處理器設計與實作

實習講義

編撰者
成大電通所計算機架構與系統研究室CASLAB

國立成功大學電機系與電腦與通信工程研究所

LAB 3: Verilog Implementation Of Arithmetic Logic Unit (ALU) 算術邏輯單元的實作

大綱

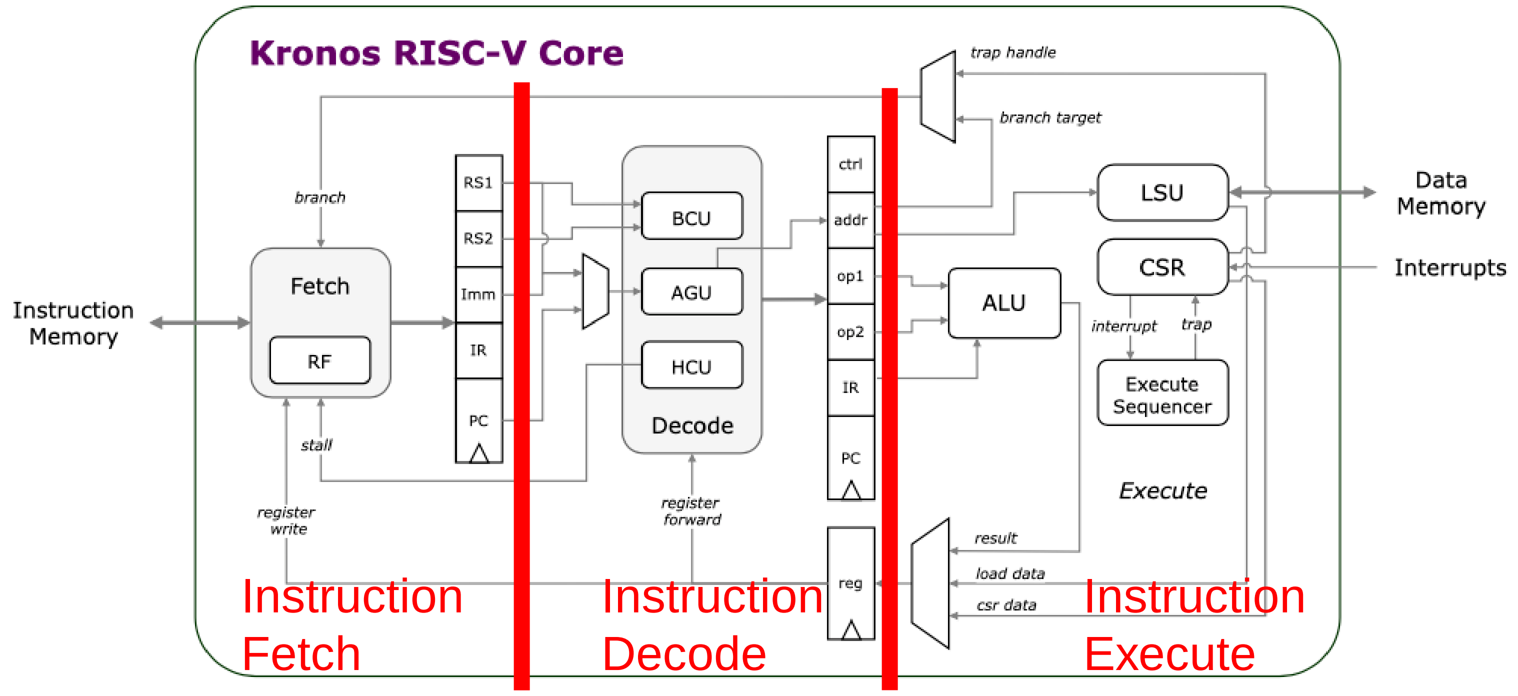
1. RISCV CPU
   - Microarchitecture
   - Control Unit
2. Verilog 基本概念
3. 實驗: 實踐ALU
   - 練習題一
   - 練習題二
   - 挑戰題

實驗目的

1. 認識 RISC-V CPU 基本的 data path & control path
2. Verilog 的基本認識
3. 認識 ALU control unit
4. Hierarchy Design in Verilog
5. 實作 ALU 之基本運算功能的行為模組

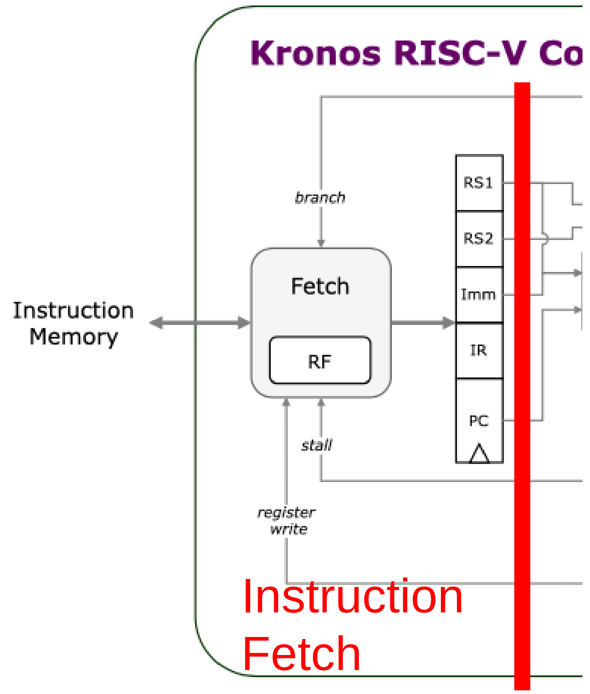
Kronos RISC-V CPU

Kronos RISC-V core architecture



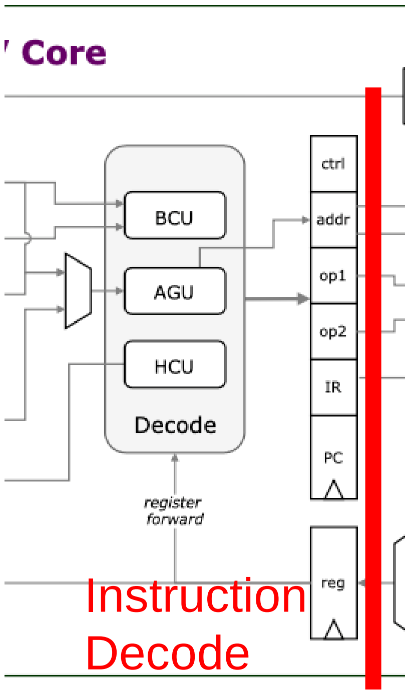
- 3-stage pipeline: Fetch, Decode, Execute
- In-order
- ISA: 32I

Kronos RISC-V core architecture



- The instruction fetch stage reads 32-bit instructions every cycle
- The PC always increments to next word (PC+4) unless the core needs to jump or to trap handler

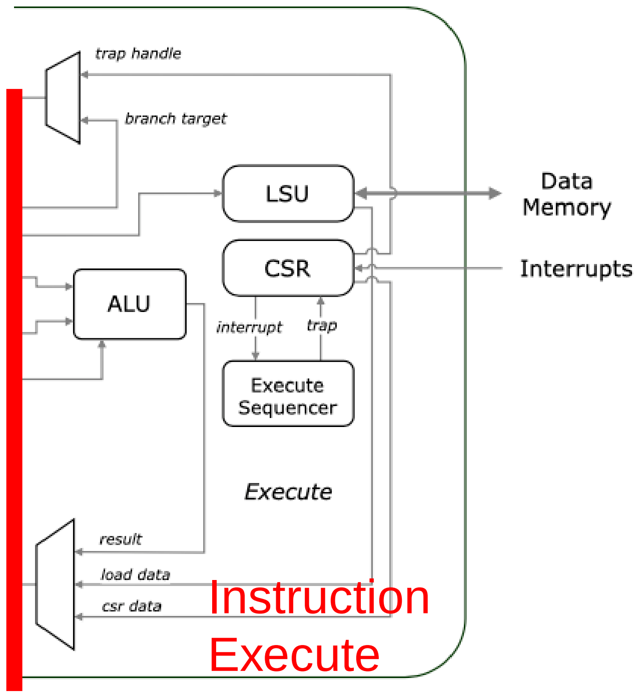
Kronos RISC-V core architecture



- In decode stage, Two operands (op1, op2) are prepared for the ALU alongside the appropriate ALU operation for the fetch instruction.

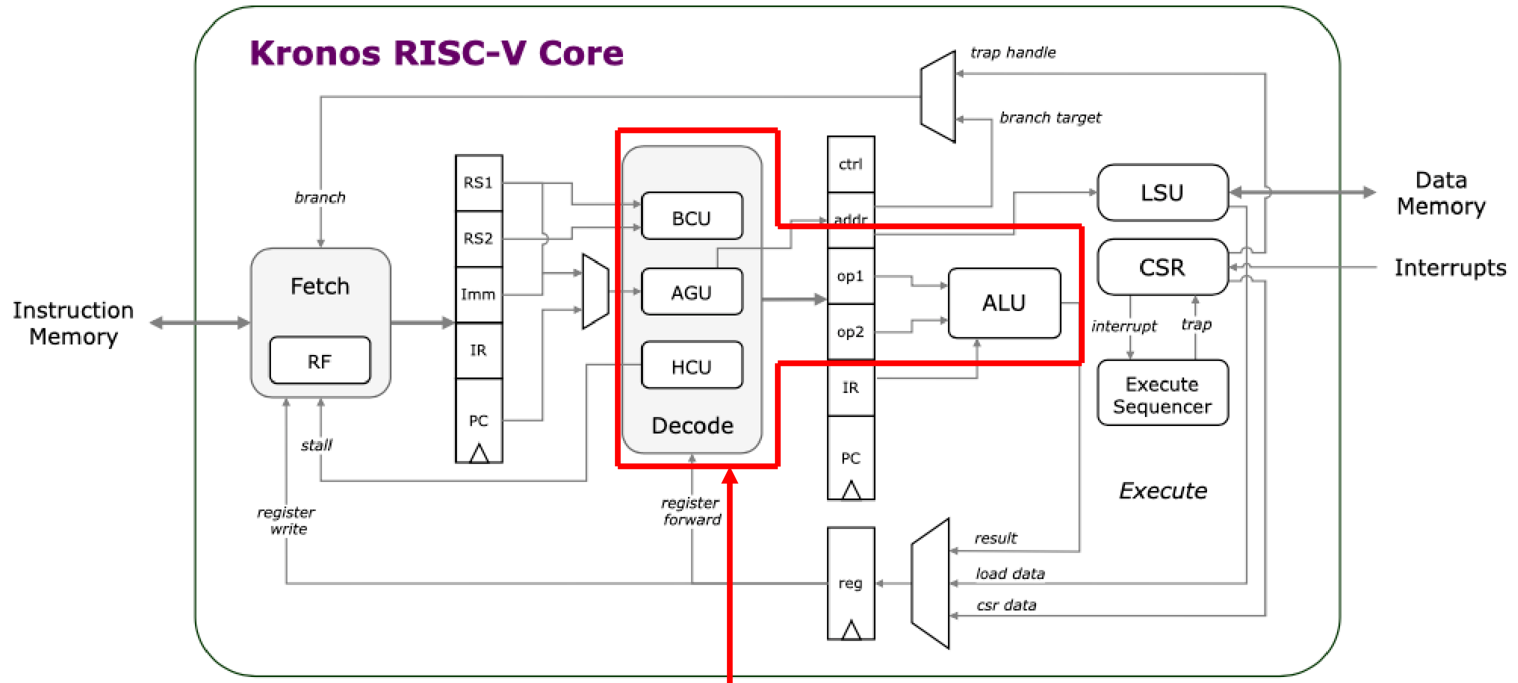
- BCU: Branch Comparison Unit
- AGU: Address Generation Unit
- HCU: Hazard Control Unit

Kronos RISC-V core architecture



- In execution stage of the Kronos core contains a ALU which is responsible for data write back, branching, sequence memory access operations (data load/store), Control-Status Register operations, catches exception & responds to interrupts.

