Computer Organization PA3

Implement a 5-stage pipelined MIPS CPU
with forwarding and hazard detection

Student ID: B11107051

Name: 李品翰

Area: 2938.768 (without IM、DM、RF)
Slack: 1.6546 + 2.5 = 4.1546 (DM size = 32)

1. Module Implementation

以下僅介紹 Part3，且本次 PA 許多地方與 PA2 相同，因此以下僅包含改動的部分與新增的 pipline、hazard detection、forward 等元件。

1.1. ALU_Control.v

- 因本次 PA 沒有 beq，正常情況下不會有 `ALU_op == 2'b01` 的狀況，因此 `Funct` 可指定成 don't care。

```
case (ALU_op)
    ...
    2'b01: Funct <= 2'bxx;
    ...
endcase
```

1.2. Control.v

- 刪除了 beq 與 j 的指令
- 因 hazard detecion 部分需要藉由 `Mem_r` 判斷是否有讀取 memory 的行為，因此不能像 PA2 一樣直接忽略，必須在 `lw` 時為 1，其他為 0。
- 與作業說明提供的圖不太一樣，這裡我新增了一個輸入 (`input Stall`)，若 Stall 則輸出 nop 指令的控制線。

```
always @(*) begin
    if (Stall) begin // nop
        Reg_dst    <= 1'bx;
        Reg_w      <= 1'b0;
        ALU_src    <= 1'bx;
        Mem_w      <= 1'b0;
        Mem_r      <= 1'b0;
        Mem_to_reg <= 1'bx;
        ALU_op     <= 2'bxx;
    end
    ...
end
```

1.3. FinalCPU.v

為了方便區分各個區域的 pin，我以 IF、ID、EX、MEM、WB 等前綴命名，例如 `ID_RdAddr`、`EX_RdAddr`、`MEM_RdAddr`、`WB_RdAddr` 分別代表各區域的 RdAddr。

1.3.1 Hazard Detection Unit

1. 此元件判斷下一個指令是否需要 Stall，此處我稍做修改，將原本應該接到多工器的輸出改為接到 Control，並命名為 Stall：

```
assign Stall = (EX_Mem_r && (
    EX_RtAddr == ID_RsAddr ||
    EX_RtAddr == ID_RtAddr
));
```

2. 另外兩個輸出皆為!Stall：

```
wire IF_ID_write = !Stall;
assign PC_Write = !Stall;
```

1.3.2 Forwarding Unit

1. 此處分別判斷 MEM -> EX 與 WB -> EX 兩種 forwarding：

```
assign Forward_1_MEM = MEM_Reg_w &&
    MEM_RdAddr != 0 && MEM_RdAddr == EX_RsAddr;
assign Forward_2_MEM = MEM_Reg_w &&
    MEM_RdAddr != 0 && MEM_RdAddr == EX_RtAddr;
assign Forward_1_WB = WB_Reg_w &&
    WB_RdAddr != 0 && WB_RdAddr == EX_RsAddr;
assign Forward_2_WB = WB_Reg_w &&
    WB_RdAddr != 0 && WB_RdAddr == EX_RtAddr;
```

2. 再以三元運算子，優先判斷 Forward_x_MEM，使兩種 forwarding 同時觸發時，只會執行 MEM -> EX 的 forwarding：

```
assign EX_ALU_Src_1 = Forward_1_MEM ? MEM_ALU_result :
    Forward_1_WB ? WB_RdData : EX_RsData;
assign EX_Mem_w_data = Forward_2_MEM ? MEM_ALU_result :
    Forward_2_WB ? WB_RdData : EX_RtData;
assign EX_ALU_Src_2 = EX_ALU_src ? EX_imm_extend : EX_Mem_w_data;
```

1.3.3 Pipeline Register

- 大部分的 Pipeline Register 皆由同一個 module，以不同 size 完成：

```
module Pipeline_Register #(
    parameter size = 1
) (
    input [size-1:0] in,
    output reg [size-1:0] out,
```

```
    input clk
);
    initial out <= 0;
    always @(posedge clk) out <= in;
endmodule
```

