# 計算機組織 Lab2

學號：111613025

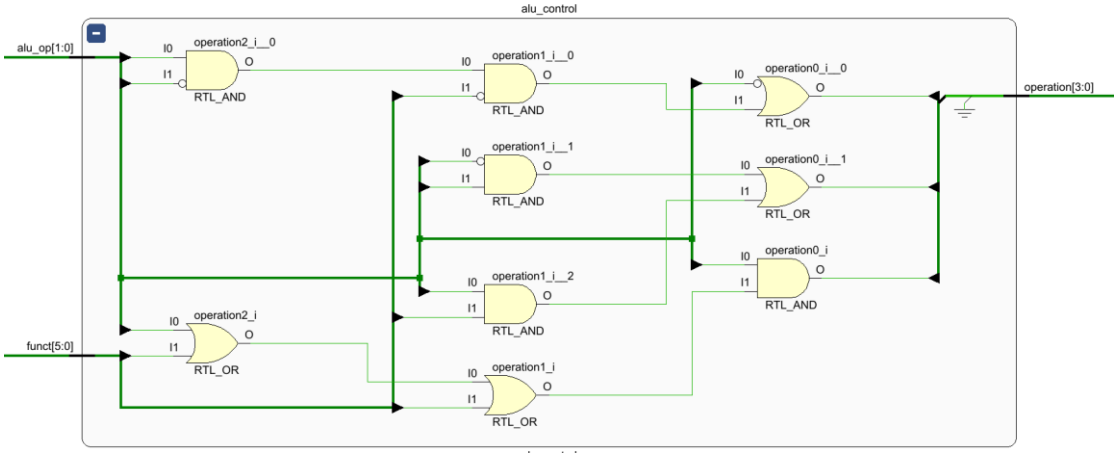
系級：材料 15

姓名：黃綵誼

# I. Architecture Diagrams

*Show your ALU control (if you have one), main control and single-cycle processor design by "Schematic" tool in Vivado, or draw them by yourself. And briefly explain them.*

1. ALU control

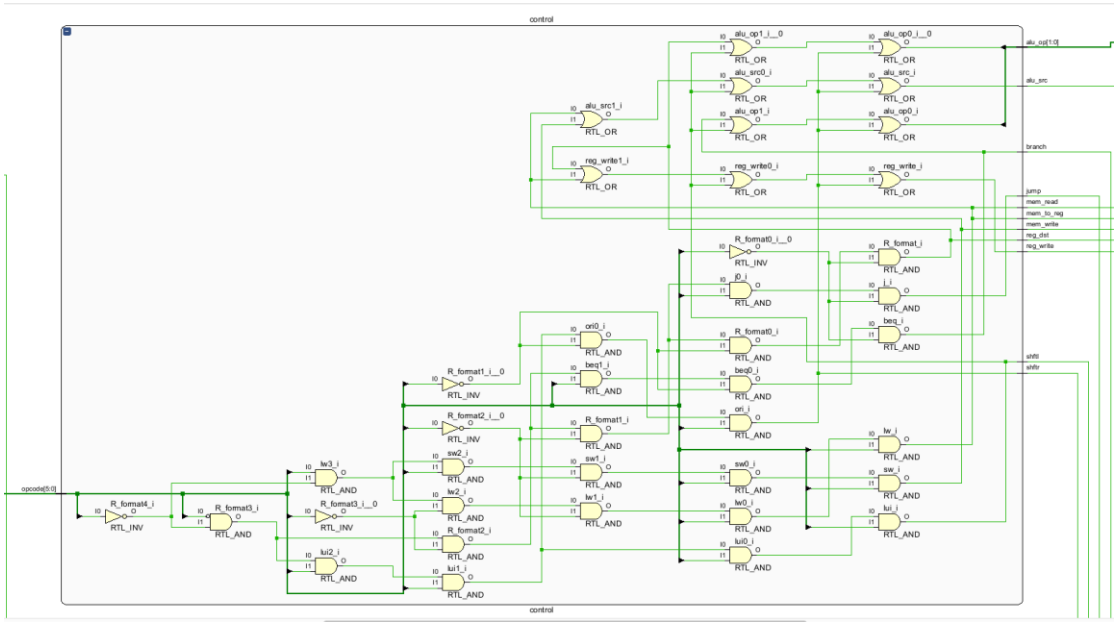



I assign each bit of operation separately according to the truth table below. For lui and ori, do “OR”(0001) operation.

