Pipelined CPU Design

110 學年度第 2 學期

老師：朱守禮　老師

學生：
第 11 組
10820109 電資三 陳詩云
10820119 電資三 翁千涵
10820130 電資三 侯孝璇

一、背景

以 midterm project 撰寫的 ALU Design 為基礎，並參照課堂所學的 Pipelined Datapath，完成 Pipelined MIPS-Lite CPU 設計。

二、方法

根據本學期課程所教授之內容並參考講義內容，實現以下所需之模組。

(1)PC：依據輸控制訊號看是否可將 PC+4 後的結果寫入 PC。此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

(2)Sign Extend：將 16 位元的有號數擴充成 32 位元的輸出。此模組為組合邏輯(Combinational Logic)。

(3)Control：根據讀進來的指令產生對應的控制訊號。模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

(4)ALU：以 1-bit 的 Full Adder 為基礎，透過訊號的控制選擇要進行 AND，OR，ADD，SUB 還是 SLT 的功能，並依據課程講義所講述的設計方式，以 Ripple Carry 的進位方式,將 32 個 1- bit 的 ALU Bit Slice 連結起來,實現成一個 32-bit 的 ALU。此模組為組合邏輯(Combinational Logic)。

(5)Register File：依照輸入的暫存器編號讀取對應的資料，並根據 clock 訊號與 RegWrite 訊號在特定時間點開放寫入資料。此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

(6)Memory：由 MemRead 訊號與 MemWrite 訊號分別控制資料的讀取與寫入，若讀取資料的條件成立則從記憶體讀取資料並存放於暫存器中;若寫入資料的條件成立，則將資料寫入記憶體指定的位址中。此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

(7)Shifter：根據課程講義所描述的 Barrel Shifter 為設計方式，透過 5 個移位量，完成 32-bit 的左移 Shifter，而此模組為組合邏輯(Combinational Logic)。

(8)HiLo：儲存除法器計算後的結果，其中 Quotient 存於 32-bit 的 Lo 暫存器，Remainder 存於 32-bit 的 Hi 暫存器。此模組為循序邏輯(Sequential Logic)，因此須以 Clock 訊號同步。

(9)Divider Hardware：32-bit 的無號數除法器，採用 3rd Version Sequential Restoring Division Hardware 來實作，而此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步，因此採用 always block 來設計。

(10)Mux：以 Dataflow Modeling 來設計，根據讀進的控制訊號輸出其對應功能所運算出的結果。本模組為組合邏輯(Combinational Logic)。

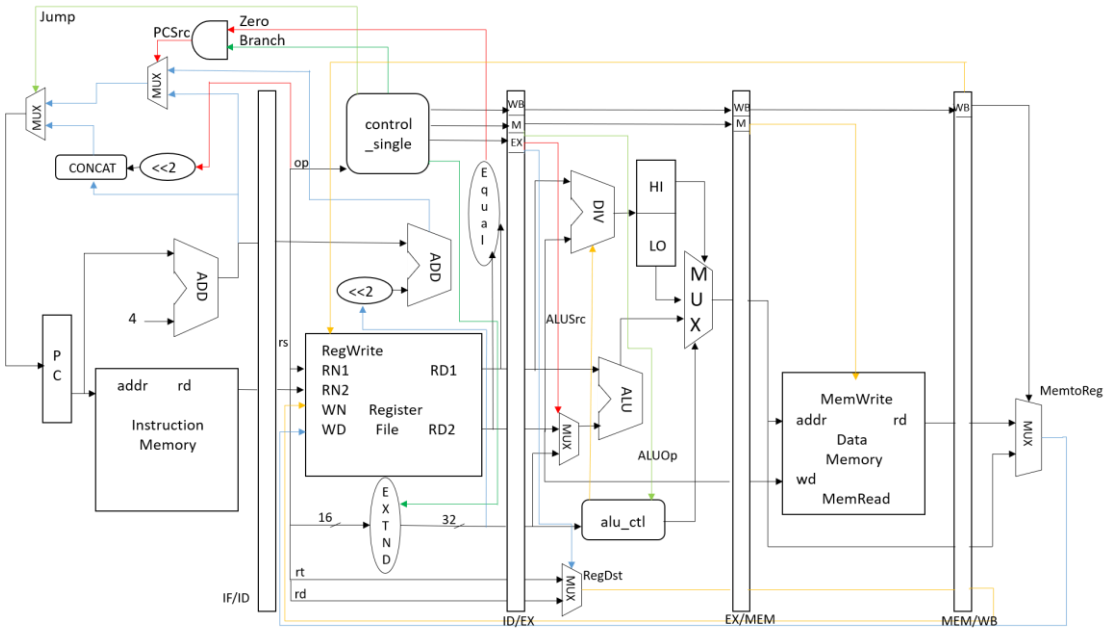
(11)Pipeline Registers：將 Datapath 的電路依照五階區隔開來，分別為 IF/ID、ID/EX、EX/MEM 及 MEM/WB，用於暫存每一階的輸出結果。而模組皆為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

