

**National Yang Ming Chiao Tung University**

# **Final Project**

## **Computer Organization**

111651070 張柏豐

Advisor: Professor Liu, Chih-Wei Ph.D.

January, 2025

# Content

<b>Chapter 1 Introduction .....</b>	<b>4</b>
1.1 Abstract .....	4
1.2 Method .....	4
<b>Chapter 2 Non-Pipeline MIPS CPU .....</b>	<b>5</b>
2.1 Relationship Between SP.v & PATTERN.v .....	5
2.1.1 Abstract .....	5
2.1.2 Architecture .....	5
2.1.3 Workflow .....	6
2.2 SP.v Part .....	7
2.2.1 Register & I/O Declaration .....	7
2.2.2 Signal Reset Part .....	7
2.2.3 Decode & ALU Compute .....	8
2.2.4 Delay Counter .....	9
2.3 PATTERN.v Part .....	11
2.3.1 Main Function (initial) .....	11
2.3.2 Check Answer Task .....	11
2.4 Waveform & Compile Result .....	13
<b>Chapter 3 Pipeline MIPS CPU .....</b>	<b>14</b>
3.1 Relationship Between SP_pipeline.v & PATTERN_p.v .....	14
3.1.1 Abstract .....	14
3.1.2 Architecture .....	14
3.1.3 Workflow .....	15
3.2 SP_pipeline.v Part .....	15
3.2.1 Register & I/O Declaration .....	16
3.2.2 IF Stage (Instruction Fetch) .....	17
3.2.3 ID Stage (Instruction Decode) .....	17

3.2.4 EX Stage (Execution) .....	18
3.2.5 MEM Stage (Memory Read / Write) .....	19
3.2.6 WB Stage (Write Back) .....	19
<b>3.3 PATTERN_p.v Part .....</b>	<b>21</b>
3.3.1 Main Function (initial) .....	21
3.3.2 Input Task .....	22
3.3.3 Check Answer Task .....	24
<b>3.4 Waveform &amp; Compile Result .....</b>	<b>26</b>
<b>Chapter 4 Conclusion .....</b>	<b>27</b>
4.1 Final Project Feedback .....	27
4.2 Course Feedback .....	27

# Chapter 1 Introduction

## 1.1 Abstract

本次的計算機組織期末專題為撰寫一個 Non-Pipeline 的 MIPS CPU 架構，以及一個 Pipeline 的 MIPS CPU 架構，依照助教提供的 TESTBED.v, TESTBED\_p.v 與包含部分提示的主 CPU Module: SP.v, SP\_pipeline.v、檢查 CPU 執行結果的 Module: PATTERN.v, PATTERN\_p.v，完成剩餘的部分，並使用 iVerilog 來編譯並執行，確保撰寫的結果正確無誤。以下是常見的 MIPS CPU 架構 (Datapath & Control System)：

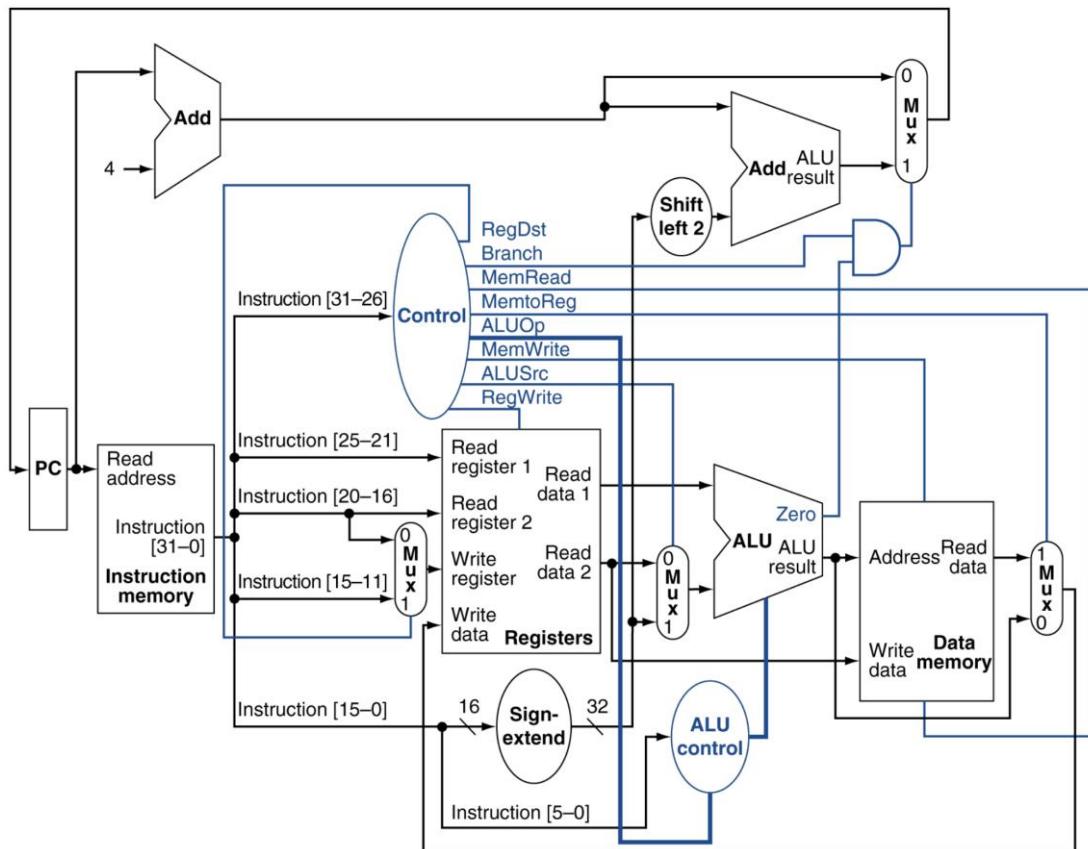


Figure 1. Simple MIPS CPU Architecture

## 1.2 Method

在接下來的實作中，我將從 Non-Pipeline 的 MIPS 架構開始，先完成 SP.v 的部分(包含 Decode, Memory Read / Write, Register Read / Write...)，接著依照自己設計的 SP.v，去設計 PATTERN.v 來進行對應的檢查。在完成過後會進一步處理 Pipeline 的情況，同樣先完成架構的設計 SP\_pipeline.v (包含 IF, ID, EX, MEM, WB 五個階段)，再接著處理 PATTERN\_p.v 來檢查設計結果。

# Chapter 2 Non-Pipeline MIPS CPU

## 2.1 Relationship Between SP.v & PATTERN.v

### 2.1.1 Abstract

對於 Non-Pipeline 的部分，因為不用將一個指令拆成多個階段，因此設計的方法較為簡單，僅需要在一個 Clock Cycle 內先從給定的 Instruction Address (PC) 找到 32-bit 的目標指令後，分解成 opcode(6-bit), rs(5-bit), rt(5-bit), rd(5-bit), shamt(5-bit), funct(6-bit), imm(16-bit)，接著再依照其對應的 opcode, funct 去 Decode，並直接進行特定的 Register Read / Write, Memory Read / Write 即可。

### 2.1.2 Architecture

為了方便後續的設計，首先繪製了其大致的架構圖，結果如下圖 Figure 2 所示。主要由幾個元件組成：PC (Program Counter), Instruction Memory, Data Register, ALU, Control System, Data Memory，分別負責不同的資料處理，接下來將詳細介紹整體的運作過程。

