計算機組織

112 學年度第 2 學期

Final Project

Pipelined ALU Design

老師： 朱守禮 老師

學生： 11127132 羅海綺

11127137 黃乙家

11127138 林雨臻

一、背景

為了使學生充分了解，該如何使用 Verilog HDL 與 Modelsim 模擬器，以 Midterm Project 所設計之 ALU Design 及計算機組織課程講義：

[1] Chapter 4:The Processor

[2] Pipelined Datapath

為基礎，設計一個 Pipelined MIPS-Lite CPU。

二、方法

1.功能說明：本 Project 包含下列 16 道 MIPS 指令。

(a) Integer Arithmetic: add, sub, and, or, sll, slt, andi

(b) Integer Memory Access: lw, sw

(c) Integer Branch: beq, j, jr

(d) Integer Multiply/Divide: multu

(e) Other Instructions: mfhi, mflo, no

2.設計要求：Datapath 與詳細架構圖(並以 PowerPoint 或 Word 設計繪製)。

(1) ALU:

(a) 設計重點說明：

- 需使用 Midterm Project 所設計之 ALU 完成 add, sub, and, or, sll, slt, andi 指令。

(2) Datapath:

(a) 設計重點說明：

- 所有指令之執行，須遵守 5-Stage Pipelined CPU 執行指令之行為。

(3) multu:

(a) 設計重點說明：

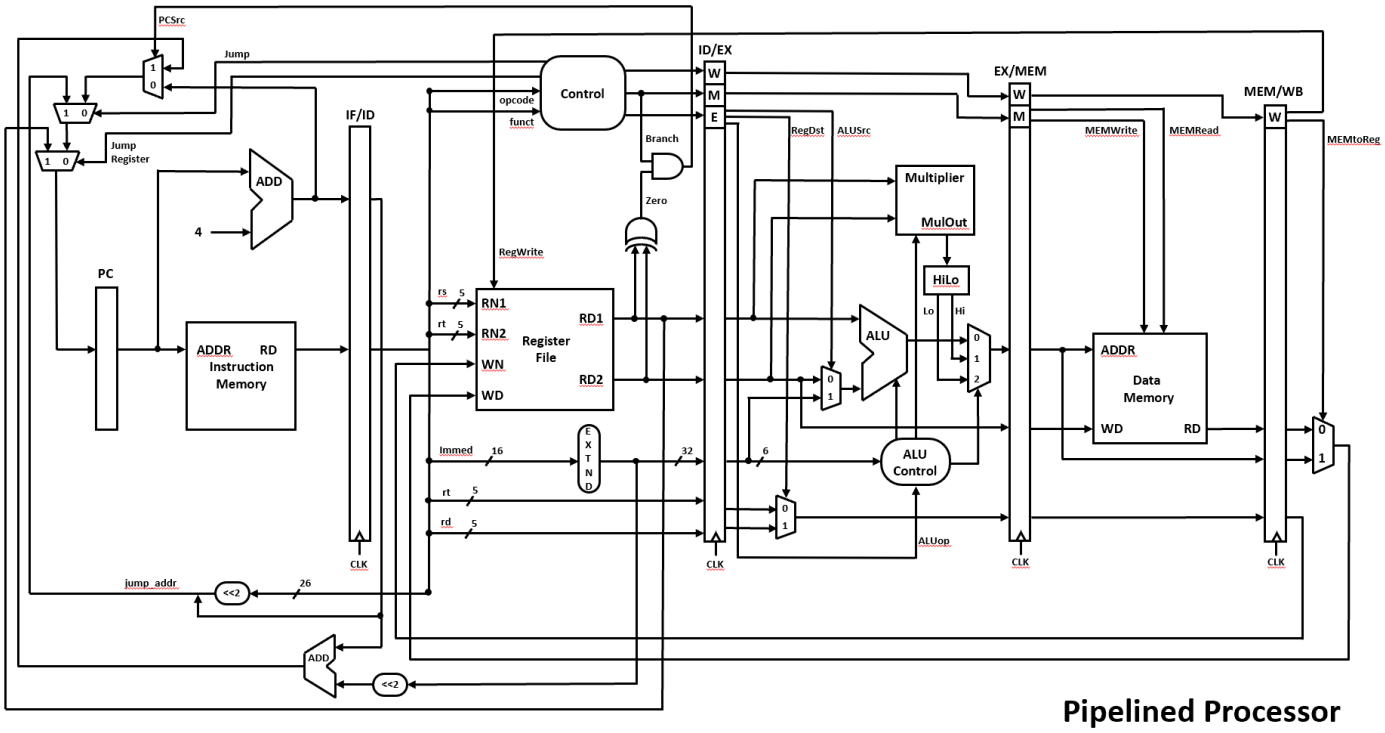
- 32-bits 無號數乘法指令。
- 需使用 Midterm Project 所設計之 Multiplier。

(5) Testbench:

(a) 設計重點說明：

- 須以讀檔的方式，讀入測試資料。
- Single Cycle 版本之 MIPS 處理器設計與 Testbench Verilog 範例，請詳見 i-Learning 上之「Single Cycle CPU and Testbench」
- 本機測時將有不同的測試資料，以確認設計正確性與完整性。

(b) Datapath 與詳細架構圖：請參考附錄(CO.pptx)。



Pipelined Processor

程式架構：

mips_pipeline:

IF - Instruction Fetch:

PC

PCADD (PC+4)

memory (InstrMem)

2-1 MUX*3

ID - Instruction Decode:

ControlUnit

sign_extend or unsign_extend

RegFile

BEQ　(從 MEM 移到 ID)