| ALUOp | | Funct field | | | | | | Operation |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | $C_3\,C_2\,C_1\,C_0$ |
| 0 | 0 | X | X | X | X | X | X | 0 0 1 0 (lw/sw) |
| X (0) | 1 | X | X | X | X | X | X | 0 1 1 0 (beq) |
| 1 | X (0) | X | X | 0 | 0 | 0 | 0 | 0 0 1 0 (add) |
| 1 | X (0) | X | X | 0 | 0 | 1 | 0 | 0 1 1 0 (sub) |
| 1 | X (0) | X | X | 0 | 1 | 0 | 0 | 0 0 0 0 (AND) |
| 1 | X (0) | X | X | 0 | 1 | 0 | 1 | 0 0 0 1 (OR) |
| 1 | X (0) | X | X | 1 | 0 | 1 | 0 | 0 1 1 1 (slt) |
| lui, ori | 1 | 1 | x | x | x | x | x | x | 0 0 0 1 |

```
module alu_control (
    input  [1:0] alu_op,    // ALUOp
    input  [5:0] funct,     // Funct field
    output [3:0] operation  // Operation
);

    /* implement "combinational" logic satisfying requirements in FIGURE 4.12 */
    assign operation[3] = 0;
    assign operation[2] = (~alu_op[1] & alu_op[0]) | (alu_op[1] & funct[1]);
    assign operation[1] = ~alu_op[1] | (alu_op[1] & ~alu_op[0] & ~funct[2]);
    assign operation[0] = alu_op[1] & (alu_op[0] | funct[0] | funct[3]);

endmodule
```

## 2. main control




First, I determine whether the opcode is R-format, lw, sw, beq, j, lui, ori.

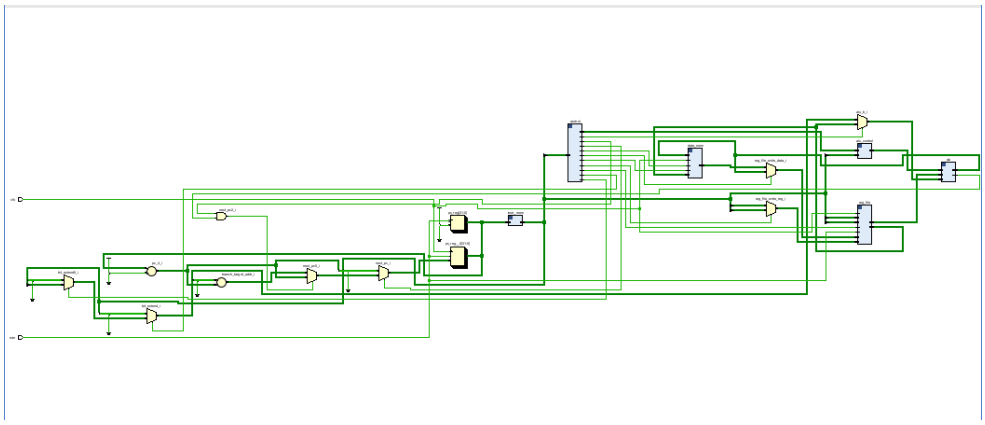
```
wire R_format;
wire lw;
wire sw;
wire beq;
wire [5:0] op;
wire j;
wire lui;
wire ori;

assign op = opcode;

assign R_format = ~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op[1] & ~op[0];
assign lw = op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
assign sw = op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0];
assign beq = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & ~op[0];
assign j = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & ~op[0];

assign lui = ~op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];
assign ori = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0];
```

Then, assign output based on the truth table below. I implement “shftr” (shift right and extend with zero) and “shftl” (shift left and extend with zero) for ori and lui respectively. Use alu_op 2’b11 for both lui and ori.

