

# CA Lab2 Report

---

## Modules Explanation

---

部分的 module 為 lab 1 沿用，因此說明沒有重寫而是僅列出修改之處。

### Adder\_32

This module simply assigns the sum of two input `a`, `b` to output `c`, which is rather trivial given that the addition of vector is built-in in Verilog.

### ALU\_Control

#### Inputs & outputs:

```
input [1:0] ALUOp;  
input [6:0] funct7;  
input [2:0] funct3;  
  
output reg [2:0] ALUctl;
```

The `ALUOp` indicates whether the output `ALUctl` depends on the instruction. More specifically, in my implementation, I set `ALUOp = 11` when the operation type is decided by both `funct3` and `funct7` (R-type), while in the I-type case I set `ALUOp = 10`, meaning that the operation is decided only by `funct3`.

The output `ALUctl` represents the operation type of current instruction, as the following:

ALUctl	Operation	ALUOp	funct3	funct7
0	and	11	111	0
1	xor	11	100	0
2	sll	11	001	0
3	add	11	000	0
3	add	10	000	N.A.
4	sub	11	000	32
5	mul	11	000	1
6	sra	10	101	N.A.

I implemented this logic by several `case` syntax. I believe there should be some technique to transform this into some logic gates, but I have no background in logic design. 🤔

## Lab 2 修改部分：

如果是 `lw` 或 `sw`，則運算必為加法；而 `beq` 時因為直接在 ID 進行判斷，所以 ALU 運算類型是 *don't care*。因此在 sequential logic 中透過對應的 ALUOp 來給出 ALUctl。

## ALU

### Inputs & outputs:

```
input [2:0] ctl;
input signed [31:0] op1, op2;
output reg [31:0] res;
```

In this module we calculate the result by executing corresponding operation of current instruction. All the operations are built-in in Verilog. Note that we have to label `op1`, `op2` as `signed` to get the correct results.

Furthermore, since we get a 32-bit `op2` even when that operand is actually a 5-bits immediate (that is, the `srai` case), we take only `op2[4:0]` into account when executing **shift right** operation.

## Control

### Inputs & outputs:

```

input [6:0] opcode;
output reg [1:0] ALUOp;
output reg ALUSrc;
output reg RegWrite;

```

The outputs will be set according to opcode :

Opcode	ALUOp	ALUSrc	RegWrite
0110011 (R-type)	11	0	1
0010011 (I-type)	10	1	1

I also implement this part with case , for readability.

## Lab 2 修改部分：

1. 新增 NoOp ，當它為 1 時就將所有 signal 設為 0 ，使得 effectively 下一輪的 EX 和下下一輪的 MEM 、下下下一輪的 WB 被 stalled 。
2. 新增 lw 、 sw 和 beq 對應的 signal 。

## MUX32

### Inputs & outputs:

```

input swt;
input [31:0] in0, in1;
output reg [31:0] res;

```

The logic in this module is very easy:  $res = in0$  when  $swt == 0$  and  $res = in1$  otherwise.

## MUX32\_2

二維版本的 MUX ，簡易的判斷 。

## Imme\_Gen

基於 opcode 來使用 instruction 不同部分生成 immediate 。如果時間夠的話還有一些優化空間 ，例如：

- IR[30:25] 永遠是 imme[10:5]
- IR[11:8] 在 sw 和 beq 時都對應到 imme[4:1]

## Sign\_Extend

Take the highest bit of input. Duplicate it by 20 times. Then combine this extension and original input into the output. Done.

## Forwarding Unit & Hazard Detection Unit

這兩個部份的邏輯基本上就是照著 lecture slide 和 spec 寫的，暫且略過。

## CPU

可以把使用 pipeline 的 CPU 想成由 IF, ID, EX, MEM, WB 、 Hazard\_Detection 及 Forwarding\_Unit 這些 module 所構成的。

對於每一個 pipeline stage，會需要從上一個 stage 紀錄一些資訊 ( ex. Rd, ALUout ) 或是訊號，並且產生出新的資訊 ( ex. ALUout )。Pipeline registers 的工作就是將這些資訊在每個 clk 更新到下一個 stage。

在每一個 stage 內部就是將各個 module 依據 interface 接好對應的訊號。

比較特別之處在於：

### 1. Stall 的機制：

Hazard\_Detection 從 EX 和 ID 判斷需要 stall 時，除了發送 NoOp 讓 Control 發出無效的訊號，也發送訊號讓 PC 及 IF/ID 在下一輪不會被改變。

### 2. Flush 的機制：

如果 ID 階段發現要 branch，代表目前 IF 的 instruction 其實並不該被執行。因此發送 signal 讓 IF/ID 進到下一輪時會把 ID 的 register 都寫為 0，這樣之後都是 nop。

## Difficulties Encountered and Solutions in This Lab

---

我認為我這次最沒寫好的地方在於 pipeline register 包成 module 似乎沒有什麼明顯的好處；反倒是應該將 IF、ID、EX 等 pipeline stage 包成 module，而其中的 registers 便是 pipeline register。這樣子可以省下一部份重複抄寫的 code ( 但還是很多 )，也可以讓主模組 CPU 比較簡潔易讀。

此外，根據特定功能會需要找不同的 stage，這時如果分為不同的模組就會比較好 debug。

## Development Environment

---

Using iverilog in **Win 11**.