Kronos RISC-V core architecture



本次實驗室主要著重於 Decoder & ALU 這部分的實作

Verilog 基本概念

HDL Introduction

- 硬體描述語言

- 具有多種描述硬體的方式
  - Behavior level
    - 只考慮模組中的功能和函數，不必考慮硬體方面的詳細電路。
  - Register Transfer level
    - 說明資料如何在暫存器中儲存和傳送，和資料處理的方式。
  - Gate level
    - 模組是由Logic gates所構成的，使用Logic gates來設計電路。
  - Switch level
    - 最低層次，設計者需知道電晶體的元件特性。

Verilog Background

- Verilog was written by gateway design automation in the early 1980
- Cadence acquired gateway in 1990
- Cadence released Verilog to the public domain in 1991
- In 1995, the language was ratified as IEEE standard 1364

Three Levels of Verilog

- Register Transfer Level

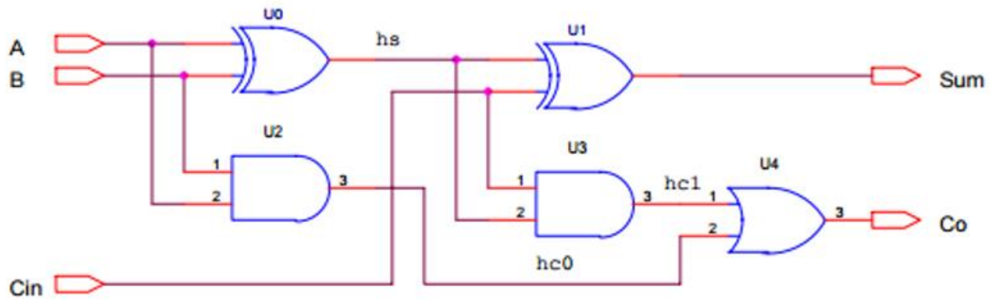
```
assign {Co, Sum} = A + B + Ci
```

- Gate Level

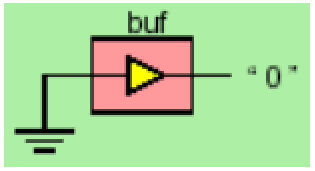
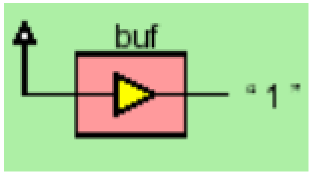
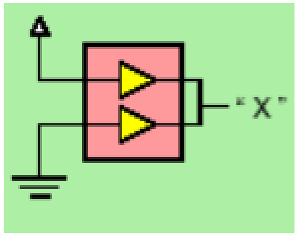
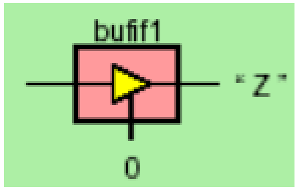
```
xor  u0(.z(hs), .a1(A), .a2(B));
xor  u1(.z(Sum), .a1(Ci), .a2(hs));
and  u2(.z(hc0), .a1(A), .a2(B));
and  u3(.z(hc1), .a1(Ci), .a2(hs));
or   u4(.z(Co), .a1(hc0), .a2(hc1));
```

- Switch Level

```
// AND gate of u2
pmos  p0(VDD, nand, A),
      p1(VDD, nand, B);
nmos  n0(nand, wire1, A),
      n1(wire1, GND, B);

pmos  p2(VDD, hc0, nand);
nmos  n2(hc0, GND, nand);
  ⋮
```

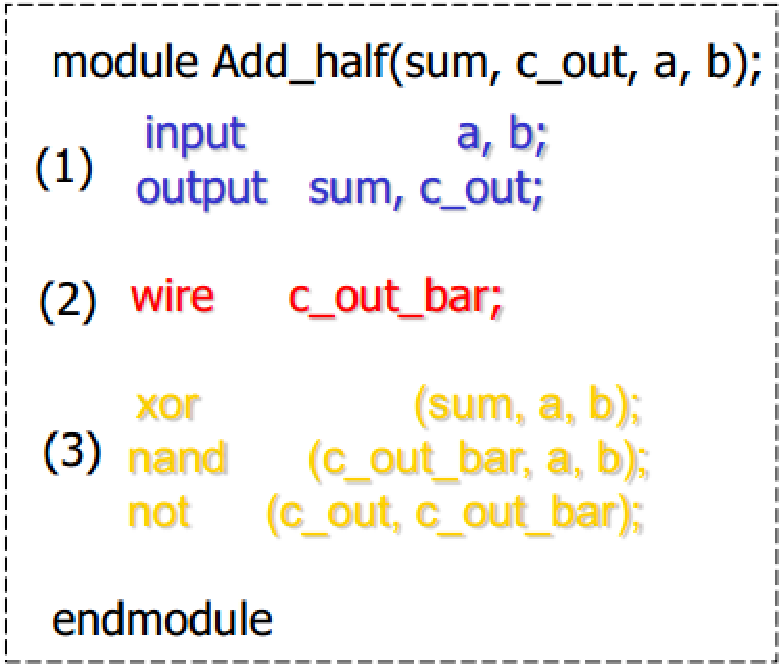


Four Logic Levels

- 
  0: logic 0 / false
- 
  1: logic 1 / true
- 
  X: unknown logic value
- 
  Z: high-impedance

Verilog Module

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



```
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
```

Verilog Operators (1/3)

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

Verilog 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 ~^

Verilog 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	?:

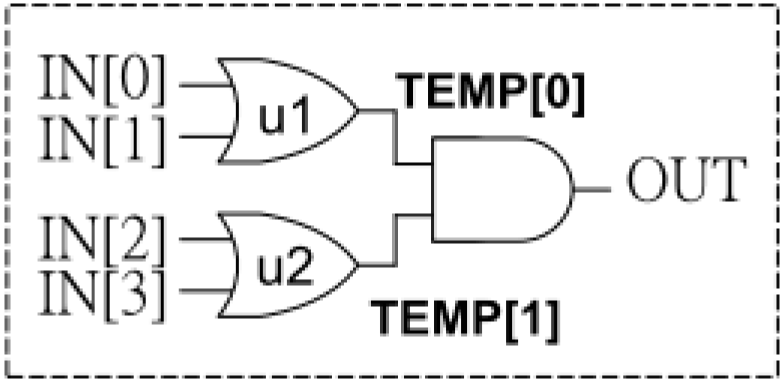
Verilog Descriptions

- Verilog 主要有三種描述硬體電路的方式
  1. Structural description
  2. “assign” description
  3. “always” description

Structural Description

- Gate level design
- Connections of sub modules at higher level module

```
module OR_AND_STRUCTURAL(IN,OUT);

input       [3:0]    IN;
output               OUT;
wire        [1:0]    TEMP;

or u1(TEMP[0], IN[0], IN[1]);
or u2(TEMP[1], IN[2], IN[3]);
and (OUT, TEMP[0], TEMP[1]);
endmodule
```

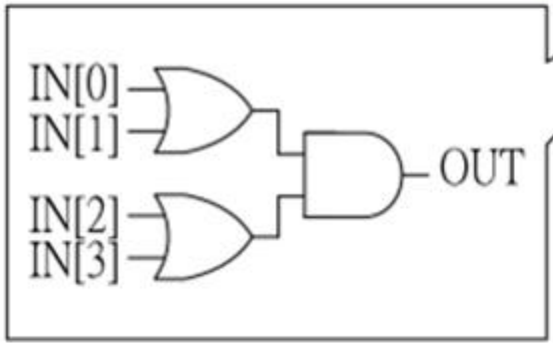


"assign" Description

- Combinational circuit

```
module OR_AND_DATA_FLOW(IN, OUT);
input       [3:0]      IN;
output                 OUT;

assign OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);

endmodule
```

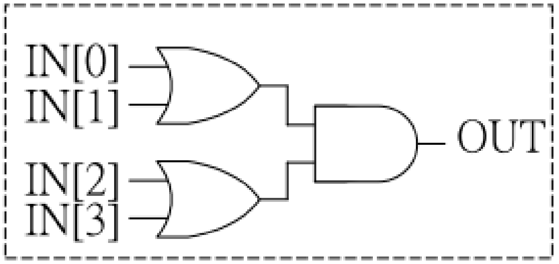
Synthesized and optimized by tools



"always" Description

- Combinational circuit
- Sequential circuit
- Output must be declared as "reg"

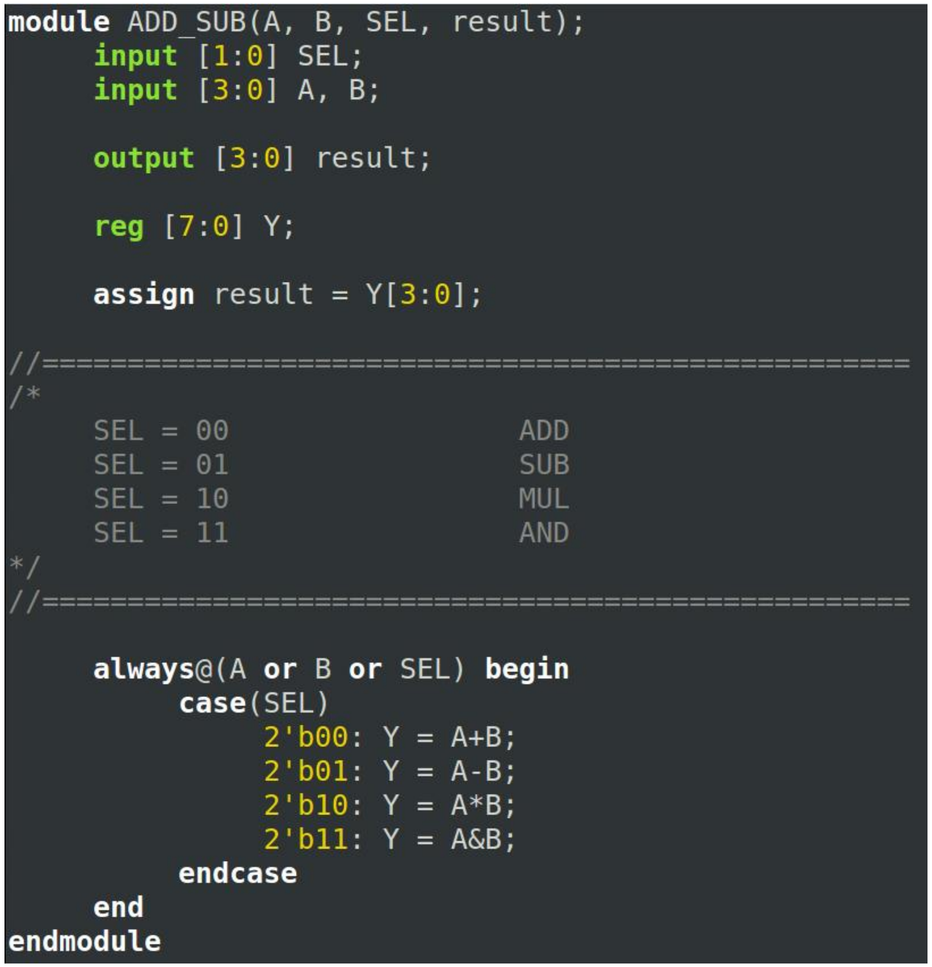
```
module OR_AND_BEHAVIORAL(IN, OUT);

input  [3:0]   IN;
output         OUT;
reg            OUT;

always @(IN)
begin
   OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);
end

endmodule
```