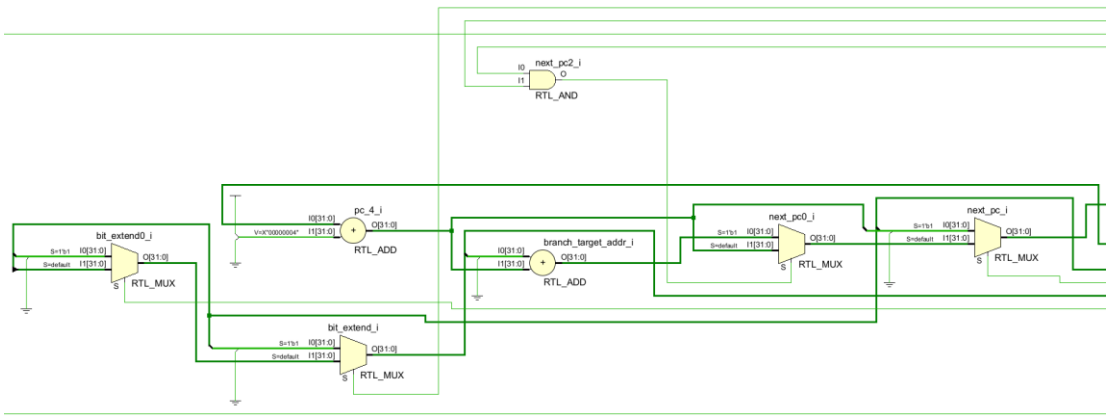
| Input or output | Signal name | R-format | lw | sw | beq | lui | ori |
|---|---|---|---|---|---|---|---|
| Inputs | Op5 | 0 | 1 | 1 | 0 | 0 | 0 |
| | Op4 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Op3 | 0 | 0 | 1 | 0 | 1 | 1 |
| | Op2 | 0 | 0 | 0 | 1 | 1 | 1 |
| | Op1 | 0 | 1 | 1 | 0 | 1 | 0 |
| | Op0 | 0 | 1 | 1 | 0 | 1 | 1 |
| Outputs | RegDst | 1 | 0 | X | X | 0 | 0 |
| | ALUSrc | 0 | 1 | 1 | 0 | 1 | 1 |
| | MemtoReg | 0 | 1 | X | X | 0 | 0 |
| | RegWrite | 1 | 1 | 0 | 0 | 1 | 1 |
| | MemRead | 0 | 1 | 0 | 0 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 | 0 | 0 |
| | Branch | 0 | 0 | 0 | 1 | 0 | 0 |
| | ALUOp1 | 1 | 0 | 0 | 0 | 1 | 1 |
| | ALUOp2 | 0 | 0 | 0 | 1 | 1 | 1 |
| | jump | 0 | 0 | 0 | 0 | 0 | 0 |
| | shftr | 0 | 0 | 0 | 0 | 0 | 1 |
| | shftl | 0 | 0 | 0 | 0 | 1 | 0 |

```
assign reg_dst = R_format;
assign alu_src = lw | sw | lui | ori;
assign mem_to_reg = lw;
assign reg_write = R_format | lw | lui | ori;
assign mem_read = lw;
assign mem_write = sw;
assign branch = beq;
assign jump = j;
assign shftl = lui;
assign shftr = ori;

assign alu_op[1] = R_format | lui | ori;
assign alu_op[0] = beq | lui | ori;
```

