

- case 敘述
  - 範例: 2X1 Multiplexor

```
module mux21 (in1, in2, sel, out)
input       in1, in2 ,sel;
output      out;
reg         out;

            always@(in1 or in2 or sel)
            begin
                        case (sel)
                                    1'b0: out = in1;
                                    1'b1: out = in2;
                        endcase
            end
endmodule
```



Behavioral level
Example : adder

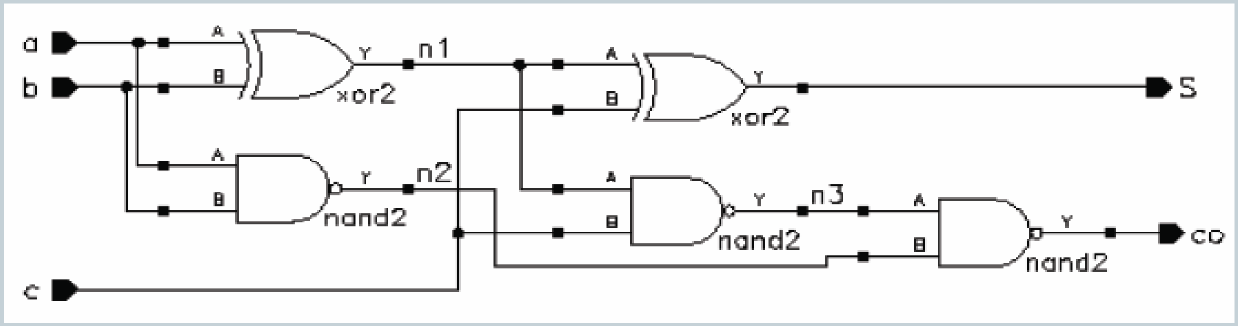
```
module add (co, s, a, b, c)
  input a, b ,c ;
  output co, s ;

  always (*) begin
    {co, s} = a + b + c;
  end
endmodule
```

Dataflow level
Example : adder

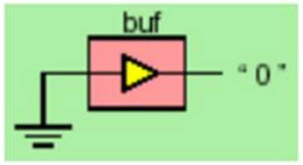
```
module add (co, s, a, b, c)
  input a, b ,c ;
  output co, s ;

  assign {co, s} = a + b + c;
endmodule
```

Gate-level
Example : adder

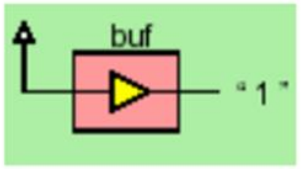
```
module add (co, s, a, b, c);
    input a, b ,c ;
    output co, s ;
    wire n1,n2,n3;
    xor(n1, a, b) ;
    xor(s, n1, c) ;
    nand(n2, a, b) ;
    nand(n3,n1, c) ;
    nand(co, n3,n2) ;
endmodule
```



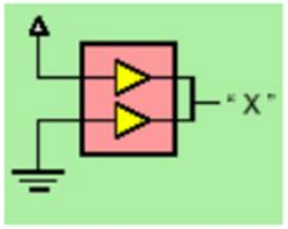
Four Value Logic

- 

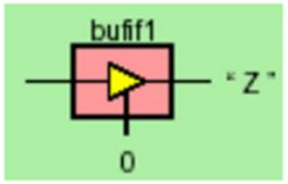
0: logic 0 / false

- 

1: logic 1 / true

- 

X: unknown logic value

- 

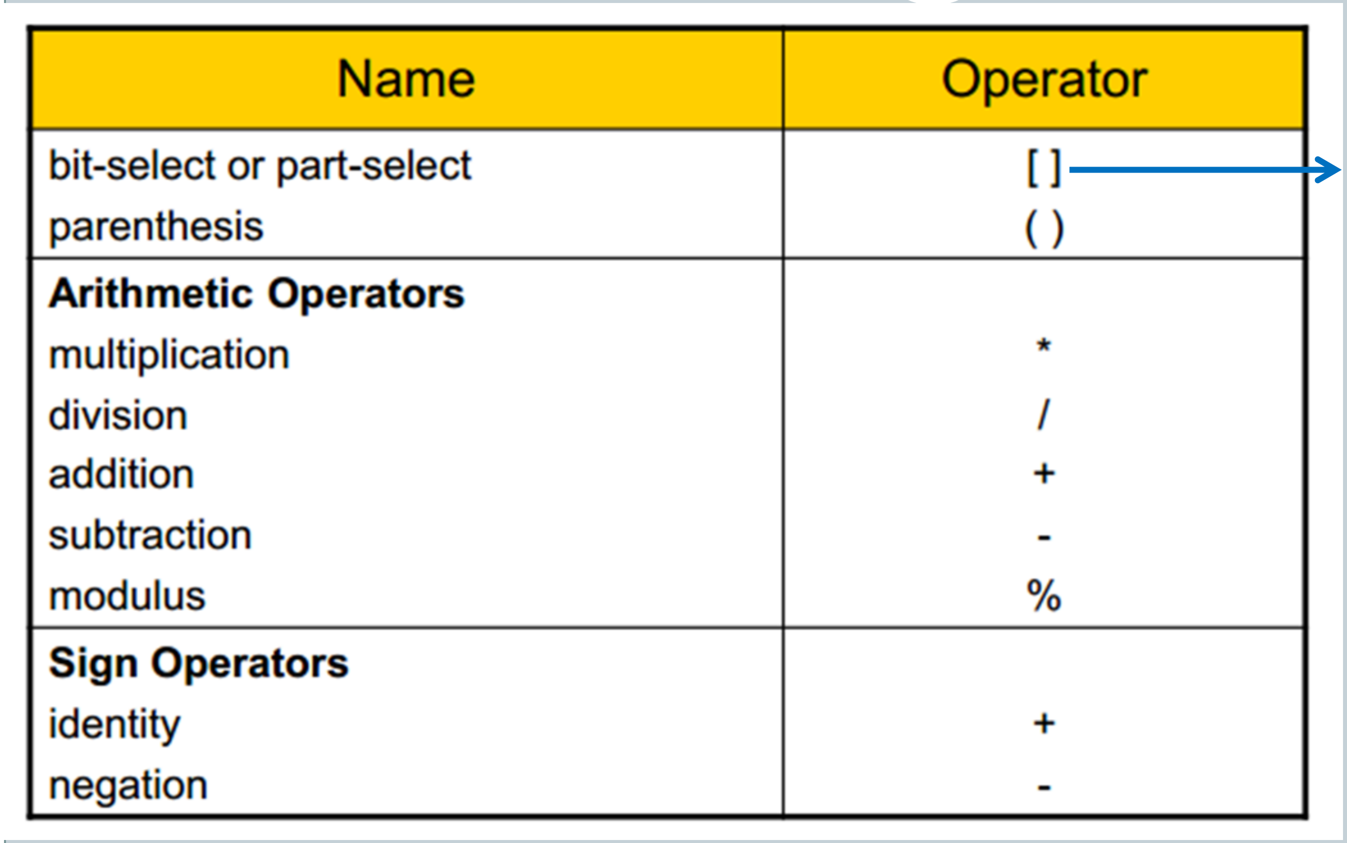
Z: high-impedance

Number representation

- **規定長度之數字(Sized numbers)**
  - 規定長度之數字以<size>'<base format> <number>來表示。
    - <Size>是以十進未來表示數字的位數(Bits)，
    - <base format>是用以定義此數字為十進位來表示數字的位數(Bits)。
      - 十進位('d或'D)
      - 十六進位('h或'H)
      - 二進位('b或'B)
      - 八進位('o或'O)
    - Example:

```
4'b1111   //  這是一個  4-bit  二進位
12'habc   //  這是一個 12-bit  十六進位數
16'd255   //  這是一個 16-bit  十進位數
```

Operators (1/3)

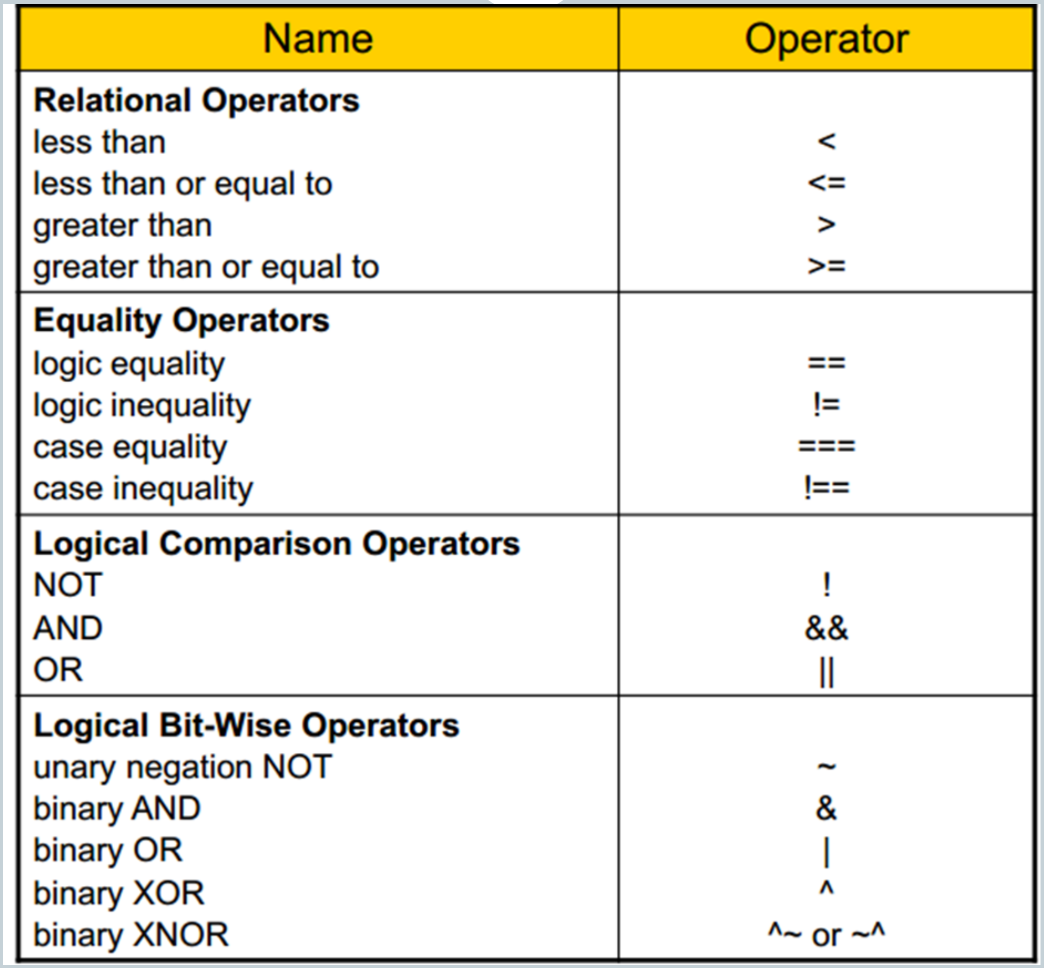


Name	Operator
bit-select or part-select	[ ]
parenthesis	( )
Arithmetic Operators	
multiplication	*
division	/
addition	+
subtraction	-
modulus	%
Sign Operators	
identity	+
negation	-

Example:

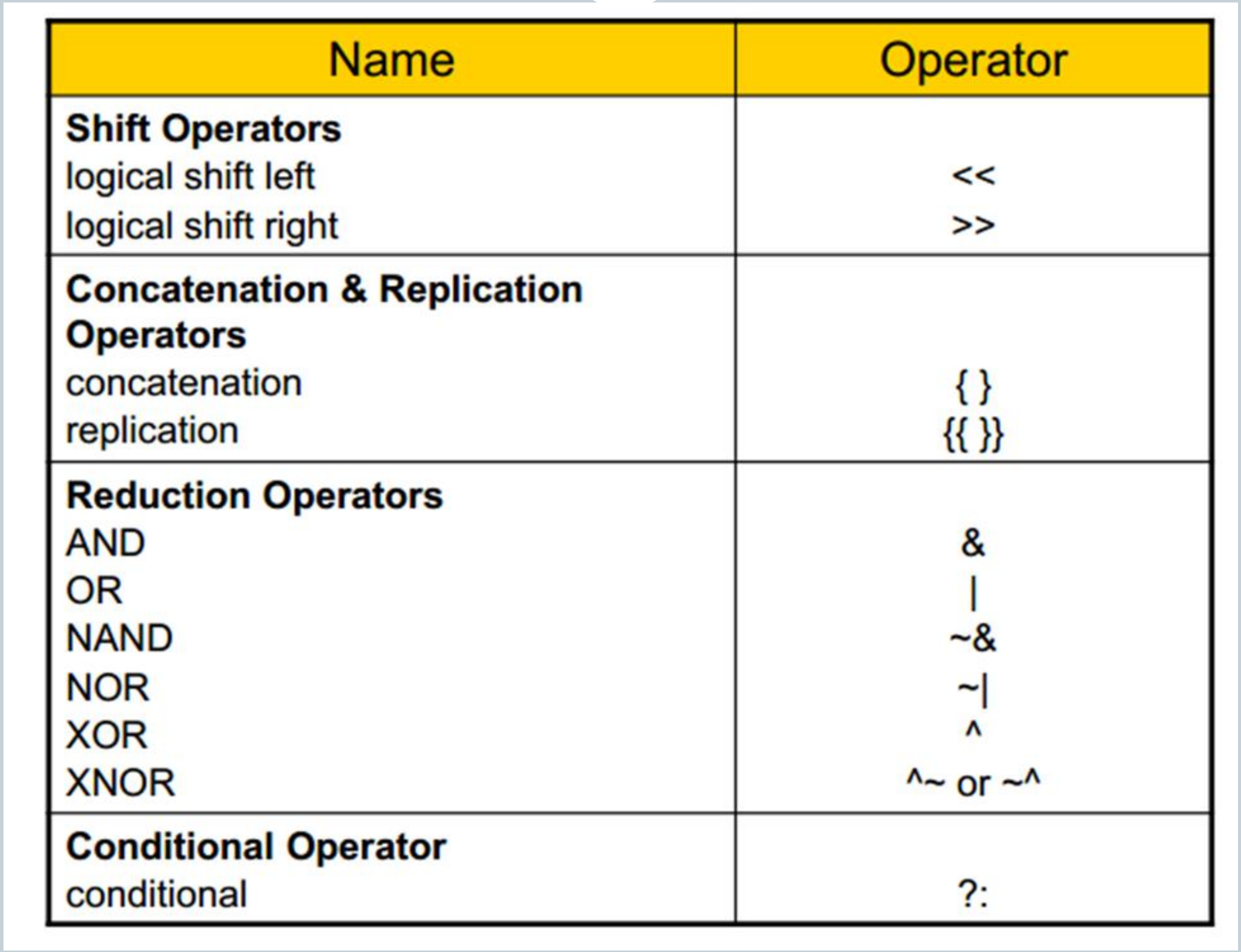
```
A = 4’b1100
X = A[3:2]      // X = 2’b11
Y = A[1:0]      // Y =  2’b00
```

Operators (2/3)



Name	Operator
Relational Operators	
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=
Equality Operators	
logic equality	==
logic inequality	!=
case equality	===
case inequality	!==
Logical Comparison Operators	
NOT	!
AND	&&
OR	||
Logical Bit-Wise Operators	
unary negation NOT	~
binary AND	&
binary OR	|
binary XOR	^
binary XNOR	^~ or ~^