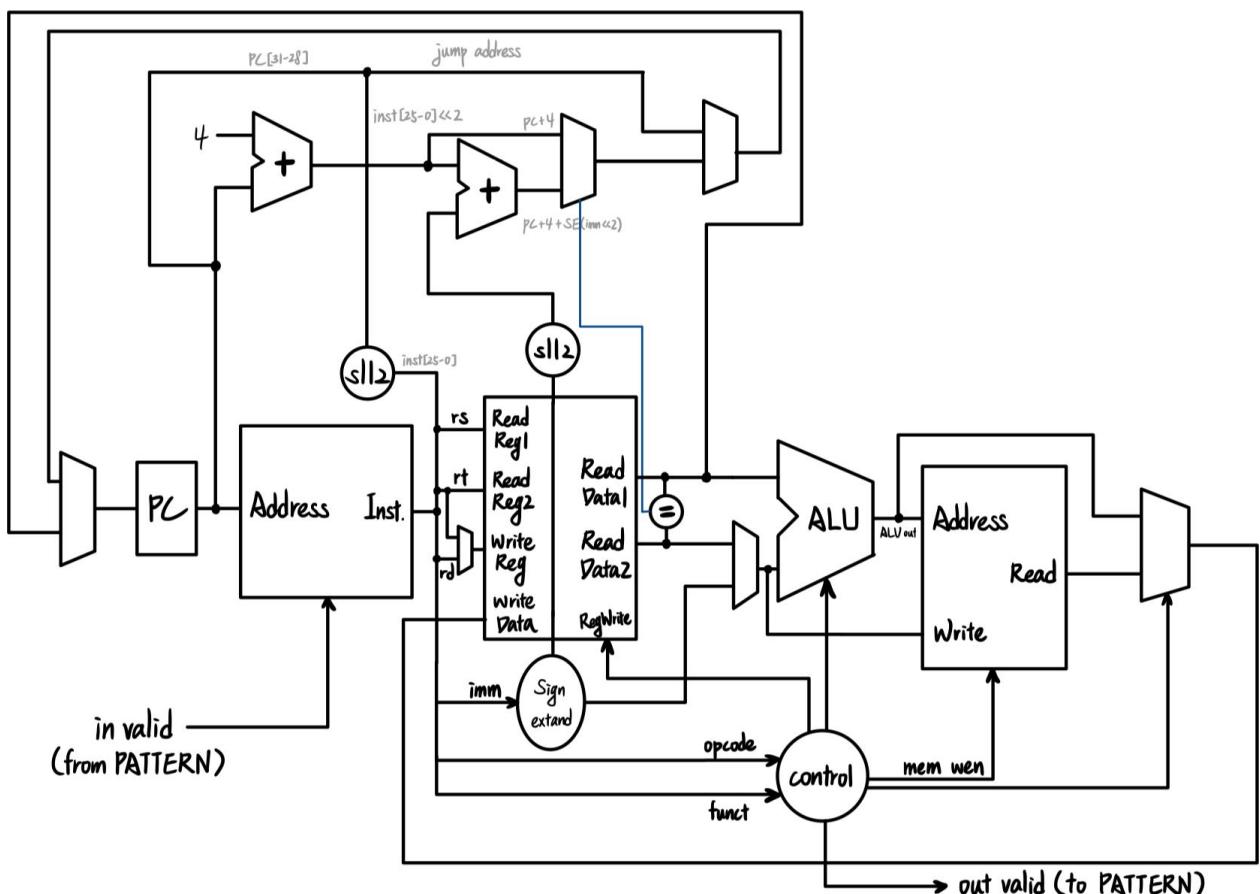


Figure 2. Non-Pipeline MIPS CPU Architecture

### 2.1.3 Workflow

整體來說，SP.v 與 PATTERN.v 的執行關係大致如下述：

1. PATTERN.v 先從給定的指令檔”instruction.txt”、記憶體檔”mem.txt”讀取資料。
2. 啟動 Reset 訊號將 SP.v 的 CPU 各暫存器與參數重置。
3. 初始化 PATTERN.v 所有的暫存器與參數。
4. PATTERN.v 檢查 SP.v 的 Reset 是否正確處理完畢。
5. PATTERN.v 開始產生週期性的 Clk 訊號給 SP.v。
6. PATTERN.v 產生一個指令並將 in\_valid 設為 1 持續 1 個 Cycle 使 SP.v 能夠讀取該指令。
7. SP.v 在下一次 Clk 的 Positive Edge 觸發時開始執行該指令的運算。
8. PATTERN.v 等待 SP.v 處理該指令，在處理完成後 SP.v 會發送 out\_valid 訊號告知。
9. PATTERN.v 在接收到 out\_valid 為 1 且 Clk 達到 Negative Edge 時就會立即執行與之相同的指令，並接著馬上驗證其結果是否與 SP.v 相同，檢查的方式為逐一檢查 32 個 Register 內儲存的數據是否相同，以及當時的 Instruction Address (PC)是否相同。
10. 若該次指令的檢查通過，回到步驟 6 並往復循環直到 1000 次的指令執行完畢。
11. 全部的指令都執行完成後，統一檢查 Memory 上 4096 個 Address 的資料是否也相同。
12. 完成任務。

## 2.2 SP.v Part

### 2.2.1 Register & I/O Declaration

首先是關於 Register 以及 SP module 的 I/O 部分宣告，除了基本常見的幾個必要 Register 之外，比較特別的是 3-bit 的 delay\_counter，該 Register 主要是用來記錄當前經過的 Clk，目的是處理對於 lw 這種需要等待 Data Memory 將資料讀出，而不能立即在當前的 Cycle 完成該指令的情況，在每個循環使用過後就會重置其值，等待下一個循環的使用。

```
1  module SP(
2      // INPUT SIGNAL
3      clk,
4      rst_n,
5      in_valid,
6      inst,
7      mem_dout,
8      // OUTPUT SIGNAL
9      out_valid,
10     inst_addr,
11     mem_wen,
12     mem_addr,
13     mem_din
14 );
15
16 //-----
17 // INPUT AND OUTPUT DECLARATION
18 //-----
19
20 input          clk, rst_n, in_valid;
21 input [31:0] inst;
22 input signed [31:0] mem_dout;    // 32-bit signal from memory
23 output reg      out_valid; // Let outputs valid
24 output reg      [31:0] inst_addr; // 32-bit address of current instruction
25 output reg      mem_wen; // Memory write enable
26 output reg      [11:0] mem_addr; // 12-bit Memory address
27 output reg signed [31:0] mem_din; // 32-bit signal to write into memory
28
29 //-----
30 // DECLARATION
31 //-----
32
33 // REGISTER FILE, DO NOT EDIT THE NAME.
34 reg signed [31:0] r [0:31];    // 32 Register file, each 32-bit
35 reg signed [31:0] pc;        // 32-bit Program counter
36 reg signed [31:0] alu_out;   // ALU output
37 reg [5:0] opcode, funct;    // Instruction field
38 reg [4:0] rs, rt, rd, shamt; // Registers
39 reg [31:0] immediate;       // immediate
40 reg [2:0] delay_counter;    // Clock counter
41 integer i;                  // for loop index
```

Figure 3. SP.v: Register & I/O Declaration Code

### 2.2.2 Signal Reset Part

接著是處理 Reset 的部分，當 PATTERN.v 傳送 Reset Signal 給 SP.v 時，SP.v 會將其中幾個比較重要的訊號重置，包括 Instruction Address (inst\_addr), out\_valid, mem\_wen (Memory Write Enable), mem\_addr (Address input), mem\_din (Data to write into memory), delay\_counter, All Register r。

```

49     // If rst_n == 0, reset all signal
50     if (!rst_n) begin
51         pc <= 0;
52         inst_addr <= 0;
53         out_valid <= 0;
54         mem_wen <= 1;           // Only read
55         mem_addr <= 0;
56         mem_din <= 0;
57         delay_counter <= 7;
58         for (i = 0; i < 32; i = i + 1) begin
59             r[i] <= 0;
60         end

```

Figure 4. SP.v: Signal Reset Code

### 2.2.3 Decode & ALU Compute

接著是 SP.v 最主要的部份，當接收到 PATTERN.v 的 in\_valid=1 時，首先先將當前接收到的指令 inst 拆解成 opcode, rs, rt, rd, shamt, funct, imm，並且將 delay\_counter 歸零，mem\_wen 則先設為讀取模式，待需要寫入時再設為 0。接著開始進行 decode，對於 R-Type 我們將計算結果先儲存到 alu\_out，之後再統一將 alu\_out 寫入 r[rd]即可；而對於 I-Type 則直接將計算結果寫入 Register r[rt]內，分支指令則直接更新 pc，待下次的 Posedge 觸發時再更新到 inst\_addr 上，其餘指令則如同講義上的定義，以下 Figure 5 為此部分的程式碼：

```

61     // If in_valid == 1 (from PATTERN), start decoding
62     end else if (in_valid) begin
63         delay_counter <= 0;
64         mem_wen <= 1;           // Only read
65         opcode = inst[31:26];
66         rs = inst[25:21];
67         rt = inst[20:16];
68         rd = inst[15:11];
69         shamt = inst[10:6];
70         funct = inst[5:0];
71         immediate = inst[15:0];
72         // Execute instruction
73         case (opcode)
74             // ===== R-Type =====
75             6'b000000: begin
76                 case (funct)
77                     6'b000000: alu_out <= r[rs] & r[rt];      // and (0x00)
78                     6'b000001: alu_out <= r[rs] | r[rt];      // or (0x01)
79                     6'b000010: alu_out <= r[rs] + r[rt];    // add (0x02)
80                     6'b000011: alu_out <= r[rs] - r[rt];    // sub (0x03)
81                     6'b000100: alu_out <= (r[rs] < r[rt])? 1:0; // slt (0x04)
82                     6'b000101: alu_out <= r[rs] << shamt;   // sll (0x05)
83                     6'b000110: alu_out <= ~r[rs] | r[rt];   // nor (0x06)
84                     6'b000111: pc <= r[rs];                // jr (0x07)
85                     default: ;
86                 endcase
87             end
88             // ===== I-Type =====
89             6'b000001: r[rt] <= r[rs] & {16'b0, immediate[15:0]}; // r[rt] = r[rs] & ZE(immm): andi (0x01)
90             6'b000010: r[rt] <= r[rs] | {16'b0, immediate[15:0]}; // r[rt] = r[rs] | ZE(immm): ori (0x02)
91             6'b000011: r[rt] <= r[rs] + {{16{immediate[15]}}, immediate[15:0]}; // r[rt] = r[rs] + SE(immm): addi (0x03)
92             6'b000100: r[rt] <= r[rs] - {{16{immediate[15]}}, immediate[15:0]}; // r[rt] = r[rs] - SE(immm): subi (0x04)
93             6'b000111: begin
94                 if (r[rs] == r[rt]) pc <= pc + 4 + {{14{immediate[15]}}, immediate[15:0], 2'b00}; // beq (0x07)
95                 else pc <= pc + 4;
96             end

```

```

97     6'b001000: begin
98         if (r[rs] != r[rt]) pc <= pc + 4 + {{14{immediate[15]}}, immediate[15:0], 2'b00}; // bne (0x08)
99         else pc <= pc + 4;
100    end
101   6'b001001: r[rt] <= immediate << 16; // lui (0x09)
102   // ===== J-Type =====
103   6'b001010: pc <= {pc[31:28], inst[25:0] << 2}; // j (0x0A)
104   6'b001011: begin
105       r[31] <= pc + 4;
106       pc <= {pc[31:28], inst[25:0] << 2}; // jal (0x0B)
107   end
108   default: ;
109 endcase
110 // Get next instruction address for those not branch instructions
111 if (opcode != 6'b000111 && opcode != 6'b001000 && opcode != 6'b001010 && ! (opcode == 6'b000000 && funct == 6'b000111)) pc <= pc + 4;

```

Figure 5. SP.v: Decode & Compute Part Code

## 2.2.4 Delay Counter

最後是 Delay Counter 的部分，也是此程式中最重要的段落，主要延遲的 Clock Cycle 為 5 個，當 delay\_counter = 0 代表為某一指令 Decode 後的第一個 Cycle，此時我們首先將 R-Type 的 alu\_out 寫入目標 Register r[rd]內；接著考慮兩種情況：lw, sw。當指令為 lw 時，首先將 mem\_wen 設為 1 才能讀取 Memory 的資料，同時將目標資料的 Address 輸入，之所以要在這個時候輸入是因為我們需要預留兩個 Cycle 的 Memory 讀取時間，等待接下來的第二次 Clk 才能讀取到 Memory 輸出的資料；至於另一種情況 sw，則是將 mem\_wen 設為 0 才能寫入 Memory。而當進入下一個 Clk 時，我們先利用前面 Decode & ALU Compute 的部分所計算出的 pc 將結果更新到 inst\_addr 上，以準備提供下一個指令讀取使用。而等到 delay\_counter = 2 時，Memory 完成讀取 lw 的資料，因此將讀取出來的 mem\_dout 寫入 Register r[rt]內，當完成這一步時，此輪的指令處理完畢，所以將 out\_valid 拉起來，讓 PATTERN.v 能夠進行檢查，並接著在一個 Cycle 後拉低 out\_valid，避免讀取到錯誤的資料。

```

112     // Clock counter part
113     end else if (delay_counter != 7) begin
114         delay_counter <= delay_counter + 1;
115         if (delay_counter == 0) begin
116             // Load alu_out into mem_addr continuously
117             mem_addr <= alu_out;
118             // For R-Type
119             if (opcode == 6'b000000 && funct != 6'b000111) r[rd] <= alu_out;
120             // For lw (Read memory part)
121             if (opcode == 6'b000101) begin
122                 mem_wen <= 1;
123                 mem_addr <= r[rs] + {{16{immediate[15]}}, immediate[15:0]};
124                 // For sw
125             end else if (opcode == 6'b000110) begin
126                 mem_wen <= 0;
127                 mem_addr <= r[rs] + {{16{immediate[15]}}, immediate[15:0]};
128                 mem_din <= r[rt];
129             end