(12)Hazard Detection：透過 ID/EX.Register、ID/EX.MemRead 以及 32 bits 的 instruction，依條件判斷 hazard 是否發生，並發送相對應的控制訊號。此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

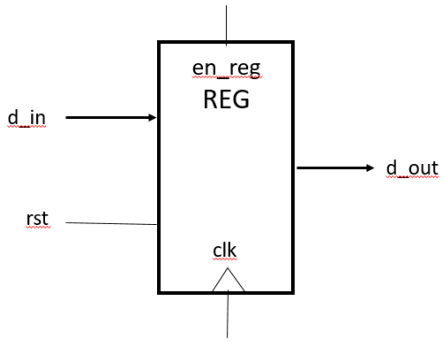
(13)Forwarding：接收 IF/ID.RegisterRs、IF/ID.RegisterRt、EX/MEM.RegisterRd、MEM/WB.RegisterRd、EX/MEM.RegisterWrite 以及 MEM/WB.RegisterWrite，判斷是為 ForwardA 還是 ForwardB。此模組為循序邏輯(Sequential Logic)，須以 Clock 訊號同步。

三、Datapath 與架構圖

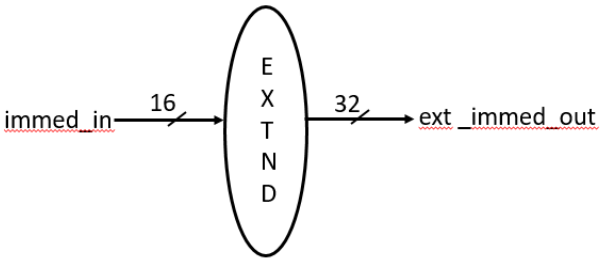
(1) Mips_signle