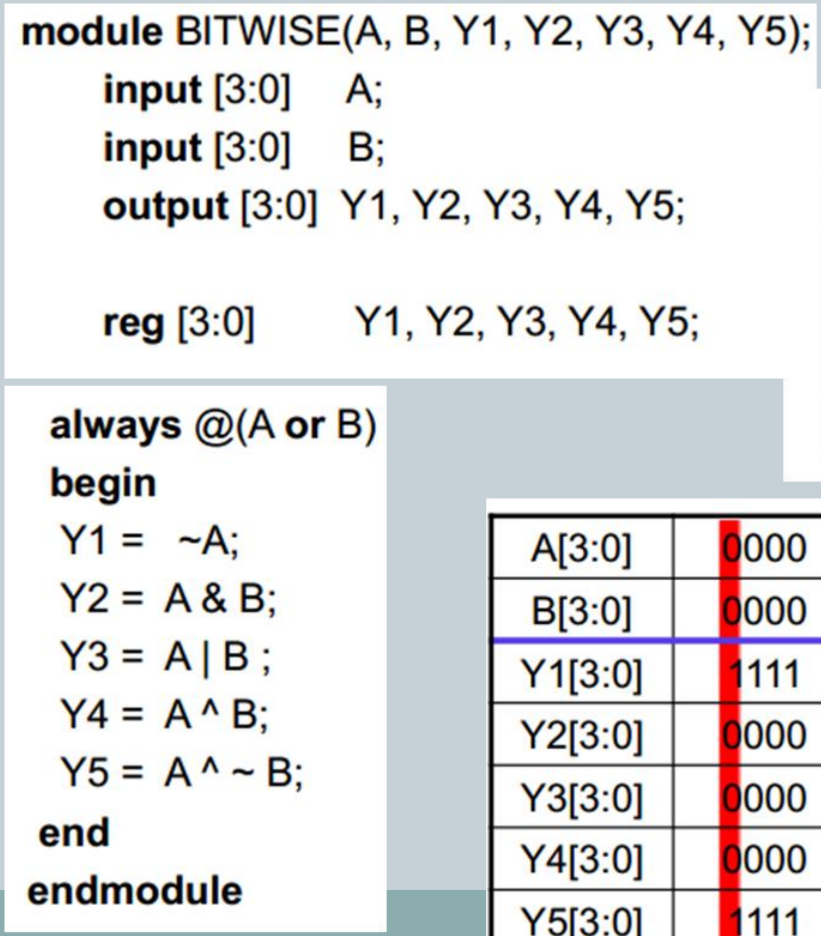
Operators (3/3)



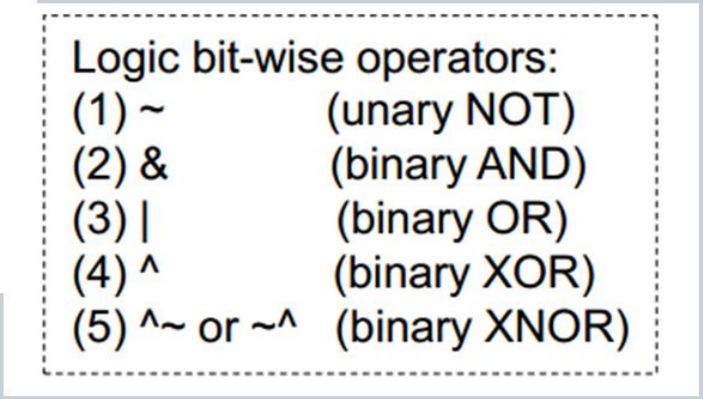
Name	Operator
Shift Operators	
logical shift left	<<
logical shift right	>>
Concatenation & Replication Operators	
concatenation	{ }
replication	{{ }}
Reduction Operators	
AND	&
OR	|
NAND	~&
NOR	~|
XOR	^
XNOR	^~ or ~^
Conditional Operator	
conditional	?:

Example of Logical Bit-Wise Operators

- Logical Bit-Wise Operators

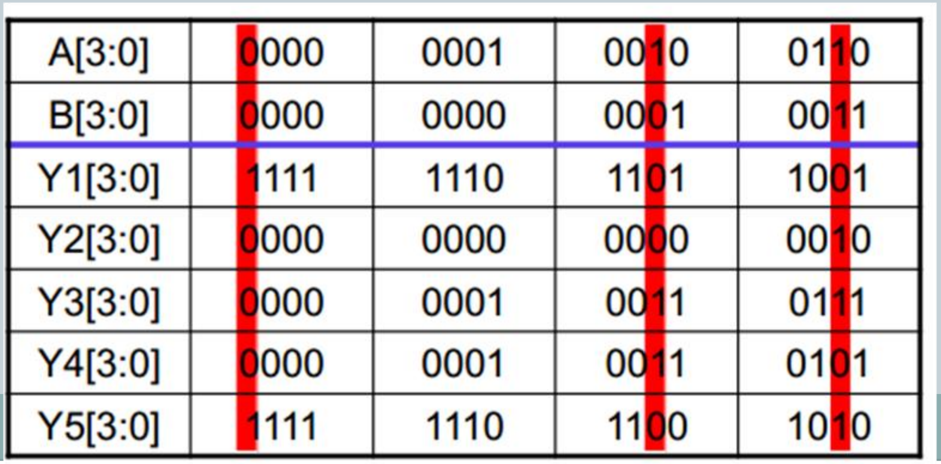


```
module BITWISE(A, B, Y1, Y2, Y3, Y4, Y5);
      input [3:0]    A;
      input [3:0]    B;
      output [3:0]  Y1, Y2, Y3, Y4, Y5;

      reg [3:0]       Y1, Y2, Y3, Y4, Y5;

  always @(A or B)
  begin
   Y1 =   ~A;
   Y2 =  A & B;
   Y3 =  A | B ;
   Y4 =  A ^ B;
   Y5 =  A ^ ~ B;
  end
 endmodule
```



Logic bit-wise operators:

(1) ~	(unary NOT)
(2) &	(binary AND)
(3) |	(binary OR)
(4) ^	(binary XOR)
(5) ^~ or ~^	(binary XNOR)

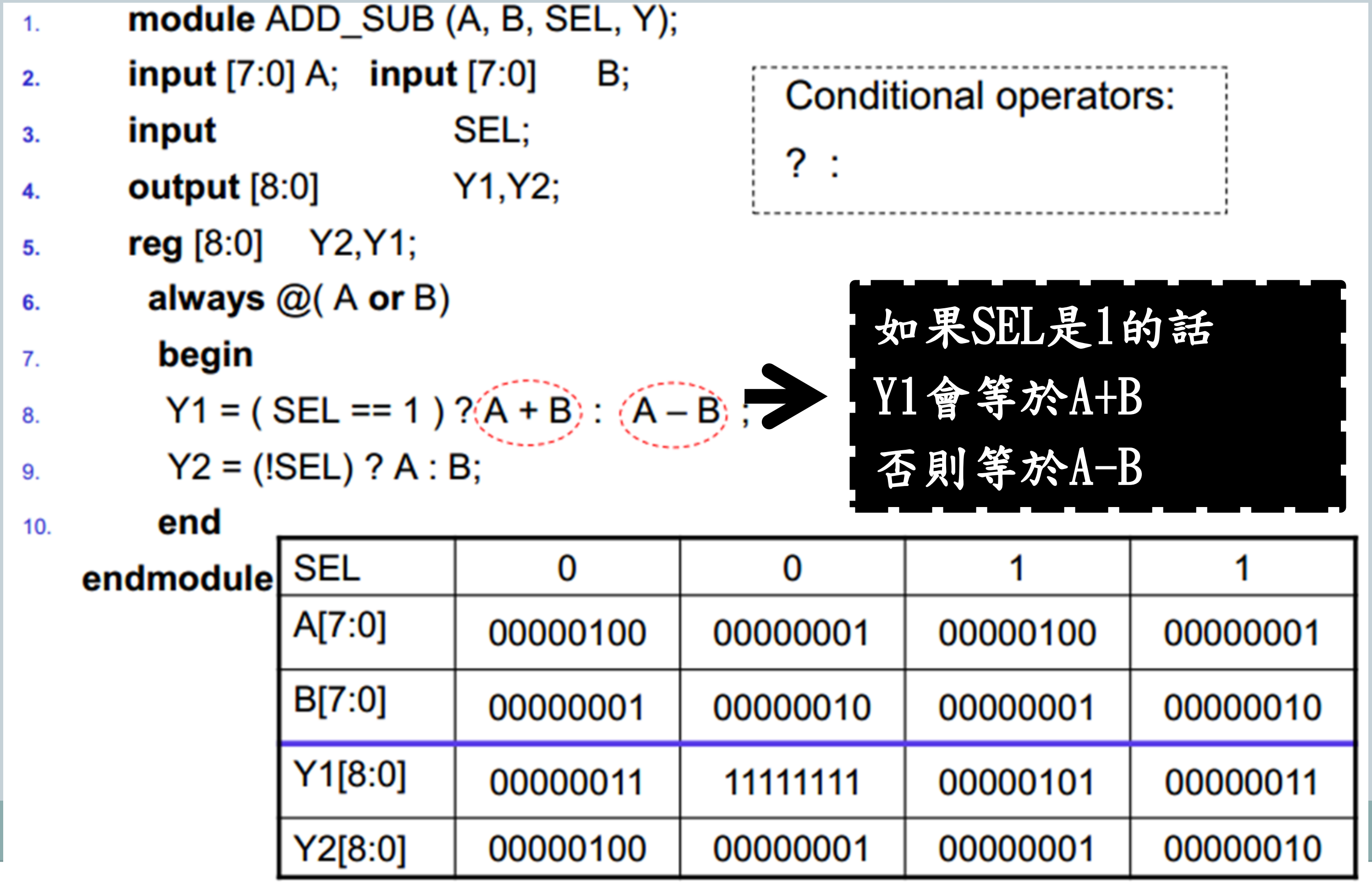


A[3:0]	0000	0001	0010	0110
B[3:0]	0000	0000	0001	0011
Y1[3:0]	1111	1110	1101	1001
Y2[3:0]	0000	0000	0000	0010
Y3[3:0]	0000	0001	0011	0111
Y4[3:0]	0000	0001	0011	0101
Y5[3:0]	1111	1110	1100	1010

Example of Conditional Operators

● Conditional operator

判斷運算式？當判斷運算式為真的時候做的運算：當判斷運算式為假的時候做的運算；



```
module ADD_SUB (A, B, SEL, Y);
input [7:0] A;   input [7:0]     B;
input                  SEL;
output [8:0]           Y1,Y2;
reg [8:0]     Y2,Y1;
  always @( A or B)
   begin
    Y1 = ( SEL == 1 ) ? A + B : A – B ;
    Y2 = (!SEL) ? A : B;
   end
endmodule
```

Conditional operators:

? :

如果SEL是1的話

Y1會等於A+B

否則等於A-B

SEL	0	0	1	1
A[7:0]	00000100	00000001	00000100	00000001
B[7:0]	00000001	00000010	00000001	00000010
Y1[8:0]	00000011	11111111	00000101	00000011
Y2[8:0]	00000100	00000001	00000001	00000010

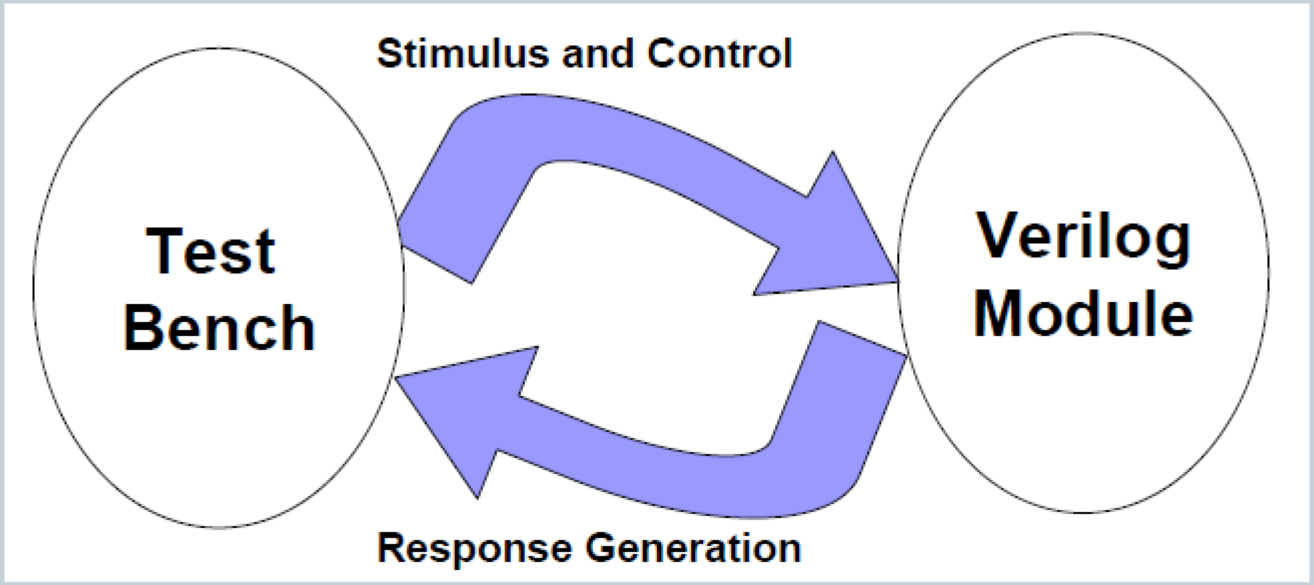
Example of Concatenation & Replication Operators

- Concatenation & Replication Operators

```
A = 1'b1
B = 2'b00
C = 2'b10
D = 3'b110

W = { B , C }                    // W= 4'b0010

X = { A , B , C , D ,3'b001  }   // X = 11'b10010110001

Z = { B , 3{ A } }               // Z = 5'b00111

H = { D[1:0]  , C ] }            // H = 4'b1010
```