3. Overall



(1) bit_extend and next_pc



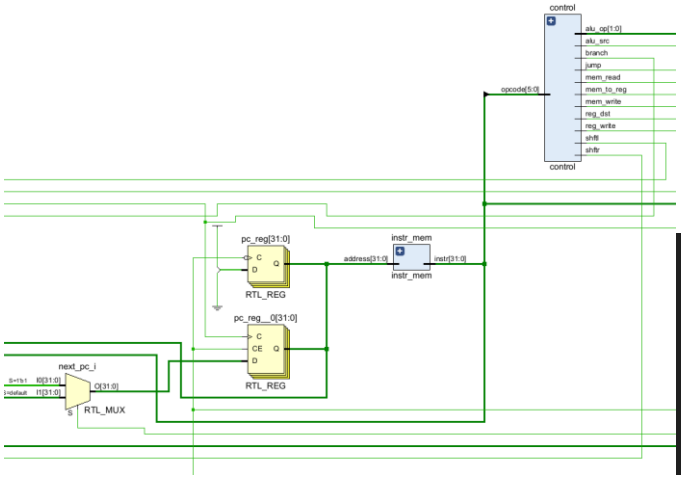

extend right 16 bit with zero when lui (shftl); extend left 16 bit with zero when ori (shftr); extend with sign bit when lw, sw, beq.

111613025 黃綵誼

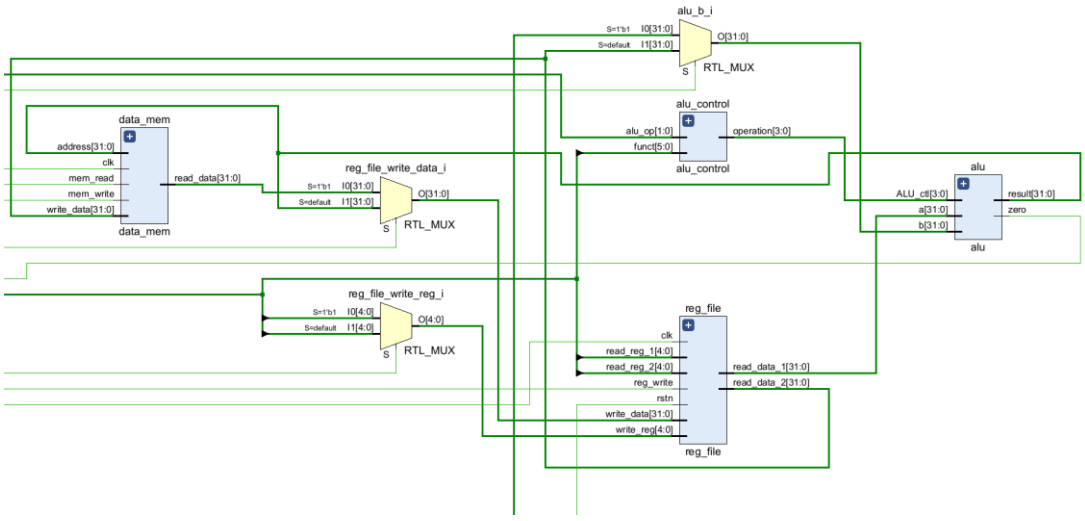
```
wire [31:0] bit_extend;
assign bit_extend = shftl ? {instr_mem_instr[15:0], 16'b0} :
                    (shftr ? {16'b0, instr_mem_instr[15:0]} :
                    {{16{instr_mem_instr[15]}}, instr_mem_instr[15:0]});
```

```
wire [31:0] branch_target_addr, jump_target_addr, next_pc;

assign jump_target_addr = {pc_4[31:28], instr_mem_instr[25:0], 2'b0};
assign branch_target_addr = {bit_extend[29:0], 2'b0} + pc_4;
assign next_pc = jump ? jump_target_addr :
                 (branch & alu_zero ? branch_target_addr : pc_4);
```