2-1 MUX*1

EX - Execute / Address Calculation:

ALU

ALUControl

Multiplier

HiLo

2-1 MUX*2

3-1 MUX*1

MEM - Memory Access (read write):

memory (DatMem)

WB - Write Back (result into register file):

2-1 MUX*1

三、結果

(1) mips_pipeline:

(a) 宣告線路:

(i) instruction bus

```
    // instruction bus
    wire[31:0] instr;
    wire[31:0] instr_id;
```

(ii) break out important fields from instruction

```
    wire [5:0] opcode, funct, funct_EX;
    wire [4:0] rs, rt, rd, shamt, shamt_EX;
    wire [15:0] immed;
    wire [31:0] extend_immed, b_offset;
    wire [25:0] jumpoffset;

    wire [4:0] rt_ex;
    wire [4:0] rd_ex;

    wire [31:0] unsign_immed; // 無符號擴展
    wire [31:0] extended_immed_id, extended_immed_ex; // 擴展結果
    wire [63:0] MultOut; // 乘法運算結果
    wire [31:0] HiOut, LoOut; // HiLo 輸出
```

(iii) datapath signals

```
    wire [4:0] rfile_wn;
    wire [31:0] rfile_rd1, rfile_rd2, rfile_wd, alu_b, alu_out, b_tgt_ex, pc_next, pc_jump,
                pc, pc_incr, dmem_rdata, jump_addr, branch_addr;

    wire [31:0] alu_or_hilo_out, alu_out_mem, alu_out_wb;

    wire [31:0] dmem_rdata_wb;

    wire [31:0] rfile_rd1_ex, rfile_rd2_ex, rfile_rd2_mem;
    wire [4:0] rfile_wn_mem, rfile_wn_wb;

    wire [1:0] SelHilo; // 選擇 Hi 或 Lo
    wire [31:0] pc_id;
```

(iv) control signals

```
    assign opcode = instr_id[31:26];
    assign rs = instr_id[25:21];
    assign rt = instr_id[20:16];
    assign rd = instr_id[15:11];
    assign shamt = instr_id[10:6];
    assign funct = instr_id[5:0];
    assign immed = instr_id[15:0];
    assign jumpoffset = instr_id[25:0];
```

(v) 分割線路

```
    wire Zero, Jump, JumpRegister;
    wire SignExtendSel;
    wire [2:0] Operation; // 控制 ALUControl
    wire [5:0] SignaltoMULTU; // 乘法運算
    wire [1:0] WB_ID, WB_EX, WB_MEM, WB_WB; // WB = {RegWrite, MemtoReg};
    wire [2:0] MEM_ID, MEM_EX, MEM_MEM; // MEM = {MemRead, MemWrite, MemtoReg};
    wire [3:0] EX_ID, EX_EX; // {RegDst, ALUOp, ALUSrc};
```

(b) Pipelining stages:

(i) IF - Instruction Fetch

```
70      reg32 PC( .clk(clk), .rst(rst), .en_reg(1'b1), .d_in(pc_next), .d_out(pc) );
71      add32 PCADD( .a(pc), .b(32'd4), .result(pc_incr) );
72      mux2 #(32) PCMUX( .sel(PCSrc), .a(pc_incr), .b(b_tgt_ex), .y(branch_addr) );
73      mux2 #(32) JMUX( .sel(Jump), .a(branch_addr), .b(jump_addr), .y(pc_jump) );
74      // mux Jump Register
75      mux2 #(32) JR_MUX( .sel(JumpRegister), .a(pc_jump), .b(rfile_rd1), .y(pc_next) );
76      memory InstrMem( .clk(clk), .MemRead(1'b1), .MemWrite(1'b0), .wd(32'd0), .addr(pc), .rd(instr) );

78    IF_ID IFID( .clk(clk), .reset(rst), .nextPC_IF(pc_incr), .RD_IF(instr), .nextPC_ID(pc_id), .RD_ID(instr_id) );
```