```

```
130      end else if (delay_counter == 1) begin
131          // Refresh program count
132          inst_addr <= pc;
133      end else if (delay_counter == 2) begin
134          // For lw (Write register part)
135          if (opcode == 6'b0000101) r[rt] <= mem_dout;
136      end else if (delay_counter == 3) begin
137          // Start output (Let PATTERN check)
138          out_valid <= 1;
139      end else if (delay_counter == 4) begin
140          // Stop output & refresh delay_counter for next cycle
141          out_valid <= 0;
142          delay_counter <= 0;
143      end
144  end
145
```

Figure 6. SP.v: Delay Counter Code

## 2.3 PATTERN.v Part

### 2.3.1 Main Function (initial)

PATTERN.v 由多個 task function 組成，並且由 initial function 去呼叫各個 task 以執行檢查的操作，下圖 Figure 7 為其程式碼，整體的運作機制就與前面 2.1.3 所列出的相對應。而助教已完成絕大多數的 task function，需要撰寫的僅有 check\_ans\_task。

```
48  initial begin
49      // read data mem & instruction
50      $readmemh("instruction.txt", instruction);
51      $readmemh("mem.txt", mem);
52
53      // initialize control signal
54      rst_n = 1'b1;
55      in_valid = 1'b0;
56      // initial variable
57      golden_inst_addr = 0;
58      for(i = 0; i < 32; i = i + 1)begin
59          golden_r[i] = 0;
60      end
61      // inst=X
62      inst = 32'bX;
63      // reset check task
64      reset_check_task;
65      // generate random idle clk
66      t = $random(seed) % 3 + 1'b1;
67      repeat(t) @(negedge clk);
68      // main pattern
69      for(pat = 0; pat < pat_num; pat = pat + 1)begin
70          input_task;
71          out_valid_wait_task;
72          check_ans_task;
73      end
74      $display("*****");
75      check_memory_task;
76      display_pass_task;
77  end
```

Figure 7. PATTERN.v: Initial Function Code

### 2.3.2 Check Answer Task

此部分的程式碼相比於 SP.v 來說容易許多，原因是我們不需要考慮 Memory Read / Write 所造成的額外延遲，可以像處理 Register 一樣直接用 mem[] 的方式來讀取、寫入 Memory，所以每條指令基本上都可以在一個 Edge Trigger 下完成，原則上這一部分就與前面 2.2.3 的部分幾乎相同，只是省略 lw, sw 需要另外用 delay\_counter 去處理的情況，並且 Register 改以 golden\_r 表示，而指令位址則是使用 golden\_inst\_addr 取代 SP.v 的 inst\_addr，下圖 Figure 8 為此部分的程式碼：

```

148 // check_ans_task
149 task check_ans_task; begin
150     // answer calculate
151     // Decode Instruction
152     opcode = instruction[golden_inst_addr>>2][31:26];
153     rs = instruction[golden_inst_addr>>2][25:21];
154     rt = instruction[golden_inst_addr>>2][20:16];
155     rd = instruction[golden_inst_addr>>2][15:11];
156     shamt = instruction[golden_inst_addr>>2][10:6];
157     func = instruction[golden_inst_addr>>2][5:0];
158     immediate = instruction[golden_inst_addr>>2][15:0];
159     address = instruction[golden_inst_addr>>2][25:0];
160     // Execute instruction
161     case (opcode)
162         // ===== R-Type =====
163         6'b000000: begin
164             case (func)
165                 6'b000000: golden_r[rd] = golden_r[rs] & golden_r[rt];           // and (0x00)
166                 6'b000001: golden_r[rd] = golden_r[rs] | golden_r[rt];           // or (0x01)
167                 6'b000010: golden_r[rd] = golden_r[rs] + golden_r[rt];          // add (0x02)
168                 6'b000011: golden_r[rd] = golden_r[rs] - golden_r[rt];          // sub (0x03)
169                 6'b000100: golden_r[rd] = (golden_r[rs] < golden_r[rt])? 1:0; // slt (0x04)
170                 6'b000101: golden_r[rd] = golden_r[rs] << shamt;            // sll (0x05)
171                 6'b000110: golden_r[rd] = ~golden_r[rs] | golden_r[rt];        // nor (0x06)
172                 6'b000111: golden_inst_addr = golden_r[rs];                  // jr (0x07)
173                 default: ;
174             endcase
175         end
176         // ===== I-Type =====
177         6'b000001: golden_r[rt] = golden_r[rs] & {16'b0, immediate[15:0]};           // andi (0x01)
178         6'b000010: golden_r[rt] = golden_r[rs] | {16'b0, immediate[15:0]};           // ori (0x02)
179         6'b000011: golden_r[rt] = golden_r[rs] + {{16{immediate[15]}}}, immediate[15:0]; // addi (0x03)
180         6'b000100: golden_r[rt] = golden_r[rs] - {{16{immediate[15]}}}, immediate[15:0]; // subi (0x04)
181         6'b000101: golden_r[rt] = mem[{{16{immediate[15]}}}, immediate[15:0]] + golden_r[rs]; // lw (0x05)
182         6'b000110: mem[golden_r[rs] + {{16{immediate[15]}}}, immediate[15:0]] = golden_r[rt]; // sw (0x06)
183         6'b000111: begin
184             if (golden_r[rs] == golden_r[rt]) golden_inst_addr = golden_inst_addr + 4 + {{14{immediate[15]}}}, immediate[15:0], 2'b00; // beq (0x07)
185             else golden_inst_addr = golden_inst_addr + 4;
186         end
187         6'b001000: begin
188             if (golden_r[rs] != golden_r[rt]) golden_inst_addr = golden_inst_addr + 4 + {{14{immediate[15]}}}, immediate[15:0], 2'b00; // bne (0x08)
189             else golden_inst_addr = golden_inst_addr + 4;
190         end
191         6'b001001: golden_r[rt] = immediate << 16;                                // lui (0x09)
192         // ===== J-Type =====
193         6'b001010: golden_inst_addr = {golden_inst_addr[31:28], address << 2}; // j (0x0A)
194         6'b001011: begin
195             golden_r[31] = golden_inst_addr + 4;
196             golden_inst_addr = {golden_inst_addr[31:28], address << 2};           // jal (0x0B)
197         end
198         default: ;
199     endcase
200     // Get next instruction address for those not branch instructions
201     if (opcode != 6'b000111 && opcode != 6'b001000 && opcode != 6'b001011 && opcode != 6'b001010 && !(opcode == 6'b000000 && func == 6'b000111)) begin
202         golden_inst_addr = golden_inst_addr + 4;
203     end

```

Figure 8. PATTERN.v: check\_ans\_task Function Code

## 2.4 Waveform & Compile Result

下面 Figure 9, 10 呈現的即為波形圖，首先在 `rst_n` 觸發後，幾個訊號被重置到特定值，接著開始產生 `Clk`，在每次的 `in_valid=1` 時讀取指令 `inst`，接著該指令執行完成後將 `out_valid` 拉起來，提供給 `PATTERN` 檢查，確定沒有問題後就繼續下一個指令，往復循環直到所有指令都完成。

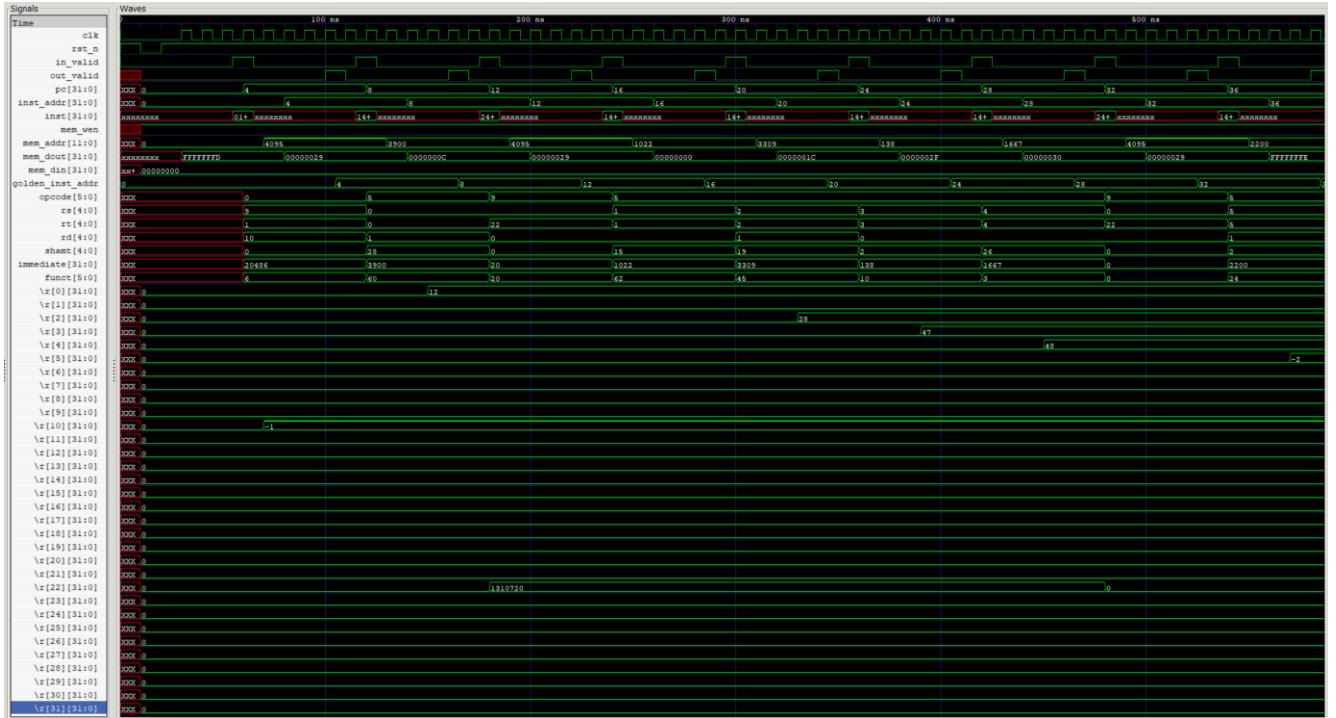


Figure 9. Waveform of Non-Pipeline MIPS Design (0-600ns)