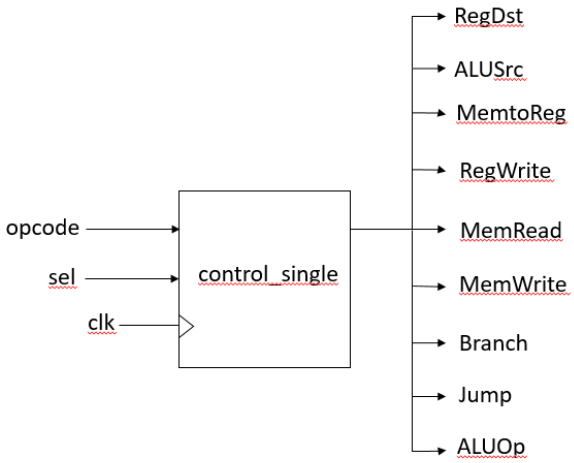
(2) reg32



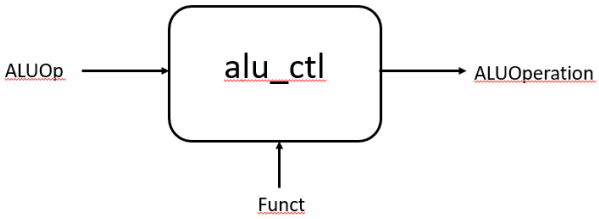
(3) sign_extend



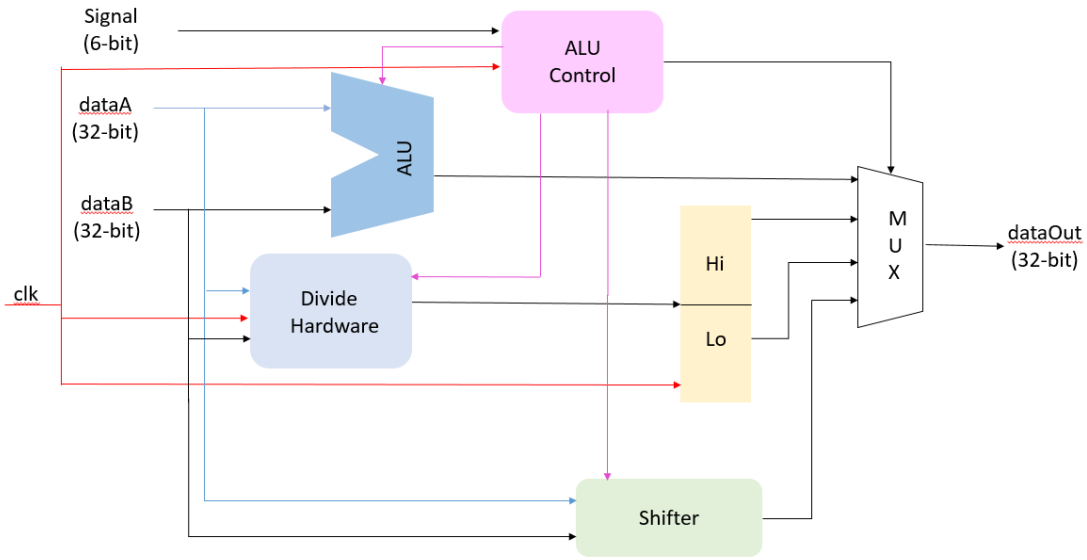
(4) control_signle



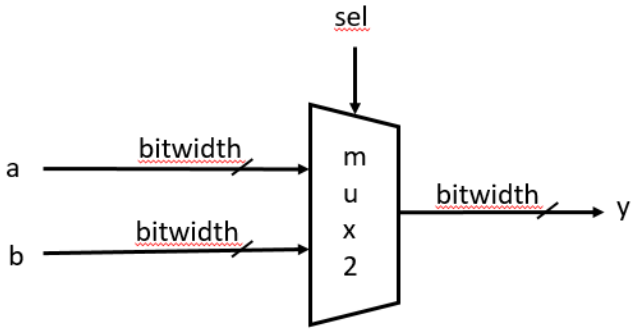
(5) alu_ctl



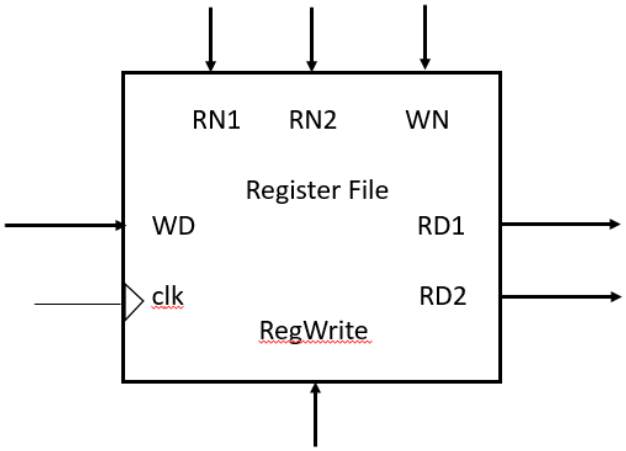
(6) TotalALU



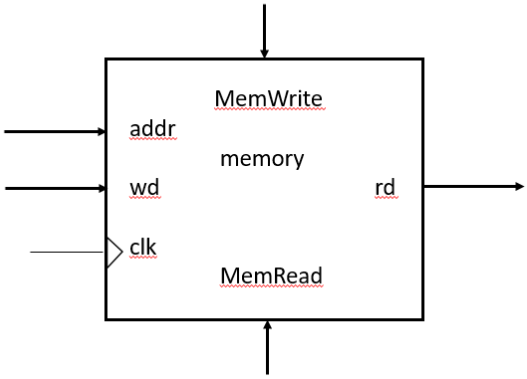
(7) mux2



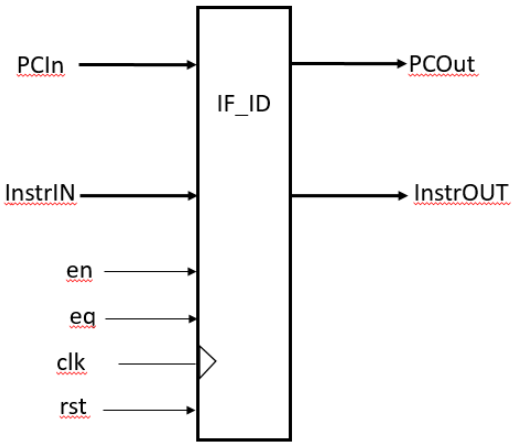
(8) reg_file



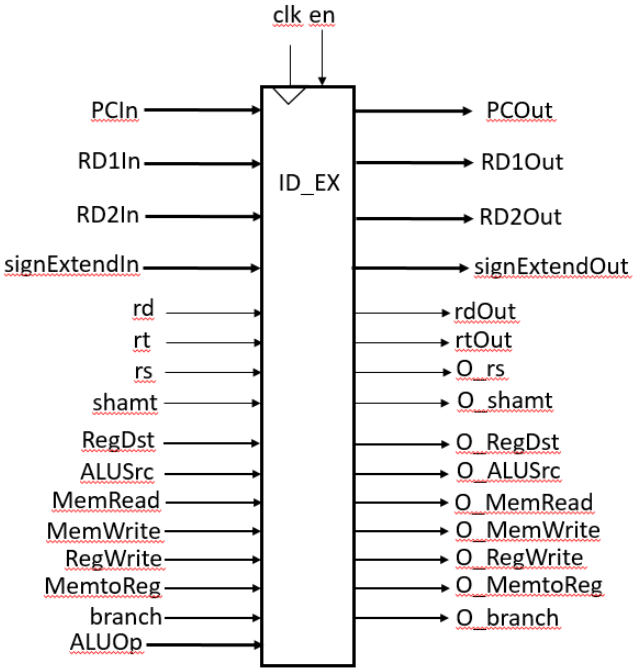
(9) memory