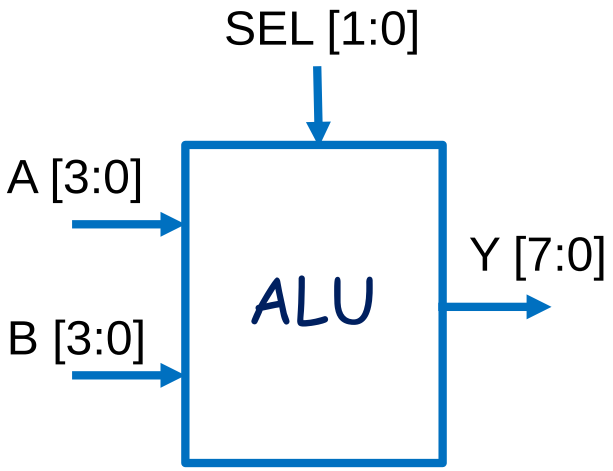
Activate OUT while any voltage transition

(0→1 or 1→0) happens at signal IN

ALU EXAMPLE



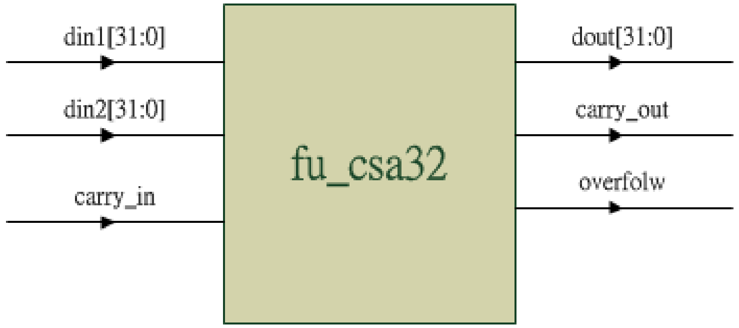
```
module ADD_SUB(A, B, SEL, result);
     input [1:0] SEL;
     input [3:0] A, B;

     output [3:0] result;

     reg [7:0] Y;

     assign result = Y[3:0];

//=====================================================
/*
     SEL = 00                    ADD
     SEL = 01                    SUB
     SEL = 10                    MUL
     SEL = 11                    AND
*/
//=====================================================

     always@(A or B or SEL) begin
          case(SEL)
               2'b00: Y = A+B;
               2'b01: Y = A-B;
               2'b10: Y = A*B;
               2'b11: Y = A&B;
          endcase
     end
endmodule
```

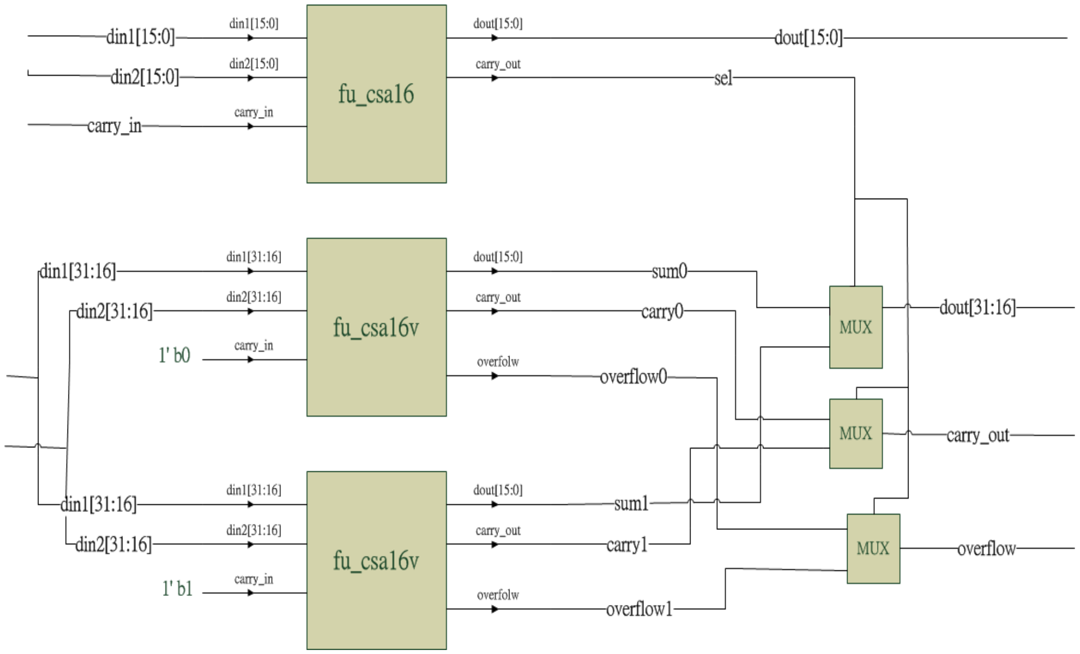


Hierarchy in Verilog

- 32-bit Carry-Select adder

```
module fu_csa32(din1, din2, carry_in, dout, carry_out, overflow);

        input   [31:0]  din1;
        input   [31:0]  din2;
        input           carry_in;
        output  [31:0]  dout;
        output          carry_out;
        output          overflow;

        wire            sel;
        wire    [15:0]  sum0, sum1;
        wire            carry0, carry1;
        wire            overflow0, overflow1;

        fu_csa16        u0(
                        .din1           (din1[15:0]),
                        .din2           (din2[15:0]),
                        .carry_in       (carry_in),
                        .dout           (dout[15:0]),
                        .carry_out      (sel)
                        );

        fu_csa16v       u1(
                        .din1           (din1[31:16]),
                        .din2           (din2[31:16]),
                        .carry_in       (1'b0),
                        .dout           (sum0),
                        .carry_out      (carry0),
                        .overflow       (overflow0)
                        );

        fu_csa16v       u2(
                        .din1           (din1[31:16]),
                        .din2           (din2[31:16]),
                        .carry_in       (1'b1),
                        .dout           (sum1),
                        .carry_out      (carry1),
                        .overflow       (overflow1)
                        );

        assign  dout[31:16] = (sel==1'b1)?sum1:sum0;
        assign  carry_out = (sel==1'b1)?carry1:carry0;
        assign  overflow = (sel==1'b1)?overflow1:overflow0;

endmodule
```





LAB 3: Verilog Implementation Of ALU

Tool used

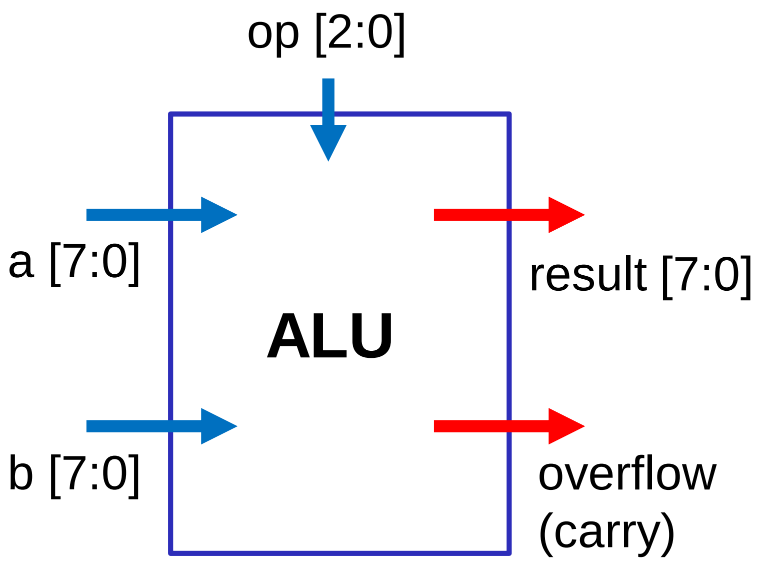
實驗環境:

1.Modelsim (Run CPU simulator)