(ii) ID - Instruction Decode

```
82      ControlUnit CTL(.clk(clk), .OpCode(opcode), .Funct(funct), .EX(EX_ID),
83      .MEM(MEM_ID), .WB(WB_ID), .ExtendSel(SignExtendSel), .Jump(Jump), .JR(JumpRegister));
84
85      // -*-*-*-*-Extender-*-*-*-*-
86      sign_extend SignExt( .immed_in(immed), .ext_immed_out(extend_immed) );
87      unsign_extend UnsignExt( .immed_in(immed), .ext_immed_out(unsign_immed) );
88      mux2 #(32) Extender(.sel(SignExtendSel), .a(extend_immed), .b(unsign_immed), .y(extended_immed_id) );
89      // -*-*-*-*-Extender-*-*-*-*-
90
91      reg_file RegFile( .clk(clk), .RegWrite(WB_WB[1]), .RN1(rs), .RN2(rt), .WN(rfile_wn_wb),
92                        .WD(rfile_wd), .RD1(rfile_rd1), .RD2(rfile_rd2) );
93
94      // branch offset shifter
95      assign b_offset = extend_immed << 2;
96      // jump offset shifter & concatenation
97      assign jump_addr = { pc_id[31:28], jumpoffset <<2 };
98      add32 BRADD( .a(pc_id), .b(b_offset), .result(b_tgt_ex) );
99      // beq
100     BEQ beq( .opcode(opcode), .zero(Zero), .a(rfile_rd1), .b(rfile_rd2) );
101     and BR_AND(PCSrc, MEM_ID[0], Zero );
103     ID_EX IDEX(.clk(clk), .reset(rst), .RT_ID(rt), .RT_EX(rt_ex),
104     .RD_ID(rd), .RD_EX(rd_ex), .RD1_ID(rfile_rd1), .RD1_EX(rfile_rd1_ex),
105     .RD2_ID(rfile_rd2), .RD2_EX(rfile_rd2_ex), .imm_ID(extended_immed_id), .imm_EX(extended_immed_ex),
106     .WB_ID(WB_ID), .WB_EX(WB_EX), .EX_ID(EX_ID), .EX_EX(EX_EX),
107     .MEM_ID(MEM_ID), .MEM_EX(MEM_EX), .func_ID(funct), .func_EX(funct_EX),
108     .shamt_ID(shamt), .shamt_EX(shamt_EX) );
112   ALU ALU(.dataA(rfile_rd1_ex), .dataB(alu_b), .control(Operation), .dataOut(alu_out), .shamt(shamt_EX) );
113   mux2 #(5) RFMUX( .sel(EX_EX[3]), .a(rt_ex), .b(rd_ex), .y(rfile_wn) );
114   mux2 #(32) ALUMUX( .sel(EX_EX[0]), .a(rfile_rd2_ex), .b(extended_immed_ex), .y(alu_b) );
115   // mux HiLo
116   MUX3_1 #(32) MUX_HiLo( .sel(SelHilo), .a(alu_out), .b(HiOut), .c(LoOut), .y(alu_or_hilo_out) );
117   ALUControl ALUCTL( .clk(clk), .ALUOp(EX_EX[2:1]), .Funct(funct_EX), .ALUOperation(Operation), .SignaltoMULTU(SignaltoMULTU), .SelHilo(SelHilo) );
118   Multiplier MULTU( .clk(clk), .reset(rst), .dataA(rfile_rd1_ex), .dataB(rfile_rd2_ex), .Signal(SignaltoMULTU), .dataOut(MultOut) );
119   HiLo HiLo( .clk(clk), .reset(rst), .MulAns(MultOut), .HiOut(HiOut), .LoOut(LoOut) );
120   EX_MEM EXMEM(.clk(clk), .reset(rst), .WB_EX(WB_EX), .MEM_EX(MEM_EX),
121   .WB_MEM(WB_MEM), .MEM_MEM(MEM_MEM), .WN_EX(rfile_wn), .WN_MEM(rfile_wn_mem),
```

(iii) EX - Execute / Address Calculation

```
129     MEM_WB MEMWB(.clk(clk), .reset(rst), .WB_MEM(WB_MEM), .WB_WB(WB_WB),
130     .ALUOut_MEM(alu_out_mem), .ALUOut_WB(alu_out_wb), .DataMEM_RD_MEM(dmem_rdata), .DataMEM_RD_WB(dmem_rdata_wb),
131     .WN_MEM(rfile_wn_mem), .WN_WB(rfile_wn_wb) );
```

(iv) MEM - Memory Access (read write)

```
125     memory DatMem( .clk(clk), .MemRead(MEM_MEM[2]), .MemWrite(MEM_MEM[1]), .wd(rfile_rd2_mem),
126                    .addr(alu_out_mem), .rd(dmem_rdata) );
129   MEM_WB MEMWB(.clk(clk), .reset(rst), .WB_MEM(WB_MEM), .WB_WB(WB_WB),
130   .ALUOut_MEM(alu_out_mem), .ALUOut_WB(alu_out_wb), .DataMEM_RD_MEM(dmem_rdata), .DataMEM_RD_WB(dmem_rdata_wb),
131   .WN_MEM(rfile_wn_mem), .WN_WB(rfile_wn_wb) );
```

(v) WB - Write Back (result into register file)

```
137     mux2 #(32) WRMUX( .sel(WB_WB[0]), .a(alu_out_wb), .b(dmem_rdata_wb), .y(rfile_wd) );
```

(2) IF - Instruction Fetch:

(a) PC:

當時脈 clk posedge 時，若非 reset 就輸出算好的下道指令位置。

(b) memory:

MemRead 或者 mem_array 有變化時，判斷 MemRead 是否為 1，若為 1 至指定記憶體位置讀取資料。

當時脈 posedge 且 MemWrite 為 1 時，將資料寫入指定記憶體位置。

(3) IF_ID:

若非 reset，傳遞 instruction 以及 nextPC。

(4) ID - Instruction Decode:

(a) ControlUnit:

判斷 OpCode，設定 control signals。

(b) reg_file:

當 RN1 或 file_array 有變化時，判斷 RN1 是否為 0，若為 0 直接賦值 RD1=zero，否則 RD1 = file_array[RN1]。

RN2 同理。

當時脈 posedge 時判斷 RegWrite 訊號是否為 1 且 WN 是否不等於 0，若皆成立，寫入資料 WN 暫存器中。

(c) unsign_extend:

無號數擴充直接補 16'b0。

(d) sign_extend:

有號數擴充將第 15 個 bit 補 16 次。

(e) BEQ:

我們將 BEQ 指令從 MEM 移到 ID 階，提早判斷並決定是否跳躍，減少 nop 數量從 3 個至 1 個。

實現方法：先判斷 opcode 是否為 BEQ，再比較兩個基底暫存器的內容是否相等，若成立則將 zero 設為 0。

(5) ID_EX:

若非 reset，傳遞訊號至下階段。

(6) EX - Execute / Address Calculation:

(a) ALUControl:

用 funct 決定 ALU 訊號。

(b) ALU:

ALU 32 位元做 AND、OR、ADD、SUB、SLT、SLL、將乘法器 64 位元的運算結果分割成 Hi、Lo、移位器左移運算。

(c) Multiplier:

