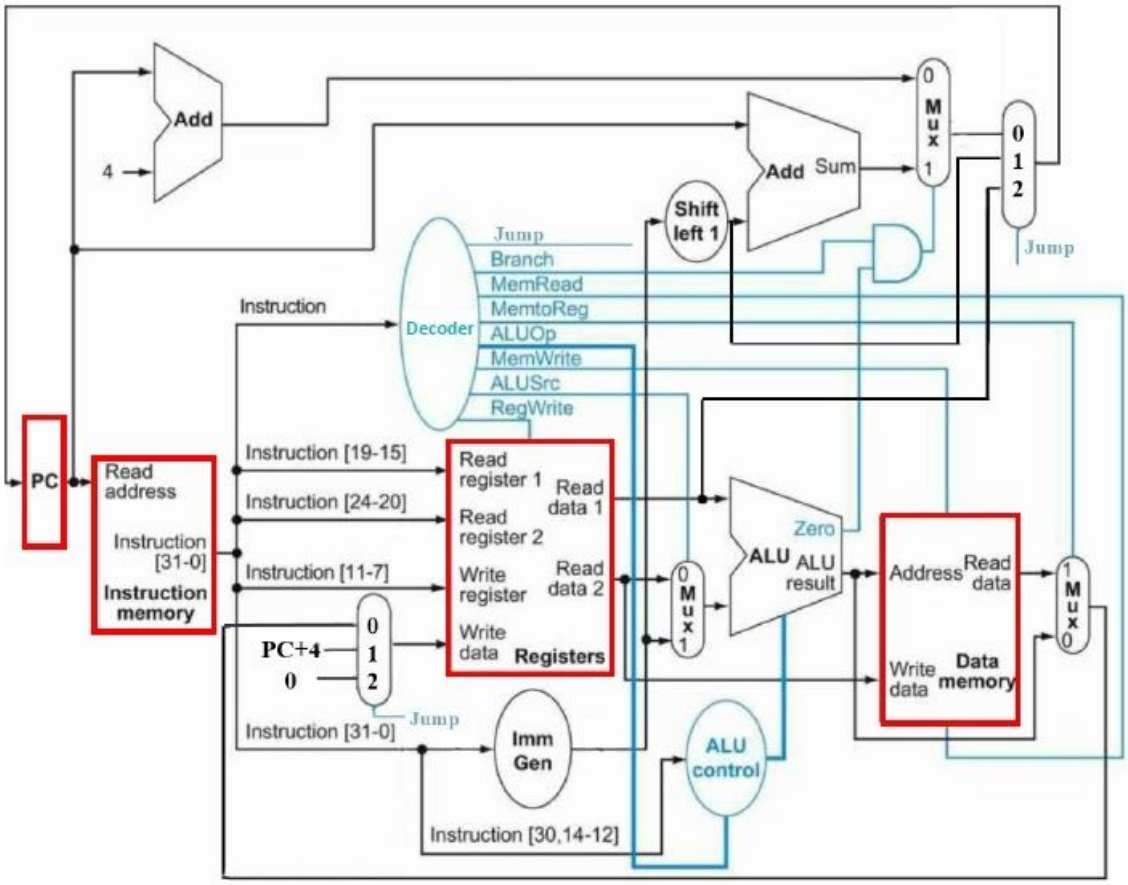
Computer Organization

Architecture diagram:



Detailed description of the implementation:

這次實驗基本上是延續LAB3而來的，因此除了助教提供的Data Memory和全新的3to1_Multiplexer之外，都是在LAB3的code中加上lw、sw、blt、bge、jal、jalr等新指令，最後在Simple_Single_CPU中把對應的i/o接起來：

1. Decoder：

我們根據instr_i[6:0]的opcode將instruction先分成5類指令格式，R-type(0)、I-type(1)、S-type(2)、B-type(3)和UJ-type(4)，接著再依據Instrucion_field、opcode及func3來設定各指令的Control Signals。

addi和sub的Func3&ALUop完全相同，如果addi的常數是負數時，會造成ALU_Ctrl判斷時發生錯誤，把ALU設成減法，所以我們將addi另外設成ALUop=2'b00，和lw/sw一樣在ALU裡面都進行加法，以避免ALU_Ctrl判斷錯誤。