Verification

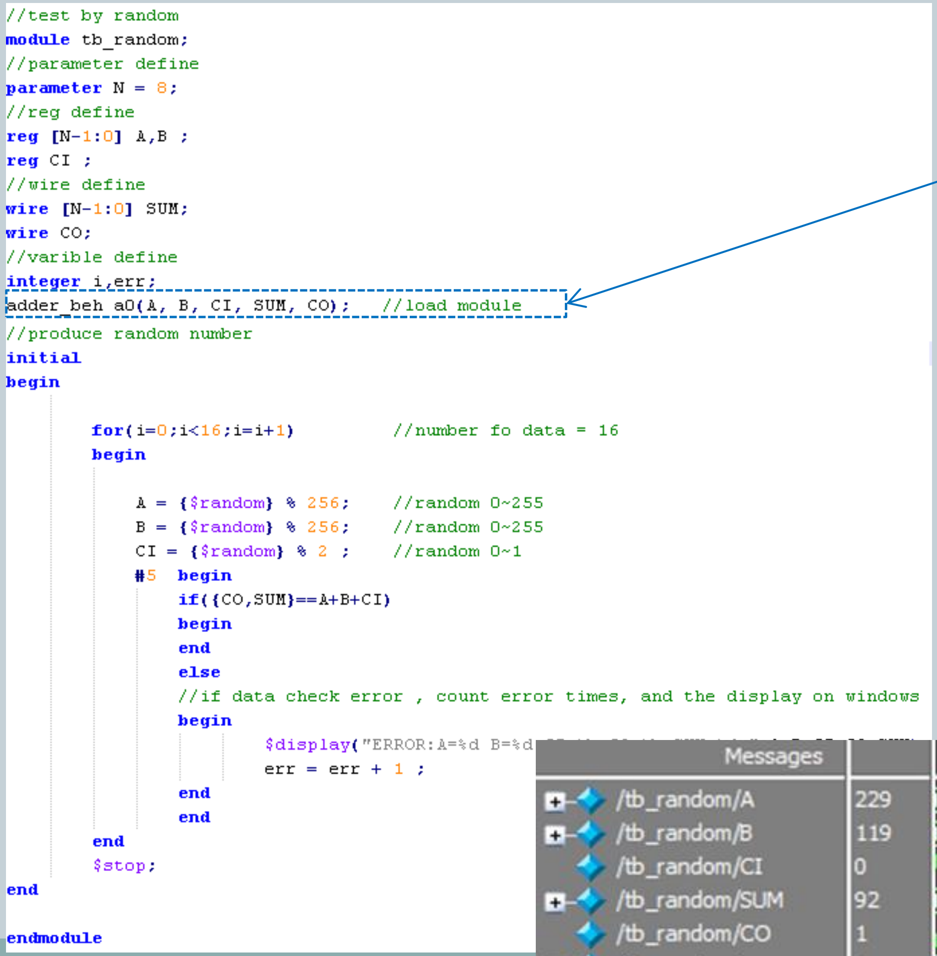
- Verilog Module



Verification

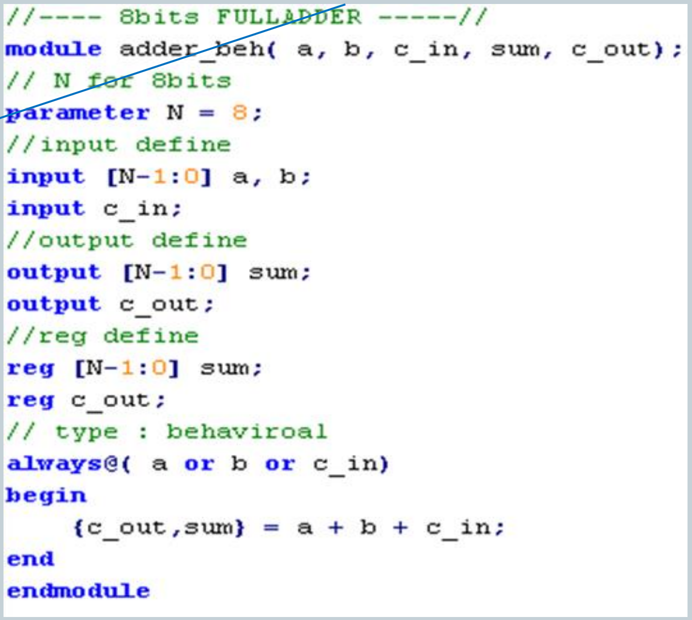
- Example: (8bits adder)

✓ test bench for 8bits adder



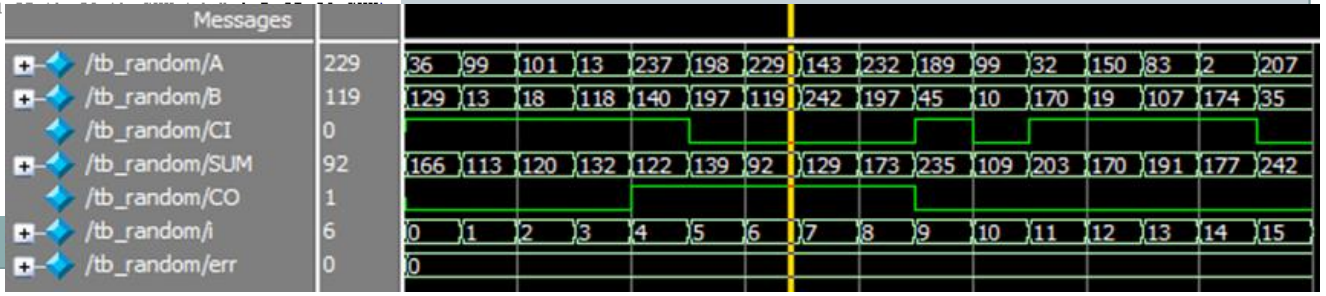
```
//test by random
module tb_random;
//parameter define
parameter N = 8;
//reg define
reg [N-1:0] A,B ;
reg CI ;
//wire define
wire [N-1:0] SUM;
wire CO;
//varible define
integer i,err;
adder_beh a0(A, B, CI, SUM, CO);   //load module
//produce random number
initial
begin

        for(i=0;i<16;i=i+1)          //number fo data = 16
        begin

            A = {$random} % 256;    //random 0~255
            B = {$random} % 256;    //random 0~255
            CI = {$random} % 2 ;    //random 0~1
            #5  begin
                if({CO,SUM}==A+B+CI)
                begin
                end
                else
                //if data check error , count error times, and the display on windows
                begin
                        $display("ERROR:A=%d B=%d
                        err = err + 1 ;
                end
                end
        end
        $stop;
end

endmodule
```

✓ 8bits adder(verilog module)



```
//---- 8bits FULLADDER -----//
module adder_beh( a, b, c_in, sum, c_out);
// N for 8bits
parameter N = 8;
//input define
input [N-1:0] a, b;
input c_in;
//output define
output [N-1:0] sum;
output c_out;
//reg define
reg [N-1:0] sum;
reg c_out;
// type : behaviroal
always@( a or b or c_in)
begin
    {c_out,sum} = a + b + c_in;
end
endmodule
```

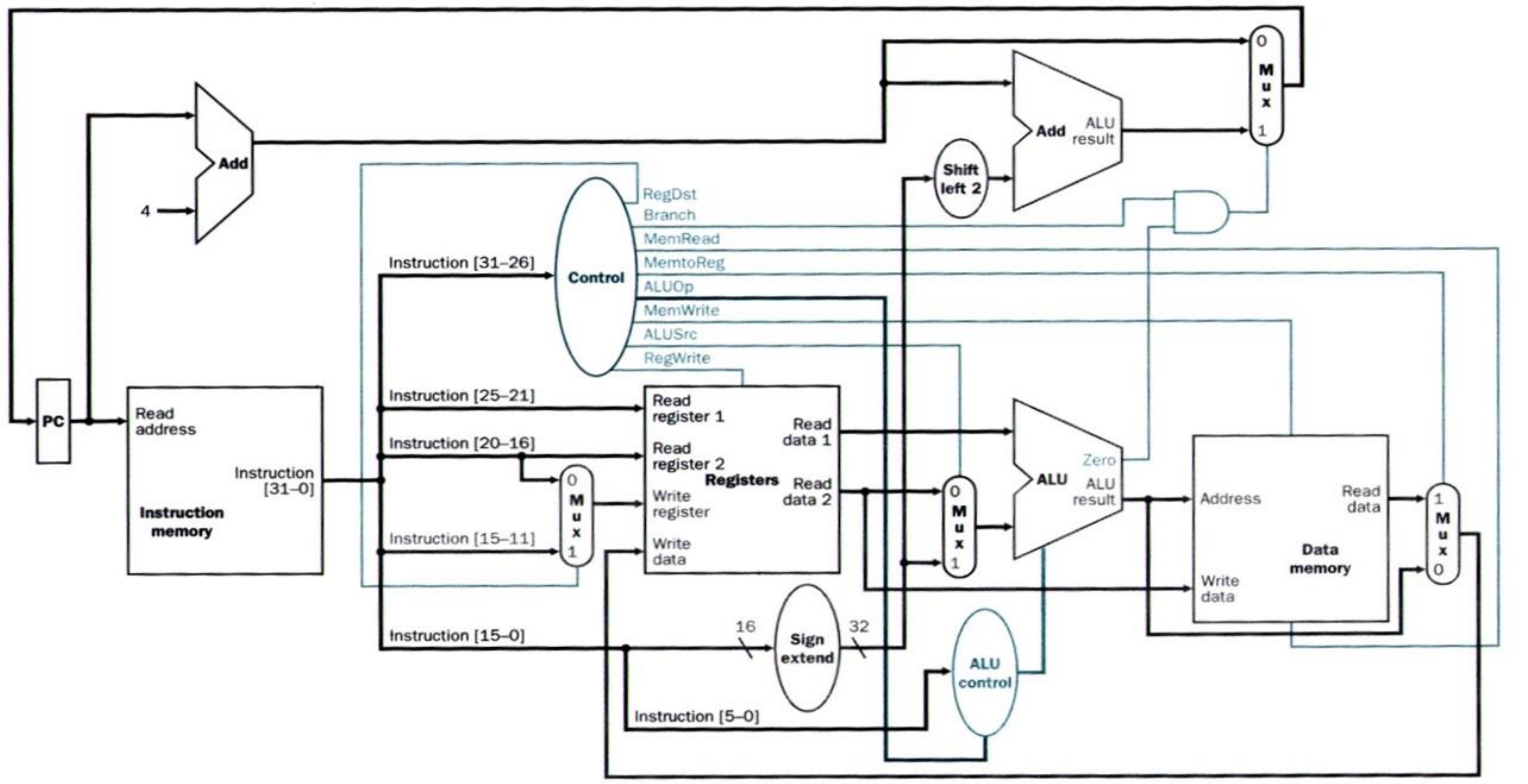
✓ Verification Wave



```
Messages
/tb_random/A     229  36  99  101 13  237 198 229 143 232 189 99  32  150 83  2   207
/tb_random/B     119  129 13  18  118 140 197 119 242 197 45  10  170 19  107 174 35
/tb_random/CI    0
/tb_random/SUM   92   166 113 120 132 122 139 92  129 173 235 109 203 170 191 177 242
/tb_random/CO    1
/tb_random/i     6    0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
/tb_random/err   0    0
```

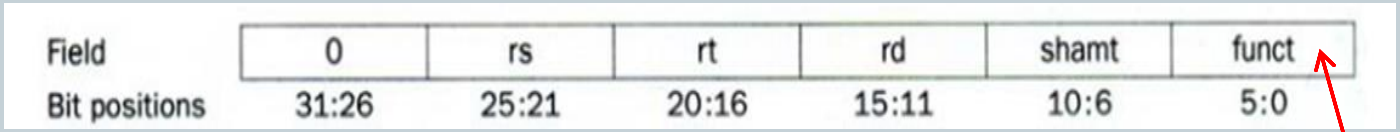
認識MIPS processor的 control

MIPS processor with control



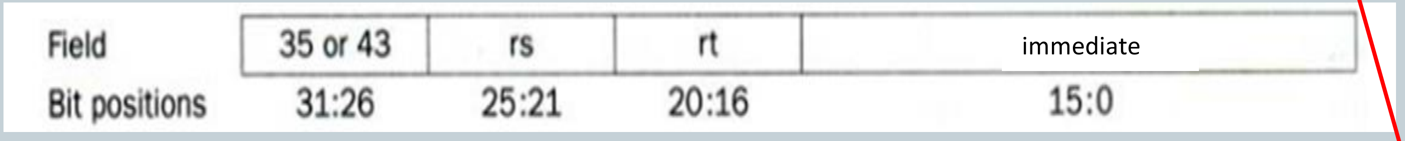
Control Unit

- Selecting the operation to perform
- Controlling the flow of data
- Information comes from the 32-bit instruction
- Instruction format : (以下只有部分指令格式)
  - R-fomrat



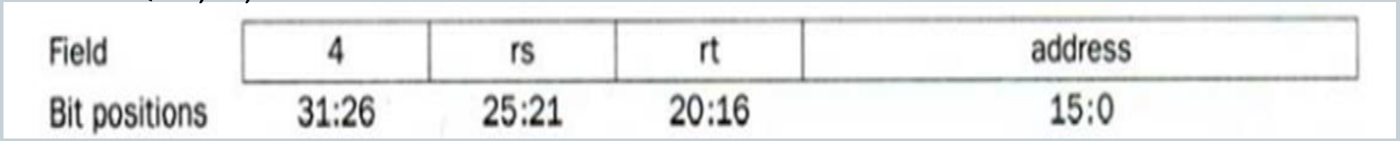
Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

  - LW rs,off16(rt) or SW rs,off16(rs)



Field	35 or 43	rs	rt	immediate
Bit positions	31:26	25:21	20:16	15:0

  - BEQ rs, rt, address



Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

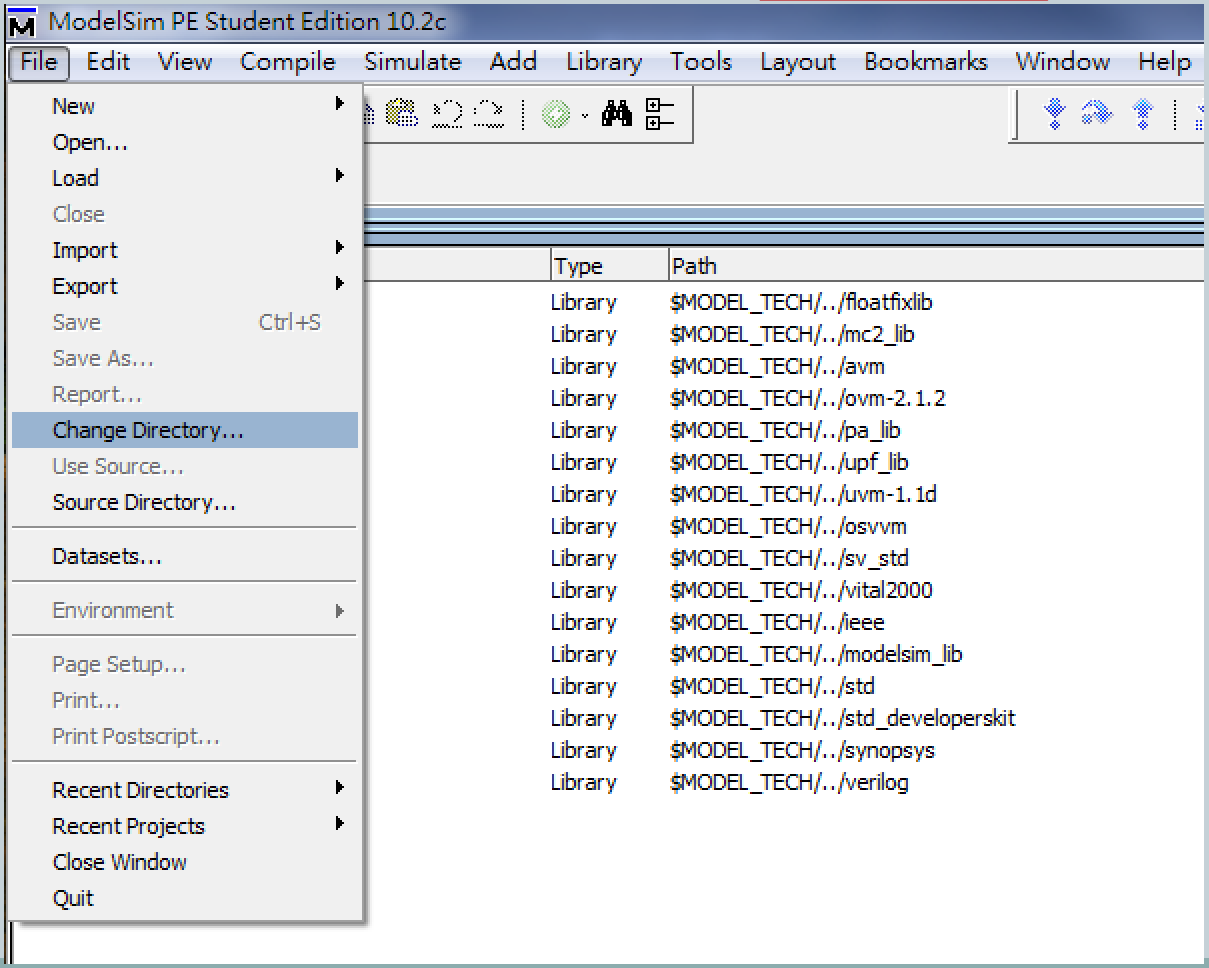
- ALU’s operation based on instruction type and <u>function code</u>

Modelsim驗證

Step.1:change to our file location

- 打開modelsim後,在File下選擇change directory到你放\..\verilog_test的地方

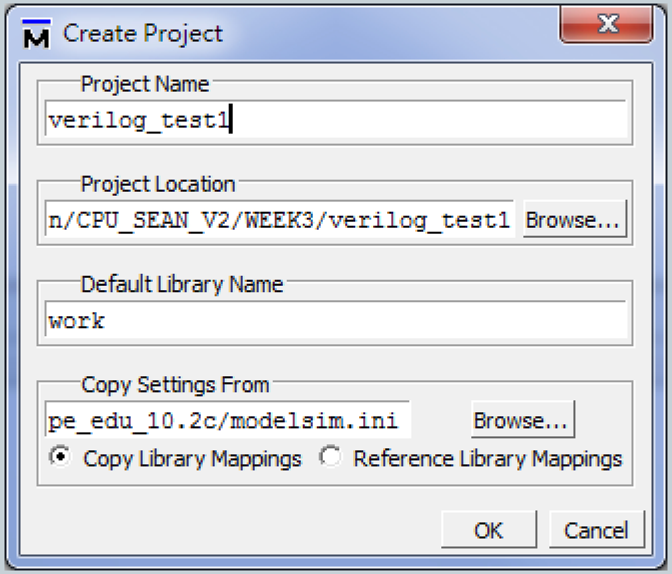
可以自己命名



Type	Path
Library	$MODEL_TECH/../floatfixlib
Library	$MODEL_TECH/../mc2_lib
Library	$MODEL_TECH/../avm
Library	$MODEL_TECH/../ovm-2.1.2
Library	$MODEL_TECH/../pa_lib
Library	$MODEL_TECH/../upf_lib
Library	$MODEL_TECH/../uvm-1.1d
Library	$MODEL_TECH/../osvvm
Library	$MODEL_TECH/../sv_std
Library	$MODEL_TECH/../vital2000
Library	$MODEL_TECH/../ieee
Library	$MODEL_TECH/../modelsim_lib
Library	$MODEL_TECH/../std
Library	$MODEL_TECH/../std_developerskit
Library	$MODEL_TECH/../synopsys
Library	$MODEL_TECH/../verilog

Step.2:new project

- 接著new一個project(名稱自訂)



Project Name: verilog_test1

Project Location: n/CPU_SEAN_V2/WEEK3/verilog_test1 Browse...