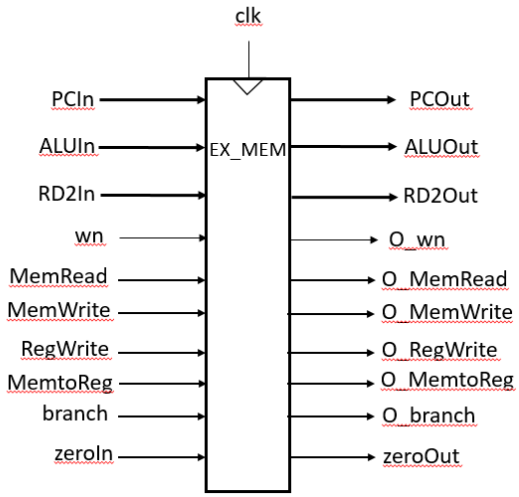
(10) IF_ID



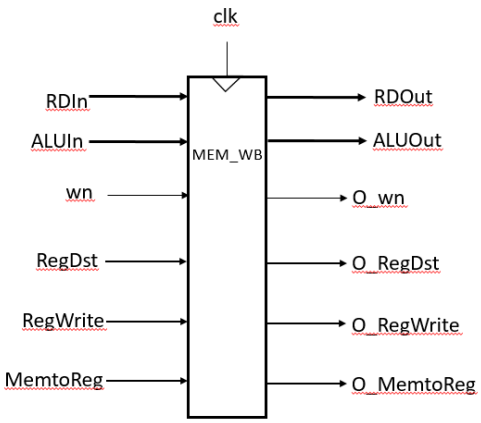
(11) ID_EX



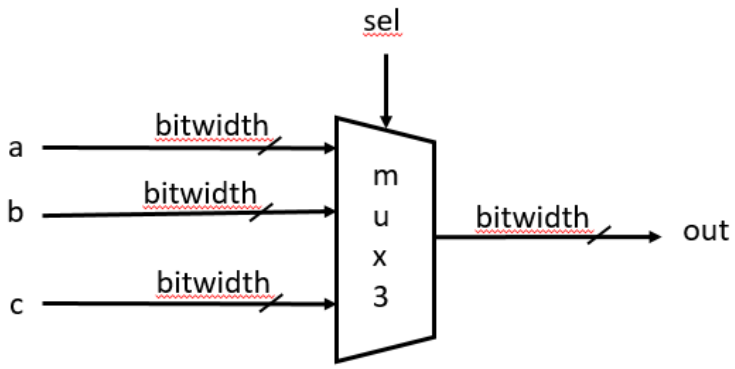
(12) EX_MEM



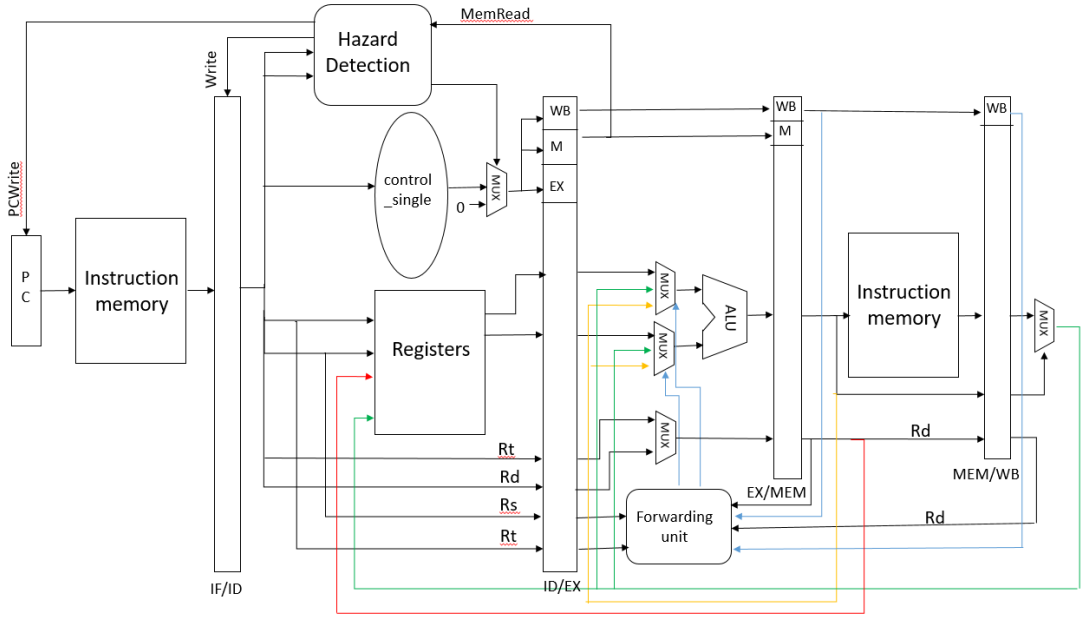
(13) MEM_WB



(14) mux3



(15) ForwardingUnit+hazarDetection



設計重點 ：

(1) 32-Bit ALU：將 1-bit 的 ALU 以 Ripple-Carry 的進位方式串接成一個完整的 32-Bit ALU，而 ALU 會透過控制訊號做出對應的動作，若控制訊號為 SUB 或 SLT，Carry In 的最低位元會輸入 1，否則便為 0，此動作是為了對被減數做 2’s complement 中加 1 的動作。

最後 SLT 可依照兩運算元相減後的正負情形來判斷兩者間大小的關係，而該正負的判斷則取決於最後 sum 的最高位元，若為 1 則表示運算元 A 小於運算元 B，為 0 則表示運算元 A 大於等於運算元 B。

(2) Shifter：dataA 為要做 shift 的對象，dataB 是指要左移幾 bit 數，根據 dataB 的最後 5 個 bit 最多可將 dataA 位移 32 bits，每一層中 dataA 的每一 bit 都會根據該層 dataB 的值決定是否要做移位，為 1 的話則該位元的輸出為 dataA 的下一個位元值，為 0 的話則不移位，維持原本的數值。

(3) Divider：先將 temp 的 Lo 擺放被除數，Hi 則清除為 0，將其左移一個 bit 後，開始做 32 回合的處理，temp 的左半邊 = temp 的左半邊-除數，若 temp 為正數，則最低位元設為 1 並左移 1bit，若 temp 為負數，則將除數加回來、最低位元設為 0 並左移 1bit。32 回合結束後，將 temp 的左半邊右移一個 bit 回去，最後 temp 的 Hi 放的是運算結果的餘數，Lo 放的則是運算結果的商。