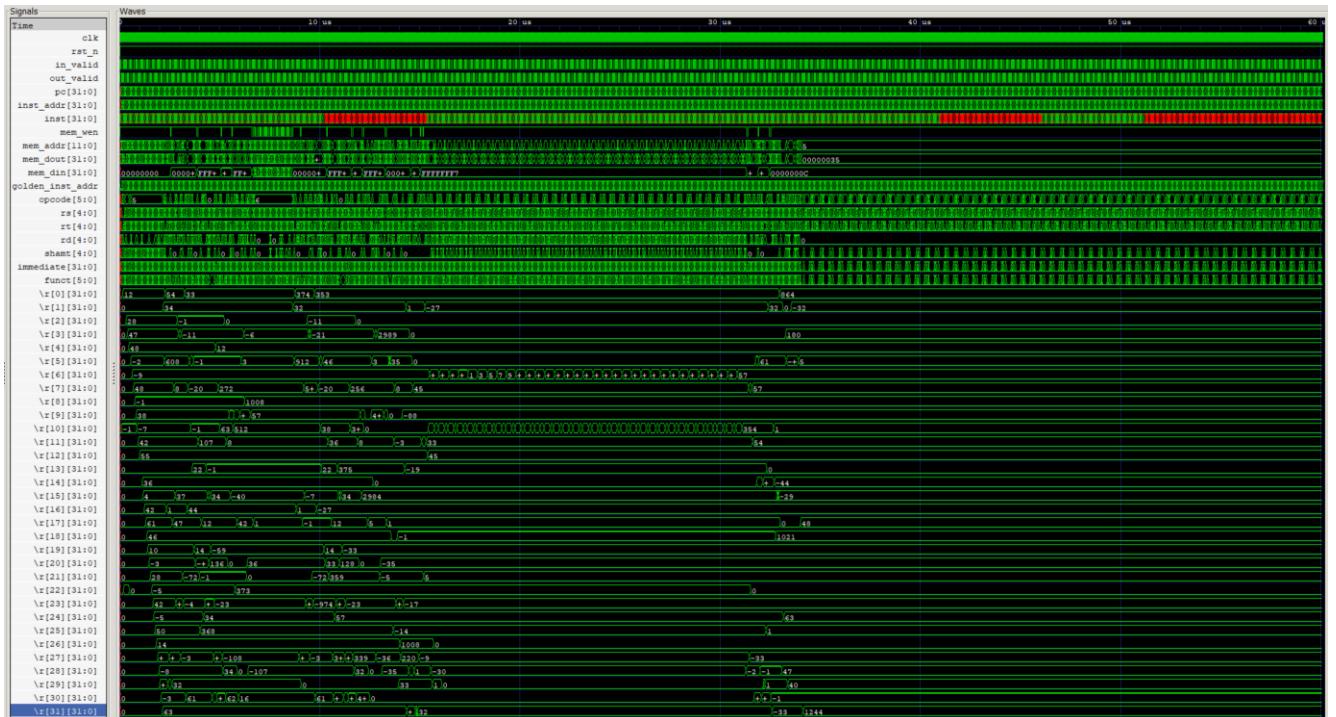


Figure 10. Waveform of Non-Pipeline MIPS Design (0-60us)

# Chapter 3 Pipeline MIPS CPU

## 3.1 Relationship Between SP\_pipeline.v & PATTERN\_p.v

### 3.1.1 Abstract

對於 Pipeline 的情況，因為需要將每個指令都拆解成 5 個階段：IF (Instruction Fetch), ID (Instruction Decode), EX (Execution), MEM (Memory Read / Write), WB (Write Back)，所以整體來說會比 Non-Pipeline 還複雜許多，整體的架構主要是寫出 5 個階段的 always block，接著每次將下一個階段需要使用到的資料暫存到中間的 Register 內，下一個 Clk 觸發時就可以直接讀取上個 Cycle 儲存到 Register 的數據，藉由這樣的方式模擬 Pipeline 的運行。

### 3.1.2 Architecture

同樣地，為了方便後續設計，先繪製了其簡略的架構圖，如下面 Figure 11 所示。主要使用的元件其實就與前面的 Non-Pipeline Design 非常相似，最大的不同在於中間多了四個暫存器 IF/ID Register, ID/EX Register, EX/MEM Register, MEM/WB Register，並且此架構與課本上提到的略有不同，原因是為了避免 Control Hazard，我們在 IF 階段如果碰到分支指令就要立即先計算出下一個指令的 PC，而不是等到 ID 階段才處理。此外，由於碰到分支指令時，是「下一個」Clock 才會將其目標 inst\_addr 載入，因此原先的分支指令在下一個 Clock 依然會往下傳遞進入到 ID 階段，為了避免這個指令在後面進行不必要的 Register / Memory 寫入，導致最後的結果錯誤，就可以透過在 ID 階段將 reg\_wen, mem\_wen 都設定為唯讀的 1，避免後續影響 PATTERN.v 的檢查結果。另外，由於 lw 在輸入 Addr 跟 Memory 讀出之間需要至少間隔一個 Cycle，所以如果放在 MEM 階段處理，WB 階段會來不及讀取到 Memory 的資料，故我們將原本的 Memory 處理從 MEM 往前移動到 EX 階段。至於 sw 指令雖然只要一個 Cycle 就可以完成寫入，但因為 Data Memory 需要靠 mem\_wen 來控制其為唯讀/寫，如果將 lw, sw 分別寫在兩個不同的 always block，則同一個 Cycle 可能就會出現兩個 block 同時控制一個相同的變數，這會發生未定義行為，也就是我們無法確保最後 mem\_wen 是 1 還是 0，因此同樣將 sw 往前移動到 EX 階段進行處理，最後的結構就如下圖所呈現的。

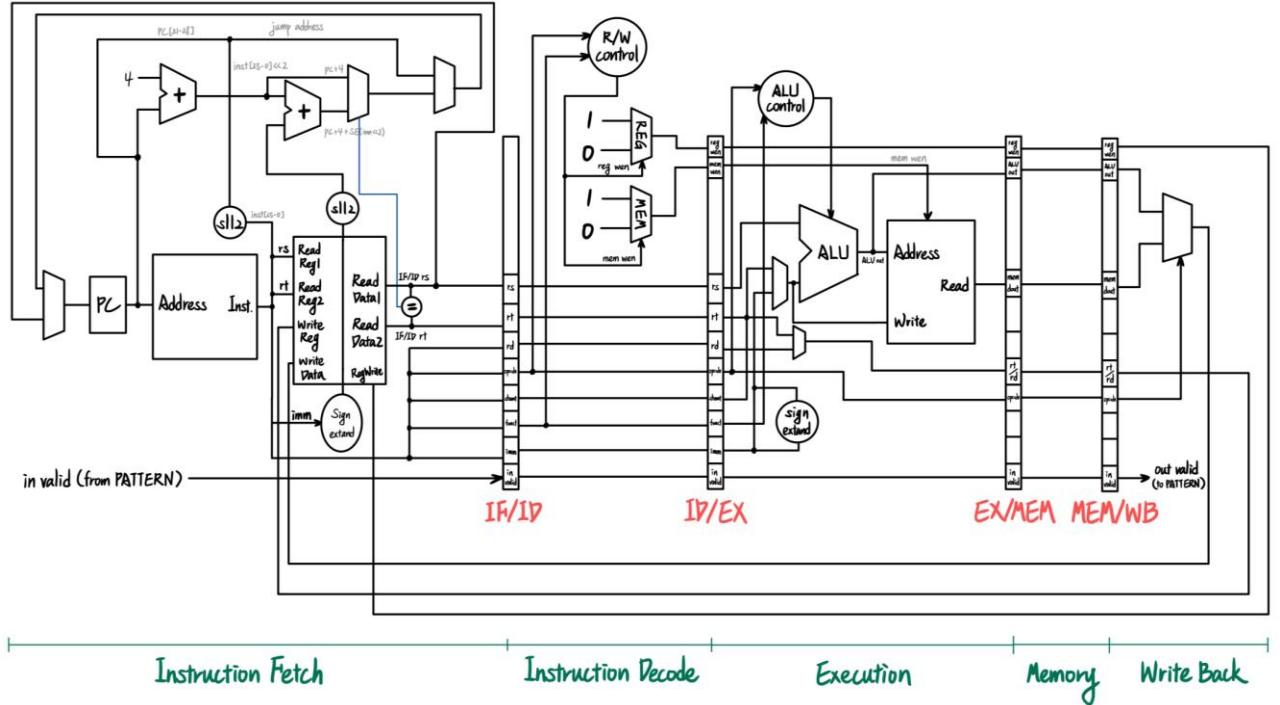


Figure 11. Non-Pipeline MIPS CPU Architecture

### 3.1.3 Workflow

整體來說，SP\_pipeline.v 與 PATTERN\_p.v 的執行關係大致如下述：

1. PATTERN\_p.v 先從給定的指令檔”instruction.txt”、記憶體檔”mem.txt”讀取資料。
2. 啟動 Reset & in\_valid 訊號將 SP\_pipeline.v 的 CPU 各暫存器與參數重置。
3. 初始化 PATTERN\_p.v 所有的暫存器與 golden\_inst\_addr\_in / out, in / out\_valid\_counter 等。
4. PATTERN\_p.v 檢查 SP\_pipeline.v 的 Reset 是否正確處理完畢。
5. PATTERN\_p.v 開始產生週期性的 Clk 訊號給 SP\_pipeline.v。
6. PATTERN\_p.v 每次會先檢查當前的 inst\_addr 是否與 SP\_pipeline.v 相同，接著讀取該 address 的指令並將 in\_valid 保持為 1，使 SP\_pipeline.v 能夠讀取該指令，同時 PATTERN\_p.v 會在這個 Cycle 先處理完分支指令的情況。
7. SP\_pipeline.v 在下一次的 Positive Edge 觸發時開始執行該指令的 IF 階段。
8. SP\_pipeline.v 的每個指令需要等待 5 個 Cycle 後才會完成到 WB 階段，此時 PATTERN\_p.v 才可以開始檢查當前所有的 Register 存的數據是否正確。
9. PATTERN\_p.v 在接收到一條新的指令後，會像 SP\_pipeline.v 一樣等待 5 個 Cycle，在要檢查 SP\_pipeline.v 的同時，在該 Clock 內像 Non-Pipeline 立即執行該指令後接著檢查校對。
10. 全部的指令都執行完成後，統一檢查 Memory 上 4096 個 Address 的資料是否也相同。