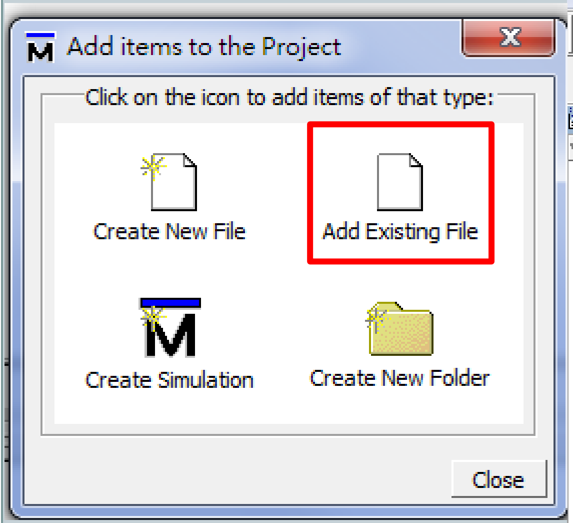
Default Library Name: work

Copy Settings From: pe_edu_10.2c/modelsim.ini Browse...

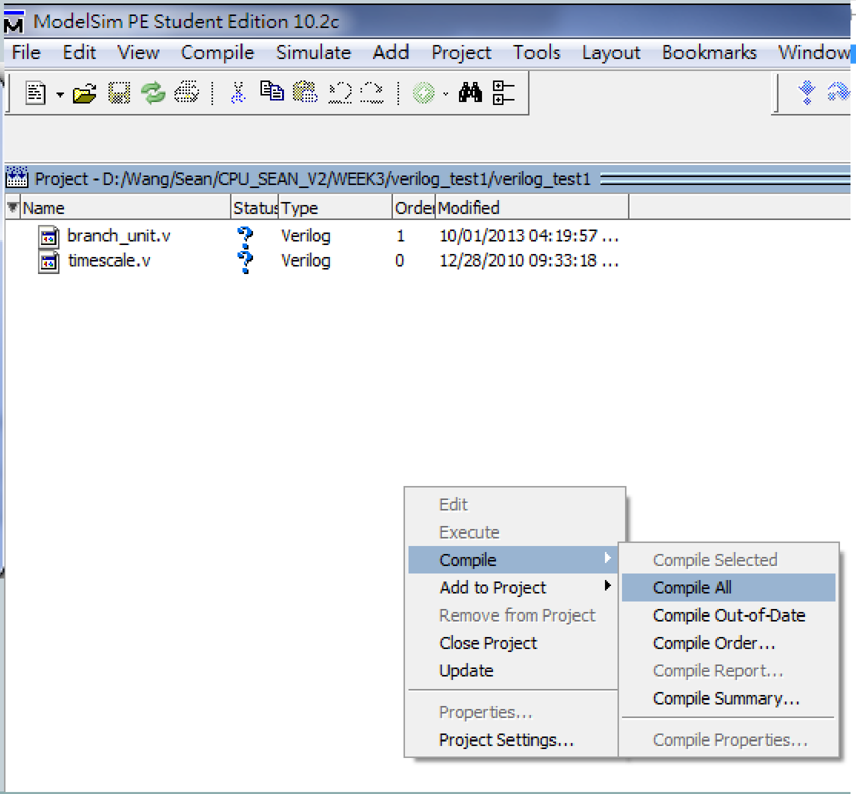
[✓ Copy Library Mappings] Reference Library Mappings

Step.3:add source code

- 新增完project後,會跳出一個視窗(如圖),點選Add Existing File將VHDL/Verilog source code加入到這個project中
- 右鍵→編譯你的VHDL/Verilog source code


Add items to the Project

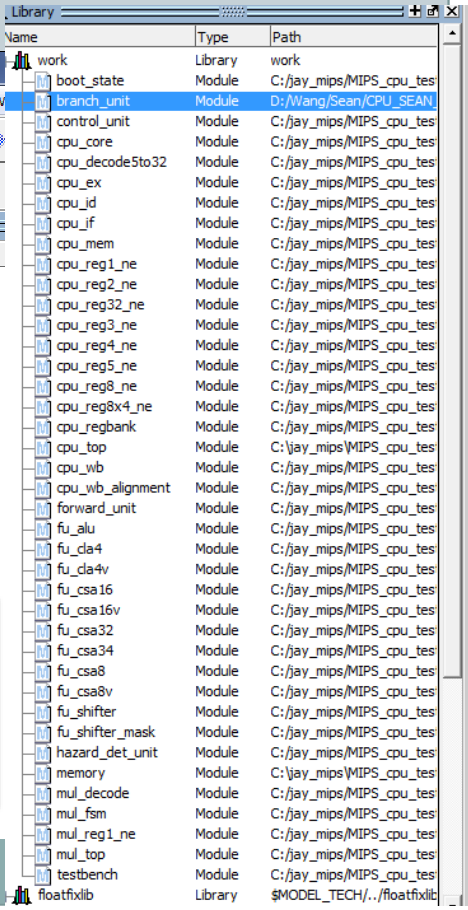
Click on the icon to add items of that type:


ModelSim PE Student Edition 10.2c

File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window

Project - D:/Wang/Sean/CPU_SEAN_V2/WEEK3/verilog_test1/verilog_test1

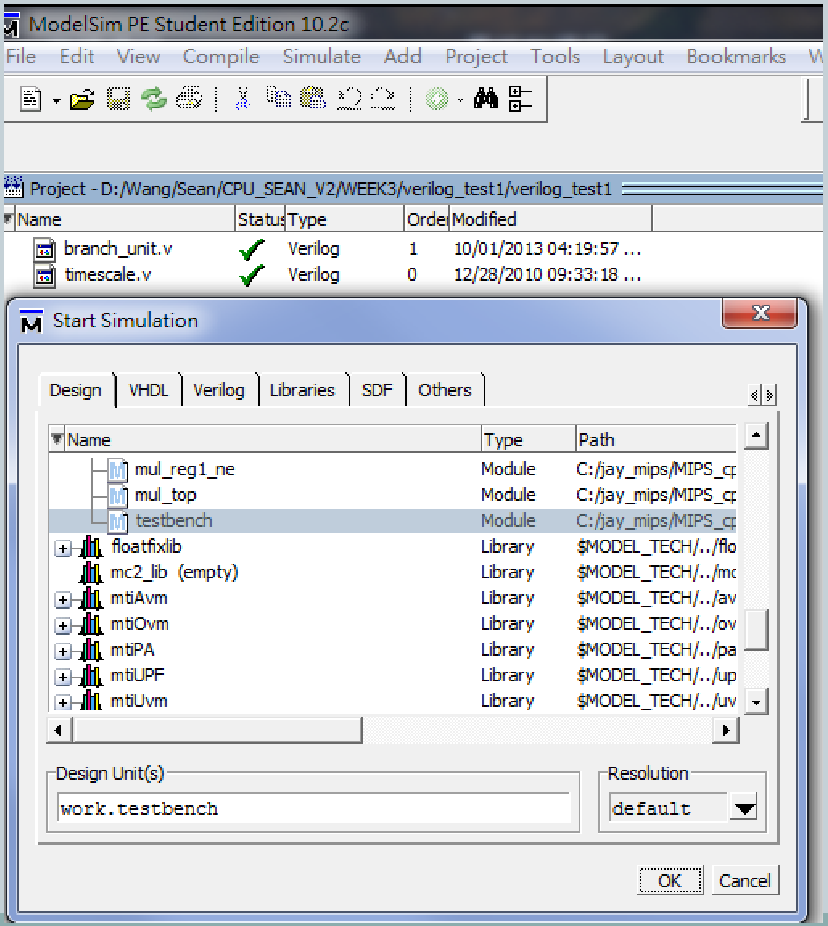
Name	Status	Type	Order	Modified
branch_unit.v	?	Verilog	1	10/01/2013 04:19:57 ...
timescale.v	?	Verilog	0	12/28/2010 09:33:18 ...


Library

Name	Type	Path
work	Library	work
boot_state	Module	C:/jay_mips/MIPS_cpu_tes
branch_unit	Module	D:/Wang/Sean/CPU_SEAN
control_unit	Module	C:/jay_mips/MIPS_cpu_tes
cpu_core	Module	C:/jay_mips/MIPS_cpu_tes
cpu_decode5to32	Module	C:/jay_mips/MIPS_cpu_tes
cpu_ex	Module	C:/jay_mips/MIPS_cpu_tes
cpu_id	Module	C:/jay_mips/MIPS_cpu_tes
cpu_if	Module	C:/jay_mips/MIPS_cpu_tes
cpu_mem	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg1_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg2_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg32_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg3_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg4_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg5_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg8_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_reg8x4_ne	Module	C:/jay_mips/MIPS_cpu_tes
cpu_regbank	Module	C:/jay_mips/MIPS_cpu_tes
cpu_top	Module	C:\jay_mips\MIPS_cpu_tes
cpu_wb	Module	C:/jay_mips/MIPS_cpu_tes
cpu_wb_alignment	Module	C:/jay_mips/MIPS_cpu_tes
forward_unit	Module	C:/jay_mips/MIPS_cpu_tes
fu_alu	Module	C:/jay_mips/MIPS_cpu_tes
fu_cla4	Module	C:/jay_mips/MIPS_cpu_tes
fu_cla4v	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa16	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa16v	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa32	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa34	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa8	Module	C:/jay_mips/MIPS_cpu_tes
fu_csa8v	Module	C:/jay_mips/MIPS_cpu_tes
fu_shifter	Module	C:/jay_mips/MIPS_cpu_tes
fu_shifter_mask	Module	C:/jay_mips/MIPS_cpu_tes
hazard_det_unit	Module	C:/jay_mips/MIPS_cpu_tes
memory	Module	C:\jay_mips\MIPS_cpu_tes
mul_decode	Module	C:/jay_mips/MIPS_cpu_tes
mul_fsm	Module	C:/jay_mips/MIPS_cpu_tes
mul_reg1_ne	Module	C:/jay_mips/MIPS_cpu_tes
mul_top	Module	C:/jay_mips/MIPS_cpu_tes
testbench	Module	C:/jay_mips/MIPS_cpu_tes
floatfixlib	Library	$MODEL_TECH/../floatfixlib

Step.4:simulate

- 按simulation, work→testbench


ModelSim PE Student Edition 10.2c
File Edit View Compile Simulate Add Project Tools Layout Bookmarks
Start Simulation
Design VHDL Verilog Libraries SDF Others
OK
Cancel

Project - D:/Wang/Sean/CPU_SEAN_V2/WEEK3/verilog_test1/verilog_test1

Name	Status	Type	Order	Modified
branch_unit.v	✓	Verilog	1	10/01/2013 04:19:57 ...
timescale.v	✓	Verilog	0	12/28/2010 09:33:18 ...

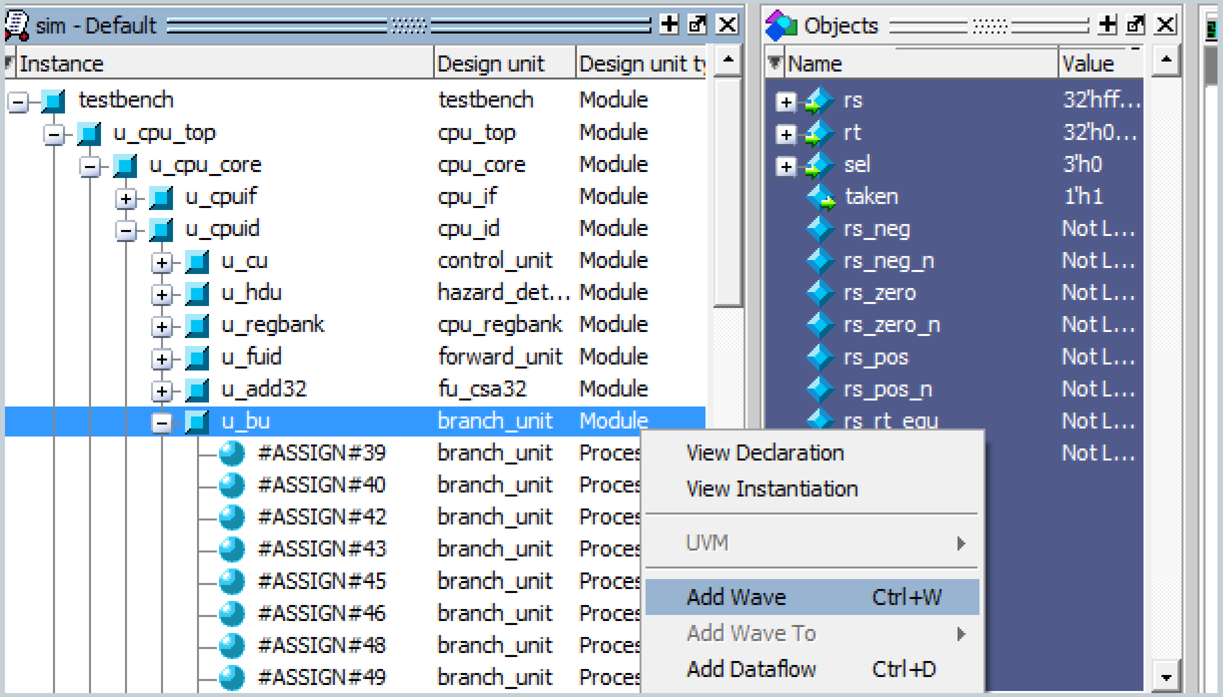
Name	Type	Path
mul_reg1_ne	Module	C:/jay_mips/MIPS_cp
mul_top	Module	C:/jay_mips/MIPS_cp
testbench	Module	C:/jay_mips/MIPS_cp
floatfixlib	Library	$MODEL_TECH/../flo
mc2_lib (empty)	Library	$MODEL_TECH/../mc
mtiAvm	Library	$MODEL_TECH/../av
mtiOvm	Library	$MODEL_TECH/../ov
mtiPA	Library	$MODEL_TECH/../pa
mtiUPF	Library	$MODEL_TECH/../up
mtiUvm	Library	$MODEL_TECH/../uv