以 MEM/WB 為例：

```
Pipeline_Register #(.size(71)) pipeline_MEM_WB (
    .in({MEM_WB, MEM_ALU_result, MEM_Mem_r_data, MEM_RdAddr}),
    .out({WB_WB, WB_ALU_result, WB_Mem_r_data, WB_RdAddr}),
    .clk(clk)
);
```

- IF/ID 因包含控制腳 IF_ID_write，需另外寫一個 module：

```
module Pipeline_IF_ID (
    input IF_ID_write,
    input [31:0] IF_Instruction,
    output reg [31:0] ID_Instruction,
    input clk
);
    initial ID_Instruction <= 0;
    always @(posedge clk)
        if (IF_ID_write) ID_Instruction <= IF_Instruction;
endmodule
```

2. Testing Result

為了閱讀方便，以下 IM.dat 的內容將以 MIPS Assembly 的方式呈現。另外，因資料皆以 addiu 或 sw 指定，因此不需要 RF.dat 與 DM.dat 也可得到相同的結果。

2.1. R Type Instruction

- **r_type.asm**

```
addiu   $1,     $0,     1
addiu   $2,     $0,     2
addiu   $3,     $0,     0xffff
addiu   $4,     $0,     0x0f0f
addiu   $5,     $0,     0x00ff
addu    $27,    $1,     $3      # 0x00010000
subu    $28,    $1,     $3      # 0xffff0002
sll     $29,    $2,     30      # 0x80000000
sll     $30,    $2,     31      # 0x00000000
```

```
or      $31,    $4,     $5      # 0x00000fff
```

- **RF.out**

```
...
00010000 // R[27]
ffff0002 // R[28]
80000000 // R[29]
00000000 // R[30]
00000fff // R[31]
```

2.2. I Type Instruction

- **i_type.asm**

```
addiu   $1,     $0,     1284
ori     $2,     $1,     17428
addiu   $3,     $0,     16
ori     $4,     $3,     1
sw      $2,     2($0)
sw      $4,     0($0)
lw      $5,     2($0)
```

- **RF.out**

```
00000000 // R[0]
00000504 // R[1]
00004514 // R[2]
00000010 // R[3]
00000011 // R[4]
00114514 // R[5]
...
```

- **DM.out**

```
00 // Addr = 0x00
00 // Addr = 0x01
00 // Addr = 0x02
11 // Addr = 0x03
45 // Addr = 0x04
14 // Addr = 0x05
...
```

2.3. Hazard Detection (Stall)

- **stall.asm**

```
addiu   $1,     $0,     0x1111      # 0x1111
sw      $1,     0($0)
# EX.Rt == ID.Rs
lw      $2,     0($0)               # 0x1111
addu    $3,     $2,     $0          # 0x1111
# EX.Rt == ID.Rt
lw      $4,     0($0)               # 0x1111
addu    $5,     $0,     $4          # 0x1111
# together
lw      $6,     0($0)               # 0x1111
addu    $7,     $6,     $6          # 0x2222
```

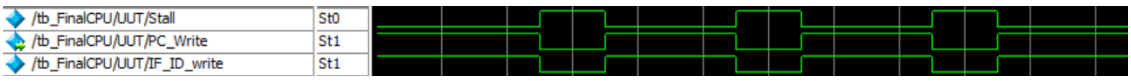
- **RF.out**

```
00000000 // R[0]
00001111 // R[1]
00001111 // R[2]
00001111 // R[3]
00001111 // R[4]
00001111 // R[5]
00001111 // R[6]
00002222 // R[7]
...
```

- **DM.out**

```
00 // Addr = 0x00
00 // Addr = 0x01
11 // Addr = 0x02
11 // Addr = 0x03
...
```