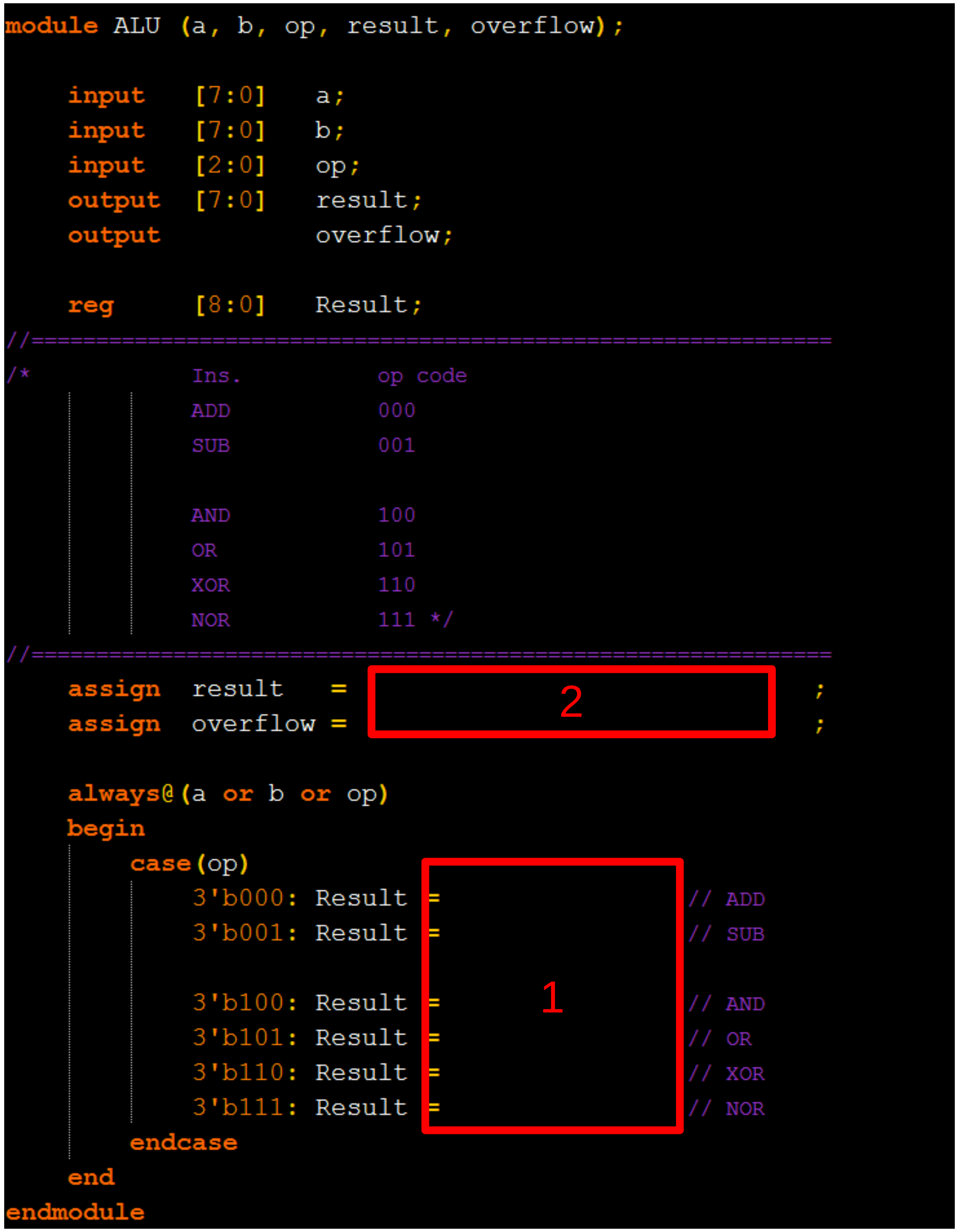
- 不會用到virtualbox

Lab 3-1 -- 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   =                               ;
    assign  overflow =                               ;

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

            3'b100: Result =                  // 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

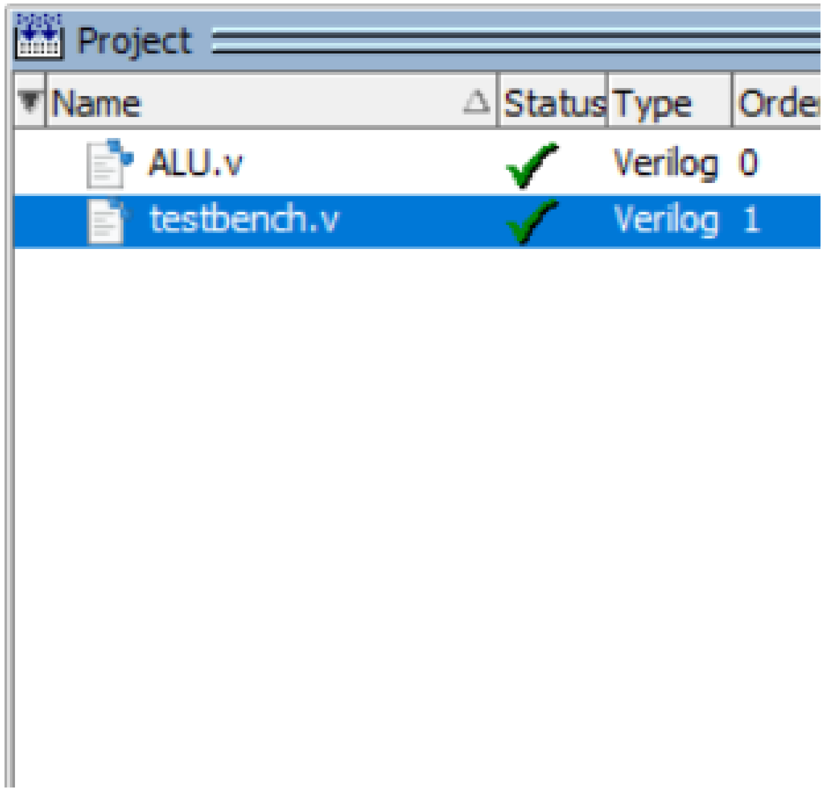
Step2: 把ALU運算完的結果
傳到output (result)
ALU在做加減法運算時，
可能會有overflow產生，
請判斷什麼時候有overflow

if (op = ADD/SUB)
check overflow

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

Modelsim驗證

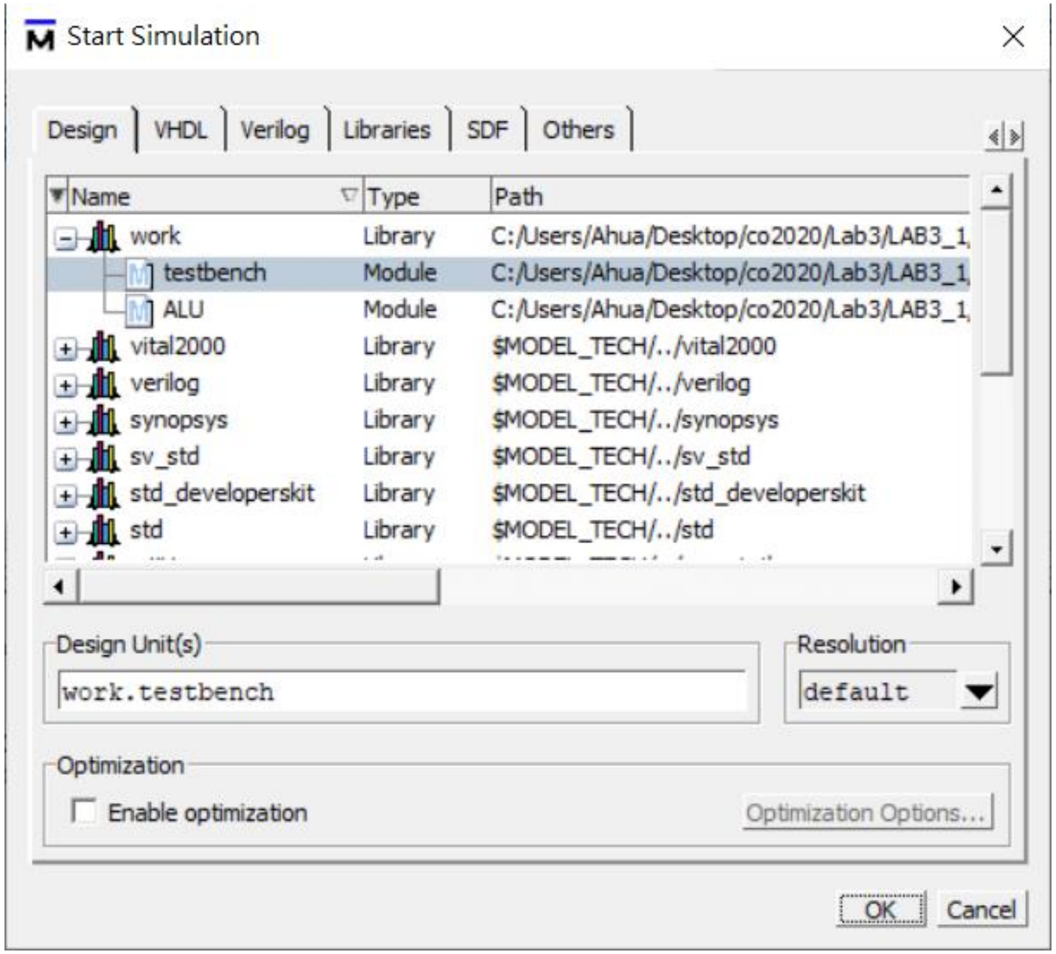
- 加入ALU.v 及 testbench.v 兩個檔案



Name	Status	Type	Orde
ALU.v	✓	Verilog	0
testbench.v	✓	Verilog	1

Modelsim驗證

- Simulation 選擇 work -> testbench



Name	Type	Path
work	Library	C:/Users/Ahua/Desktop/co2020/Lab3/LAB3_1
testbench	Module	C:/Users/Ahua/Desktop/co2020/Lab3/LAB3_1
ALU	Module	C:/Users/Ahua/Desktop/co2020/Lab3/LAB3_1
vital2000	Library	$MODEL_TECH/../vital2000
verilog	Library	$MODEL_TECH/../verilog
synopsys	Library	$MODEL_TECH/../synopsys
sv_std	Library	$MODEL_TECH/../sv_std
std_developerskit	Library	$MODEL_TECH/../std_developerskit
std	Library	$MODEL_TECH/../std