Design Unit(s): work.testbench

Resolution: default

Step.5:Add Wave

- 加入你要觀察的部分，觀察波形圖是否正確。
- 若你有事先儲存要觀察波型的檔案，可以直接讀取”XXX.do”檔。
  - 讀取方式: File→Load→Marco File…
  - 儲存方式: 點擊Wave視窗後，File→Save Format…



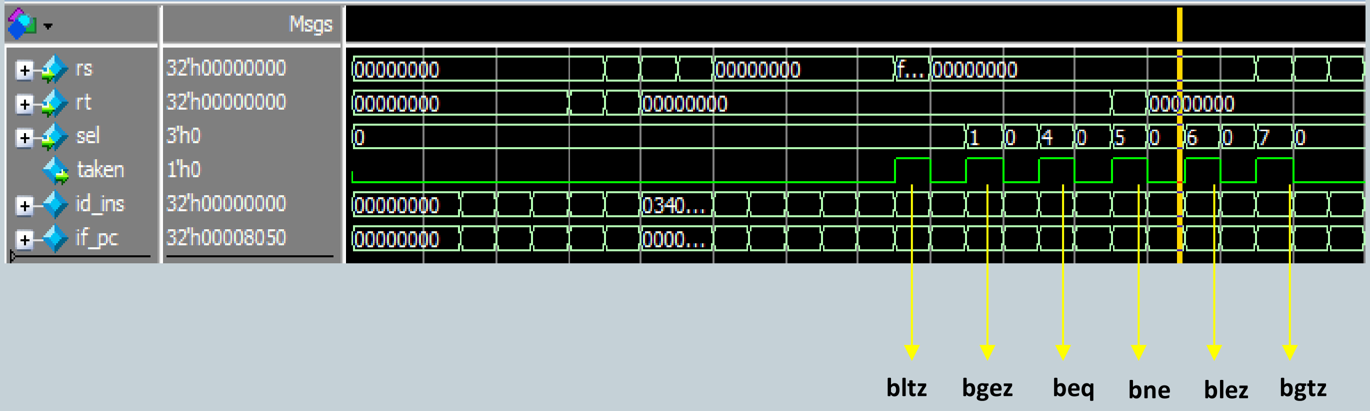
Instance	Design unit	Design unit ty
testbench	testbench	Module
u_cpu_top	cpu_top	Module
u_cpu_core	cpu_core	Module
u_cpuif	cpu_if	Module
u_cpuid	cpu_id	Module
u_cu	control_unit	Module
u_hdu	hazard_det…	Module
u_regbank	cpu_regbank	Module
u_fuid	forward_unit	Module
u_add32	fu_csa32	Module
u_bu	branch_unit	Module
#ASSIGN#39	branch_unit	Proces
#ASSIGN#40	branch_unit	Proces
#ASSIGN#42	branch_unit	Proces
#ASSIGN#43	branch_unit	Proces
#ASSIGN#45	branch_unit	Proces
#ASSIGN#46	branch_unit	Proces
#ASSIGN#48	branch_unit	Proces
#ASSIGN#49	branch_unit	Proces

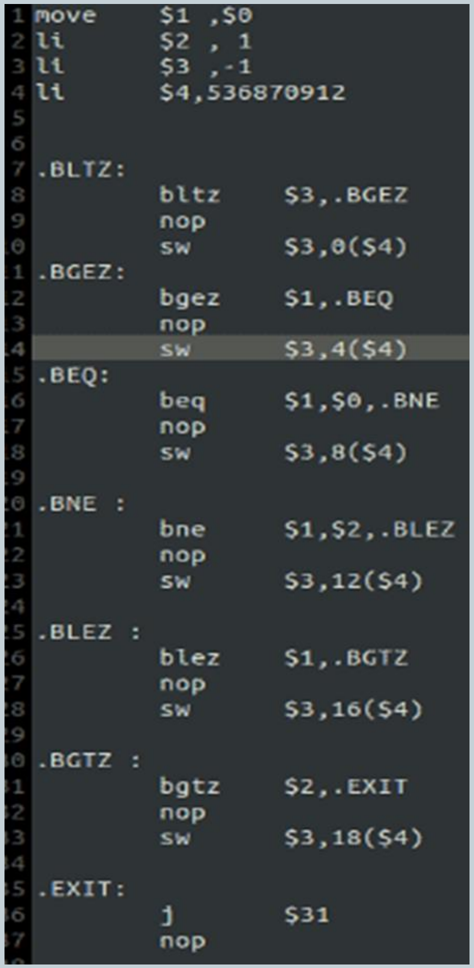
Name	Value
rs	32'hff…
rt	32'h0…
sel	3'h0
taken	1'h1
rs_neg	Not L…
rs_neg_n	Not L…
rs_zero	Not L…
rs_zero_n	Not L…
rs_pos	Not L…
rs_pos_n	Not L…
rs_rt_equ	Not L…
	Not L…

Step.6:Obeserve Wave

- 右圖為用來驗證硬體(branch_unit)的組合語言程式，這個程式的行為是當遇到每個有條件的branch指令，每個跳躍的條件都會成立。

- 下圖為用來觀察硬體行為是否有誤的波形圖。

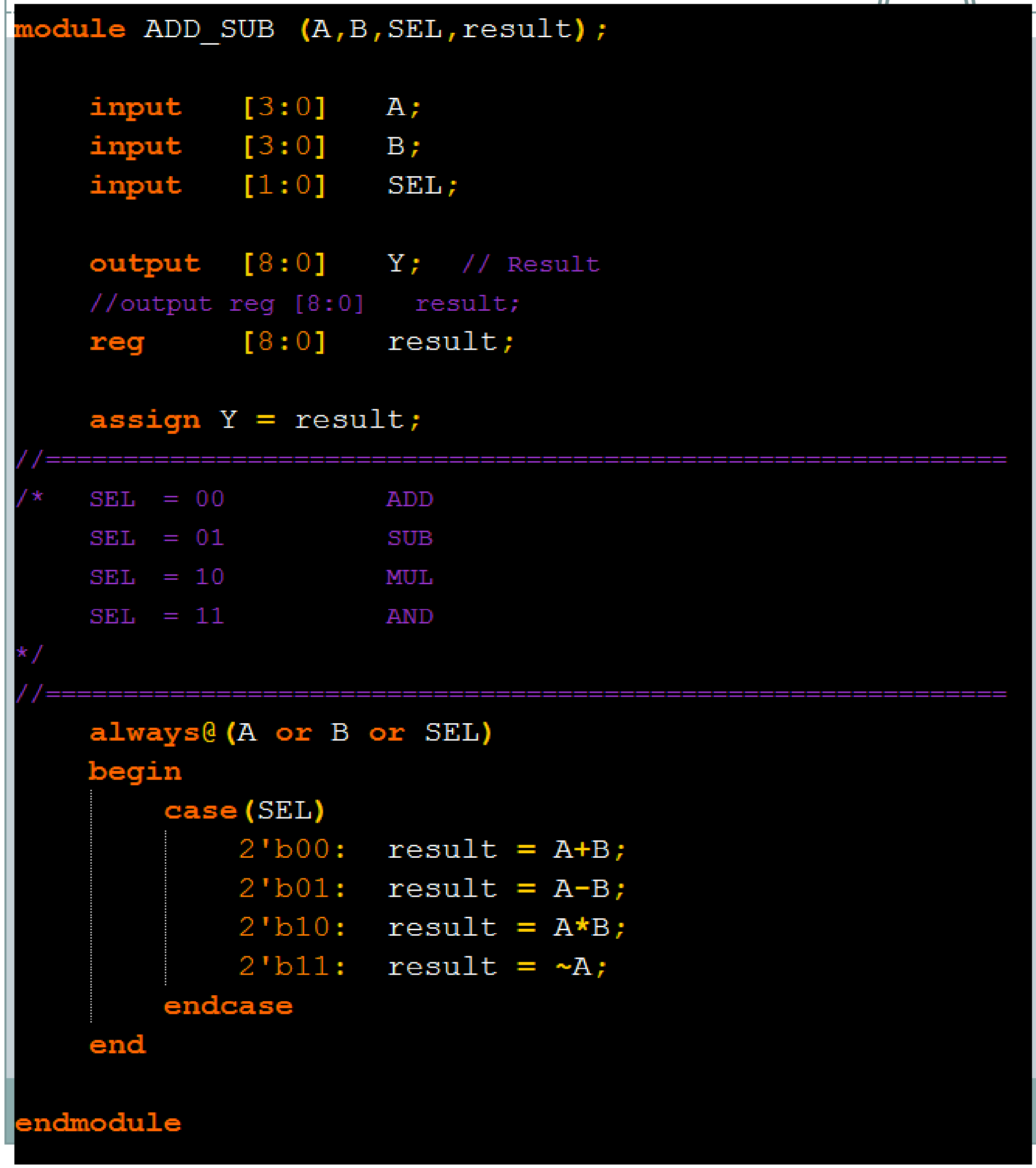




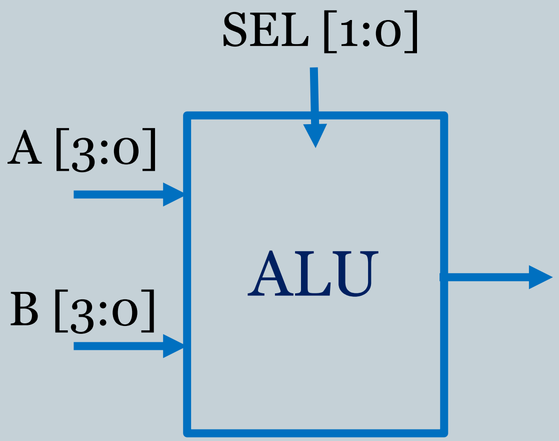
```
move      $1 ,$0
li        $2 , 1
li        $3 ,-1
li        $4,536870912


.BLTZ:
          bltz      $3,.BGEZ
          nop
          sw        $3,0($4)
.BGEZ:
          bgez      $1,.BEQ
          nop
          sw        $3,4($4)
.BEQ:
          beq       $1,$0,.BNE
          nop
          sw        $3,8($4)

.BNE :
          bne       $1,$2,.BLEZ
          nop
          sw        $3,12($4)

.BLEZ :
          blez      $1,.BGTZ
          nop
          sw        $3,16($4)

.BGTZ :
          bgtz      $2,.EXIT
          nop
          sw        $3,18($4)

.EXIT:
          j         $31
          nop
```

以Verilog實作 Arithmetic Logic Unit (ALU)

ALU EXAMPLE

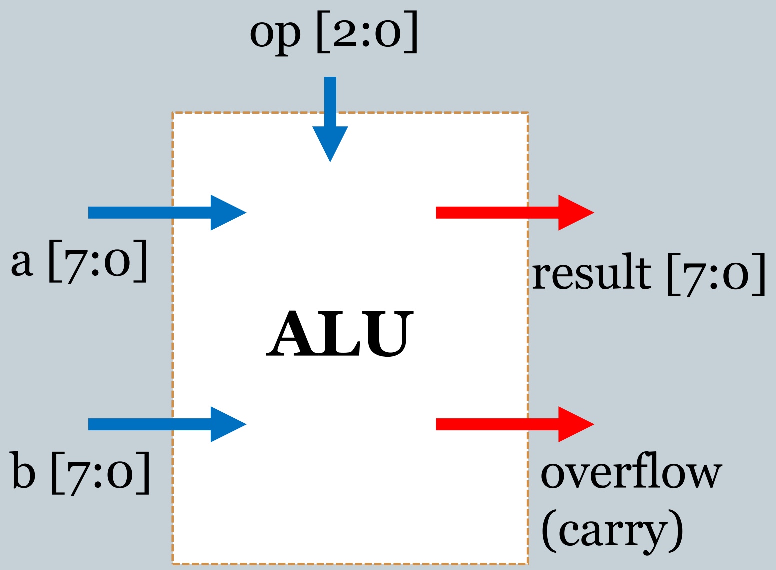


```
module ADD_SUB (A,B,SEL,result);

    input    [3:0]    A;
    input    [3:0]    B;
    input    [1:0]    SEL;

    output   [8:0]    Y;   // Result
    //output reg [8:0]   result;
    reg      [8:0]    result;

    assign Y = result;
//=====================================================================
/*  SEL  = 00            ADD
    SEL  = 01            SUB
    SEL  = 10            MUL
    SEL  = 11            AND
*/
//=====================================================================
    always@(A or B or SEL)
    begin
        case(SEL)
            2'b00:  result = A+B;
            2'b01:  result = A-B;
            2'b10:  result = A*B;
            2'b11:  result = ~A;
        endcase
    end

endmodule
```