(2) other parts

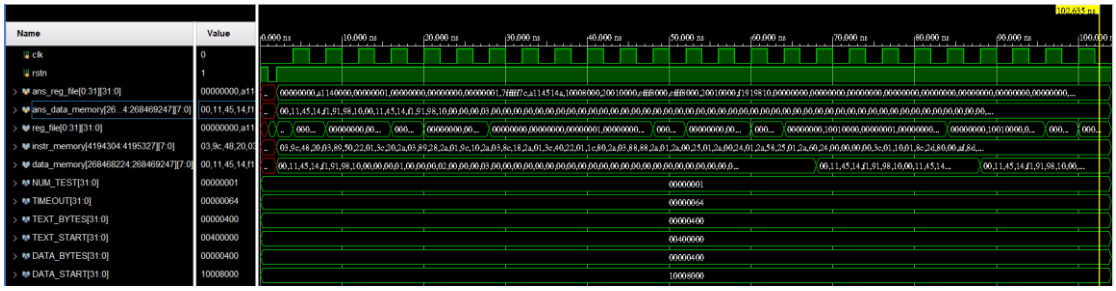



```
always @(posedge clk) begin
    if (rstn) begin
        pc <= next_pc;
    end
end

always @(negedge rstn) begin
    pc <= TEXT_START;
end
```

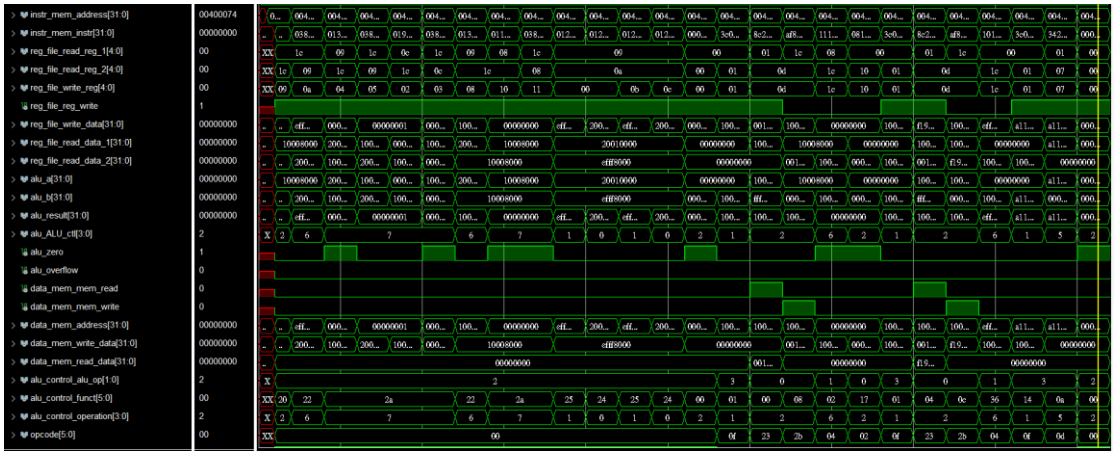


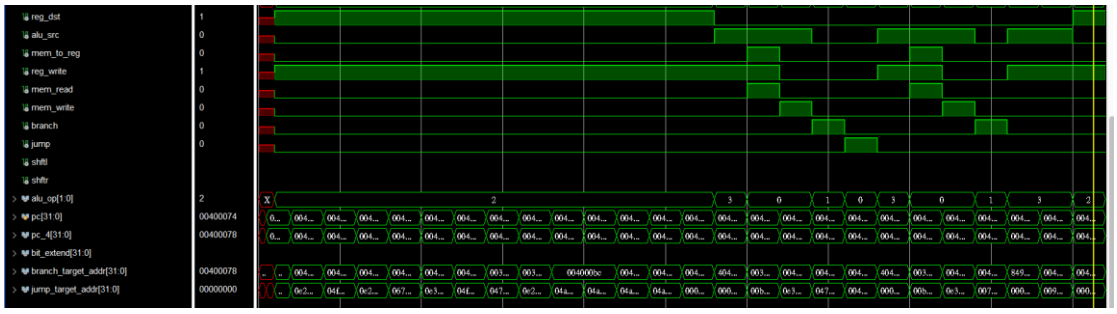
# II. Experimental Result

1. Show the waveform screen shot of the test we provided.




signal from *single_cycle.v* :




2. What other cases you've tested? Why you choose them?

I’ve tested a double loop assembly code (bubble sort algorithm) to test beq and j.