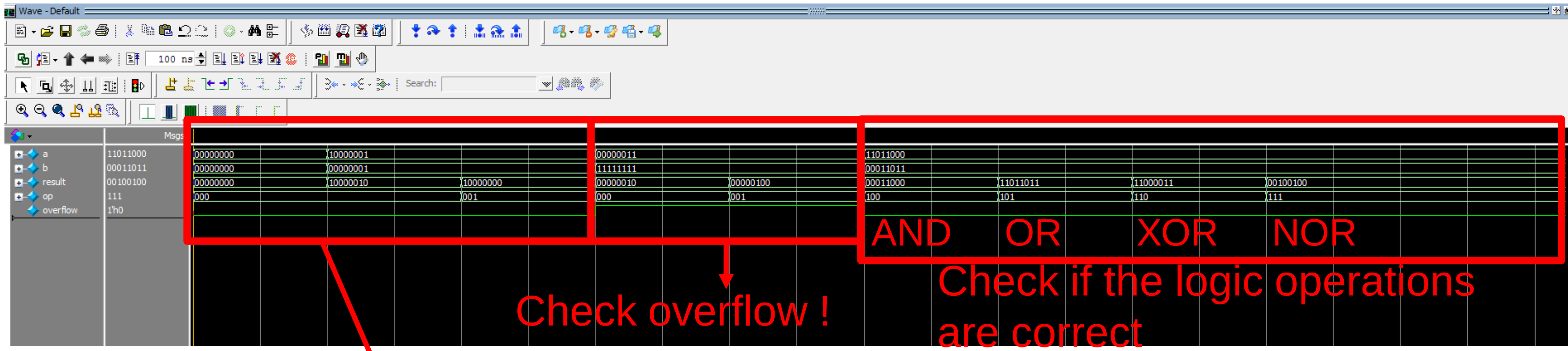
Design Unit(s): work.testbench

Resolution: default

Optimization

☐ Enable optimization

預期結果

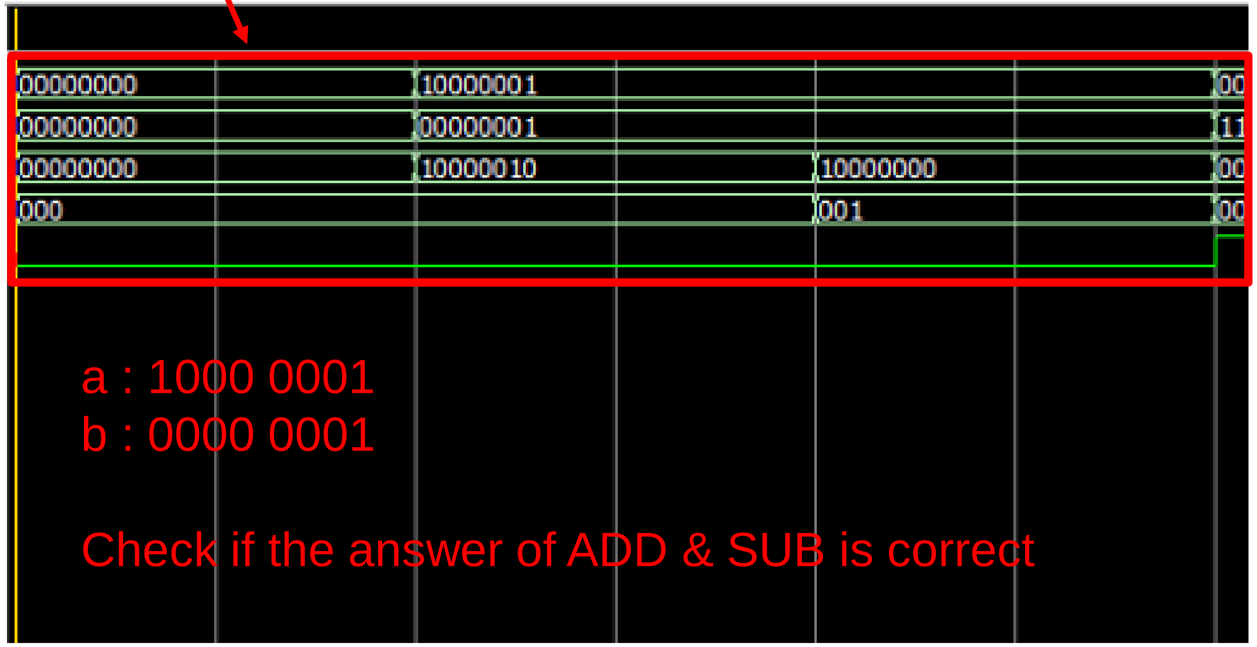


```
a         11011000   00000000  10000001            00000011  11111111            11011000
b         00011011   00000000  00000001            11111111                      00011011
result    00100100   00000000  10000010  10000000  00000010            00000100  00011000  11011011  11000011  00100100
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
```

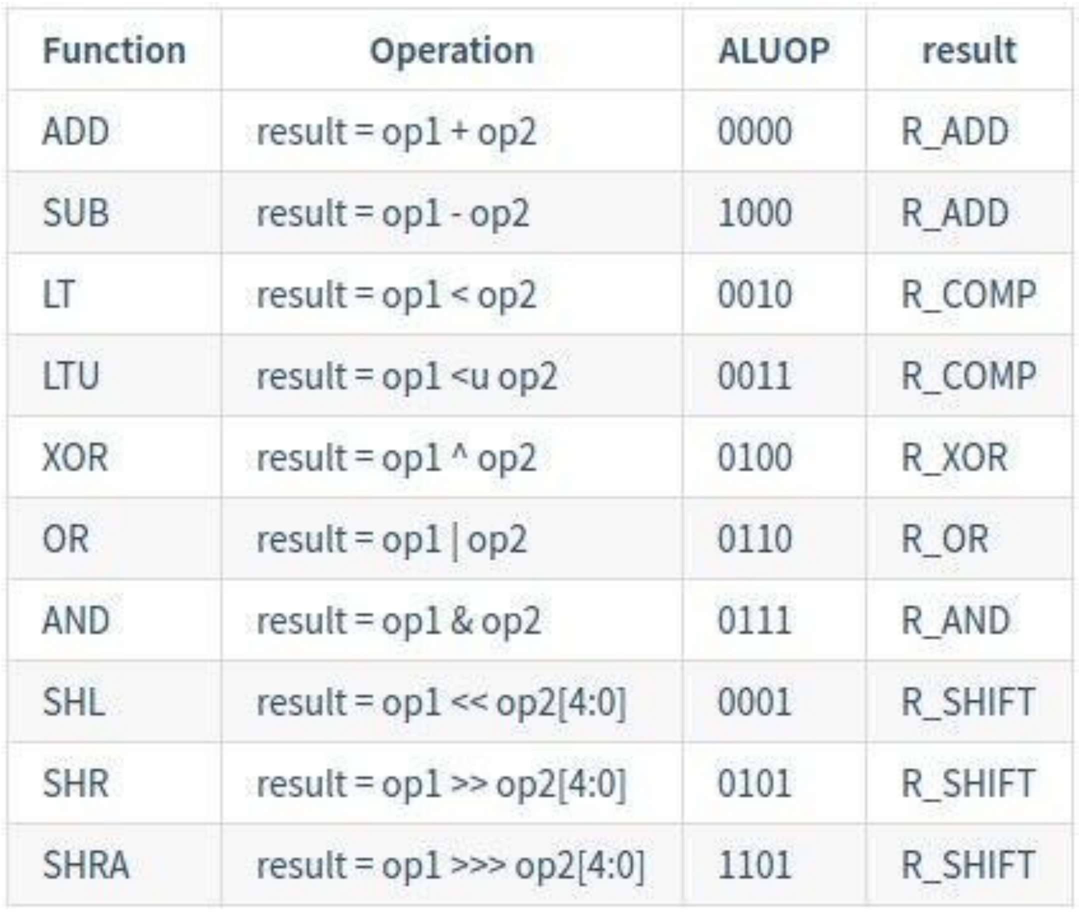
a : 1000 0001
b : 0000 0001

Check if the answer of ADD & SUB is correct

Lab 3-2 ALU Operation Implementation

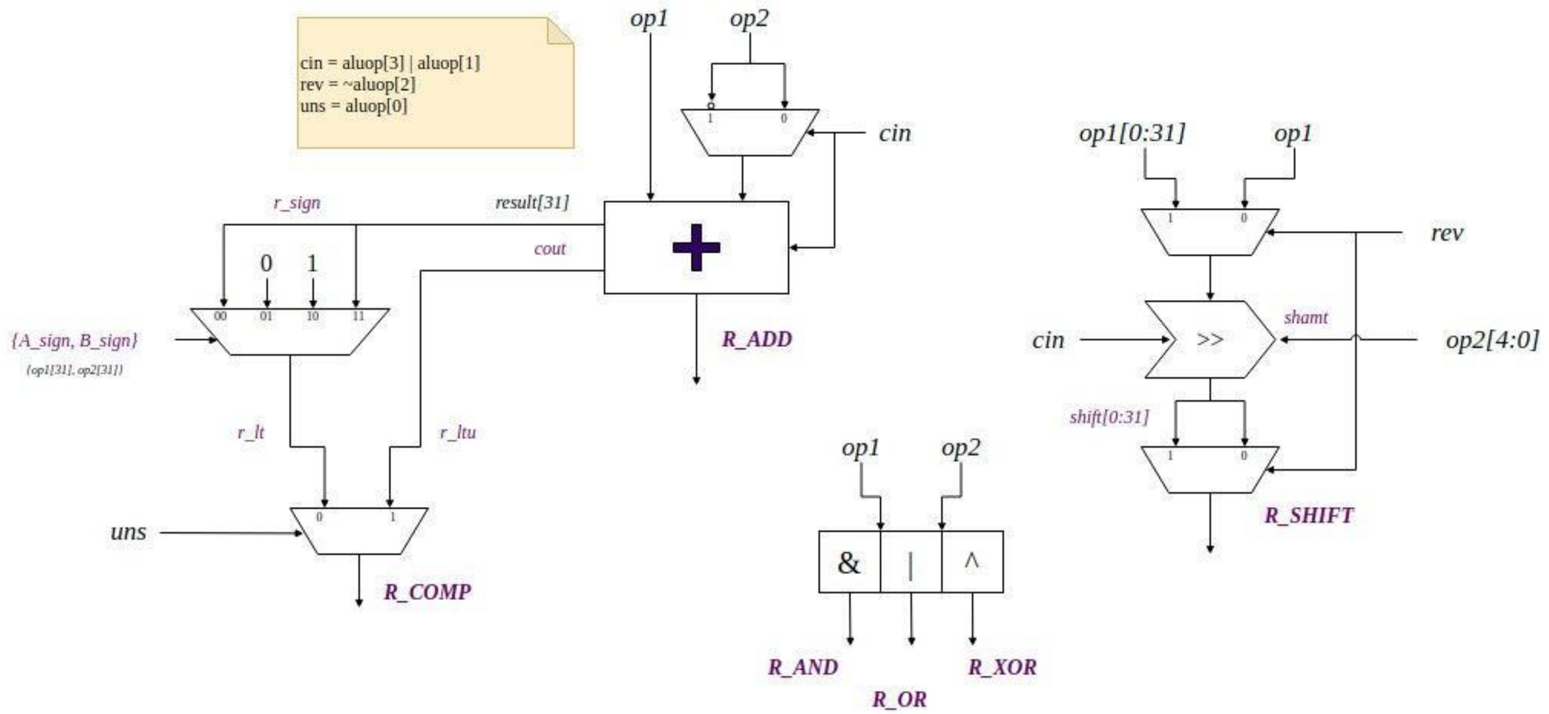
- 請同學以Verilog完成這顆 RISC-V CPU 中alu.v 的空白部分，並跑助教的程式來驗證ALU運算結果
  - 因為這顆 RISC-V CPU 經過簡化之後**並沒有實現所有 RISC-V 的指令**，詳細的運算指令和 ALUop decode 請參考 Table 1.

Table 1: ALU 運算指令



Function	Operation	ALUOP	result
ADD	result = op1 + op2	0000	R_ADD
SUB	result = op1 - op2	1000	R_ADD
LT	result = op1 < op2	0010	R_COMP
LTU	result = op1 <u op2	0011	R_COMP
XOR	result = op1 ^ op2	0100	R_XOR
OR	result = op1 | op2	0110	R_OR
AND	result = op1 & op2	0111	R_AND
SHL	result = op1 << op2[4:0]	0001	R_SHIFT
SHR	result = op1 >> op2[4:0]	0101	R_SHIFT
SHRA	result = op1 >>> op2[4:0]	1101	R_SHIFT