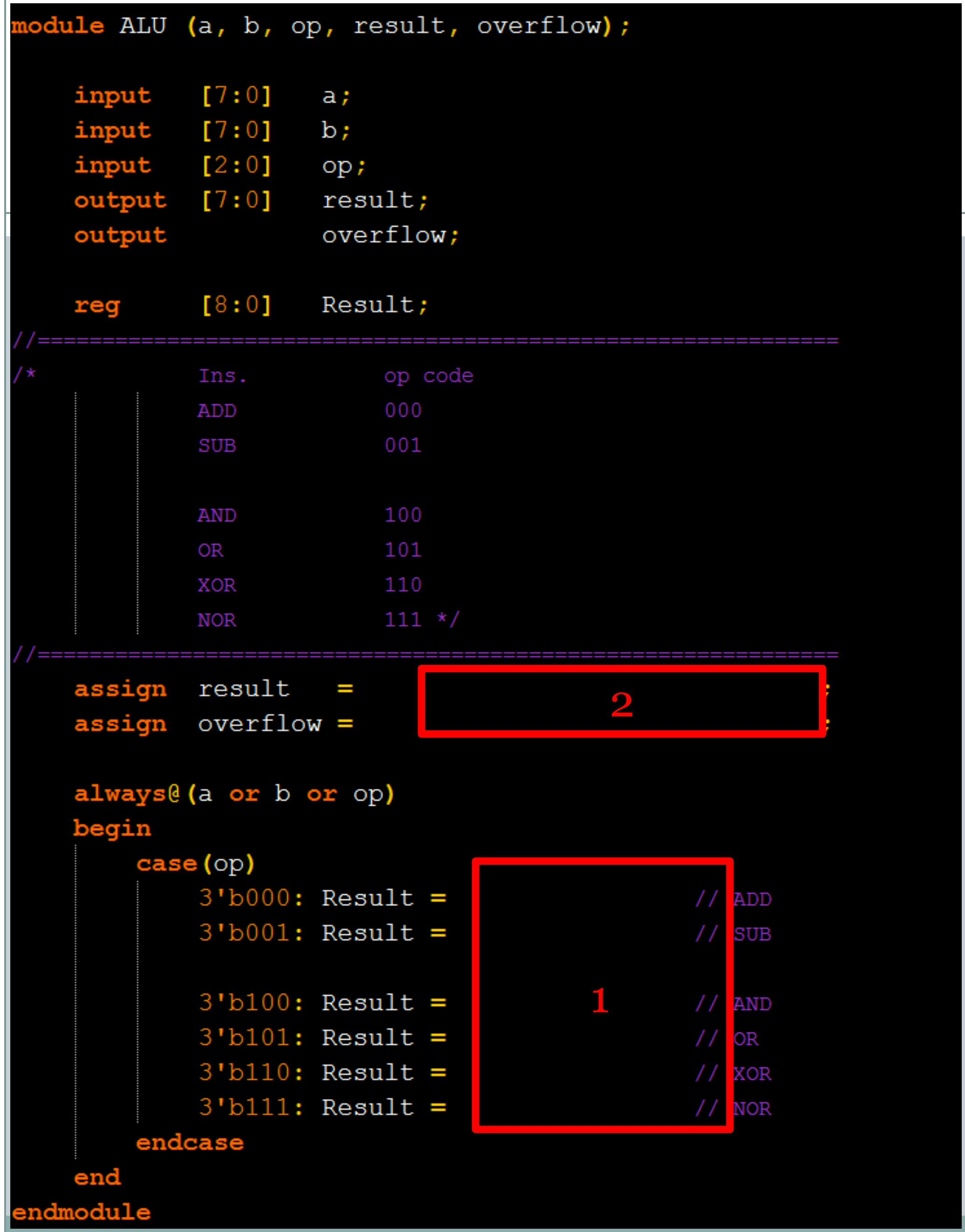


練習題(一) : Simple ALU

- 前面的範例是一個只有加減法的ALU，請同學根據前面的範例，用verilog完成新的ALU，並且完成驗證



Instruction	op Code
ADD	000
SUB	001
AND	100
OR	101
XOR	110
NOR	111



```
module ALU (a, b, op, result, overflow);

    input   [7:0]   a;
    input   [7:0]   b;
    input   [2:0]   op;
    output  [7:0]   result;
    output          overflow;

    reg     [8:0]   Result;
//=====================================================
/*          Ins.            op code
            ADD             000
            SUB             001

            AND             100
            OR              101
            XOR             110
            NOR             111 */
//=====================================================
    assign  result   =     [          2          ];
    assign  overflow =     [                     ];

    always@(a or b or op)
    begin
        case(op)
            3'b000: Result =  [                 ] // ADD
            3'b001: Result =  [                 ] // SUB

            3'b100: Result =  [        1        ] // AND
            3'b101: Result =  [                 ] // OR
            3'b110: Result =  [                 ] // XOR
            3'b111: Result =  [                 ] // NOR
        endcase
    end
endmodule
```

Step1: 根據不同的op code 決定
ALU的運算

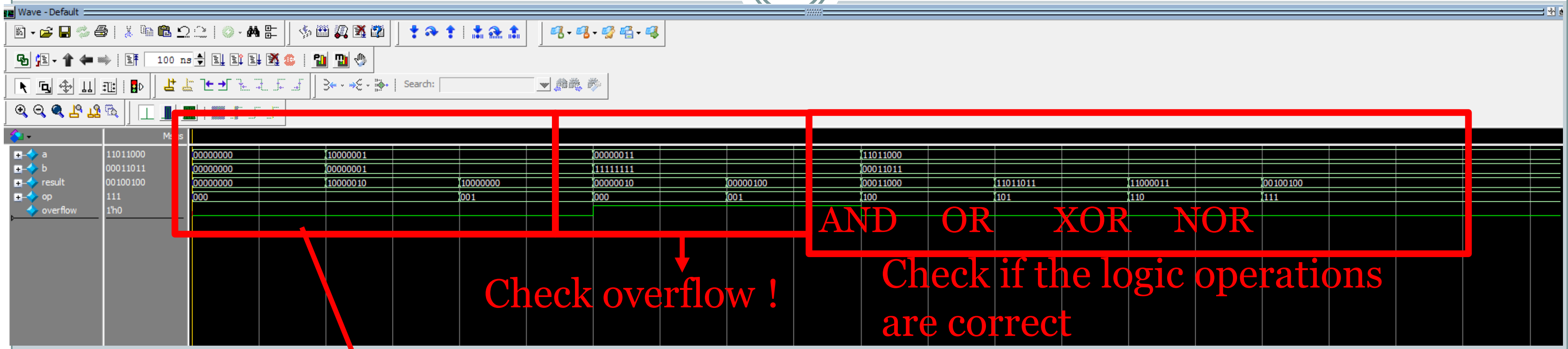
Step2: 把ALU運算完的結果
傳到output (result)
ALU在做加減法運算時，
可能會有overflow產生，
請判斷什麼時候有overflow

練習題(一)ALU加減法為無號數的加減法

if (op = ADD/SUB)
check overflow

$$\begin{tikzcd}\boxed{\text{Result [8:0]}} \arrow[d] \\ \boxed{\text{result [7:0]}}\end{tikzcd}$$

練習題(一)：驗證

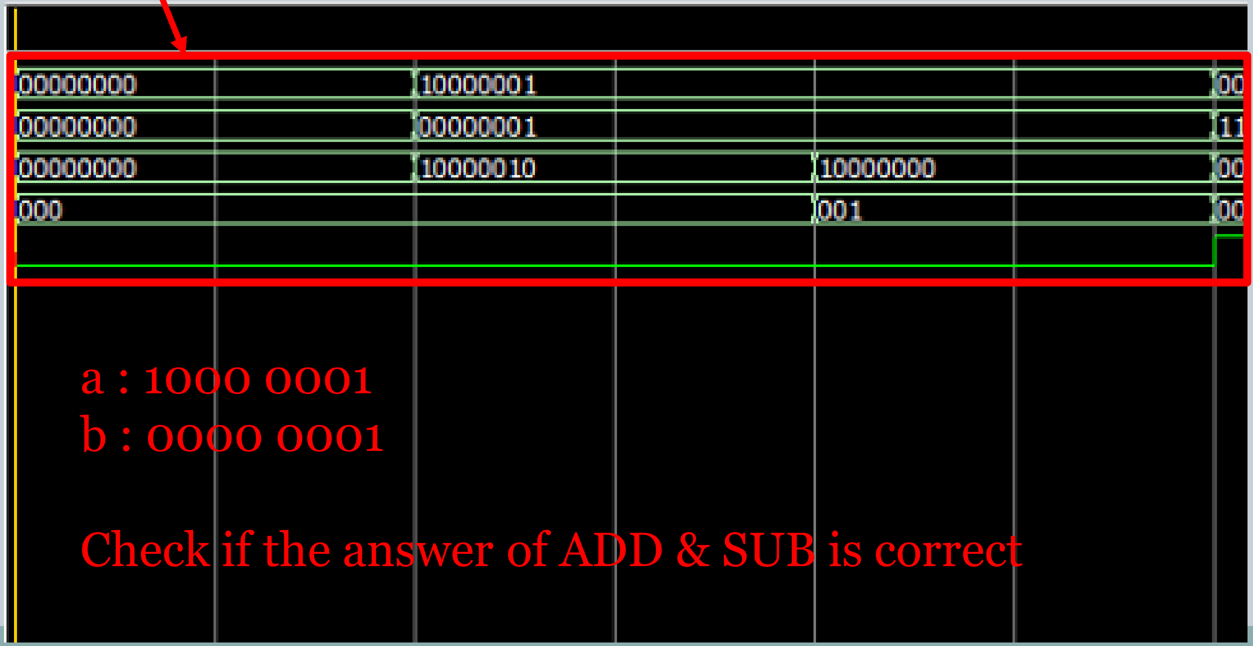


```
Wave - Default
Search:
                 Msgs
a         11011000   00000000  10000001  00000011  11011000
b         00011011   00000000  00000001  11111111  00011011
result    00100100   00000000  10000010  10000000  00000010  00000100  00011000  11011011  11000011  00100100
op        111        000       001       000       001       100       101       110       111
overflow  1'h0
```

AND OR XOR NOR

Check overflow !

Check if the logic operations are correct



```
00000000  10000001  00
00000000  00000001  11
00000000  10000010  10000000  00
000       001       00
```

a : 1000 0001
b : 0000 0001

Check if the answer of ADD & SUB is correct

練習題(二) :實作ALU指令

- 打開資料夾verilog_test_alu，並請同學以Verilog完成這顆MIPS CPU 中fu_alu.v 的空白部分，並跑助教的程式(code.mif)來驗證ALU運算結果。

- 因為這顆MIPS CPU經過簡化之後**並沒有實現所有MIPS的指令**，MIPS詳細的運算指令和ALUop decode請參考Table 1與Table 2.

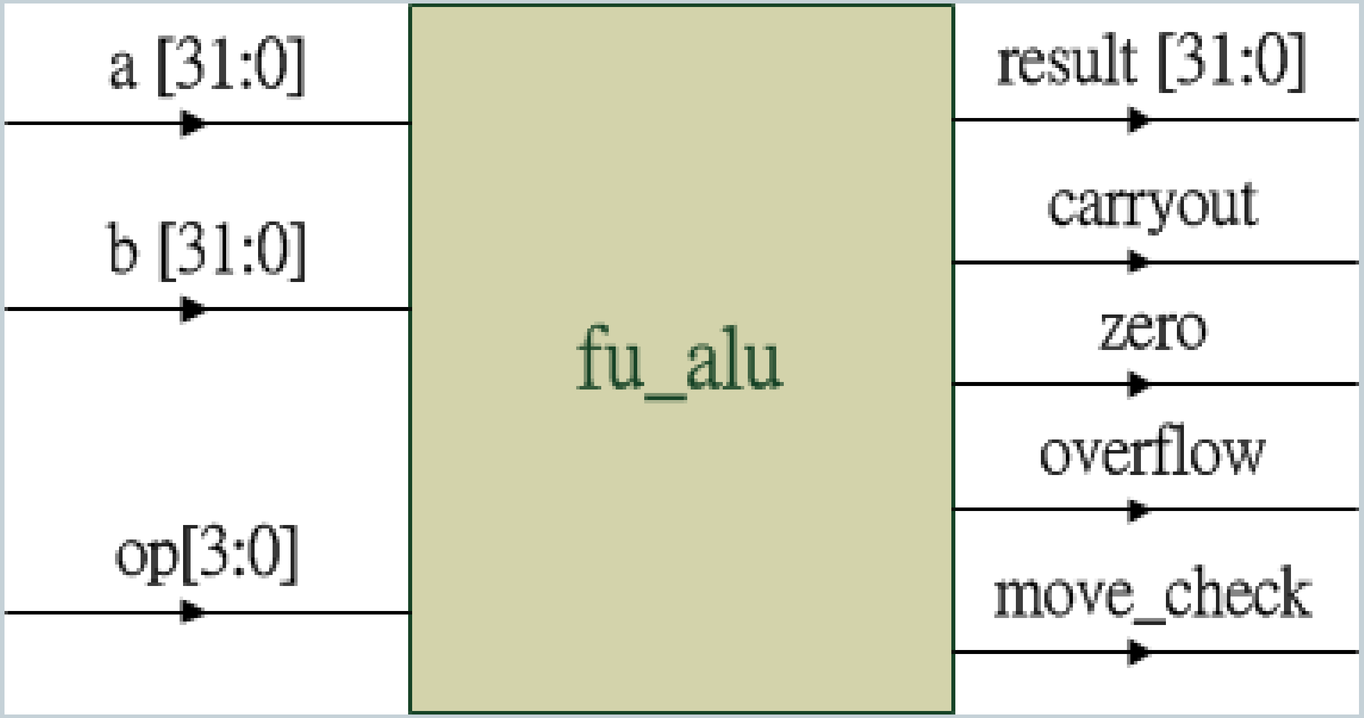
Table 1: ALU運算指令

ARITHMETIC OPERATIONS	
ADD RD , RS , RT	RD = RS + RT (OVERFLOW TRAP)
ADDU RD , RS , RT	RD = RS + RT
SUB RD , RS , RT	RD = RS – RT (OVERFLOW TRAP)
SUBU RD , RS , RTRD = RS – RT	RD = RS – R
AND RD , RS , RT	RD = RS & RT
OR RD , RS , RT	RD = RS | RT
XOR RD , RS , RT	RD = RS ⊕ RT
NOR RD , RS , RT	RD = ~(RS | RT)
ADDI RD , RS , CONST 16	RD = RS + CONST 16 ± (OVERFLOW TRAP)
ADDIU RD , RS , CONST 16	RD = RS + CONST 16 ±
SLT RD , RS , RT	RD = (RS ± < RT ±) ? 1 : 0
SLTU RD , RS , RT	RD = (RS∅ < RT∅) ? 1 : 0
ANDI RD , RS , CONST 16	RD = RS & CONST 16 ∅
ORI RD , RS , CONST 16	RD = RS | CONST 16 ∅
XORI RD , RS , CONST 16	RD = RS ⊕ CONST 16 ∅
LUI RD , CONST 16	RD = CONST 16 << 16

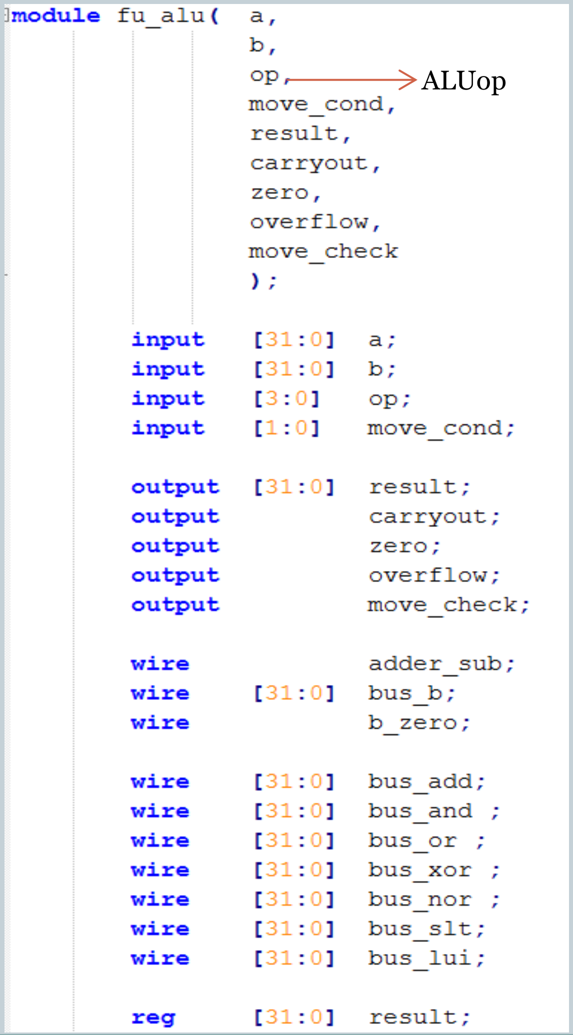
Table 2: ALUop & ALU Function

	OPCODE	funct	ALUop	ALU Function
ADD	00 0000	10 0000	0000	ADD
ADDU	00 0000	10 0001	0001	ADD
SUB	00 0000	10 0010	0010	SUB
SUBU	00 0000	10 0011	0011	SUB
AND	00 0000	10 0100	0100	AND
OR	00 0000	10 0101	0101	OR
XOR	00 0000	10 0110	0110	XOR
NOR	00 0000	10 0111	0111	NOR
ADDI	00 1000	xx xxxx	1000	ADD
ADDIU	00 1001	xx xxxx	1001	ADD
SLT	00 0000	10 1010	1010	SUB
SLTU	00 0000	10 1011	1011	SUB
ANDI	00 1100	xx xxxx	1100	AND
ORI	00 1101	xx xxxx	1101	OR
XORI	00 1110	xx xxxx	1110	XOR
LUI	00 1111	xx xxxx	1111	NOP