## 3.2 SP\_pipeline.v Part

### 3.2.1 Register & I/O Declaration

首先同樣是 Register 以及 SP\_pipeline module 的 I/O 部分宣告，除了基本常見的幾個必要 Register 之外，對於 Pipeline 每個階段的交界處都需要額外的 Register，主要取決於後面幾個階段會使用到的訊號，像是 EX\_MEM\_alu\_out 代表這個 Register 會用來將當前指令 EX 階段計算出的 alu\_out 儲存到 EX\_MEM\_alu\_out，在下一個 Cycle 內，該指令進行到下一個階段 MEM 就可以直接取用 EX\_MEM\_alu\_out 的資料進行計算，其它部分依此類推，其中我們也只保留後面幾個階段會使用到的訊號，其餘用不到的就不會再往下傳送。

```
1  module SP_pipeline(
2    // INPUT SIGNAL
3    clk,
4    rst_n,
5    in_valid,
6    inst,
7    mem_dout,
8    // OUTPUT SIGNAL
9    out_valid,
10   inst_addr,
11   mem_wen,
12   mem_addr,
13   mem_din
14 );
15
16 //-----
17 //  INPUT AND OUTPUT DECLARATION
18 //-----
19
20 input          clk, rst_n, in_valid;
21 input signed [31:0] inst;
22 input signed [31:0] mem_dout; // 32-bit signal from memory
23 output reg      out_valid; // Let outputs valid
24 output reg      [31:0] inst_addr; // 32-bit address of current instruction
25 output reg      mem_wen; // Memory write enable
26 output reg      [11:0] mem_addr; // 12-bit Memory address
27 output reg signed [31:0] mem_din; // 32-bit signal to write into memory
28
29 //-----
30 //  DECLARATION
31 //-----
32
33 // REGISTER FILE, DO NOT EDIT THE NAME.
34 reg signed [31:0] r [0:31]; // 32 Register file, each 32-bit
35 reg signed [31:0] jalpc; // 32-bit jal pc register
36 integer i; // for loop index
37
38 // IF-ID Registers
39 reg [31:0] IF_ID_immediate;
40 reg [5:0] IF_ID_opcode, IF_ID FUNCT;
41 reg [4:0] IF_ID_rs, IF_ID_rt, IF_ID_rd, IF_ID_shamt;
42 reg      IF_ID_in_valid;
43 // ID-EX Registers
44 reg [31:0] ID_EX_immediate;
45 reg [5:0] ID_EX_opcode, ID_EX_FUNCT;
46 reg [4:0] ID_EX_rs, ID_EX_rt, ID_EX_rd, ID_EX_shamt;
47 reg      ID_EX_reg_wen, ID_EX_mem_wen, ID_EX_in_valid;
48 // EX-MEM Registers
49 reg signed [31:0] EX_MEM_alu_out;
50 reg [5:0] EX_MEM_opcode;
51 reg [4:0] EX_MEM_rt_rd;
52 reg      EX_MEM_reg_wen, EX_MEM_mem_wen, EX_MEM_in_valid;
53 // MEM-WB Registers
54 reg signed [31:0] MEM_WB_alu_out;
55 reg [5:0] MEM_WB_opcode;
56 reg [4:0] MEM_WB_rt_rd;
57 reg      MEM_WB_reg_wen, MEM_WB_in_valid;
```

Figure 12. SP\_pipeline.v: Register & I/O Declaration Code

### 3.2.2 IF Stage (Instruction Fetch)

首先是 Instruction Fetch 的部分，對於 Reset 的階段就與前面的幾個 Code 差不多，將特定的訊號歸零或拉到 1，因此接下來我們只會分析 in\_valid 觸發的情況。下面 Figure 13 為其程式碼，就如前面 Figure 11 呈現的一樣，此階段不只單純會進行 Instruction Fetch，同時還會處理分支指令 jr, beq, bne, j, jal，因此在這個階段就會先將 Instruction Decode 出 opcode, rs, rt, funct 等，這些數據的主要用途是判斷並立即計算分支指令的目標位址，立即更新下一個 inst\_addr，接著會直接將 rs, rt, rd, opcode 等寫入第一個 Register: IF/ID。另外，我另外使用了 IF\_ID\_in\_valid 來控制這個階段的執行，當 PATTERN\_p.v 將 in\_valid 設為 1 後，IF 階段在執行時就會同步將 IF\_ID\_in\_valid 也從 0 設為 1，目的是告訴下一階段 ID 可以開始執行了。而在最後 PATTERN\_p.v 讀取的 1000 次指令結束後，就會將 in\_valid 設為 0，此時 IF 也會接著將 IF\_ID\_in\_valid 設回 0，也就是逐個階段的開始&逐個階段的結束。

```
82      // If in_valid == 1 (from PATTERN), start decoding branch instructions (jr, beq, bne, j, jal)
83  end else if (in_valid) begin
84      IF_ID_in_valid <= 1;           // Let next stage ID valid
85      IF_ID_opcode <= inst[31:26];
86      IF_ID_rs <= inst[25:21];
87      IF_ID_rt <= inst[20:16];
88      IF_ID_rv <= inst[15:11];
89      IF_ID_shamt <= inst[10:6];
90      IF_ID_funct <= inst[5:0];
91      IF_ID_immediate <= inst[15:0];
92      case (inst[31:26])
93          6'b000000: if (inst[5:0] == 6'b000011) inst_addr <= r[inst[25:21]];                                // jr  (0x00)
94          6'b000111: begin
95              if (r[inst[25:21]] == r[inst[20:16]]) inst_addr <= inst_addr + 4 + {14{inst[15]}}, inst[15:0], 2'b00; // beq (0x07)
96              else inst_addr <= inst_addr + 4;
97          end
98          6'b001000: begin
99              if (r[inst[25:21]] != r[inst[20:16]]) inst_addr <= inst_addr + 4 + {14{inst[15]}}, inst[15:0], 2'b00; // bne (0x08)
100             else inst_addr <= inst_addr + 4;
101         end
102         6'b001010: inst_addr <= {inst_addr[31:28], inst[25:0] << 2};                                // j   (0x0A)
103         6'b001011: begin
104             jalpc <= inst_addr + 4;                                // jal (0x0B)
105             inst_addr <= {inst_addr[31:28], inst[25:0] << 2};
106         end
107         default: ;
108     endcase
109     // Get next instruction address for those not branch instructions
110     if (inst[31:26] != 6'b000011 && inst[31:26] != 6'b001000 && inst[31:26] != 6'b001010 && inst[31:26] != 6'b001011 && !(inst[31:26] == 6'b000000 && inst[5:0] == 6'b000111)) begin
111         inst_addr <= inst_addr + 4;
112     end
113 end else begin
114     // End instruction (End IF)
115     IF_ID_in_valid <= 0;
116 end
```

Figure 13. SP\_pipeline.v: IF Stage Code

### 3.2.3 ID Stage (Instruction Decode)

接著是 Instruction Decode 的部分，由於 IF 階段已經將 inst 拆解完成了，因此直接將上一個階段 Decode 得到的 IF\_ID\_opcode / rs / rt / rd / shamt / funct / imm 都直接往下一個 Register: ID/EX 寫入即可。至於此階段主要的操作是決定 Data Register 與 Data Memory 的讀/寫控制訊號 reg\_wen, mem\_wen，以下共有四組情況：

1. reg\_wen = 1, mem\_wen = 1: 當指令為 Branch 時將兩者都設為 Read 避免寫入數據導致錯誤
2. reg\_wen = 0, mem\_wen = 1: 當指令為 lw 時將 reg 設為 Write、mem 設為 Read

3. reg\_wen = 1, mem\_wen = 0: 當指令為 sw 時將 reg 設為 Read、mem 設為 Write

4. reg\_wen = 0, mem\_wen = 1: 剩餘的指令則都是要寫入 Register 的，因此與 lw 設定相同

最後，就與前面 IF 階段相似的，同樣使用 ID\_EX\_in\_valid 來控制下一個階段的開始與否。

```
133 // If IF_ID_in_valid == 1 (from IF), start decoding instructions
134 end else if (IF_ID_in_valid) begin
135   ID_EX_in_valid <= 1;           // Let next stage EX valid
136   ID_EX_opcode <= IF_ID_opcode;
137   ID_EX_rs <= IF_ID_rs;
138   ID_EX_rt <= IF_ID_rt;
139   ID_EX_rd <= IF_ID_rd;
140   ID_EX_shamt <= IF_ID_shamt;
141   ID_EX_funct <= IF_ID_funct;
142   ID_EX_immediate <= IF_ID_immediate;
143   if ((IF_ID_opcode == 6'b000000 && IF_ID_funct == 6'b000111) || IF_ID_opcode == 6'b000111 || IF_ID_opcode == 6'b001000 || IF_ID_opcode == 6'b001010 || IF_ID_opcode == 6'b001011) begin
144     ID_EX_req_wen <= 1;        // Register: Read
145     ID_EX_mem_wen <= 1;        // Memory: Read
146   end else if (IF_ID_opcode == 6'b000101) begin // lw
147     ID_EX_req_wen <= 0;        // Register: Write
148     ID_EX_mem_wen <= 1;        // Memory: Read
149   end else if (IF_ID_opcode == 6'b000110) begin // sw
150     ID_EX_req_wen <= 1;        // Register: Read
151     ID_EX_mem_wen <= 0;        // Memory: Write
152   end else begin
153     ID_EX_req_wen <= 0;        // Register: Write
154     ID_EX_mem_wen <= 1;        // Memory: Read
155   end
156 end else begin
157   // End ID
158   ID_EX_in_valid <= 0;
159 end
```