(4) Datapath：由 PC、ADD、Control Unit、Instruction Memory、Register File、EXTEND、ALU、ALU Control、Data Memory 與 MUX 等，多個 component 所組成的電路。每一階會由 Pipeline Registers 區隔開來，Instruction Fetch 的輸出會存放在 IF/ID 暫存器裡；Instruction Decode 的輸出會存放在 ID/EX 暫存器中；Execute/Address Calculation 的輸出會存放在 EX/MEM 暫存器裡；Memory Access(Read/write)的輸出則存放在 MEM/WB 暫存器中，藉此完成 Pipelined Execution。而其中會透過 Control Unit 送控制訊號給各個 MUX，幫助選擇該指令所需之輸出。

(5) Branch：當讀取到的指令被解譯為 beq 或 bne 後，會判斷 RD1 與 RD2 是兩者是否相等。若為 beq 且 RD1 與 RD2 相等，或為 bne 且 RD1 與 RD2 不相等的話，則改變控制訊號 PCSrc，再透過 MUX 選擇輸出，PC+4 與經過 extend 後再做 2-bit 左移的 offset 之相加後，再寫回 PC。

(6) Jump: 當讀取到的指令被解譯為 j 後，會將 PC+4 取四位元([31:28])再接上指令裡經過左移兩位元的 address([25:0])，根據控制訊號 Jump，透過 MUX 輸出該結果後，再寫回 PC。

(7) Memory Access:當讀取到的指令被解譯為 lw 時，會根據控制訊號 RegDst 使 Register File 的 WN 放的是 rt 暫存器，再透過控制訊號 ALUSrc，使 MUX 選擇讓經過 extend 成 32 bits 的 offset 作為輸出，使 RD1 與其能夠作為 ALU 的兩個 input，再透過控制訊號 Operation 來選擇要讓 ALU 做相加的工作，最後進入 Data Memory，會再透過控制訊號 MemtoReg 來決定是否要將資料寫回 Register File 的 WD。

若為 sw，會根據控制訊號 RegDst 使 Register File 的 RN2 放的是 rt 暫存器，再透過控制訊號 ALUSrc，使 MUX 選擇讓經過 extend 成 32 bits 的 offset 作為輸出，使 RD1 與其能夠作為 ALU 的兩個 input，再透過控制訊號 Operation 來選擇要讓 ALU 做相加的工作，輸出將作為 Data Memory 中的 ADDR，而 RD2 也會作為 Data Memory 中，要寫入的資料 WD。

(8) ori：當指令被解譯為 ori 時，其作用在於將指令[15:0]的 immed 擴充至 32 bits，再與 rs 裡的值進行 OR 運算，再將結果存至 rt。

(9) nop：當指令被解譯為 nop 時，則為 no operation

四、結果與討論

(1) 模擬結果：

```
  0, reading data: Mem[          0] => 2385444864
  0, reg_file[ 0] =>          0 (Port 1)
  0, reg_file[ 0] =>          0 (Port 2)
  0, PC:          0
  0, LW

  1, reading data: Mem[          4] =>  372310019
  1, reg_file[17] =>          2 (Port 1)
  1, reg_file[15] =>         21 (Port 2)
  1, PC:          4
  1, BNE

  2, reg_file[17] =>          2 (Port 2)
  2, PC:          4
  2, wd:          x
  2, NOP

  3, reading data: Mem[          8] =>  305201154
  3, reading data: Mem[          2] =>        256
  3, reg_file[ 0] =>          0 (Port 1)
  3, reg_file[ 0] =>          0 (Port 2)
  3, PC:          8
  3, BEQ

  4, reg_file[17] =>          2 (Port 1)
  4, reg_file[17] =>          2 (Port 2)
  5, reg_file[15] <=        256 (Write)
  4, PC:          8
  4, wd:        256
  4, NOP

  5, reading data: Mem[         20] =>   36735008
  5, reg_file[ 0] =>          0 (Port 1)
  5, reg_file[ 0] =>          0 (Port 2)
  5, PC:         20
  5, wd:          x
  5, ADD
```

