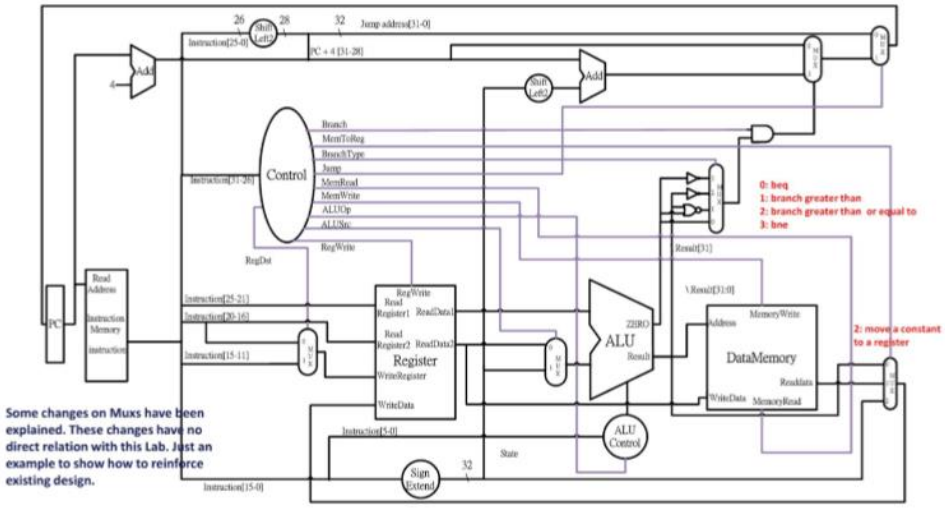
Computer Organization

Architecture diagrams:



Hardware module analysis:

Program Counter：

目前指令的位置。

Decoder：

判斷目前所要執行的 operation 為何，輸出 RegWrite_o/ ALU_op_o/ ALUSrc_o/ RegDst_o/ Branch_o/ MemToReg_o/ Jump_o/ MemRead_o/ MemWrite_o 等控制訊號。

Instruction Memory：

將目前要執行的指令存到 Instruction Memory。

Register File：

依照 R-type/I-type 來讀取 1-2 個 register 檔案內容並輸出做運算。

ALU：

依據收到 ALU Control 的訊號，將輸入的值作相對應的 operation。

(這次相較上次增加了 lw / sw / jal)

Adder：

Adder1

將 pc 的指向下一個指令(pc+4)

Adder2

依據分支給予的偏移量加上原本 pc 的值來更新 pc 的值。

ALU Control：

控制 ALU 所要做的 operation。

Shift Left 2：

因為 instruction 的單位為 word(32 bit)，所以要向左移 2 bit 將單位轉為以 byte(8 bit)來計算

Sign Extend：

像是 immediate 等的值在 Iinstruction 中只存放 16 bit，因此需要透過 sign extend 為 32 bit 才能當作輸入。

MUX

MUX_2to1

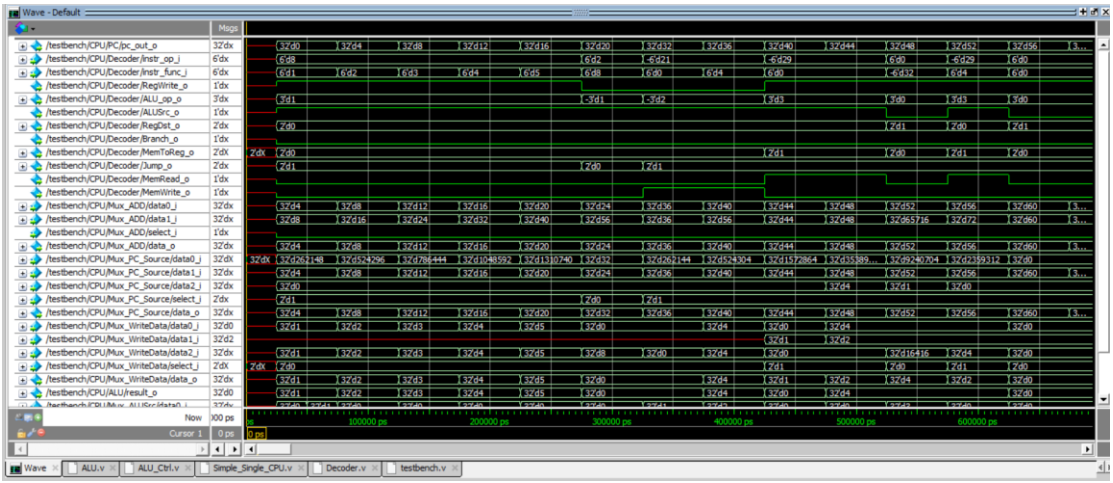
1. Mux_ALUSrc (choose which data (reg_read_data2/ sign_ext_out) be written into reg)
2. Mux_ADD (choose which data (add1_out/ add2_out) be written into Mux_PC_Source)