Figure 14. SP\_pipeline.v: ID Stage Code

### 3.2.4 EX Stage (Execution)

再來是 ALU Execution 的部分，同時也包含了 lw & sw 的 Memory Read / Write，其中 ALU 的計算結果一律儲存到下一個 Register: EX/MEM 的 EX\_MEM\_alu\_out，並且在碰到 lw 時，將 mem\_wen 設為 1(sw 則設為 0)，且寫入 EX\_MEM\_alu\_out 的數據同時也寫入 mem\_addr，用來指定 Memory 的目標位址，而 lw 指令接著在進行到 WB 階段時就能夠取得該數據。

```
172 // If ID_EX_in_valid == 1 (from ID), start computing with ALU & Read / Write Memory
173 end else if (ID_EX_in_valid) begin
174   EX_MEM_in_valid <= 1;           // Let next stage MEM valid
175   EX_MEM_mem_wen <= ID_EX_mem_wen;
176   EX_MEM_reg_wen <= ID_EX_req_wen;
177   EX_MEM_opcode <= ID_EX_opcode;
178   case (ID_EX_opcode)
179     // ===== R-Type =====
180     6'b000000: begin
181       case (ID_EX_funct)
182         6'b000000: EX_MEM_alu_out <= r[ID_EX_rs] & r[ID_EX_rt];           // and (0x00)
183         6'b000001: EX_MEM_alu_out <= r[ID_EX_rs] | r[ID_EX_rt];           // or (0x01)
184         6'b000010: EX_MEM_alu_out <= r[ID_EX_rs] + r[ID_EX_rt];           // add (0x02)
185         6'b000011: EX_MEM_alu_out <= r[ID_EX_rs] - r[ID_EX_rt];           // sub (0x03)
186         6'b000100: EX_MEM_alu_out <= (r[ID_EX_rs] < r[ID_EX_rt])? 1:0;    // slt (0x04)
187         6'b000101: EX_MEM_alu_out <= r[ID_EX_rs] << ID_EX_shamt;          // sll (0x05)
188         6'b000110: EX_MEM_alu_out <= ~r[ID_EX_rs] | r[ID_EX_rt];          // nor (0x06)
189         default: ;
190       endcase
191       EX_MEM_rt_rd <= ID_EX_rd; // Pass rd as the destination register for the WB stage
192     end
193     // ===== I-Type =====
194     6'b000001: EX_MEM_alu_out <= r[ID_EX_rs] & {16'b0, ID_EX_immediate[15:0]};           // andi (0x01)
195     6'b000010: EX_MEM_alu_out <= r[ID_EX_rs] | {16'b0, ID_EX_immediate[15:0]};           // ori (0x02)
196     6'b000011: EX_MEM_alu_out <= r[ID_EX_rs] + {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0]; // addi (0x03)
197     6'b000100: EX_MEM_alu_out <= r[ID_EX_rs] - {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0]; // subi (0x04)
198     6'b000101: begin
199       EX_MEM_alu_out <= r[ID_EX_rs] + {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0];           // lw (0x05)
200       mem_wen <= 1;
201       mem_addr <= r[ID_EX_rs] + {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0];
202     end
203     6'b000110: begin
204       EX_MEM_alu_out <= r[ID_EX_rs] + {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0];           // sw (0x06)
205       mem_wen <= 0;
206       mem_addr <= r[ID_EX_rs] + {16{ID_EX_immediate[15]}}, ID_EX_immediate[15:0];
207       mem_din <= r[ID_EX_rt];
208     end
209     6'b001001: EX_MEM_alu_out <= ID_EX_immediate << 16;                                // lui (0x09)
210   endcase
```

```

211      // Pass rt as the destination register for the WB stage (Except for R-Type Instructions)
212      if (ID_EX_opcode != 6'b000000) EX_MEM_rt_rd <= ID_EX_rt;
213  end else begin
214      // End EX
215      EX_MEM_in_valid <= 0;
216  end
217 end

```

Figure 15. SP\_pipeline.v: EX Stage Code

### 3.2.5 MEM Stage (Memory Read / Write)

對於 Memory 的部分，由於前面 EX 階段已經處理完畢了，因此這個階段就只要將來自 EX 階段的數據( e.g. alu\_out, rt\_rd, reg\_wen, opcode )繼續往下一階段的暫存器 Register: MEM/WB 傳送即可。其中比較特別的是 MEM\_WB\_rt\_rd 這個暫存器，它儲存的是接下來在 WB 階段要將數據寫入 Data Register 時，用來指定目標暫存器的位址，也就是 rd (R-Type), rt (I-Type) 的其中一個，同時一樣有 MEM\_WB\_in\_valid 用來控制下一階段的執行。

```

219  // Stage 4: MEM (Memory Read / Write)
220  always @(posedge clk or negedge rst_n) begin
221      // Reset signal
222      if (!rst_n) begin
223          MEM_WB_alu_out <= 0;
224          MEM_WB_rt_rd <= 0;
225          MEM_WB_reg_wen <= 1; // Only read
226          MEM_WB_opcode <= 0;
227          // If EX_MEMORY_in_valid == 1 (from EX), just pass the data that WB stage need
228      end else if (EX_MEMORY_in_valid) begin
229          MEM_WB_in_valid <= 1;           // Let next stage WB valid
230          MEM_WB_reg_wen <= EX_MEMORY_reg_wen;
231          MEM_WB_alu_out <= EX_MEMORY_alu_out;
232          MEM_WB_rt_rd <= EX_MEMORY_rt_rd;
233          MEM_WB_opcode <= EX_MEMORY_opcode;
234      end else begin
235          // End MEM
236          MEM_WB_in_valid <= 0;
237      end
238  end

```

Figure 16. SP\_pipeline.v: MEM Stage Code

### 3.2.6 WB Stage (Write Back)

對於 Write Back Stage 主要就只有三種情況要處理：

1. lw 指令：對於 lw 指令，其目標暫存器直接使用 MEM/WB 暫存器內的 rt\_rd 即可，至於要寫入的資料則來自於 Data Memory 所讀取出來的結果 mem\_dout。
2. jal 指令：jal 之所以要寫入，是因為它需要將 PC + 4 存到 r[31] 內，為了避免該暫存器出現 WAW 或 RAW 的 Data Hazard，就統一將這個暫存器的寫入操作保留到 WB 階段再來一起執行，由 opcode 來判斷。
3. Others：對於剩下的其他指令，寫入的數據源一律都是來自於 alu\_out，目標暫存器則同樣直接使用前面的 rt\_rd，接著寫入數據即可完成。

最後 WB 的階段，會將 out\_valid 設為 1，此時 PATTERN\_p.v 就會開始進行檢查，一直到最後所有的 1000 筆指令都執行完畢，接收到由前一階段 MEM\_WB\_in\_valid 為 0 的訊號時，意味著所有程式已執行完畢，故直接把 out\_valid 降回 0，結束 PATTERN\_p.v 的檢查。

```
240 // Stage 5: WB (Write Back)
241 always @(posedge clk or negedge rst_n) begin
242     if (!rst_n) begin
243         // Do not need to restart any signal
244         // If MEM_WB_in_valid == 1 (from MEM), write data into the destination register
245     end else if (MEM_WB_in_valid) begin
246         out_valid <= 1;                                // Let output valid (Let PATTERN check)
247         if (MEM_WB_reg_wen == 0) begin                // Register: Write
248             if (MEM_WB_opcode == 6'b000101) r[MEM_WB_rt_rd] <= mem_dout;    // For lw
249             else r[MEM_WB_rt_rd] <= MEM_WB_alu_out;           // For other instructions
250         end
251         if (MEM_WB_opcode == 6'b001011) r[31] <= jalpc;          // For jal
252     end else begin
253         // End WB
254         out_valid <= 0;
255     end
256 end
257
258 endmodule
```

Figure 17. SP\_pipeline.v: WB Stage Code

### 3.3 PATTERN\_p.v Part

#### 3.3.1 Main Function (initial)

首先對於 Pipeline 的 PATTERN\_p.v，整體的運作機制其實就與 Non-Pipeline 非常相似，最主要的差別在於 main pattern 的部分。在 Non-Pipeline 中，每一輪都是先由 PATTERN.v 產生一個 inst\_addr，接著傳送給 SP.v 完成該指令的執行後，PATTERN.v 會直接先檢查 32 個 Registers，並檢查當前的 inst\_addr 是否正確，反覆執行直到所有指令都跑完後，就會開始檢查一輪的 Memory 資料是否正確。而對於 Pipeline 的情況，PATTERN\_p.v 會先檢查當前的 inst\_addr 是否與 SP\_pipeline.v 的相同，再來才會檢查 32 個 Registers，最後所有指令執行完畢一樣檢查 Memory，其餘的部分基本上都是相同的。

```
52   initial begin
53     // read data mem & instruction
54     $readmemh("instruction.txt", instruction);
55     $readmemh("mem.txt", mem);
56
57     // initialize control signal
58     rst_n = 1'b1;
59     in_valid = 1'b0;
60
61     // initial variable
62     golden_inst_addr_in = 0;
63     golden_inst_addr_out = 0;
64     //*****
65     stage1_inst_addr = 4;
66     //*****
67     in_valid_counter = 0;
68     out_valid_counter = 0;
69     latency = -1;
70     for(i = 0; i < 32; i = i + 1)begin
71       golden_r[i] = 0;
72     end
73
74     // inst=X
75     inst = 32'bX;
76
77     // reset check task
78     reset_check_task;
79
80     // generate random idle clk
81     t = $random(seed) % 3 + 1'b1;
82     repeat(t) @(negedge clk);
83
84     // main pattern
85     while(out_valid_counter < execution_num)begin
86       input_task;
87       check_ans_task;
88       @(negedge clk);
89     end
90
91     // check out_valid
92     check_memory_and_out_valid;
93     display_pass_task;
94   end
```

Figure 18. PATTERN\_p.v: Initial Function Code

### 3.3.2 Input Task

相較於 Non-Pipeline，Pipeline 不能單純讀取一個指令後，等待 SP 的 out\_valid 拉起再直接計算結果，原因是依照題目規定，Pipeline 需要保持 out\_valid 持續拉起直到最後所有指令執行完畢，因此我們無法用該訊號來判斷何時 PATTERN\_p.v 要執行該指令並檢查結果，故此處我使用的方法是仿照 SP\_pipeline.v 的設計，同樣加上多個階段的 Cycle 延遲，因此 PATTERN\_p.v 在剛讀取到一個新指令的時候，並不會在當下這個 Clk 就完成該指令的執行並檢查結果，而是會往下級傳送，而當前位於第五級的指令就會在當下這個 Clk 執行，順序是 Posedge 時 SP\_pipeline.v 該指令的 WB 階段執行完畢，接著下一個 Negedge 時 PATTERN\_p.v 直接在該 Clk 內完成相同的一個指令執行，並立即檢查 Register 結果，往復循環直到最後結束。