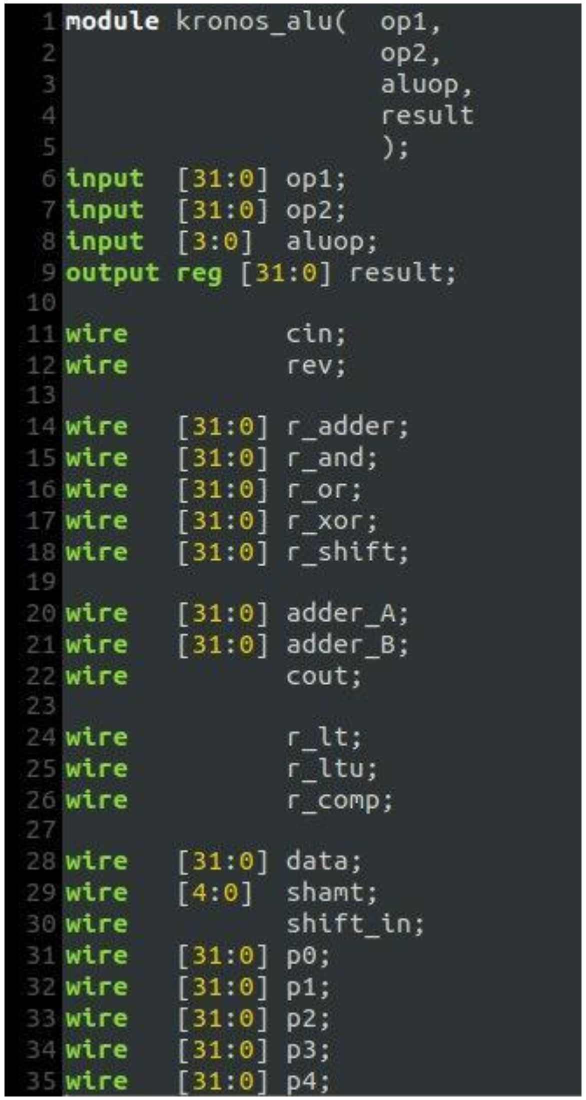
Kronos ALU Module Design

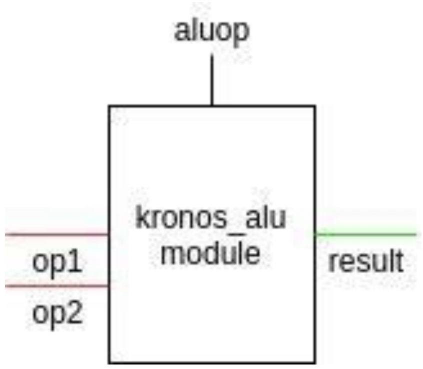


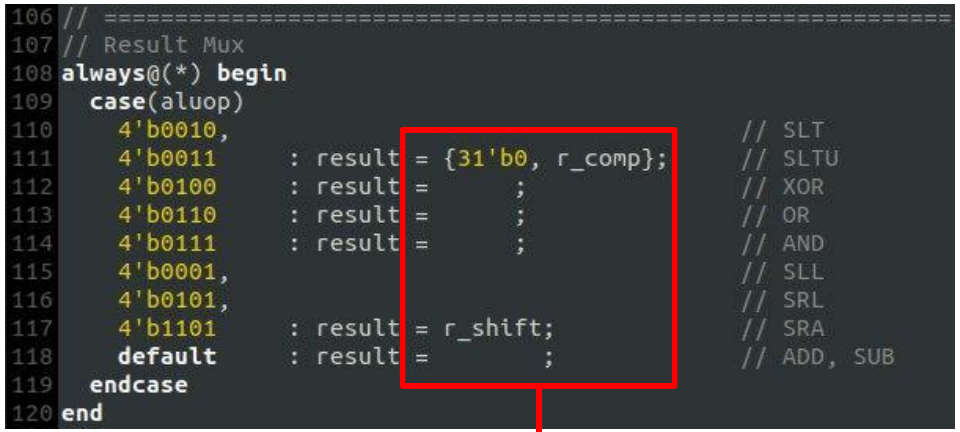
Function	Operation	ALUOP	result
ADD	result = op1 + op2	0000	R_ADD
SUB	result = op1 - op2	1000	R_ADD
LT	result = op1 < op2	0010	R_COMP
LTU	result = op1 <u op2	0011	R_COMP
XOR	result = op1 ^ op2	0100	R_XOR

OR	result = op1 | op2	0110	R_OR
AND	result = op1 & op2	0111	R_AND
SHL	result = op1 << op2[4:0]	0001	R_SHIFT
SHR	result = op1 >> op2[4:0]	0101	R_SHIFT
SHRA	result = op1 >>> op2[4:0]	1101	R_SHIFT

Lab3-2 Kronos ALU module


```verilog
module kronos_alu(  op1,
                    op2,
                    aluop,
                    result
                    );
input  [31:0] op1;
input  [31:0] op2;
input  [3:0]  aluop;
output reg [31:0] result;

wire          cin;
wire          rev;

wire   [31:0] r_adder;
wire   [31:0] r_and;
wire   [31:0] r_or;
wire   [31:0] r_xor;
wire   [31:0] r_shift;

wire   [31:0] adder_A;
wire   [31:0] adder_B;
wire          cout;

wire          r_lt;
wire          r_ltu;
wire          r_comp;

wire   [31:0] data;
wire   [4:0]  shamt;
wire          shift_in;
wire   [31:0] p0;
wire   [31:0] p1;
wire   [31:0] p2;
wire   [31:0] p3;
wire   [31:0] p4;
```

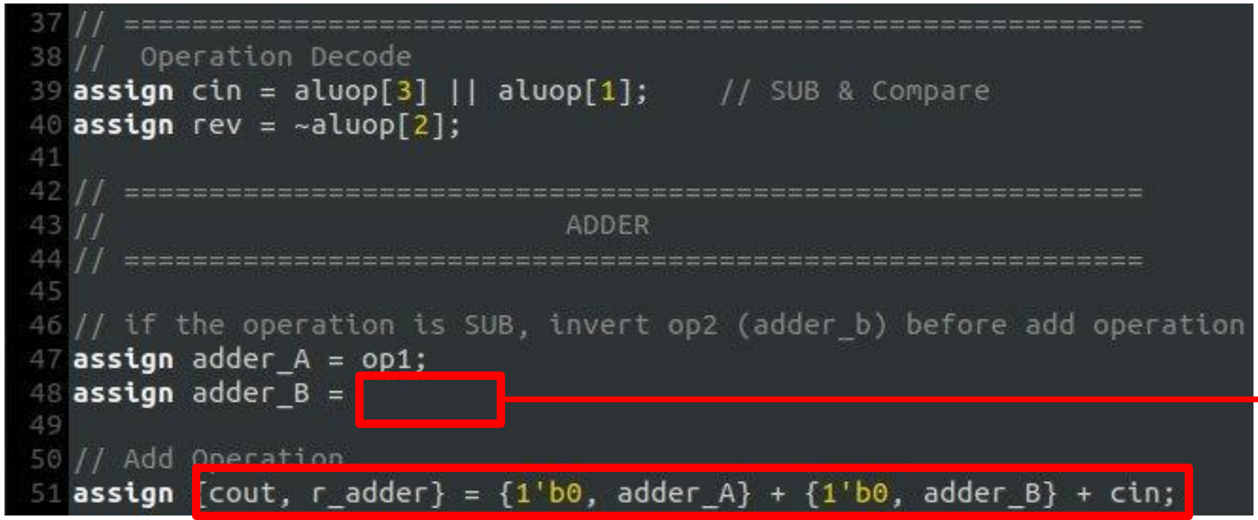



```verilog
// ================================================================
// Result Mux
always@(*) begin
  case(aluop)
    4'b0010,                                        // SLT
    4'b0011      : result = {31'b0, r_comp};        // SLTU
    4'b0100      : result =       ;                 // XOR
    4'b0110      : result =       ;                 // OR
    4'b0111      : result =       ;                 // AND
    4'b0001,                                        // SLL
    4'b0101,                                        // SRL
    4'b1101      : result = r_shift;                // SRA
    default      : result =          ;              // ADD, SUB
  endcase
end
```

A Result Mux is used to select output from ALU

Lab3-2 ALU Function

- ADD, SUB Operation



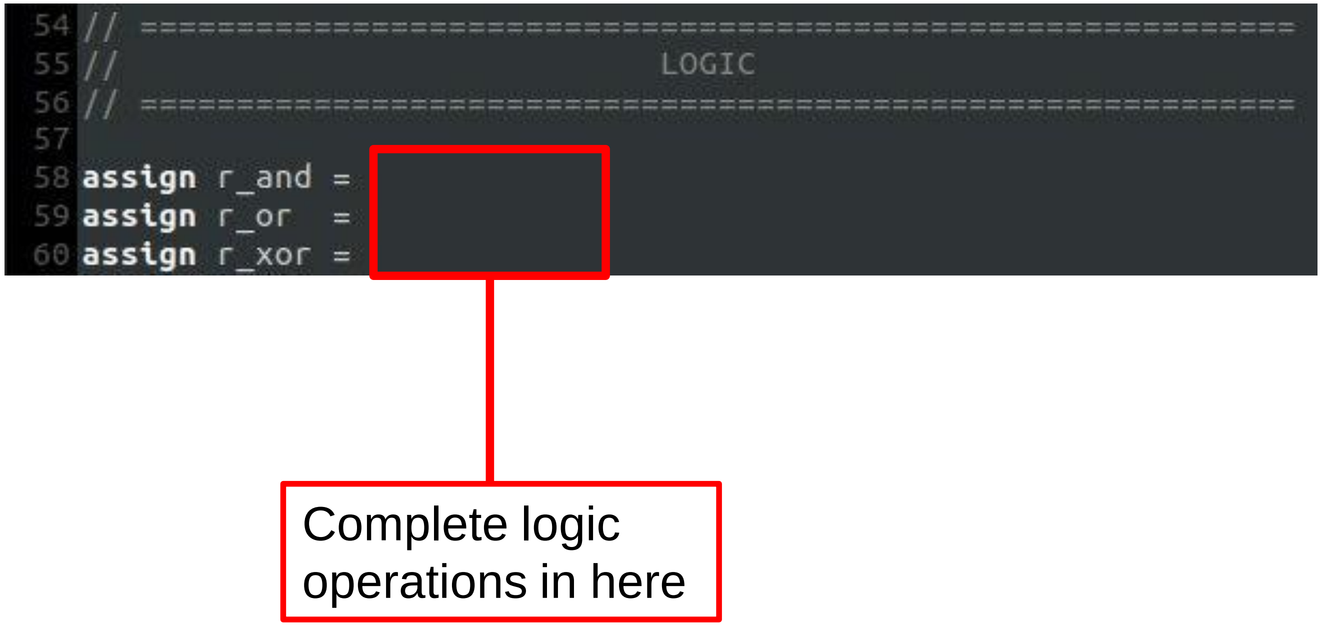
```
// ============================================================
//  Operation Decode
assign cin = aluop[3] || aluop[1];     // SUB & Compare
assign rev = ~aluop[2];

// ============================================================
//                         ADDER
// ============================================================

// if the operation is SUB, invert op2 (adder_b) before add operation
assign adder_A = op1;
assign adder_B = 

// Add Operation
assign {cout, r_adder} = {1'b0, adder_A} + {1'b0, adder_B} + cin;
```