jal和jalr的ALUop是Don't care，但還是把他們的ALUop設為00，讓我們能方便Debug。

```
// 0:R-type, 1:I-type, 2:S-type, 3:B-type, 4:UJ-type, 5:error
assign Instr_field = (opcode==7'b1101111) ? 4:      //UJ-type(jal)
                     (opcode==7'b1100011) ? 3:      //B-type
                     (opcode==7'b0100011) ? 2:      //S-Type
                     (opcode==7'b0000011) ? 1:      //I-type(LW)
                     (opcode==7'b0010011) ? 1:      //I-type(arithmetic)
                     (opcode==7'b1100111) ? 1:      //I-type(jalr)
                     (opcode==7'b0110011) ? 0:      //R-type
                                            5;      //error

assign Ctrl_o = (Instr_field==0)                        ?10'b0000100010:( //R-type              ALUop=10
                (Instr_field==1 && opcode==7'b0000011) ?10'b0011110000:( //I-type(LW)          ALUop=00
                (Instr_field==1 && opcode==7'b1100111) ?10'b1000100000:( //I-type(jalr)        -
                (Instr_field==1 && opcode==7'b0010011
                                && funct3==3'b000)      ?10'b0010100000:( //I-type(addi)        ALUop=00
                (Instr_field==1)?10'b0010100010:(                         //I-type(arithmetic)  ALUop=10
                (Instr_field==2)?10'b0010001000:(                         //S-type(SW)          ALUop=00
                (Instr_field==3)?10'b0000000101:(                         //B-type              ALUop=01
                (Instr_field==4)?10'b0100100000:(                         //UJ-type(jal)        -
                                                 0)))))))));

assign Jump     = {Ctrl_o[9:8]};
assign ALUSrc   = Ctrl_o[7];
assign MemtoReg = Ctrl_o[6];
assign RegWrite = Ctrl_o[5];
assign MemRead  = Ctrl_o[4];
assign MemWrite = Ctrl_o[3];
assign Branch   = Ctrl_o[2];
assign ALUOp    = {Ctrl_o[1:0]};
```

2. ALU_Ctrl：

所有指令可以利用input的ALUop訊號分成3種：
load/store(00)、Branch(01)、arithmetic(10)

1. **load/store**：I-type(load) & S-type & addi

因為load和store都在ALU裡面進行加法，所以input為ALUop==2'b00時，output ALU_Ctrl==4'b0010。

2. **Branch**：beq 、 bne、blt、bge

Branch的ALUop都是01，但我們alu裡面的運算是利用behavirol implementation來定義這四個指令，所以要藉由Function3 (instr[2:0])區分這四個指令，再給予他們不同的ALU_Ctrl。
beq：instr[2:0] == 3'b000時，ALU_Ctrl = 4'b0110
bne：instr[2:0] == 3'b001時，ALU_Ctrl = 4'b0011
blt ：instr[2:0] == 3'b100時，ALU_Ctrl = 4'b0100
bge：instr[2:0] == 3'b101時，ALU_Ctrl = 4'b0101

3. **算術運算**：R-tpye & I-type

在ALU裡進行同一種運算，會有R-type和I-type兩種指令，例如add和addi都是在ALU裡面進行加法；xor和xori都是在ALU裡面進行xor。
我們進一步發現：add和addi的Funct3完全一樣，也就是在ALU進行同一種運算的R-type和I-type指令，Funct3會一樣，所以將這類型(addi, add)的指令歸類在算術運算，利用Funct3給予不同的ALU_Ctrl就好。
需要注意的地方是加法和減法的Funct3都是000，所以需要進一步先判斷是否為減法 (instr[3:0]=4'b1000)。

補充：instr[3] = instruction[30]；instr[2:0] = instruction[14:12] (=Funct3)

```
module ALU_Ctrl(
    input       [3:0]   instr,
    input       [1:0]   ALUOp,
    output wire [3:0] ALU_Ctrl_o
    );

/* Write your code HERE */
assign ALU_Ctrl_o = (ALUOp==2'b00)                          ? 4'b0010 : //load/store -> ALU = add
                    ((ALUOp==2'b01) && (instr[2:0]==3'b000)) ? 4'b0110 : //beq
                    ((ALUOp==2'b01) && (instr[2:0]==3'b001)) ? 4'b0011 : //bne
                    ((ALUOp==2'b01) && (instr[2:0]==3'b100)) ? 4'b0100 : //blt
                    ((ALUOp==2'b01) && (instr[2:0]==3'b101)) ? 4'b0101 : //bge
                    ((ALUOp==2'b10) && (instr==4'b1000))     ? 4'b0110 : //ALU = sub
                    ((ALUOp==2'b10) && (instr[2:0]==3'b000)) ? 4'b0010 : //ALU = add
                    ((ALUOp==2'b10) && (instr[2:0]==3'b111)) ? 4'b0000 : //ALU = and
                    ((ALUOp==2'b10) && (instr[2:0]==3'b110)) ? 4'b0001 : //ALU = or
                    ((ALUOp==2'b10) && (instr[2:0]==3'b010)) ? 4'b0111 : //ALU = SLT (set less than)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b001)) ? 4'b1111 : //ALU = sll (shift left logical)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b100)) ? 4'b1110 : //ALU = xor (exclusive or)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b101)) ? 4'b1001 : //ALU = sra (shift right arithmetic)
                                                               4'b0011;  //ALU = Error detect


endmodule
```