但這樣的設計會出現一個 Data Hazard 的問題，就是當 SP\_pipeline.v 的 WB 階段指令正在寫入資料到某一個 Register 內，下一個 Cycle 的新指令（會進入 IF 階段）恰好是涉及 Register 資料的分支指令（e.g. jr, beq, bne），若它所使用到的 Register 又剛好與前一個 Cycle 的 WB 目標 Register 相同，由於分支指令會在 IF 階段就執行，雖然對於 SP\_pipeline.v 指令來說確實不會有問題發生，但對於 PATTERN\_p.v 的檢查就會出問題。PATTERN\_p.v 在讀取到新指令並傳送給 SP\_pipeline.v 之前會「先」處理分支指令的部分，然而在這個時候位於第五級的 WB 指令尚未執行，因此它取到的 Register 值就會是舊的，需要等待當前這個 Cycle 結束後才會更新完成。為了解決這個問題，我的方法是在 PATTERN\_p.v 的 input\_task 階段就直接先檢查第五級的指令，如果發現該指令會寫入的目標 Register 恰好與當前新的分支指令使用到的相同，則直接先計算出其值提供給分支指令使用，而「不會」實際寫入，真正的寫入操作依然會等到該 Cycle 的最後面要檢查 Registers 前才實際寫入，避免影響到中途的其他指令。以下的 input\_task 大致上分成兩個部分，第一個部分是檢查第五級的指令目標寫入 Register 是否與當前的分支指令相同，第二個部分才是實際的分支指令執行。另外，其中的 stage\_valid 是為了確保第五級當前確實是有指令要執行的，否則讀取到錯誤的訊號可能導致未定義行為發生。下面的 Figure 19 為第一部分的程式碼、Figure 20 則是第二部分的程式碼。

```

159 // Pre-decoding for those branch instructions
160 opcode = instruction[golden_inst_addr_in>>2][31:26];
161 rs = instruction[golden_inst_addr_in>>2][25:21];
162 rt = instruction[golden_inst_addr_in>>2][20:16];
163 rd = instruction[golden_inst_addr_in>>2][15:11];
164 shamt = instruction[golden_inst_addr_in>>2][10:6];
165 func = instruction[golden_inst_addr_in>>2][5:0];
166 immediate = instruction[golden_inst_addr_in>>2][15:0];
167 address = instruction[golden_inst_addr_in>>2][25:0];
168 // Save current instruction address temporarily
169 stage1_inst_addr = golden_inst_addr_in;
170 // rs, rt data for branch instruction address calculation
171 new_rs_data = golden_r[rs];
172 new_rt_data = golden_r[rt];
173 // For those rs, rt register should write before branch instruction calculation
174 // (Which just calculate for pc, the actual write will start later)
175 if (stage_valid) begin
176     n_opcode = instruction[stage_inst_addr>>2][31:26];
177     n_rs = instruction[stage_inst_addr>>2][25:21];
178     n_rt = instruction[stage_inst_addr>>2][20:16];
179     n_rd = instruction[stage_inst_addr>>2][15:11];
180     n_shamt = instruction[stage_inst_addr>>2][10:6];
181     n_func = instruction[stage_inst_addr>>2][5:0];
182     n_imm = instruction[stage_inst_addr>>2][15:0];
183     case (n_opcode)
184         // ===== R-Type =====
185         6'b000000: begin // R-type instructions
186             case (n_func)
187                 6'b000000: begin
188                     if (n_rd == rs) new_rs_data = golden_r[n_rs] & golden_r[n_rt]; // and (0x00)
189                     else if (n_rd == rt) new_rt_data = golden_r[n_rs] & golden_r[n_rt];
190                 end
191                 6'b000001: begin
192                     if (n_rd == rs) new_rs_data = golden_r[n_rs] | golden_r[n_rt]; // or (0x01)
193                     else if (n_rd == rt) new_rt_data = golden_r[n_rs] | golden_r[n_rt];
194                 end
195                 6'b000010: begin
196                     if (n_rd == rs) new_rs_data = golden_r[n_rs] + golden_r[n_rt]; // add (0x02)
197                     else if (n_rd == rt) new_rt_data = golden_r[n_rs] + golden_r[n_rt];
198                 end
199                 6'b000011: begin
200                     if (n_rd == rs) new_rs_data = golden_r[n_rs] - golden_r[n_rt]; // sub (0x03)
201                     else if (n_rd == rt) new_rt_data = golden_r[n_rs] - golden_r[n_rt];
202                 end
203                 6'b000100: begin
204                     if (n_rd == rs) new_rs_data = (golden_r[n_rs] < golden_r[n_rt])? 1:0; // slt (0x04)
205                     else if (n_rd == rt) new_rt_data = (golden_r[n_rs] < golden_r[n_rt])? 1:0;
206                 end
207                 6'b000101: begin
208                     if (n_rd == rs) new_rs_data = golden_r[n_rs] << shamt; // sll (0x05)
209                     else if (n_rd == rt) new_rt_data = golden_r[n_rs] << shamt;
210                 end
211                 6'b000110: begin
212                     if (n_rd == rs) new_rs_data = ~golden_r[n_rs] | golden_r[n_rt]; // nor (0x06)
213                     else if (n_rd == rt) new_rt_data = ~golden_r[n_rs] | golden_r[n_rt];
214                 end
215                 default: ;
216             endcase
217         end
218         // ===== I-Type =====
219         6'b000001: begin
220             if (n_rt == rs) new_rs_data = golden_r[n_rs] & {16'b0, n_imm[15:0]}; // andi(0x01)
221             else if (n_rt == rt) new_rt_data = golden_r[n_rs] & {16'b0, n_imm[15:0]};
222         end
223         6'b000010: begin
224             if (n_rt == rs) new_rs_data = golden_r[n_rs] | {16'b0, n_imm[15:0]}; // ori (0x02)
225             else if (n_rt == rt) new_rt_data = golden_r[n_rs] | {16'b0, n_imm[15:0]};
226         end
227         6'b000011: begin
228             if (n_rt == rs) new_rs_data = golden_r[n_rs] + {{16{n_imm[15]}}, n_imm[15:0]}; // addi(0x03)
229             else if (n_rt == rt) new_rt_data = golden_r[n_rs] + {{16{n_imm[15]}}, n_imm[15:0]};
230         end
231         6'b000100: begin
232             if (n_rt == rs) new_rs_data = golden_r[n_rs] - {{16{n_imm[15]}}, n_imm[15:0]}; // subi(0x04)
233             else if (n_rt == rt) new_rt_data = golden_r[n_rs] - {{16{n_imm[15]}}, n_imm[15:0]};
234         end
235         6'b000101: begin
236             if (n_rt == rs) new_rs_data = mem[{{16{n_imm[15]}}, n_imm[15:0]} + golden_r[n_rs]]; // lw (0x05)
237             else if (n_rt == rt) new_rt_data = mem[{{16{n_imm[15]}}, n_imm[15:0]} + golden_r[n_rs]];
238         end

```

```

238         end
239         6'b001001: begin
240             if (n_rt == rs) new_rs_data = n_imm << 16;                                // lui (0x09)
241             else if (n_rt == rt) new_rt_data = n_imm << 16;
242         end
243         // ===== J-Type =====
244         6'b001011: begin
245             if (rs == 31) new_rs_data = jalpc;                                              // jal (0x0B)
246             else if (rt == 31) new_rt_data = jalpc;
247         end
248         default: ;
249     endcase
250 end

```

Figure 19. PATTERN\_p.v: Input Task Data Hazard Controller Code

```

251     // Pre-decoding for those branch instructions
252     case (opcode)
253         6'b000000: if (func == 6'b000111) golden_inst_addr_in = new_rs_data;
254         6'b000111: begin
255             if (new_rs_data == new_rt_data) golden_inst_addr_in = golden_inst_addr_in + 4 + {{14{immediate[15]}}, immediate[15:0], 2'b00}; // beq(0x07)
256             else golden_inst_addr_in = golden_inst_addr_in + 4;
257         end
258         6'b001000: begin
259             if (new_rs_data != new_rt_data) golden_inst_addr_in = golden_inst_addr_in + 4 + {{14{immediate[15]}}, immediate[15:0], 2'b00}; // bne(0x08)
260             else golden_inst_addr_in = golden_inst_addr_in + 4;
261         end
262         6'b001010: golden_inst_addr_in = {golden_inst_addr_in[31:28], address << 2};                                         // j (0x0A)
263         6'b001011: begin
264             jalpc = golden_inst_addr_in + 4;
265             golden_inst_addr_in = {golden_inst_addr_in[31:28], address << 2};                                         // jal(0x0B)
266         end
267         default: ;
268     endcase
269     // Get next instruction address for those not branch instructions
270     if (opcode != 6'b000111 && opcode != 6'b001000 && opcode != 6'b001010 && opcode != 6'b001011 && !(opcode == 6'b000000 && func == 6'b000111)) begin
271         golden_inst_addr_in = golden_inst_addr_in + 4;
272     end

```

Figure 20. PATTERN\_p.v: Input Task Branch Instruction Controller Code

### 3.3.3 Check Answer Task

PATTERN\_p.v 的最後這一部分就簡單許多了，基本上與 Non-Pipeline 的 PATTERN.v 相同，只不過省略了所有的分支指令（因為在前面的 input\_task 就執行完畢了），其中前半部分的 Pipeline waiting 就是用來等待 SP\_pipeline.v Cycle 用的，下方 Figure 21 為其程式碼，透過每個 Clk 將 Instruction 往下一級傳送，就可以與其運作機制相同。Figure 22 則是單純的各指令直接運算，就與 Non-Pipeline 相似：

```

287     // check_ans_task
288     task check_ans_task; begin
289         // Pipeline waiting
290         if (in_valid) begin
291             stage2_valid <= 1;
292             stage2_inst_addr <= stage1_inst_addr; // Passing instruction
293         end else stage2_valid <= 0;
294         if (stage2_valid) begin
295             stage3_valid <= 1;
296             stage3_inst_addr <= stage2_inst_addr; // Passing instruction
297         end else stage3_valid <= 0;
298         if (stage3_valid) begin
299             stage4_valid <= 1;
300             stage4_inst_addr <= stage3_inst_addr; // Passing instruction
301         end else stage4_valid <= 0;
302         if (stage4_valid) begin
303             stage5_valid <= 1;
304             stage5_inst_addr <= stage4_inst_addr; // Passing instruction
305         end else stage5_valid <= 0;
306         if (stage5_valid) begin
307             stage_valid <= 1;
308             stage_inst_addr <= stage5_inst_addr; // Passing instruction
309         end else stage_valid <= 0;

```

Figure 21. PATTERN\_p.v: check\_ans\_task Pipeline Waiting Part Code

```

311 // check out_valid
312 if(out_valid)begin
313     // answer calculate (actual write)
314     opcode = instruction[stage_inst_addr>>2][31:26];
315     rs = instruction[stage_inst_addr>>2][25:21];
316     rt = instruction[stage_inst_addr>>2][20:16];
317     rd = instruction[stage_inst_addr>>2][15:11];
318     shamt = instruction[stage_inst_addr>>2][10:6];
319     func = instruction[stage_inst_addr>>2][5:0];
320     immediate = instruction[stage_inst_addr>>2][15:0];
321     // R-type
322     // I-type
323     // PC & jump, beq...etc.
324     // hint: it's necessary to consider sign extention while calculating
325     case (opcode)
326         // ===== R-Type =====
327         6'b000000: begin
328             case (func)
329                 6'b000000: golden_r[rd] = golden_r[rs] & golden_r[rt];           // and (0x00)
330                 6'b000001: golden_r[rd] = golden_r[rs] | golden_r[rt];           // or  (0x01)
331                 6'b000010: golden_r[rd] = golden_r[rs] + golden_r[rt];          // add (0x02)
332                 6'b000011: golden_r[rd] = golden_r[rs] - golden_r[rt];          // sub (0x03)
333                 6'b000100: golden_r[rd] = (golden_r[rs] < golden_r[rt])? 1:0; // slt (0x04)
334                 6'b000101: golden_r[rd] = golden_r[rs] << shamt;              // sll (0x05)
335                 6'b000110: golden_r[rd] = ~(golden_r[rs] | golden_r[rt]);       // nor (0x06)
336                 default: ;
337             endcase
338         end
339         // ===== I-Type =====
340         6'b000001: golden_r[rt] = golden_r[rs] & {16'b0, immediate[15:0]}; // andi (0x01)
341         6'b000010: golden_r[rt] = golden_r[rs] | {16'b0, immediate[15:0]}; // ori  (0x02)
342         6'b000011: golden_r[rt] = golden_r[rs] + {{16{immediate[15]}}}, immediate[15:0]; // addi (0x03)
343         6'b000100: golden_r[rt] = golden_r[rs] - {{16{immediate[15]}}}, immediate[15:0]; // subi (0x04)
344         6'b000101: golden_r[rt] = mem[{{16{immediate[15]}}}, immediate[15:0]] + golden_r[rs]; // lw   (0x05)
345         6'b000110: mem[golden_r[rs] + {{16{immediate[15]}}}, immediate[15:0]] = golden_r[rt]; // sw   (0x06)
346         6'b001001: golden_r[rt] = immediate << 16;                           // lui  (0x09)
347         6'b001011: golden_r[31] = jalpc;                                         // jal  (0x0B)
348         default: ;
349     endcase