ALU module

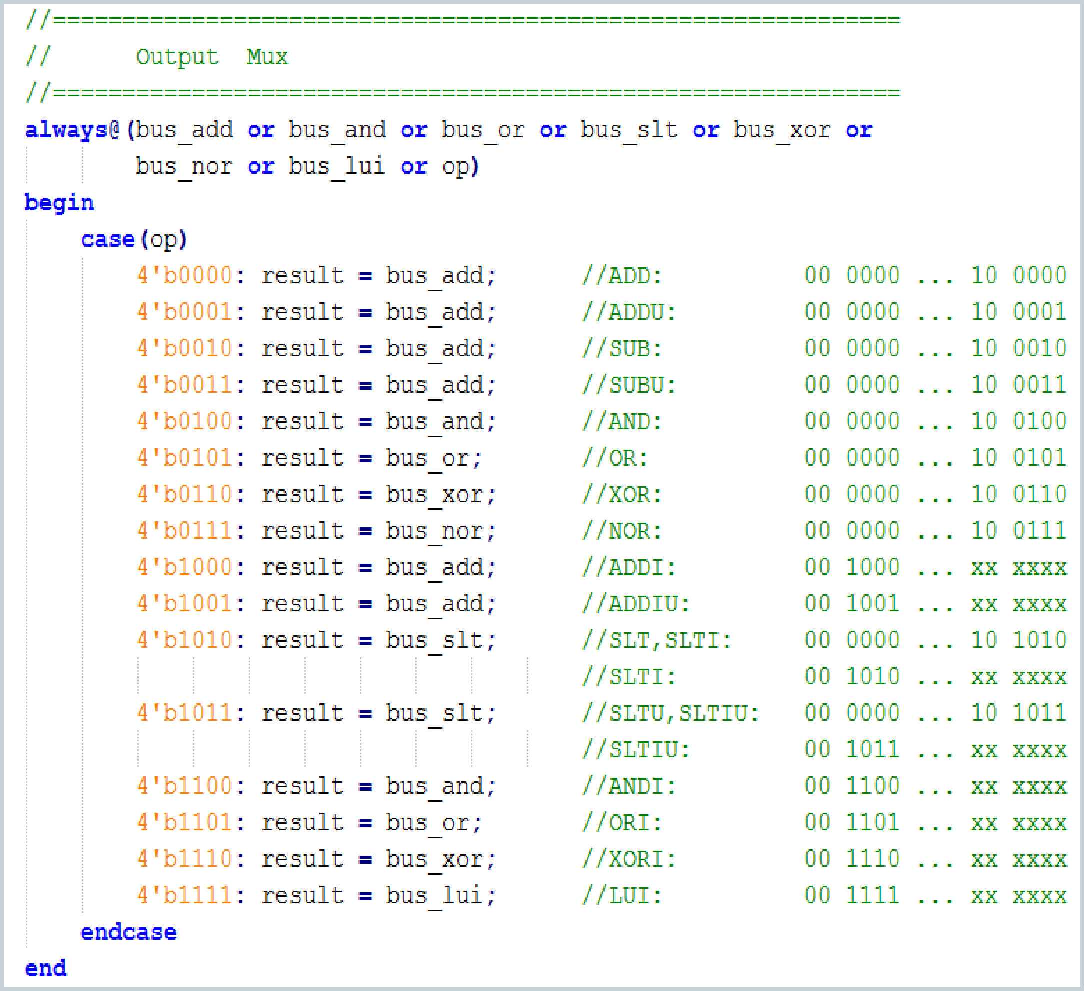

a [31:0]
b [31:0]
op[3:0]
fu_alu
result [31:0]
carryout
zero
overflow
move_check

ALU module



```
module fu_alu(  a,
                b,
                op,
                move_cond,
                result,
                carryout,
                zero,
                overflow,
                move_check
                );

        input   [31:0]  a;
        input   [31:0]  b;
        input   [3:0]   op;
        input   [1:0]   move_cond;

        output  [31:0]  result;
        output          carryout;
        output          zero;
        output          overflow;
        output          move_check;

        wire            adder_sub;
        wire    [31:0]  bus_b;
        wire            b_zero;

        wire    [31:0]  bus_add;
        wire    [31:0]  bus_and ;
        wire    [31:0]  bus_or ;
        wire    [31:0]  bus_xor ;
        wire    [31:0]  bus_nor ;
        wire    [31:0]  bus_slt;
        wire    [31:0]  bus_lui;

        reg     [31:0]  result;
```

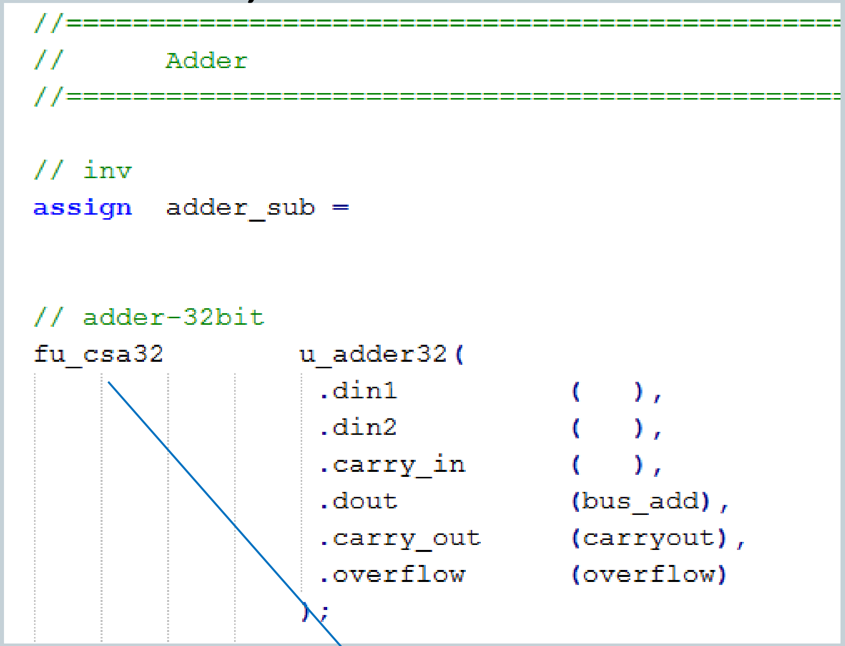
ALUop



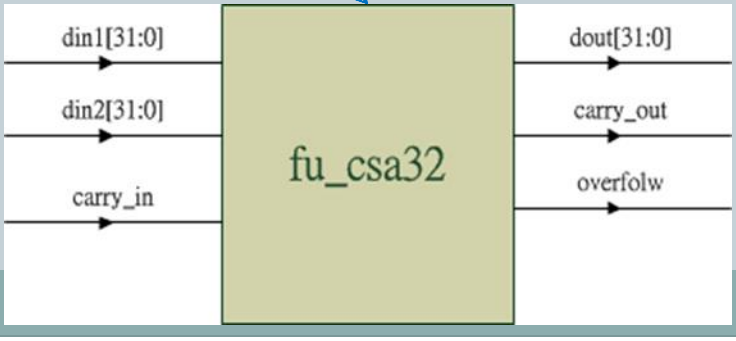
```
//==============================================================
//      Output  Mux
//==============================================================
always@(bus_add or bus_and or bus_or or bus_slt or bus_xor or
        bus_nor or bus_lui or op)
begin
    case(op)
        4'b0000: result = bus_add;      //ADD:          00 0000 ... 10 0000
        4'b0001: result = bus_add;      //ADDU:         00 0000 ... 10 0001
        4'b0010: result = bus_add;      //SUB:          00 0000 ... 10 0010
        4'b0011: result = bus_add;      //SUBU:         00 0000 ... 10 0011
        4'b0100: result = bus_and;      //AND:          00 0000 ... 10 0100
        4'b0101: result = bus_or;       //OR:           00 0000 ... 10 0101
        4'b0110: result = bus_xor;      //XOR:          00 0000 ... 10 0110
        4'b0111: result = bus_nor;      //NOR:          00 0000 ... 10 0111
        4'b1000: result = bus_add;      //ADDI:         00 1000 ... xx xxxx
        4'b1001: result = bus_add;      //ADDIU:        00 1001 ... xx xxxx
        4'b1010: result = bus_slt;      //SLT,SLTI:     00 0000 ... 10 1010
                                        //SLTI:         00 1010 ... xx xxxx
        4'b1011: result = bus_slt;      //SLTU,SLTIU:   00 0000 ... 10 1011
                                        //SLTIU:        00 1011 ... xx xxxx
        4'b1100: result = bus_and;      //ANDI:         00 1100 ... xx xxxx
        4'b1101: result = bus_or;       //ORI:          00 1101 ... xx xxxx
        4'b1110: result = bus_xor;      //XORI:         00 1110 ... xx xxxx
        4'b1111: result = bus_lui;      //LUI:          00 1111 ... xx xxxx
    endcase
end
```

ALU Function

- ADD , SUB



```
//==============================================
//      Adder
//==============================================

// inv
assign  adder_sub =


// adder-32bit
fu_csa32         u_adder32(
                  .din1          (   ),
                  .din2          (   ),
                  .carry_in      (   ),
                  .dout          (bus_add),
                  .carry_out     (carryout),
                  .overflow      (overflow)
                  );
```



- 數的表示法:2的補數表示法(有號數)
- 數的運算:
  - 加法
    - ex. 2 + 3 = 5

      0010+0011 = 0101
  - 減法
    - ex. 2 - 3 = -1
    - 數值 - 數值 = 數值 + (-數值)

      0010+(1100+1) = 1111

  # -3為[3的1的補數] + 1

  0011 3

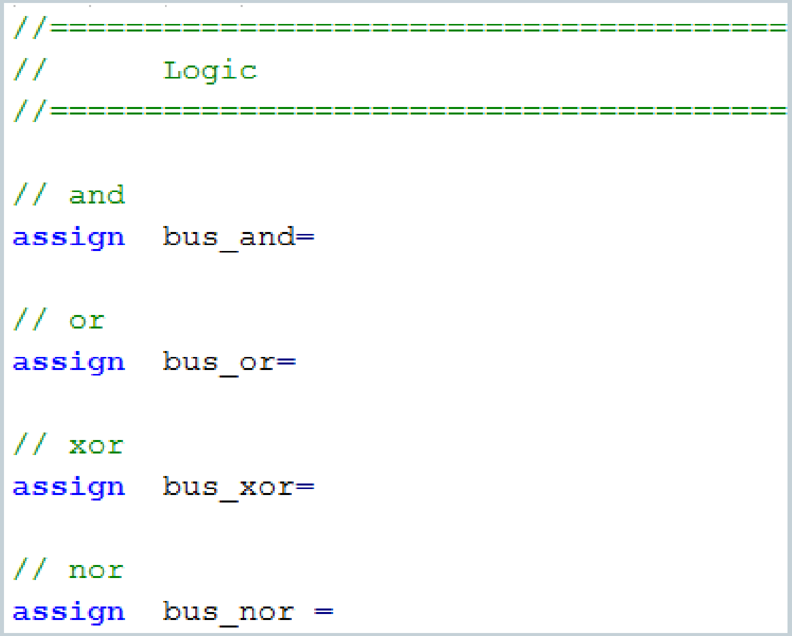
  ↓ 1的補數

  $\underline{1100\ \ +1}$

  1101 -3

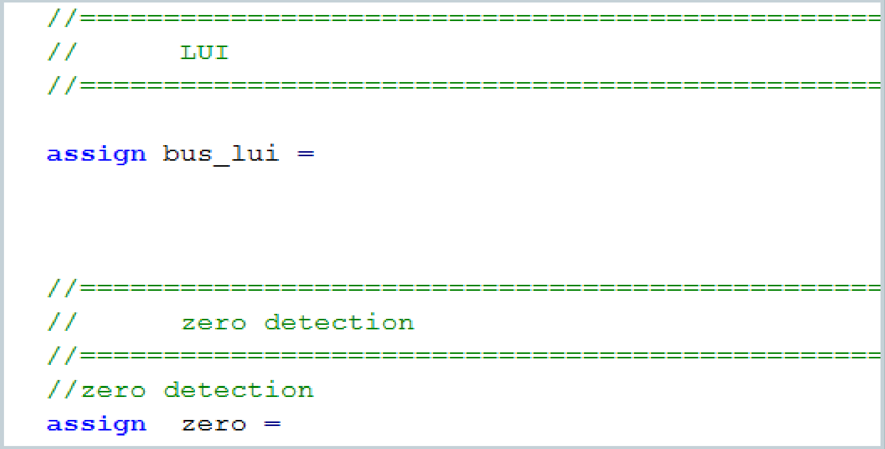
ALU Function

- Logic



```
//=======================================
//       Logic
//=======================================

// and
assign  bus_and=

// or
assign  bus_or=

// xor
assign  bus_xor=

// nor
assign  bus_nor =
```

- LUI , zero



```
//===============================================
//       LUI
//===============================================

assign bus_lui =



//===============================================
//       zero detection
//===============================================
//zero detection
assign  zero =
```

ALU驗證程式

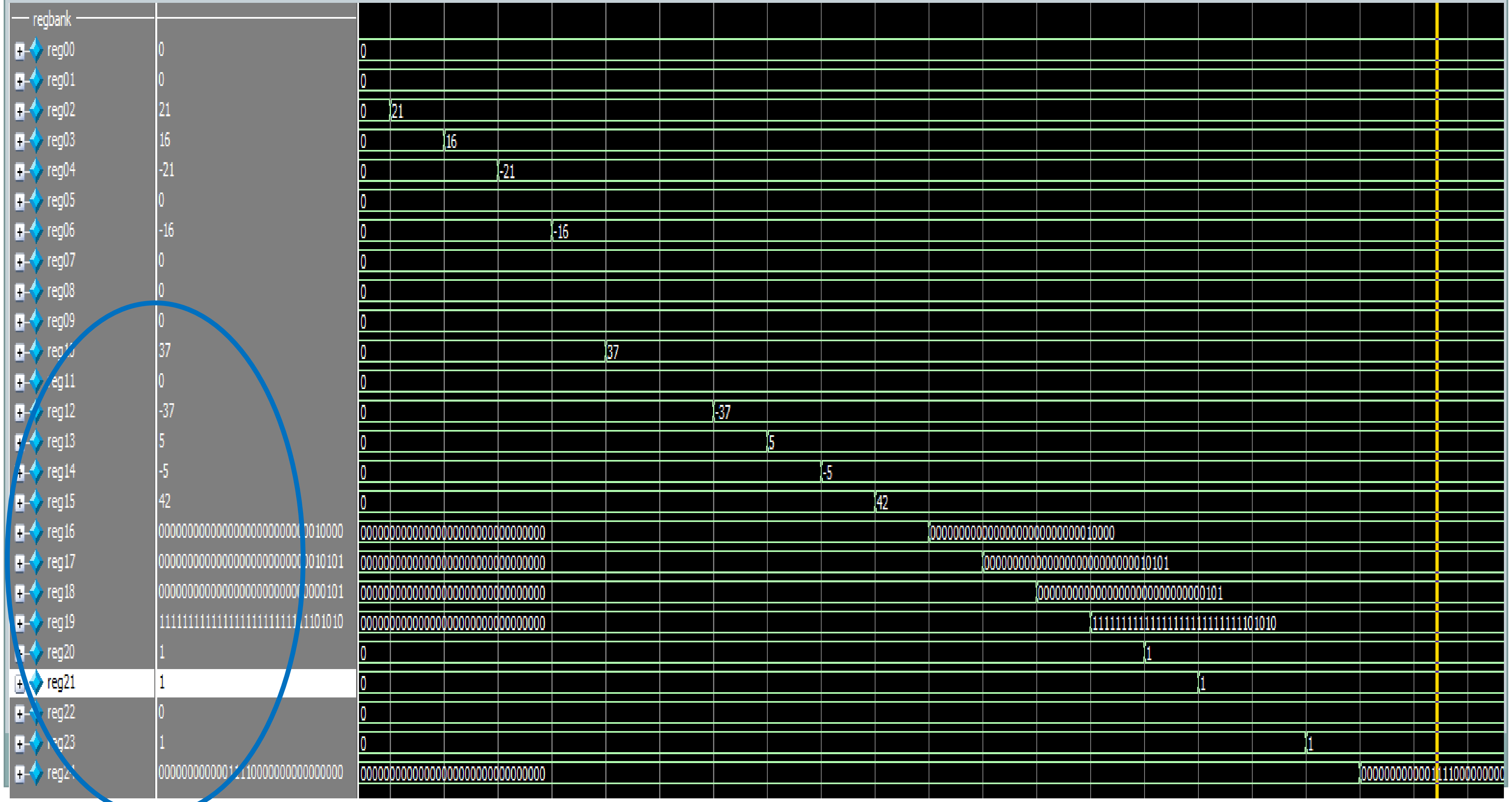
- 簡單的組語驗證（答案印在右邊）✓ 此份組語，驗證了藍色底的算數指令。
  - ➢ 此份組語程式，已轉為code.mif檔，且存在於verilog_test_branch/software_mips/下

```
move     $1 ,$0
li   $2 , 21       #10101
li   $3 , 16 #10000
li   $4 , -21      #1..101011
li   $6, -16 #1..111000

   add       $10,$2,$3             #37
   add       $11,$2,$4             #0
   add       $12, $4,$6            #-37
   sub       $13, $2,$3            #5
   sub       $14 ,$4,$6            #-5
   sub       $15, $2,$4            #42
   and       $16, $2,$3            #10000
   or        $17, $2,$3            #10101
   xor       $18, $2,$3            #101
   nor       $19, $2,$3            #1..101010
   slt       $20, $3,$2            #1
   slt       $21, $4,$6            #1
   sltu $22,$4,$3                  #0
   sltu $23,$3,$4                  #1
   lui       $24, 0xf

.EXIT:
    j    $31
    nop
```