執行期中 project 的乘法運算。

(d) HiLo:

若非 reset，將 HiLo 設為乘法運算的結果。

(7) EX_MEM:

若非 reset，傳遞訊號至下階段。

(8) WB - Write Back (result into register file):

用一個 2-1MUX 判斷是否要寫回 reg_file。

四、討論

(1) Modelsim 驗證結果：

1. // lw $s1(17,2), $t7(15,21), 0

00

00

2F

8E

// nop

00

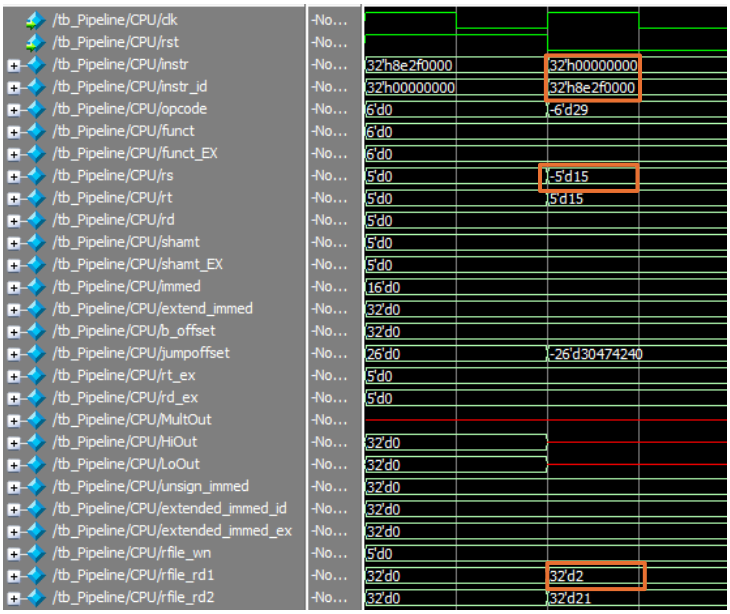
00

00

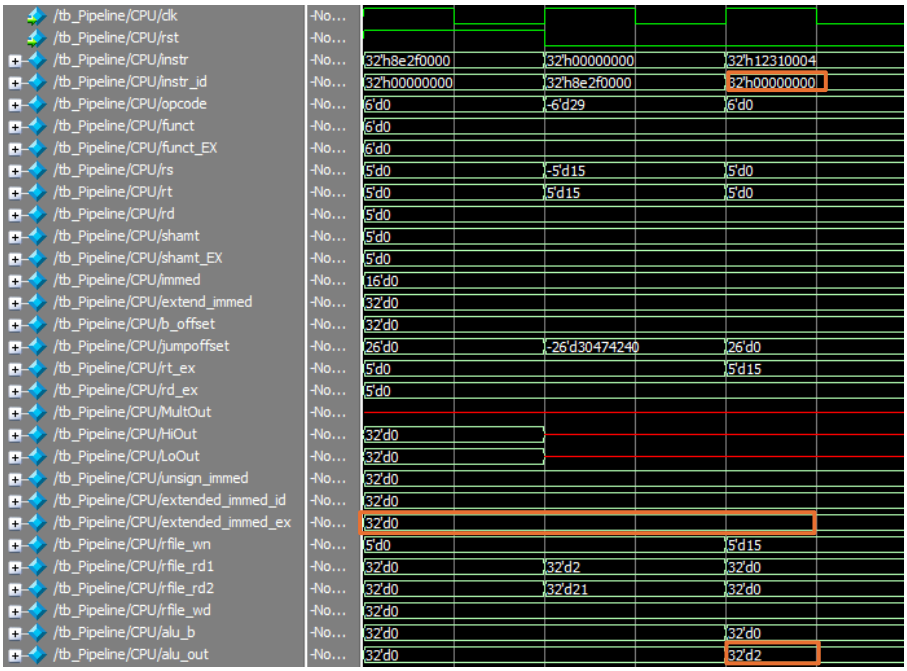
00

第一個 cycle 先 fetch 指令，因為電腦以 little Endian 的方式讀取，所以顯示 8E2F0000

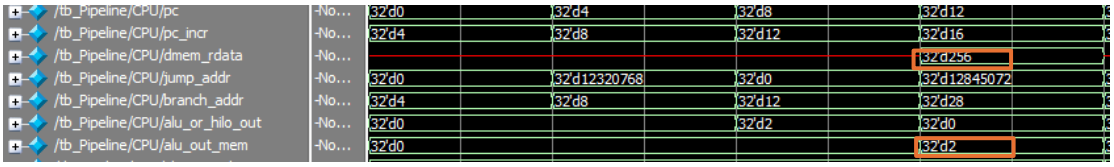
第二個 cycle 將 lw 指令解碼，並 fetch nop 指令，因為 I-type 只有兩個暫存器，而 lw 只用到 rdl，因此我們這時只需要看 rs 的位置與 rdl 的值。