111613025 黃綵誼

```
main:
    # Load values of int_a, int_b, int_c, int_d
    lw $t3, int_a
    lw $t4, int_b
    lw $t5, int_c
    # Bubble sort algorithm
    li $a0, 1
    li $t1, 2           # Set the maximum number of comparisons
    li $t0, 0           # Outer loop counter (i)

outer_loop:
    slt $t8, $t0, $t1   # if i>=3: end
    beq $t8, $zero, end_sort
    li $t2, 0           # j = 0
    add $t0, $t0, $a0    # i++

inner_loop:
    slt $t8, $t2, $t1   # if j>=3: outer loop
    beq $t8, $zero, outer_loop  # Exit the loop if no more comparisons needed

swap_loop:
    # Compare and swap int_a and int_b
    slt $t9, $t3, $t4
    beq $t9, $zero, swap_ab

    # Compare and swap int_b and int_c
    slt $t9, $t4, $t5
    beq $t9, $zero, swap_bc
```

```
next_iteration:
    # Increment outer loop counter
    add $t2, $t2, $a0    # j++
    j inner_loop        # Continue inner loop

end_sort:
    # End of program
    sw $t3, int_a
    sw $t4, int_b
    sw $t5, int_c
    li      $a0, 0x11111111 # test li (lui, ori)
```

```
#### tb_single_cycle.sv ####
==== Test  0 RUNNING ====
number of instructions: 29 , exit @ 0x00400074
run 25 cycles
==== Test  0 PASSED ====
==== Test  1 RUNNING ====
number of instructions: 38 , exit @ 0x00400098
run 72 cycles
==== Test  1 PASSED ====
#### Test Result ####
Passed  2 : 0 1
Failed  0 :
#### all passed!
```

111613025 黃綵誼

# III. Answer the following Questions

1. When does write to register/memory happen during the clock cycle? How about read?

   Write happens at negedge of clock cycle, while read happens at posedge. This can ensure that the data input is stable.

2. Translate the "branch" pseudo instructions ( blt , bgt , ble , bge ) in the Green Card into real instructions. Only at register can be modified, and other common registers should not be modified.

   (1) blt $t1, $t2, Loop ➔ slt $at, $t1, $t2; bne $at, $zero, Loop
   t1 < t2 ➔ at != 0

   (2) bgt $t1, $t2, Loop ➔ slt $at, $t2, $t1; bne $at, $zero, Loop
   t1 > t2 ➔ t2 < t1 ➔ at != 0

   (3) ble $t1, $t2, Loop ➔ slt $at, $t2, $t1; beq $at, $zero, Loop
   t1 <= t2 ➔ t2 !< t1 ➔ at == 0

   (4) bge $t1, $t2, Loop ➔ slt $at, $t1, $t2; beq $at, $zero, Loop
   t1 >= t2 ➔ t1 !< t2 ➔ at == 0

3. Give a single beq assembly instruction that causes infinite loop. (consider that there's no delay slot)

```
Loop:
    add $t0, $t2, $t3
    beq $t0, $t0, Loop
```

   Because $t0 always equals to $t0

4. The j instruction can only jump to instructions within the "block" defined by "(PC+4)[31:28]". Design a method to allow j to jump to the next block (block number + 1) using another j.

   First, jump to the end of the current block and enter the next block, then we can jump to the start of the next block.

```
j current_block_end
current_block_end:
    j next_block_start
```

5. Why a Single-Cycle Implementation Is Not Used Today?

   Because the clock period must fit the longest delay of the instruction, which may cause much idle time, violating design principle “making the common case fast.” This is inefficient when implementing various instructions, so we will improve performance by pipelining.

## IV. Problem Encountered & Solution(optional)

*List some important problem you’ve met during this lab and the solution.*

1. The main problem I encounter is that I don’t know how to implement the lui and ori instruction.

   Solution: Ask friends for help and list out the truth table.

## V. Feedback(optional)

*Anything you want to say to TA team about this lab. How can we improve the lab?*

Thank you for all the patient and detailed response to the questions on Teams. I think It would be better if there is more detail regarding to which files should be added to simulation source and maybe more testcase.