ARITHMETIC OPERATIONS		
ADD	RD , RS , RT	RD = RS + RT (OVERFLOW TRAP)
ADDU	RD , RS , RT	RD = RS + RT
SUB	RD , RS , RT	RD = RS – RT (OVERFLOW TRAP)
SUBU	RD , RS , RTRD = RS – RT	RD = RS – R
AND	RD , RS , RT	RD = RS & RT
OR	RD , RS , RT	RD = RS | RT
XOR	RD , RS , RT	RD = RS ⊕ RT
NOR	RD , RS , RT	RD = ~(RS | RT)
ADDI	RD , RS , CONST 16	RD = RS + CONST 16 ± (OVERFLOW TRAP)
ADDIU	RD , RS , CONST 16	RD = RS + CONST 16 ±
SLT	RD , RS , RT	RD = (RS ± < RT ±) ? 1 : 0
SLTU	RD , RS , RT	RD = (RS∅ < RT∅) ? 1 : 0
ANDI	RD , RS , CONST 16	RD = RS & CONST 16 ∅
ORI	RD , RS , CONST 16	RD = RS | CONST 16 ∅
XORI	RD , RS , CONST 16	RD = RS ⊕ CONST 16 ∅
LUI	RD , CONST 16	RD = CONST 16 << 16

ALU預期結果

- View→Wave (打開wave 視窗)
- File→Load→Marco File... →wave_alu.do (載入要觀察的部分)



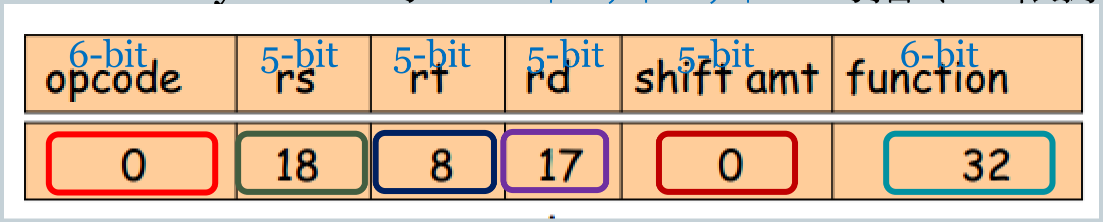
```
regbank
reg00   0
reg01   0
reg02   21
reg03   16
reg04   -21
reg05   0
reg06   -16
reg07   0
reg08   0
reg09   0
reg10   37
reg11   0
reg12   -37
reg13   5
reg14   -5
reg15   42
reg16   00000000000000000000000000010000
reg17   00000000000000000000000000010101
reg18   00000000000000000000000000000101
reg19   11111111111111111111111111101010
reg20   1
reg21   1
reg22   0
reg23   1
reg24   00000000000011110000000000000000
```

挑戰題

在5-stage pipeline的 CPU當中，IF(Instruction Fetch)負責去拿instruction，並且把這個 32-bit 的 instruction 傳給下一級， Decode 階段則負責把 IF 準備好的 32-bit instruction 做解碼的動作。

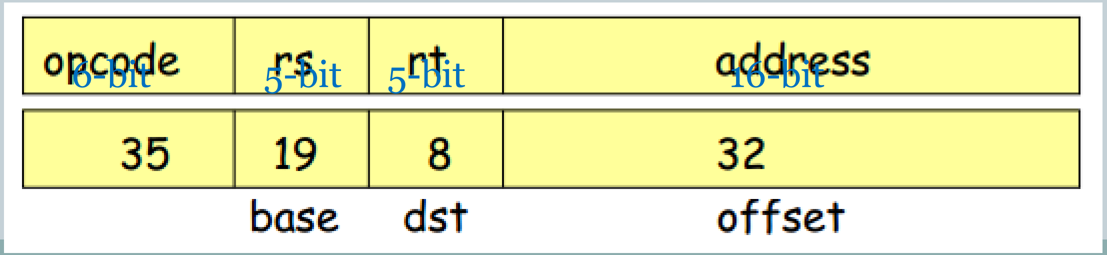
挑戰題當中，請同學用 Verilog 實作出一個簡單的 Decode Stage

Example:
Assembly code 為 add $s1, $s2, $t0 的指令，根據表格可以得到



6-bit opcode	5-bit rs	5-bit rt	5-bit rd	5-bit shift amt	6-bit function
0	18	8	17	0	32

Machine code 為 00000010010010001000100000100000
這一串32-bit 的 machine 就會是IF拿到的指令，並且傳到Decode 做為 input



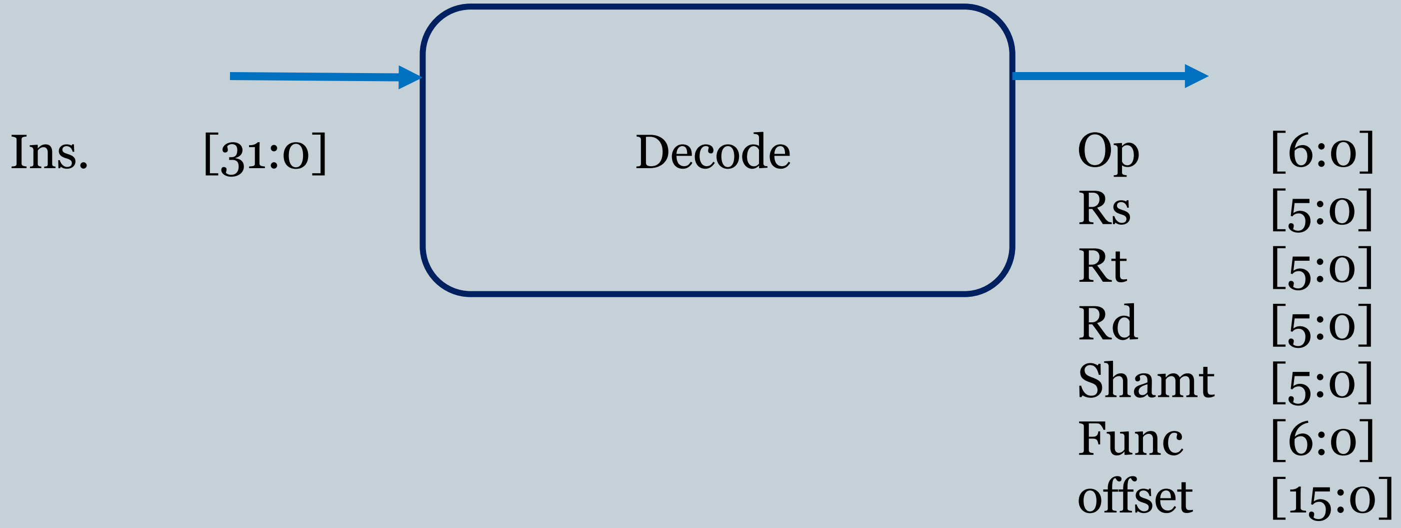
opcode 6-bit	rs 5-bit	rt 5-bit	address 16-bit
35	19	8	32
	base	dst	offset

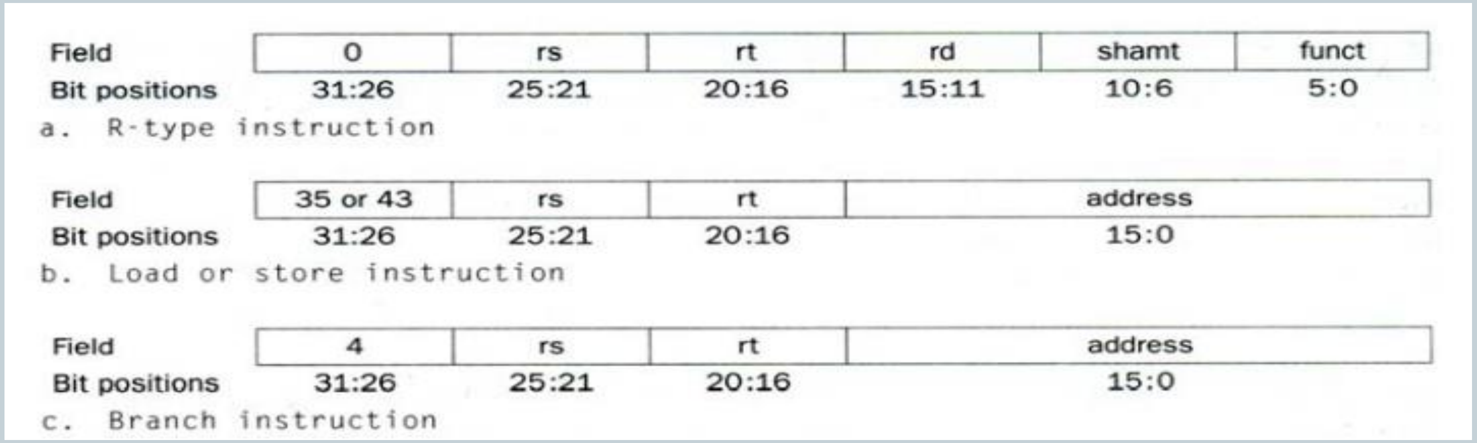
挑戰題

我們要實作的Decode 就是負責把 input instruction 對應的 opcode, Rs, Rt, Rd, shamt, function, offset 傳出來

```
Ex: input    ins = 000000 11111 00111 01001 00000 000011
    output   op       = 000000
             Rs       =  11111
             Rt       =  00111
             Rd       =  01001
             shamt    =  00000
             function = 000011
             op = 0，是 R-type的指令，所以 offset = 0
    input    ins = 100011 00001 01000 0000000000100000
    output   op       = 100011
             Rs       =  00001
             Rt       =  01000
             offset   = 0000000000100000
             op = 0，是 I-type的指令，所以 Rd =0, shamt =0, func = 0
```

挑戰題





Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

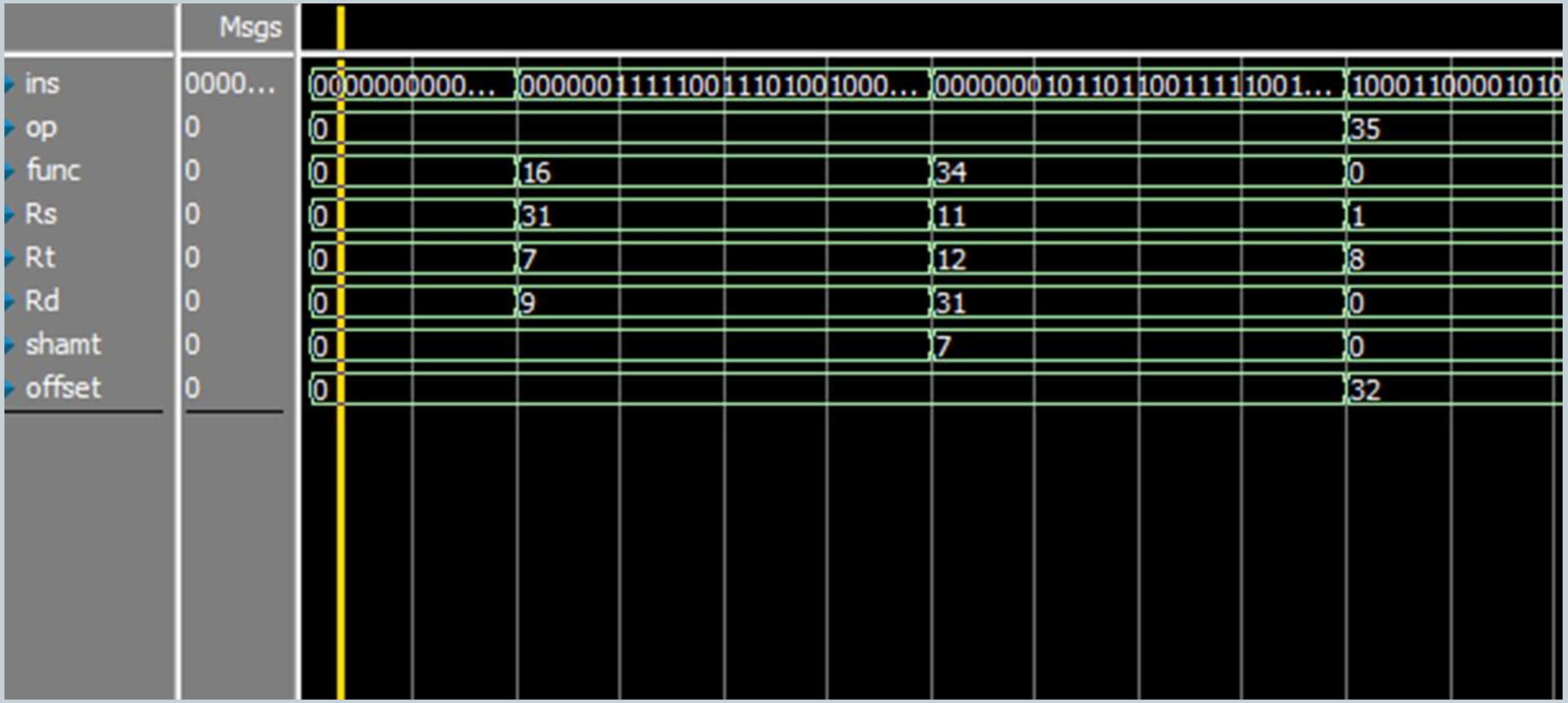
Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

結果驗證



```
        Msgs
ins     0000...   0000000000...  000000111110011101001000...  000000010110110011111001...  10001100001010
op      0         0                                                                          35
func    0         0              16                            34                            0
Rs      0         0              31                            11                            1
Rt      0         0              7                             12                            8
Rd      0         0              9                             31                            0
shamt   0         0                                            7                             0
offset  0         0                                                                          32
```