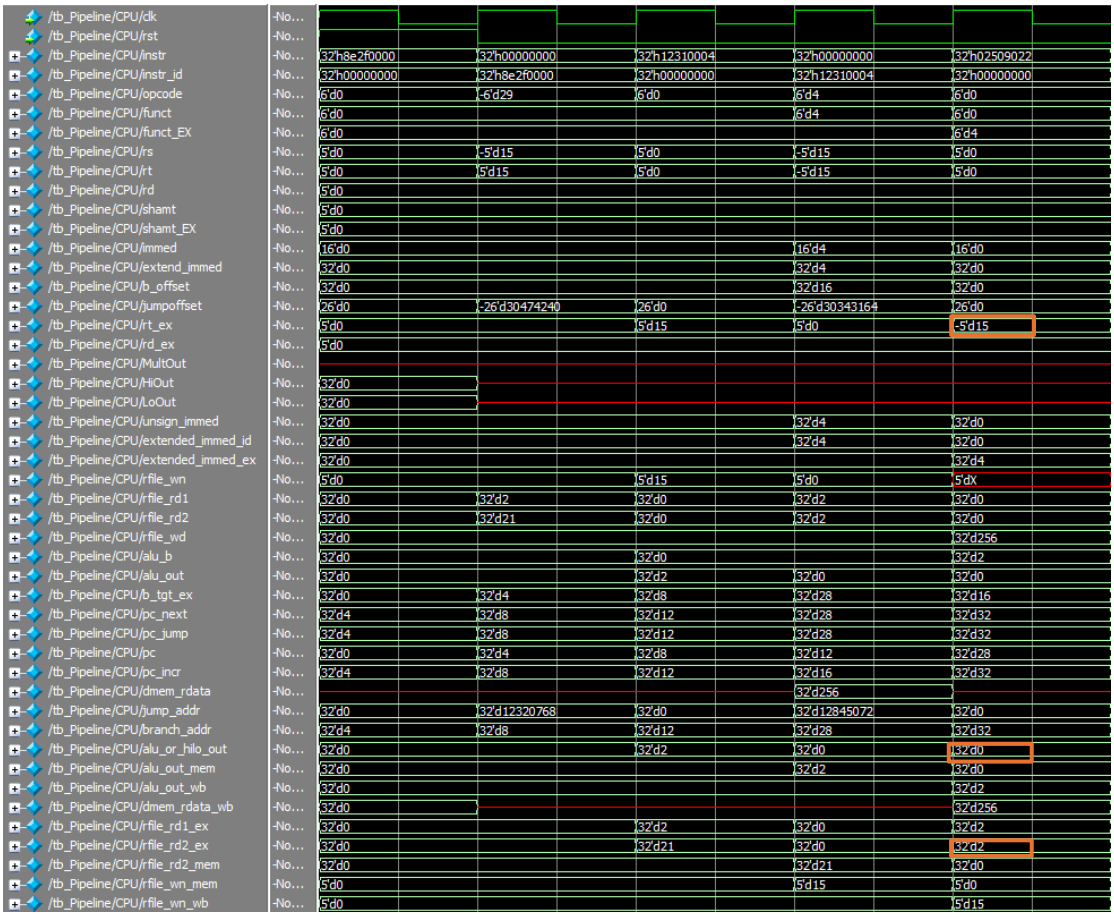
第三個 cycle，lw 進行運算，我們給的立即值為 0，因此 ALU 進行運算後的位置為 2，同時 nop 進行解碼。