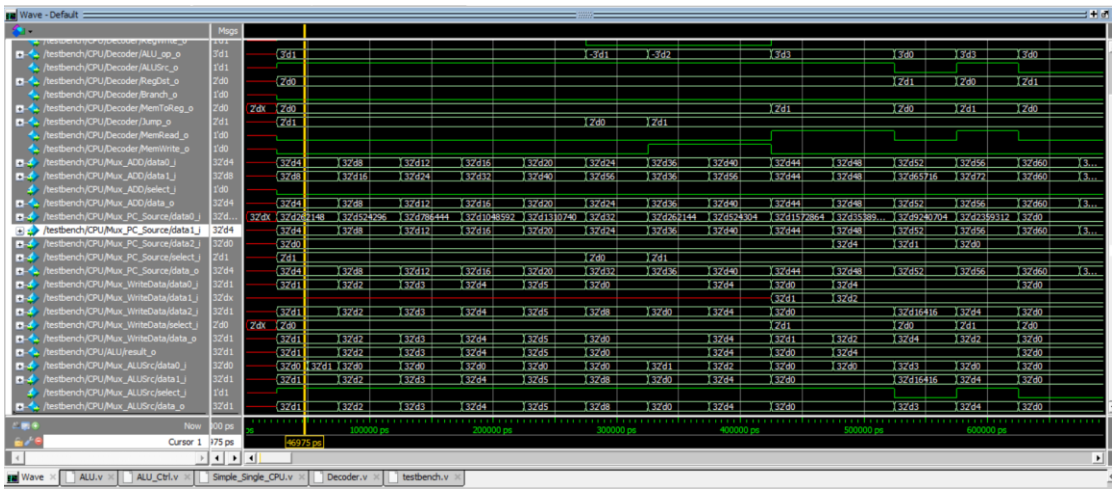
MUX_3to1

1. Mux_Write_Reg (data written to reg)
2. Mux_PC_Source (update pc address)
3. Mux_WriteData (data written into reg)

Finished part:

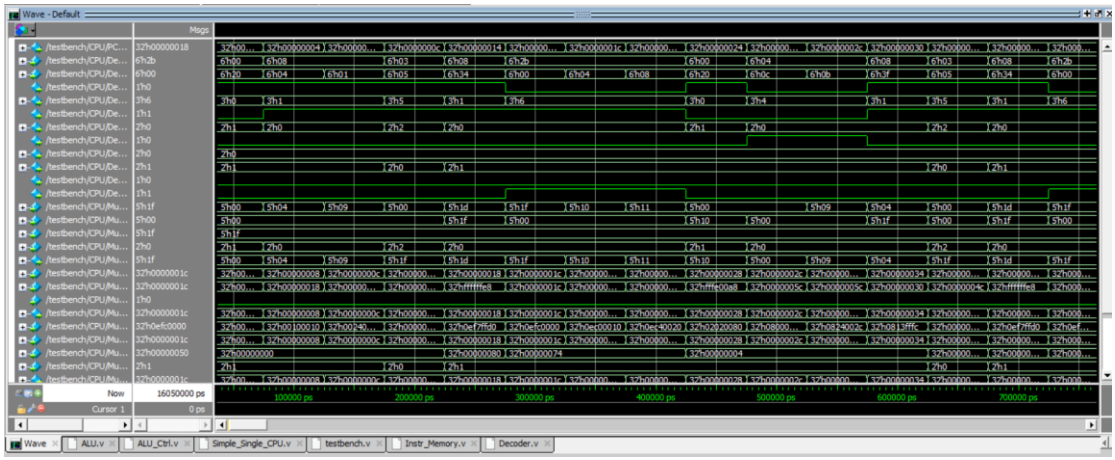
Test1

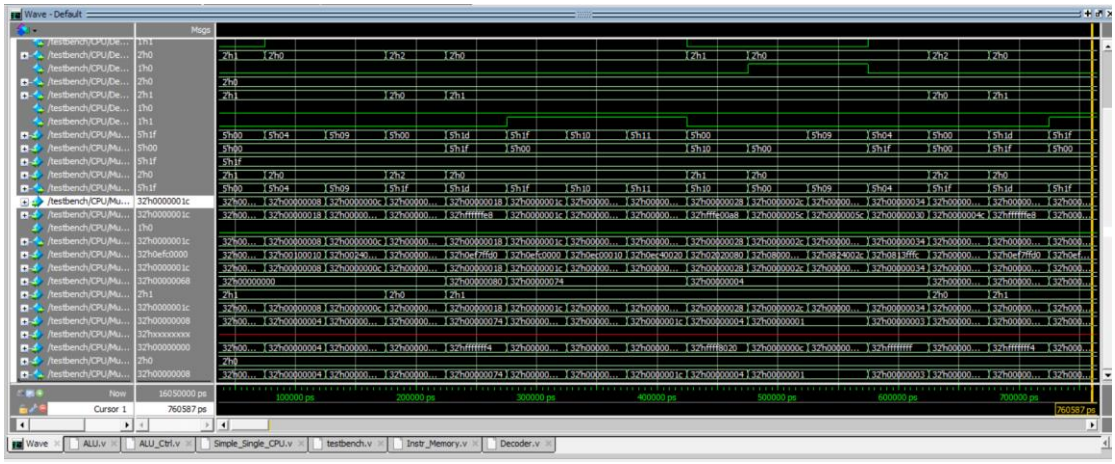




```
# Data Memory =          1,          2,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
# Registers
# R0 =          0, R1 =          1, R2 =          2, R3 =          3, R4 =          4, R5 =          5, R6 =          1, R7 =          2
# R8 =          4, R9 =          2, 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 =        128, R30 =          0, R31 =          0
# ** Note: $stop    : D:/12tmp/CO_Lab3/CO_Lab3/code/testbench.v(36)
#    Time: 16050 ns  Iteration: 0  Instance: /testbench
```

Test2





```
# Data Memory =          0,          0,         56,          4,         68,         18,          1,         56
# Data Memory =          3,         68,          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
# Registers
# R0 =          0, R1 =          0, R2 =          5, R3 =          0, R4 =          1, R5 =          0, R6 =          0, R7 =          0
# R8 =          0, R9 =          1, R10 =          0, R11 =          0, R12 =          0, R13 =          0, R14 =          0, R15 =          0
# R16 =          2, R17 =         68, R18 =          0, R19 =          0, R20 =          0, R21 =          0, R22 =          0, R23 =          0
# R24 =          0, R25 =          0, R26 =          0, R27 =          0, R28 =          0, R29 =         92, R30 =          0, R31 =         56
# ** Note: $stop    : D:/12tmp/CO_Lab3/CO_Lab3/code/testbench.v(36)
#    Time: 16050 ns  Iteration: 0  Instance: /testbench
# Break in Module testbench at D:/12tmp/CO_Lab3/CO_Lab3/code/testbench.v line 36
```

Problems you met and solutions:

1. **SW / LW 的地址錯誤**

   Sol:在 ALU 中將 SW/LW 的的指加上立即值(offset)，否則存到的地址都為同一個。

2. **Jal 存值方式錯誤，最後 ra 跳回的指令錯誤**

   Sol:透過將 Mux_Write_Reg 改為 MUX_3to1，當收到 jal 時 RegDst 設為 2，輸入 RegDst 為 31。

3. **在 test2 時 reg 內存值有誤**

   Sol：由於我是在 ALU 中處理 jal 及 jr 的地址存值及其 offest，而游於藝開始我將傳入 ALU 中的的記憶體位置傳成 RS 的讀取範圍(instr_memory_out[25:21])，之後將傳入值改為 reg_read_data1 就沒問題了。

Summary:

這次 LAB 相較於前兩次寫比較慢的原因主要都是指令之間沒有傳好而致，例如：遇到 jal 指令後要將的指存起來、該如何存，遇到 jr 指令時又要如何將 reg 中的值讀出來，還有在遇到 sw/lw 時的指的 offset 要如何去加，都是這次 LAB 主要遇到的問題。