ADD & SUB result are stored in r_adder

If the operation is SUB, invert op2 before add operation

Lab3-2 ALU Function

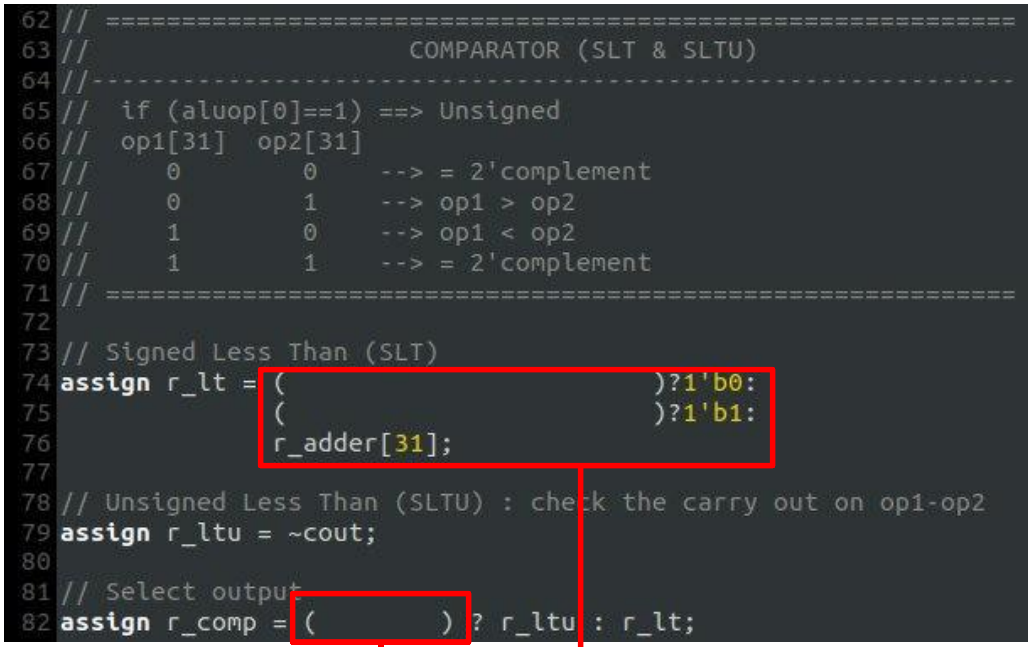
- Logic Operation



```
// ==============================================================
//                              LOGIC
// ==============================================================

assign r_and =
assign r_or  =
assign r_xor =
```

Lab3-2 ALU Function

- Comparator (SLT & SLTU)

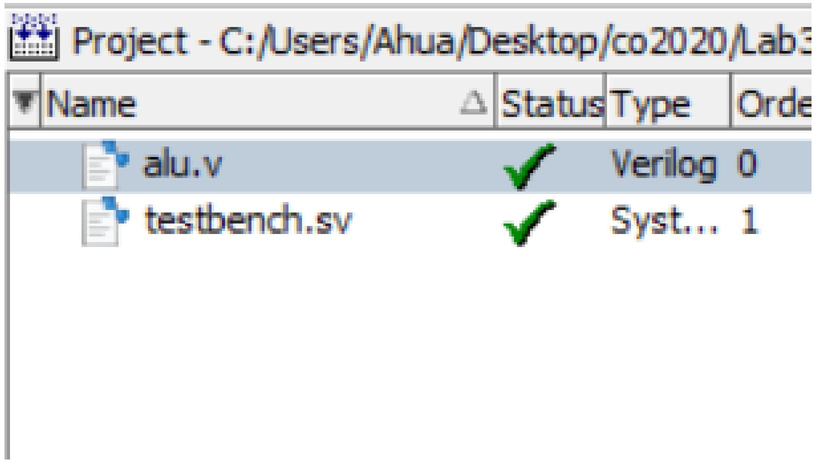


```
// ==============================================================
//                       COMPARATOR (SLT & SLTU)
//--------------------------------------------------------------
//  if (aluop[0]==1) ==> Unsigned
//  op1[31]  op2[31]
//     0        0     --> = 2'complement
//     0        1     --> op1 > op2
//     1        0     --> op1 < op2
//     1        1     --> = 2'complement
// ==============================================================

// Signed Less Than (SLT)
assign r_lt = (                         )?1'b0:
              (                         )?1'b1:
              r_adder[31];

// Unsigned Less Than (SLTU) : check the carry out on op1-op2
assign r_ltu = ~cout;

// Select output
assign r_comp = (         ) ? r_ltu : r_lt;
```

Complete Signed Less Than & Unsigned Less Than function

Lab3-2 ALU 驗證結果

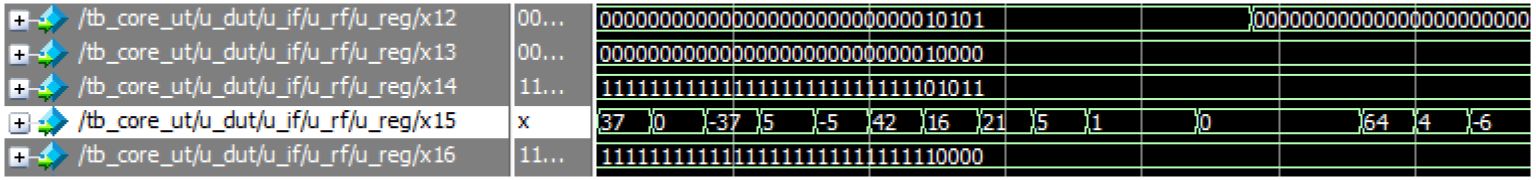
- Change directory to LAB3_2
- Create project
- Add **alu.v** & testbench.sv
- Compile
- Simulate



Project - C:/Users/Ahua/Desktop/co2020/Lab3

Name	Status	Type	Orde
alu.v	✓	Verilog	0
testbench.sv	✓	Syst...	1

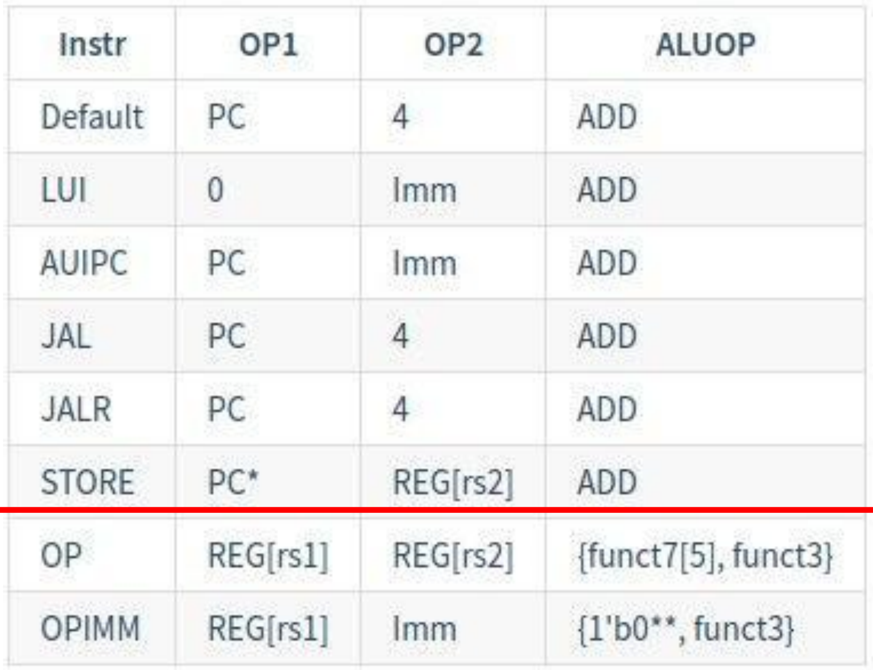
Expected waveform



```
/tb_core_ut/u_dut/u_if/u_rf/u_reg/x12  00...  00000000000000000000000000010101                    00000000000000000000000
/tb_core_ut/u_dut/u_if/u_rf/u_reg/x13  00...  00000000000000000000000000010000
/tb_core_ut/u_dut/u_if/u_rf/u_reg/x14  11...  11111111111111111111111111101011
/tb_core_ut/u_dut/u_if/u_rf/u_reg/x15  x      37  0  -37  5  -5  42  16  21  5  1  0  64  4  -6
/tb_core_ut/u_dut/u_if/u_rf/u_reg/x16  11...  11111111111111111111111111110000
```

Challenge 挑戰題

- ALU 在 EX-stage 會需要兩個 operands, op1 和 op2, 而這它們在 decoder 的預設值分別為 PC 和 constant 4，這是為計算出下一個 Instruction 的地址。ALU 預設運算則是 ADD。



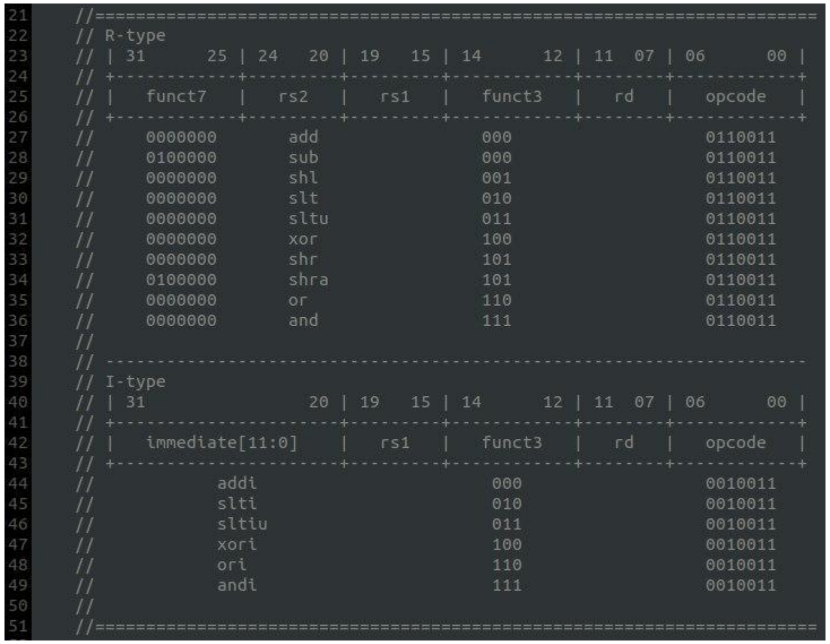
Instr	OP1	OP2	ALUOP
Default	PC	4	ADD
LUI	0	Imm	ADD
AUIPC	PC	Imm	ADD
JAL	PC	4	ADD
JALR	PC	4	ADD
STORE	PC*	REG[rs2]	ADD
OP	REG[rs1]	REG[rs2]	{funct7[5], funct3}
OPIMM	REG[rs1]	Imm	{1'b0**, funct3}

這次挑戰題主要實作部分

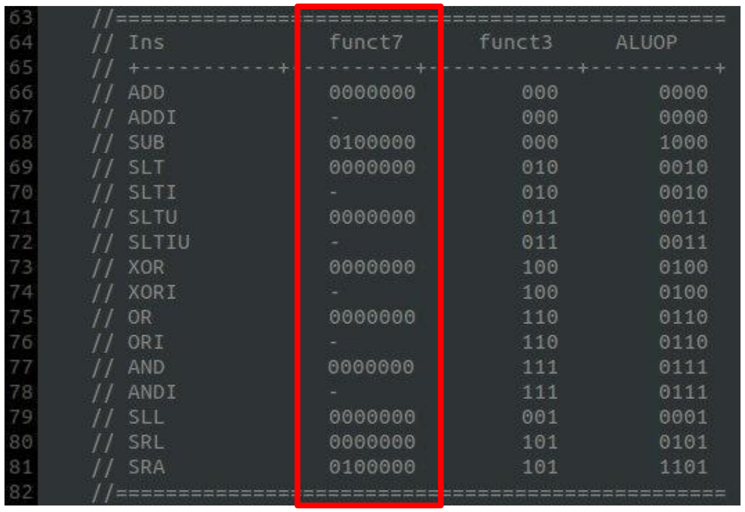
Challenge 挑戰題

- 本次挑戰題希望同學透過助教提供的部分所需的 R-type 和 I-type 指令，來完成 RISC-V 的 Decoder (部分)，並學習 RISC-V ISA 的設計。