第四個 cycle，lw 進行 memeory access，我們從 ADDR 讀到暫存器的位置=2，得到的記憶體的資料=256。



第五個 cycle，lw 將計算後暫存器 write back to register file 的 wn(rt)。



2.// beq　　$s1, $s1, 4

04

00

31

12

// nop

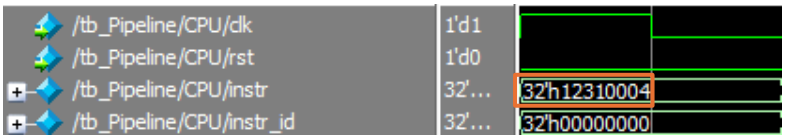
00

00

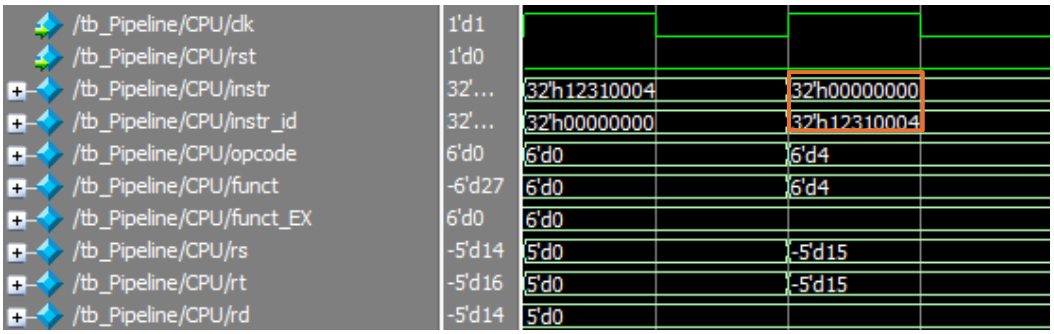
00

00

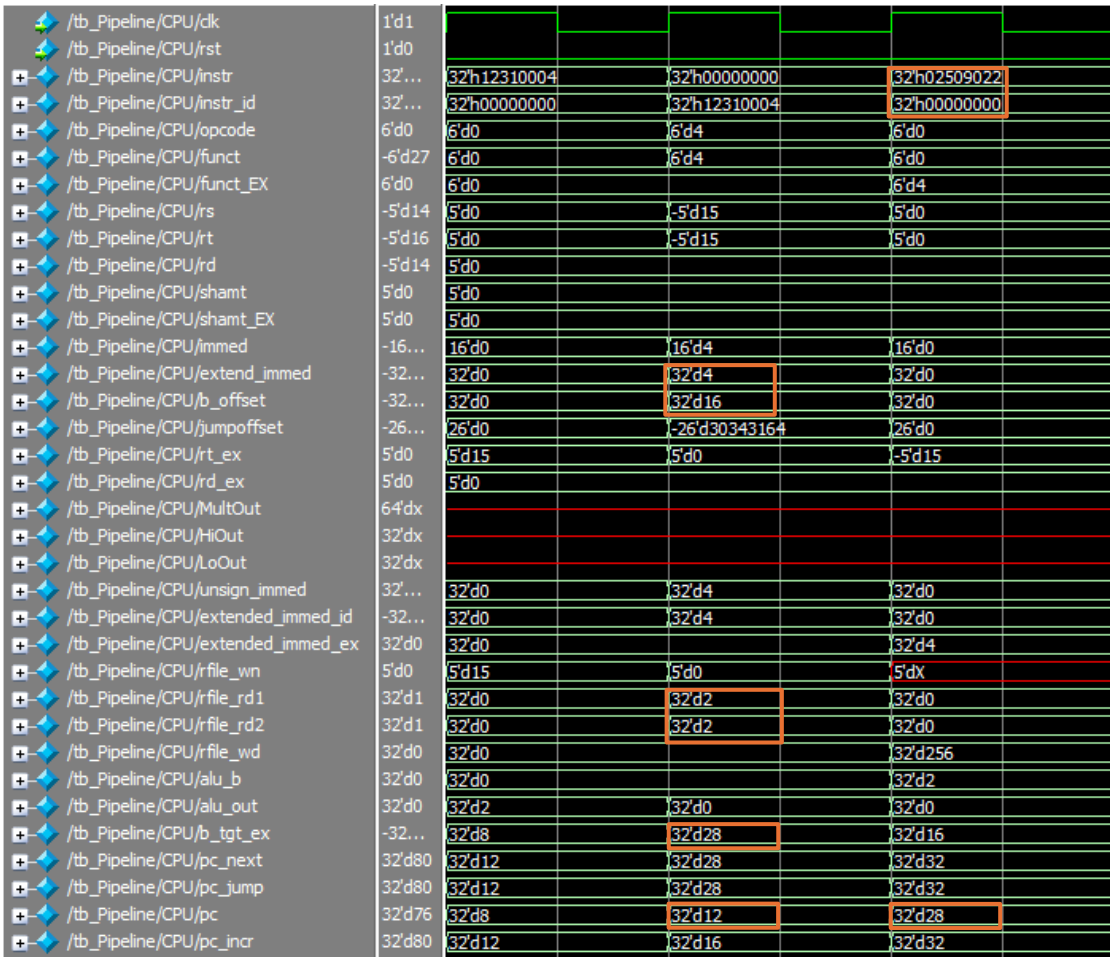
第一個 cycle 先 fetch beq 指令。



第二個 cycle 將 beq 指令解碼，fetch nop 指令。



第三個 cycle 判斷 rd1、rd2 值是否相等，由圖可知相等，偏移量為 4， offset 為 16，將 offset 與 PC+4 相加，得出 28，回傳給 pc。



3.

// sub $s2(18), $s0(16,1), $s2(18,3)