3. alu：

依據input的ALU_control (4-bit)訊號進行運算：

一般load/store、I-type、R-type都會有正常的result及zero。

少數需要判斷條件的B-type、slt，需要依據是否要jump來控制Zero：

當需要跳躍時，result==1，且zero==1；

不需跳躍時，result==0，且zero==0。

4. Imm_Gen：

根據opcode把指令分類成4類，並利用Type這個自己設定的internal signal來區分類別，完成Immediate Generate。

Type = 2’b00：B-type (beq, bne, blt, bge)

Type = 2’b01：I-type (jalr, lw, arithmetic)

Type = 2’b10：UJ-type (jal)

Type = 2’b11：S-type (sw)

Implementation results:

Test_data_1

```
# PC =        76
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          1,          2,          3
# Registers
# R0 =          0, R1 =         16, R2 =        128, R3 =          0, R4 =          0, R5 =          4, R6 =          5, R7 =          6
# R8 =          1, R9 =          2, R10 =          3, R11 =          0, R12 =          0, R13 =          0, R14 =          0, R15 =          0
# R16 =          0, R17 =          0, R18 =          0, R19 =          0, R20 =          0, R21 =          0, R22 =          0, R23 =          0
# R24 =          0, R25 =          0, R26 =          0, R27 =          0, R28 =          0, R29 =          0, R30 =          0, R31 =          0
#
```

Test_data_2

```
# PC =        44
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,         28
# Registers
# R0 =          0, R1 =         16, R2 =        128, R3 =          0, R4 =          0, R5 =          1, R6 =          0, R7 =         28
# R8 =          0, R9 =          0, R10 =          0, R11 =          0, R12 =          0, R13 =          0, R14 =          0, R15 =          0
# R16 =          0, R17 =          0, R18 =          0, R19 =          0, R20 =          0, R21 =          0, R22 =          0, R23 =          0
# R24 =          0, R25 =          0, R26 =          0, R27 =          0, R28 =          0, R29 =          0, R30 =          0, R31 =          0
```

Problems encountered and solutions:

```
/* Write your code HERE */
assign ALU_Ctrl_o = (ALUOp==2'b00)                                  ? 4'b0010 : //load/store/addi -> ALU = add
                    ((ALUOp==2'b01) && (instr[2:0]==3'b000)) ? 4'b0110 : //beq
                    ((ALUOp==2'b01) && (instr[2:0]==3'b001)) ? 4'b0011 : //bne
                    ((ALUOp==2'b01) && (instr[2:0]==3'b100)) ? 4'b0100 : //blt
                    ((ALUOp==2'b01) && (instr[2:0]==3'b101)) ? 4'b0101 : //bge
                    ((ALUOp==2'b10) && (instr[3:0]==4'b1000))? 4'b0110 : //ALU = sub
                    ((ALUOp==2'b10) && (instr[2:0]==3'b000)) ? 4'b0010 : //ALU = add
                    ((ALUOp==2'b10) && (instr[2:0]==3'b111)) ? 4'b0000 : //ALU = and
                    ((ALUOp==2'b10) && (instr[2:0]==3'b110)) ? 4'b0001 : //ALU = or
                    ((ALUOp==2'b10) && (instr[2:0]==3'b010)) ? 4'b0111 : //ALU = SLT (set less than)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b001)) ? 4'b1111 : //ALU = sll (shift left logical)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b100)) ? 4'b1110 : //ALU = xor (exclusive or)
                    ((ALUOp==2'b10) && (instr[2:0]==3'b101)) ? 4'b1001 : //ALU = sra (shift right arithmetic)
                                                                 4'b0011;  //ALU = Error detect
```

Q1. 當addi的常數為負的時候，由於addi跟sub的ALUop和Funct3完全相同，且instr[30]都會是1，在ALU_Ctrl進行判斷時，addi和sub的ALU_Ctrl都是instr[3:0]=4’b1000，會將addi判斷成sub，也就是ALU在執行「add：負數」的時候會變成「sub：負數」。

A1. 把addi在decoder中另外提出，設其ALUop=2’b00，讓addi和lw一樣在ALU裡面都進行加法運算。

Q2. 當遇到jal的指令時，jal的PC會固定跳到2048，我們重複檢查了Imm_Gen的多次，確認了立即值的輸出並沒有錯。

A2. 後來才發現設定內部訊號Type時，沒有把Type宣告成2-bit，所以jal的Type=2'b10會永遠出錯，這部分除錯花了我們最久時間。

Comment:

這次Lab4跟Lab3時間有些緊湊，雖然是因為兩次lab的重複性高，不過有些人會lab3拖太久會導致lab4也一起被拖累，我覺得助教可以lab3跟lab4同時公告，讓同學有心理準備。

這次我們有利用星期一的時間去找TA問問題，很感謝TA的親切回答，並露了一手Debug的絕活給我們看，讓我們在能在之後的Lab花更少的時間Debug。