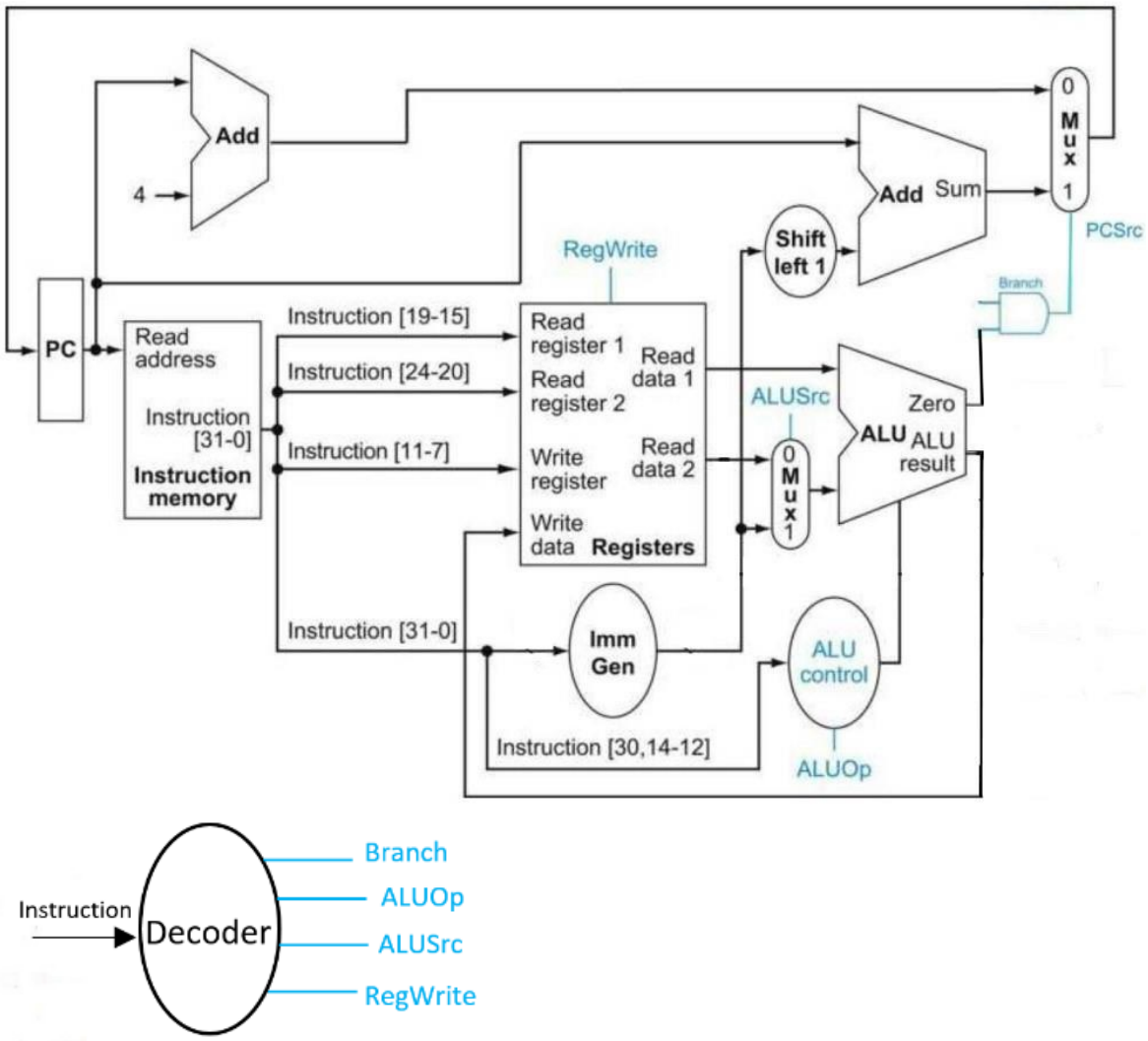
Lab 2

0712245 高魁駿
0712238 林彥彤

Architecture diagram



Implemented instruction

- R-type instrucitons (ALU_OP==10)
  - add
  - sub
  - and
  - or
  - xor
  - slt (Set on Less Than)
  - sll
  - sra
  - srl

- I-tpye instructions (ALU_OP==11)
  - addi (Add Immeidate)
  - slti (Set on Less Than Immeidate)
  - ori (Or Immeidate)
  - andi
  - srai
  - slli
  - srli

- SB-tpye instructions (ALU_OP==01)
  - beq (Branch on Equal)
  - bne (Branch on not Equal)