22

90

50

02

// nop

00

00

00

00

// nop

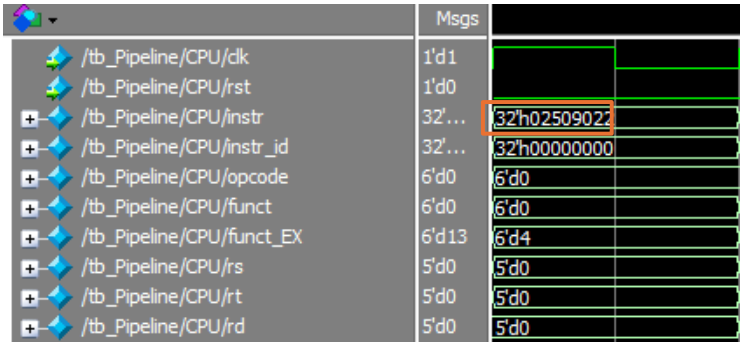
00

00

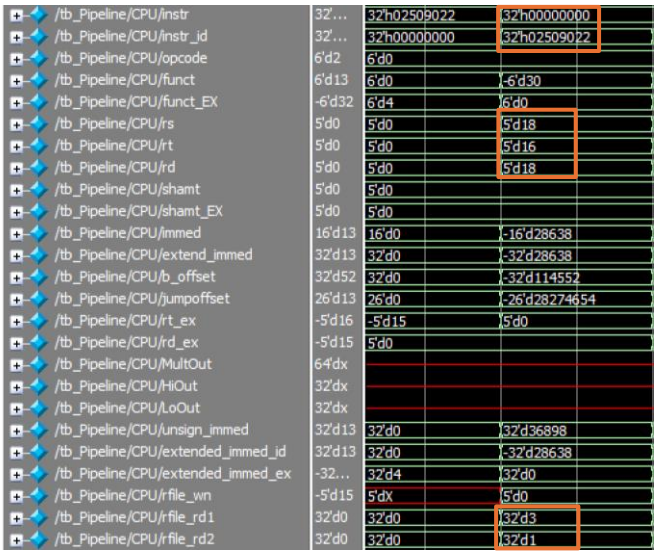
00

00

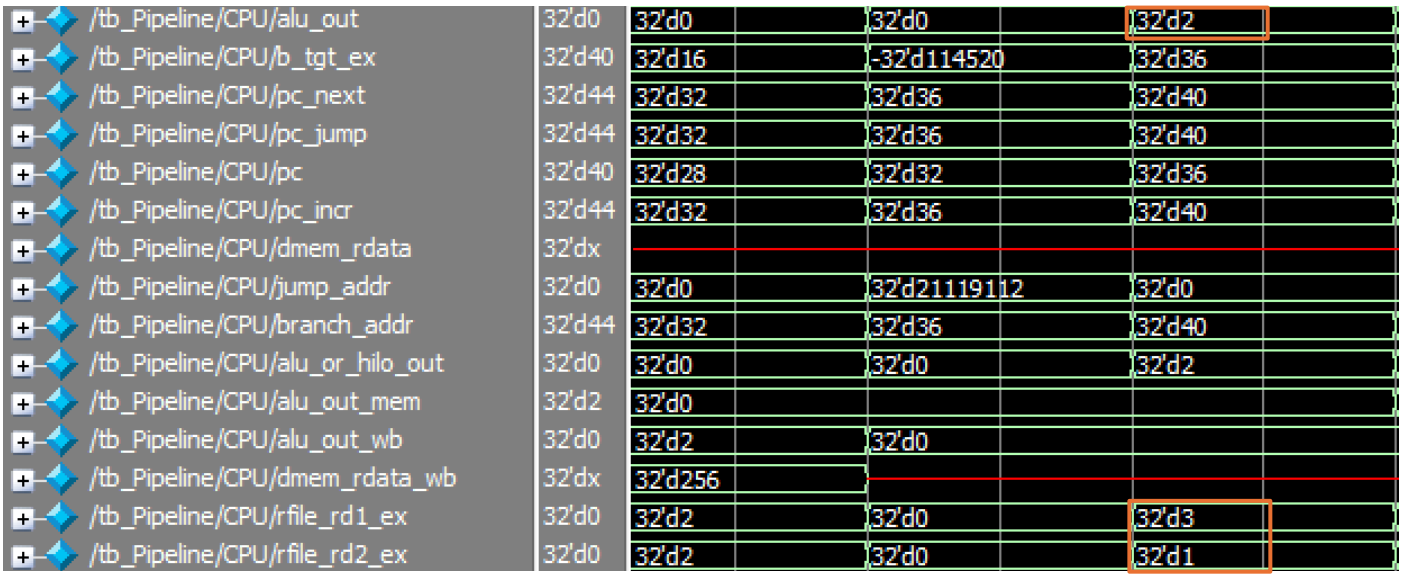
第一個 cycle 先 fetch sub 指令。



第二個 cycle 會將指令解碼，找到 rs、rt 的位置，rd1、rd2 為暫存器的值。並 fetch nop 指令。

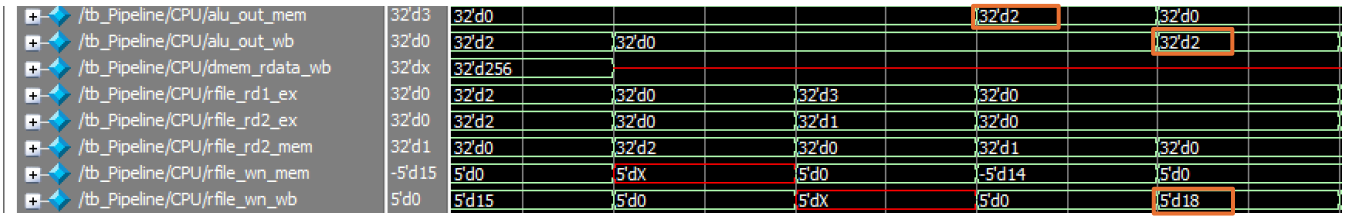


第三個 cycle 會將 rd1、rd2 放入 ALU 進行運算，得出 2。



第四個 cycle，由於 R-type 不需要經過 data memory，但還是需要傳遞。

第五個 cycle，要將 ALU 計算完的值 write back to rd，這時 rd 暫存器的位置為 18，值為 2。



4.