這邊可以很清楚發現 I-type 指令需要更多位元數存取 Immediate[11:0] 值。舉例相同 opcode 的 ADD 和 ADDI 的區別可以透過 funct7 的值來做判斷



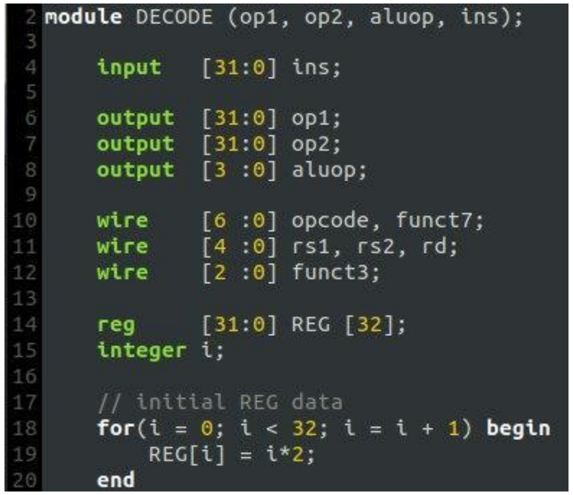
```
//==========================================================================
// R-type
// | 31        25 | 24    20 | 19    15 | 14         12 | 11   07 | 06         00 |
// +--------------+----------+----------+---------------+---------+--------------+
// |   funct7     |   rs2    |   rs1    |    funct3     |   rd    |   opcode     |
// +--------------+----------+----------+---------------+---------+--------------+
//     0000000        add                 000                       0110011
//     0100000        sub                 000                       0110011
//     0000000        shl                 001                       0110011
//     0000000        slt                 010                       0110011
//     0000000        sltu                011                       0110011
//     0000000        xor                 100                       0110011
//     0000000        shr                 101                       0110011
//     0100000        shra                101                       0110011
//     0000000        or                  110                       0110011
//     0000000        and                 111                       0110011
//
// -------------------------------------------------------------------------------
// I-type
// | 31                    20 | 19    15 | 14         12 | 11   07 | 06         00 |
// +-------------------------+----------+---------------+---------+--------------+
// |   immediate[11:0]       |   rs1    |    funct3     |   rd    |   opcode     |
// +-------------------------+----------+---------------+---------+--------------+
//            addi                         000                      0010011
//            slti                         010                      0010011
//            sltiu                        011                      0010011
//            xori                         100                      0010011
//            ori                          110                      0010011
//            andi                         111                      0010011
//
//==========================================================================
```



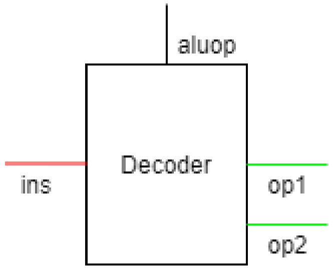
```
//================================================================
// Ins              funct7        funct3      ALUOP
// +--------------+-----------+------------+-----------+
// ADD              0000000         000         0000
// ADDI             -               000         0000
// SUB              0100000         000         1000
// SLT              0000000         010         0010
// SLTI             -               010         0010
// SLTU             0000000         011         0011
// SLTIU            -               011         0011
// XOR              0000000         100         0100
// XORI             -               100         0100
// OR               0000000         110         0110
// ORI              -               110         0110
// AND              0000000         111         0111
// ANDI             -               111         0111
// SLL              0000000         001         0001
// SRL              0000000         101         0101
// SRA              0100000         101         1101
//================================================================
```

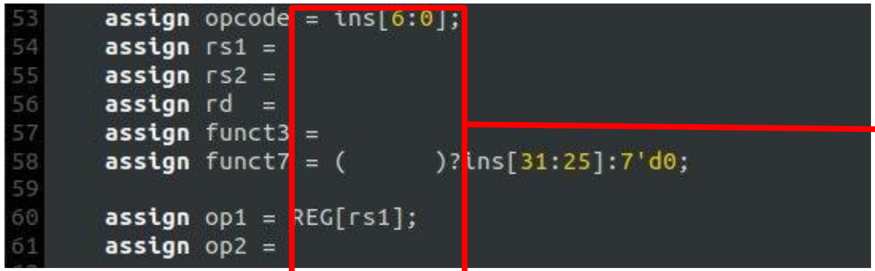
Challenge 挑戰題

- 實作 RISC-V Decoder Module

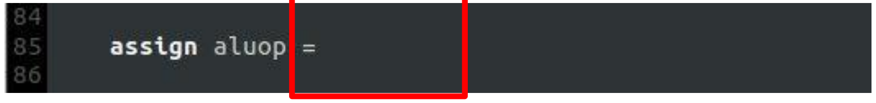


```
module DECODE (op1, op2, aluop, ins);

    input   [31:0] ins;

    output  [31:0] op1;
    output  [31:0] op2;
    output  [3 :0] aluop;

    wire    [6 :0] opcode, funct7;
    wire    [4 :0] rs1, rs2, rd;
    wire    [2 :0] funct3;

    reg     [31:0] REG [32];
    integer i;

    // initial REG data
    for(i = 0; i < 32; i = i + 1) begin
        REG[i] = i*2;
    end
```





```
    assign opcode = ins[6:0];
    assign rs1 =
    assign rs2 =
    assign rd  =
    assign funct3 =
    assign funct7 = (        )?ins[31:25]:7'd0;

    assign op1 = REG[rs1];
    assign op2 =
```



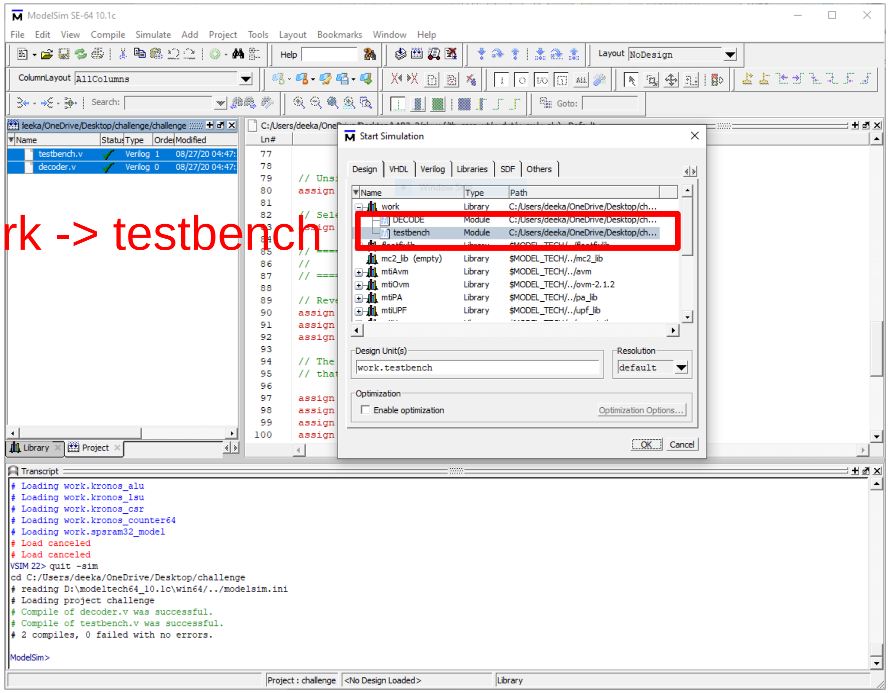
```

    assign aluop =

```

請同學透過前面助教提供的資料, 完成以下 ALU Operation Decode

Challenge 結果驗證

- Change directory to Challenge
- Create project
- Add files
- Compile
- Simulate

work -> testbench



Name	Status	Type	Order	Modified
testbench.v	✓	Verilog	1	08/27/20 04:47:
decoder.v	✓	Verilog	0	08/27/20 04:47:

Start Simulation

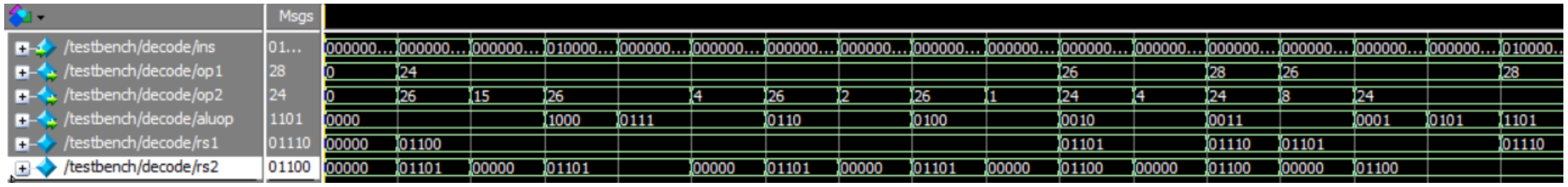
Name	Type	Path
work	Library	C:/Users/deeka/OneDrive/Desktop/ch...
DECODE	Module	C:/Users/deeka/OneDrive/Desktop/ch...
testbench	Module	C:/Users/deeka/OneDrive/Desktop/ch...
mc2_lib (empty)	Library	$MODEL_TECH/../mc2_lib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.1.2
mtiPA	Library	$MODEL_TECH/../pa_lib
mtiUPF	Library	$MODEL_TECH/../upf_lib

Design Unit(s): work.testbench
Resolution: default
Optimization: ☐ Enable optimization

```
# Loading work.kronos_alu
# Loading work.kronos_lsu
# Loading work.kronos_csr
# Loading work.kronos_counter64
# Loading work.spsram32_model
# Load canceled
# Load canceled
VSIM 22> quit -sim
cd C:/Users/deeka/OneDrive/Desktop/challenge
# reading D:\modeltech64_10.1c\win64/../modelsim.ini
# Loading project challenge
# Compile of decoder.v was successful.
# Compile of testbench.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

Project : challenge
<No Design Loaded>
Library

Expected waveform



	Msgs																	
/testbench/decode/ins	01...	000000...	000000...	000000...	010000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	000000...	010000...
/testbench/decode/op1	28	0	24									26		28	26			28
/testbench/decode/op2	24	0	26	15	26		4	26	2	26	1	24	4	24	8	24		
/testbench/decode/aluop	1101	0000			1000	0111		0110		0100		0010		0011		0001	0101	1101
/testbench/decode/rs1	01110	00000	01100									01101		01110	01101			01110
/testbench/decode/rs2	01100	00000	01101	00000	01101		00000	01101	00000	01101	00000	01100	00000	01100	00000	01100		

實驗結報

- 結報格式(每組一份)
  - 封面 (第幾組+組員)
  - 實驗內容(程式碼註解、結果截圖)
  - 實驗心得
- 繳交位置
  - ftp://140.116.164.225/ port: 21
  - 帳號/密碼 : ca_lab/Carch2020
  - Deadline: 10/19 18:00pm
- TA Contact Information:
  - 助教信箱 : anita19961013@gmail.com
  - Lab : 92617
  - Office hour : (Tuesday)8:00pm~10:00pm