- 從圖中可看見一共觸發了 3 次 stall



2.4. Forward

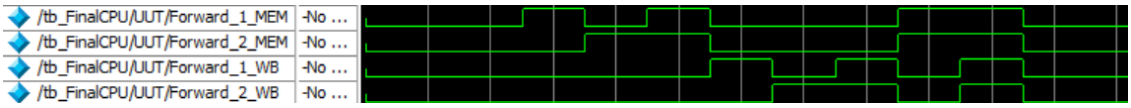
- **stall.asm**

```
# MEM -> EX forward
addiu    $1,     $0,     0x1111   # 0x1111
addu     $2,     $1,     $0       # 0x1111
addu     $3,     $0,     $2       # 0x1111
addu     $4,     $3,     $3       # 0x2222
# WB -> EX forward
addu     $5,     $3,     $0       # 0x1111
addu     $6,     $0,     $4       # 0x2222
addu     $7,     $5,     $5       # 0x2222
# together
addu     $7,     $7,     $7       # 0x4444
addu     $7,     $7,     $7       # 0x8888
```

- **RF.out**

```
00000000 // R[0]
00001111 // R[1]
00001111 // R[2]
00001111 // R[3]
00002222 // R[4]
00001111 // R[5]
00002222 // R[6]
00008888 // R[7]
...
```

- 從圖中可看見各種不同 forward 的觸發組合：



3. Compare with PA2

- 調整 DM size = 32
- Area 皆已去掉 RF、DM、IM
- PA3 的 Slack 已加上 2.5

	Slack	**Area**	**Power**
PA2	4.3441	1829.548	2.95 mW
PA3	4.1546	2938.768	3.24 mW

理論上來說，pipeline 後的最高 clock cycle 應該要是原本的 5 倍，然而事實上是 Slack 降低了，以下分析幾種可能的原因：

- critical path 的時間占比太高，導致即便將它劃分為一個 pipeline 區域，也不會提高整體速度。e.g. 原本的時間占比為 1:1:20:1:1，總和為 24 個時間單位，此時即便將其拆分成五個區域，也需要配合最慢的部分，導致最後仍須要 20 個時間單位。
- 配合上一點，不僅速度沒有增加，還可能因為多出來的控制線導致延遲提高，進而降低整體 slack。
- 或許實際電路速度確實能達到幾乎 5 倍的效果，但 Openroad 計算 Slack 的邏輯並非我所想的那樣，導致結果不如預期。

Area 與 Power 的部分，因為增加了 Hazard、Forward 與許多 Pipeline Register，導致 Area 與 Power 上升，符合預期。

4. Memory Rethinking: Implementing Multi-level Cache

IM 與 DM 皆可新增 Cache，以下以 DM 舉例：

- **CPU:** 需將 DM 改成 Cache，並判斷 Cache 是否 miss，若是則暫停指令，直到 Cache 得到 data 之後才能繼續；若 hit，則與一般的讀取/寫入動作一樣。
- **Cache:** 連接 CPU 與 memory，連接 CPU 方面，除了需要原本 DM 的接腳外，還需要新增 `Read_Miss` 與 `Write_Miss` 2 個接腳，在 miss 時使其變為 1，得到 data 後才變回 0；連接 DM 方面，則較類似 CPU 存取 DM 的行為。從外部看來類似於：

```
module Cache (
    // CPU方面
    input Cache_r, Cache_w,
    input [31:0] Cache_addr, Cache_w_data,
    output Read_Miss, Write_Miss,
    output [31:0] Cache_r_data,
    // DM方面
    output Mem_r, Mem_w,
    output [31:0] Mem_addr, Mem_w_data,
    intput [31:0] Mem_r_data,
);
```

5. Conclusion and Insights

本次 PA 成功完成了 pipeline CPU，老實說設計圖上的各個元件寫起來並沒有很複雜，真正痛苦的是在最後把他們接起來的時候，畢竟線實在是太多了，但這也讓我感受到腳位命名的重要性，好的命名確實能讓我快速了解一條線在圖中的位置、功能。

當初在學習 pipeline CPU 時，還以為 PA3 會包含 beq、j 這兩個指令，這讓我十分焦慮，心想最後該不會連功能正確都無達成了吧，幸好教授與助教們大發慈悲，沒有加入這兩個指令，感恩教授、讚嘆助教！