// j 13

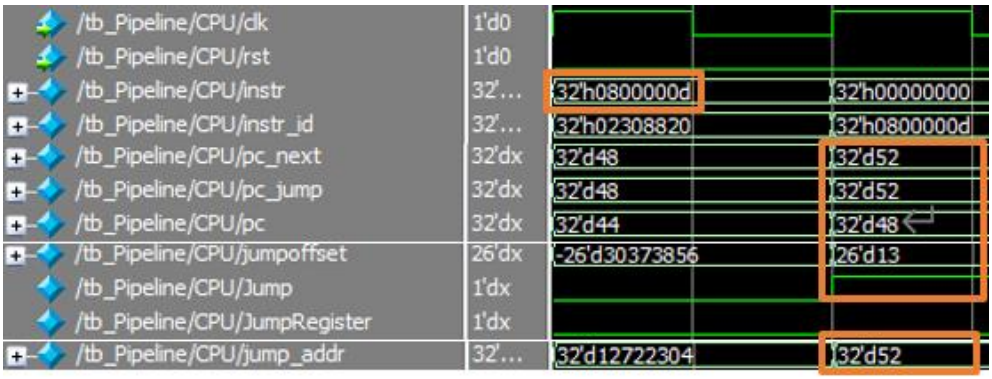
0D

00

00

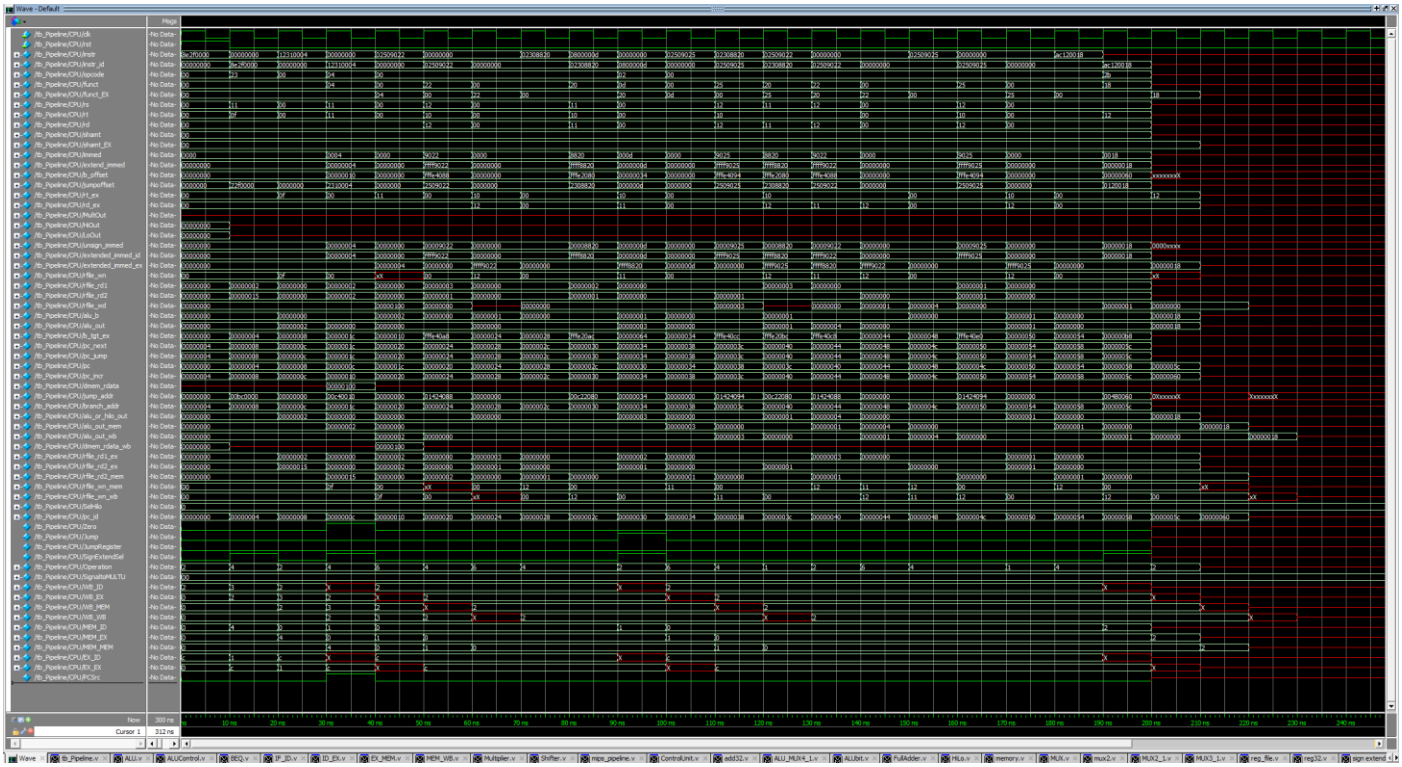
08

第一個 cycle fetch jump 指令，第二個 cycle 將 jumpoffset 與 pc+4[31:28]concat，得出 jump_addr=52，同時 control Unit 會傳出 jump 指令給 jumpMUX 去選擇，最後將 pc_next 設為 52。



五、結論

完整波型圖：



各組員分工方式與負責項目：

(1) 分工方式：

組員討論寫法後，一人寫一支程式，其他人透過加註解及撰寫書面報告的方式理解程式，為求所有人都理解程式內容。

(2) 負責項目：

羅海綺：書面報告(驗證程式)、畫圖、部分 module

黃乙家：mips_pipeline、串接所有程式、debug

林雨臻：書面報告(模組說明)、部分 module

心得感想：

這次的 project 要實作 MIPS 5-Stage Pipeline Processor ，此次的架構是基於期中所寫的模組做擴充而完成的，本來我們認為應該只需要多寫一些串接各階段訊號的 module 再加一個統整的 module 就可以完成了，但實際寫後卻發現其實並沒有想像中的那麼簡單，因為課本中的圖並沒有完整畫出所有指令的訊號傳遞方法，尤其是我們這屆的特殊指令 jr 跟 andi 因為之前對他們的使用並不了解，因此我們需要上網查其他架構圖，再逐一推算出指令的傳遞路線跟每個 signal 的值 。

在驗證波形圖時，展開後那一長串的訊號常常令我們看的眼花撩亂，且也常會發現有線路少傳遞或者指令進行了不對的運算，而此時就需要回去將相關連的 module 都再看過一次尋找錯誤的來源，或者再加入少傳遞的訊號，並將那部分指令的內容補齊，且因為電路是同步運行的，因此雖然不是每到指令都會用到所有 signal，但他們卻會在電路中被同步改變，在 debug 時要回來對訊號時，就要顧及出現錯誤的地方可能是之前的哪道指令造成的。

這次的 project 遇到 hazard 問題時，雖然只需要使用軟體解法就好，但因為我們一開始以為要需要硬體解法，因此多繞了不少彎路，此外我們還發現儘管已經加入 nop 進 instr 了，但執行乘法器時乘數跟被乘數的值卻會被 nop 蓋掉，發生 data hazard ，這才發現期中時所寫的乘法器除了最初發現的不會被重置的問題之外，還需要將 counter 寫入乘法器中，使得 32 次運算都直接在乘法器中進行就好，否則在訊號傳遞的過程中因為接收到新的值(nop)RN1 跟 RN2 都會被改成 0。

此次的 project 使我們學習到了很多，雖然並沒有實作硬體解 hazard 的方法，但卻還是具有一定的難度，但不得不說，成功完成後的成就感會讓我們覺得自己的努力並沒有白費。