```

Figure 22. PATTERN\_p.v: check\_ans\_task Answer Calculate Part Code

### 3.4 Waveform & Compile Result

下方 Figure 23, 24 呈現的則是 Pipeline 形式的波形圖，同樣在 `rst_n` 觸發後，幾個訊號被重置到特定值，接著開始產生 `Clk`，且在 `in_valid` 拉起來後的第 5 個 Cycle，第一條指令的 WB 階段完成，故將 `out_valid` 拉起，直到最後才降下去，整體的運作機制就與我們預設的相同。

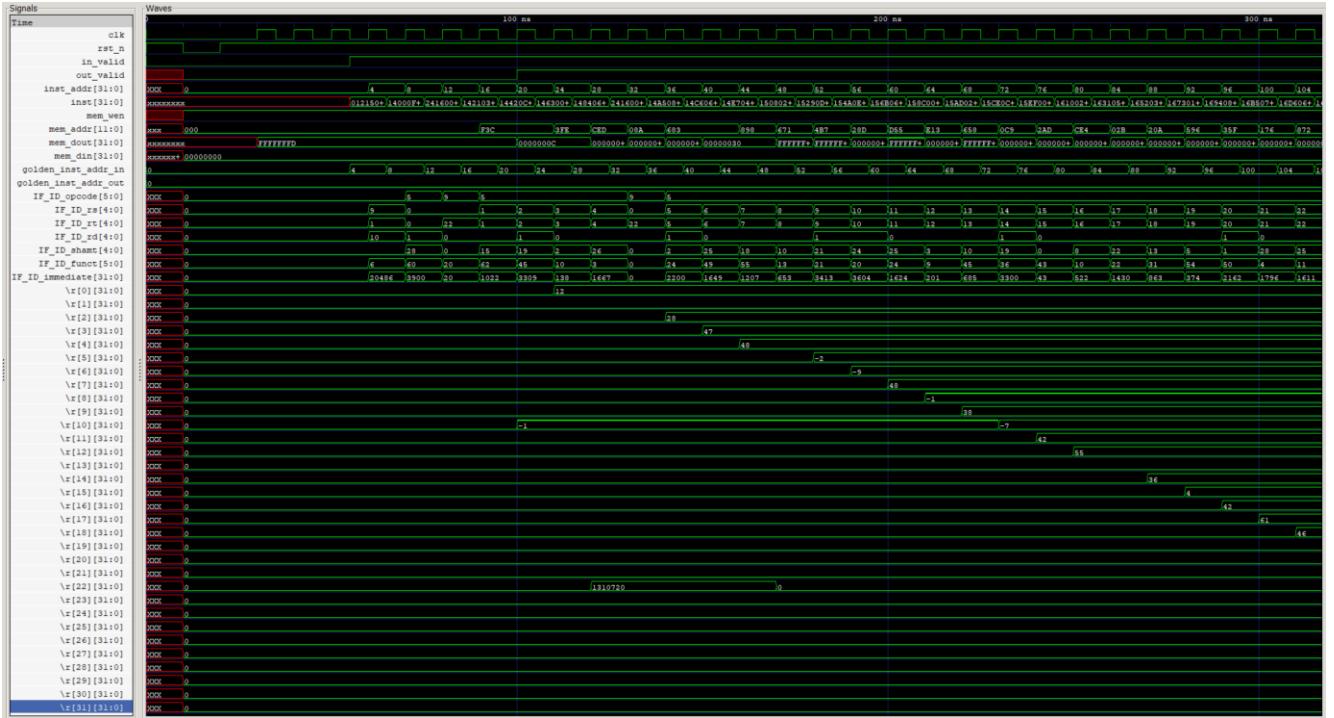


Figure 23. Waveform of Pipeline MIPS Design (0-300ns)

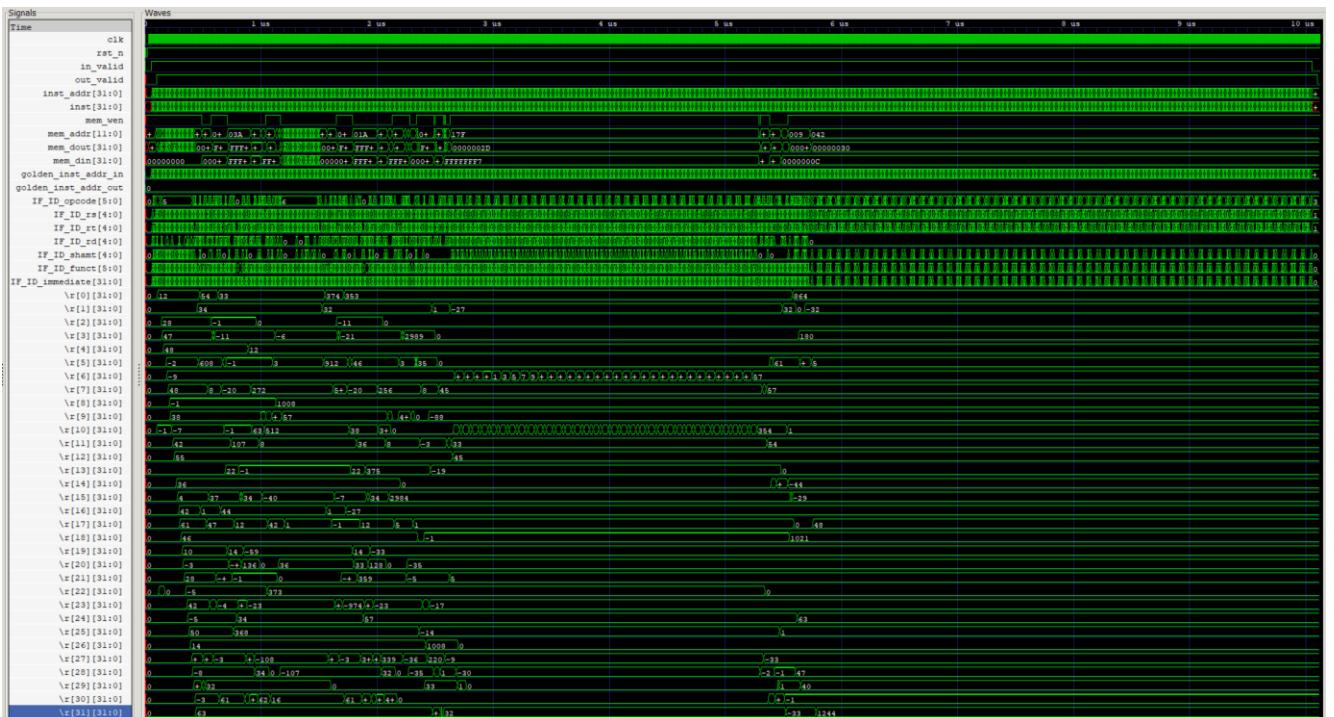


Figure 24. Waveform of Pipeline MIPS Design (0-10us)

## Chapter 4 Conclusion

### 4.1 Final Project Feedback

這次的期末專題是我第一次接觸 Verilog 這個程式語言，之前因為沒有修過數位電路與系統，所以整個專題的進行過程遇到了重重阻礙，起初甚至連「`<=`」跟「`=`」的差異都不知道。大致上來說，這次專題的進行方式就是由 ChatGPT 提供給我大致的架構做為參考，像是 Pipeline 分成五個 Stage 的 always block 設計就是由它所建議的，不過儘管它提供我大致的架構參考，每個 block 內的設計幾乎完全不可用，最終還是要我自己寫出來，而且最一開始我連波形要怎麼看都不知道，也不能理解 SP, PATTERN, TESTBED 彼此的關係，是隨著多次的 debug，不斷嘗試調整設計解決錯誤，最終才逐漸上手。光是 Non-Pipeline 的部分我就花了大約 1 個多禮拜的時間才完成，中間大約有 60-70% 的時間都是在摸索這套工具以及 Verilog，真的是我碰過最困難的期末專題，好險期末專題讓我們完成的時間夠長，否則如果提前一個禮拜絕對寫不出來。另外，雖然自己設計的 SP, PATTERN 通過了，但我比較擔心的還是助教那邊的測資與 SP, PATTERN，畢竟檢查結構是自己設計的，非常有可能其實兩個都是錯的，但因為結果相同導致最後顯示 Pass，只能希望最後的 Demo 可以順利通過。

而對於此次 Final Project 的 Feedback，希望老師可以在學期的課堂中多花一些時間、空幾堂課出來教 Verilog，畢竟當初選修計算機組織時，完全不知道期末專題是需要有 Verilog 底子的，在完全不熟悉這些工具的情況下要完成，真的太過困難了，或是說可以在學期間穿插 1、2 個比較簡單的 Verilog 作業讓我們練習等等。最後，不知道助教以後是否能夠提供一個標準的 PATTERN.v 檔讓我們能夠確保自己的設計是正確的，雖然其中的檢查結構會與 SP.v 非常像，但因為兩者還是有所差異，至少能夠讓我們有一個基本的架構參考，對於沒有 Verilog 底子的同學也比較友善，否則忙了半天或許最後的結果是錯的，15 分的學期成績幾乎都拿不到會更難受。

### 4.2 Course Feedback

這學期正式縮減到十六週後，一轉眼就結束了，整學期的課程節奏其實進行得很快，尤其計算機組織中間的幾個章節內容非常多，花了非常多的時間才消化完，不過後面的內容其實也都算蠻有趣的，而且考試有蠻明確的準備方向，所以不會是有太大負擔的一門課。最後，也很感謝劉志尉教授跟助教在這學期提供的協助，遇到的問題幾乎都有得到對應的解答，期望在未來修讀其他門課時也都能夠如此順利。