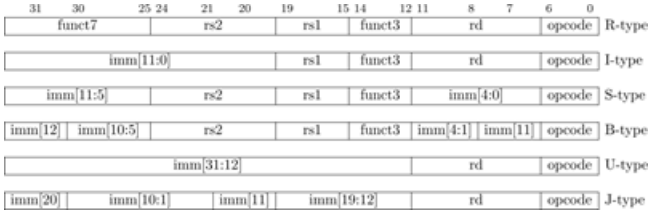
指令	ALU_OP	ALU_ctrl	(instr[30], instr[14:12])
add	10	0010	0000
sub	10	0110	1000
and	10	0000	0111
or	10	0001	0110
xor	10	1101	0100
slt	10	0111	0010
srl	10	0100	1101
sll	10	0011	0001
sra	10	0101	1101
addi	11	0010	0000
slti	11	0111	0010
ori	11	0001	0110
srli	11	0100	1101
andi	11	0000	0111
srai	11	0101	1101
slli	11	0011	0001
beq	01	1010	000
bne	01	1011	001

Detailed description of the implementation

1. Adder1
這個加法器主要是在做PC+4的作用，背後的理念是基於make common case fast, 因為程式的執行就是一行行的從記憶體讀取指令，大部分時間PC都只是穩定的往下個記憶體位址移動讀取，而且這樣讓PC提早產出的方法有助pipeline的使用(讓下一個stage可以使用PC)。

2. Decoder
Decoder的目的就是看32bit 指令前七碼的值，並且依據這些值來建立每種指令專有的datapath。另外，指令編碼也要考慮decoder的存在，讓每種7 bit operation code都只對應到一種datapath.

3.Imm_Gen
ImmGen會根據不同type的opcode來決定如何組出Immediate value。根據查表opcode在Instruction的0-6bit，因此會先取出opcode。之後依據下圖來組成type的Immediate value，但是在運算時都是使用32bit做運算，實際上就是把最前面一個bit的MSB直接拿來填滿32bit中。



4.Alu_contrl
ALU_control 需要知道ALU時機要做的運算，所以要解析指令是屬於何種型別，在解析型別時會先需要看指令的OP field 是哪種。

5.Shifter
這個模組會將32bit value直接左移一格，有可能會想說32bit signed 跟 unsigned 對資料直接左移兩格會不會得到對應的結果(ex. -2 shift left 1 會不會得到 -4)，後來實測是會的，所以這個模組對有號無號整數具有通用性。

6.Adder2
這個加法器是拿來運算出跳轉時對應的位址（如果有要跳轉的話），數值來源來自於已經加4的PC 值跟指令中32bit sign extended 的 immediate值。

7.Mux_ALUSrc
由於指令有兩種型態，ALU的資料來源可能會是暫存器的值或是指令中的16 bit immediate value。

8.ALU
這個模組，執行所有的運算，就跟指令要求的一樣直觀，只是在這裡有令指令並沒有實踐，像是BEQ，ALU實際得到要做運算是減法指令，最後用ALU 的ZERO輸出確認是否為零就能知道兩數是否相等

9.Mux_PC_Source
因為PC在這個指令架構中會有兩種移動方式，第一種是序列讀取，第二種是直接跳轉，所以需要一個多工器來讓PC接受兩種不同結果帶來的值。第一種移動方式來自於 Adder1(第一項)，而第二種移動方式來自於Adder2(第七項)。
唯一一點要注意的是選擇訊號來自於decoder 給的branch訊號 跟 ALU 的zero訊號進行and運算，因為就算預設的datapath可能會因為branch系列指令而跳轉，我們還是需要透過ALU運算是否有達到跳轉條件(ex. BEQ 只在 兩數相等時跳轉)。

Implementation results

- Test case 1

```
# Loading work.ALU_Ctrl
# Loading work.alu
VSIM 26> run -all
# r0 =   0, r1 =  21, r2 =   9, r3 =   1,
# r4 =  20, r5 =   1, r6 =   0, r7 =   0,
# r8 =   0, r9 =   0, r10 =  0, r11 =  0
# ** Note: $stop    : C:/Modeltech_pe_edu_10.4a/lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Modeltech_pe_edu_10.4a/lab3/testbench.v line 32
```

- Test case 2

```
# r0 =   0, r1 =   0, r2 =   0, r3 =   0,
# r4 =   0, r5 =   0, r6 =   2, r7 =   5,
# r8 =   7, r9 =   9, r10 =  0, r11 =  0
# ** Note: $stop    : C:/Modeltech_pe_edu_10.4a/lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Modeltech_pe_edu_10.4a/lab3/testbench.v line 32
```

- Test case 3

```
# r0 =   0, r1 =   0, r2 =   0, r3 =   0,
# r4 =   0, r5 =   0, r6 =   0, r7 =   0,
# r8 =   0, r9 =   0, r10 =  2, r11 =  2
# ** Note: $stop    : C:/Modeltech_pe_edu_10.4a/lab3/testbench.v(32)
#    Time: 1010 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at C:/Modeltech_pe_edu_10.4a/lab3/testbench.v line 32
```

Problems encountered and solutions

- Problem: Failed when implementing sra (result_o = $signed(src2_i) >>> src1_i)
- Solution: It turned out to be that not only src2_i should be signed but also the result_o itself, so we declared result_o as signed then fixed.

Lesson learnt (if any):

- Understand how to distribute control code to functions through decoder and ALUcontrol.
- Understand how a simple single cpu works