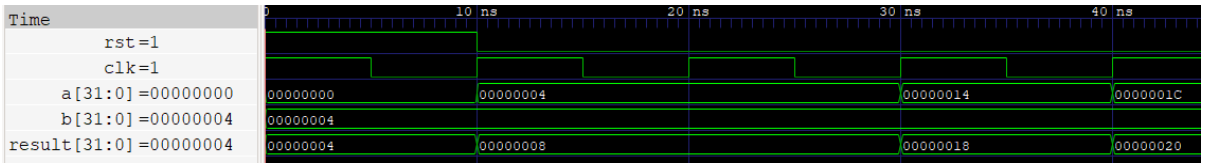
```
  6, reading data: Mem[         24] =>  134217735
  6, reg_file[17] =>          2 (Port 1)
  6, reg_file[16] =>          1 (Port 2)
  6, PC:         24
  6, J

  7, reading data: Mem[         28] =>   38834213
  7, reg_file[ 0] =>          0 (Port 1)
  7, reg_file[ 0] =>          0 (Port 2)
  7, PC:         28
  7, wd:          x
  7, OR

  8, reading data: Mem[         32] =>   36735008
  8, reg_file[18] =>          3 (Port 1)
  8, reg_file[16] =>          1 (Port 2)
  8, PC:         32
  8, wd:          0
  8, ADD

  9, reading data: Mem[         36] =>   38834210
  9, reg_file[17] =>          2 (Port 1)
  9, PC:         36
  9, wd:          3
  9, SUB

 10, reg_file[17] <=          3 (Write)
 10, reg_file[18] =>          3 (Port 1)
 10, reading data: Mem[         40] =>   38834213
 10, PC:         40
 10, wd:          x
 10, OR

 11, reading data: Mem[         44] => 2886860824
 11, PC:         44
 11, SW

 12, reg_file[18] <=          3 (Write)
```

```
 12, reg_file[18] <=          3 (Write)
 12, reading data: Mem[         48] =>   38797339
 12, reg_file[ 0] =>          0 (Port 1)
 12, reg_file[18] =>          3 (Port 2)
 13, reg_file[17] <=          3 (Write)
 12, PC:         48
 12, wd:          3
 12, DIVU

 13, reading data: Mem[         52] =>          0
 13, reg_file[18] =>          3 (Port 1)
 13, reg_file[16] =>          1 (Port 2)
 13, PC:         52
 13, wd:          2
 13, NOP

 14, reg_file[18] <=          2 (Write)
 14, reg_file[ 0] =>          0 (Port 1)
 14, reg_file[ 0] =>          0 (Port 2)
 15, reg_file[18] <=          3 (Write)
 15, writing data: Mem[         24] <=          3
 14, PC:         56
 14, wd:          3
 14, NOP

 15, PC:         60
 15, wd:          x
 15, NOP

 16, PC:         64
 16, wd:          0
 16, NOP

 17, PC:         68
 17, wd:          0
 17, NOP

 18, PC:         72
 18, wd:          0
 18, NOP
```

```
 45, reading data: Mem[        180] =>      36880
 45, PC:        180
 45, wd:          0
 45, MFHI

 46, reading data: Mem[        184] =>      36882
 46, PC:        184
 46, wd:          0
 46, MFLO

 47, reading data: Mem[        188] =>    1218688
 47, PC:        188
 47, wd:          0
 47, SLL

 48, reading data: Mem[        192] =>   38967338
 48, reg_file[18] =>          3 (Port 2)
 48, PC:        192
 48, wd:          0
 48, SLT

 49, reading data: Mem[        196] =>  913571860
 49, reg_file[18] =>          3 (Port 1)
 49, PC:        196
 49, ORI

 50, reg_file[18] <=          0 (Write)
 50, reg_file[20] =>          5 (Port 2)
 50, reg_file[19] =>          4 (Port 1)
 50, reading data: Mem[        200] =>          x
 51, reg_file[18] <=          3 (Write)
 50, PC:        200
 50, control_single unimplemented opcode  x
 51, reg_file[ x] =>          x (Port 1)
 51, reg_file[ x] =>          x (Port 2)
 51, PC:        200
 52, reg_file[19] <=         12 (Write)
 51, control_single unimplemented opcode  x
 53, reg_file[19] <=          0 (Write)
```

(2) GTK wave 輸出圖形:

1.ADD module(PC+4)



當正緣觸發時,a 與 b 相加,並以 result 作為 output。因此為針對 PC+4 的 ADD,因此 a 為當前指令的位址，而 b 恆為 4。

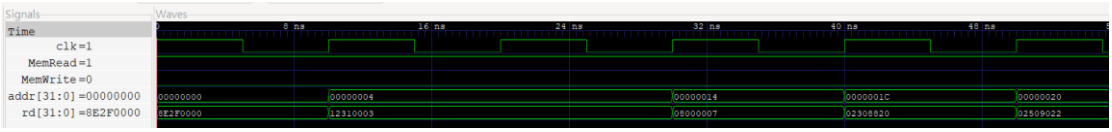
e.g. 0ns：a = 00000000, b = 00000004, result = 00000004

10ns：a = 00000004, b = 00000004, result = 00000008

20ns：a = 00000014, b = 00000004, result = 00000018

而 20ns 時，a 之所以不是 00000008 而是 00000014 的原因在於，上一道指令為 beq 指令，因而修改過 PC。

2. Instruction Memory



在 Instruction Memory 中，MemRead 訊號恆設置為 1 且 MemWrite 訊號設置為 0，並在正緣觸發時，對 addr 做定址，從 Memory 中取出指令並存放在 rd 暫存器中。

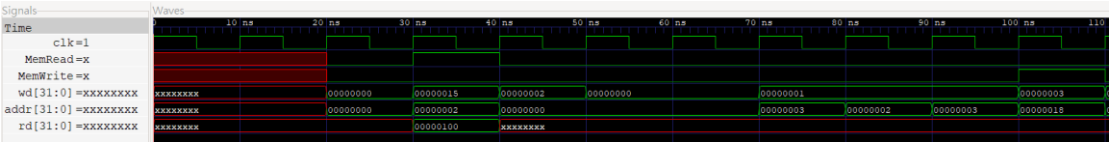
e.g. addr = 00000000, rd = 8E2F0000 lw $t7, $s1

addr = 00000004, rd = 12310003 beq $s1, $s1, 3

addr = 00000014, rd = 08000007 j 7

與(1)的 e.g 對比，執行到 00000014 的指令時，因為指令為 jump，因此取出下一個要執行之位址時不會是原先的 00000018，而是 0000001C。

3. Data Memory

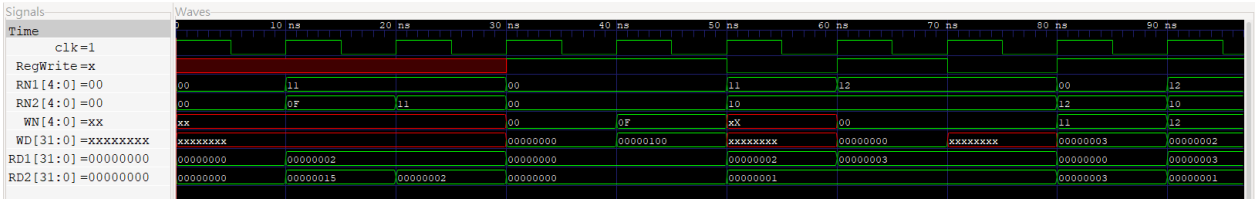


在 Data Memory 中，會根據控制訊號 MemRead 與 MemWrite 來決定對 Data Memory 的讀與寫，在前 20ns 時 MemRead 與 MemWrite 因 Control Unit 尚未發送訊號，因此為無訊號的狀態，wd、addr 與 rd 則為 don’t care。

而直到 30ns 時為例，正緣觸發且 MemRead 訊號為 1，意味著會對 Data Memory 做讀取，add 收到欲存取之記憶體位址，wd 收到欲寫入之資料，rd 則會輸出讀取到的記憶體資料。

再以 100ns 時為例,正緣觸發且 MemWrite 訊號為 1,意味著要對 Data Memory 做寫入，而 MemRead 訊號為 0，因此 rd 仍保持 don’t care 的狀態。

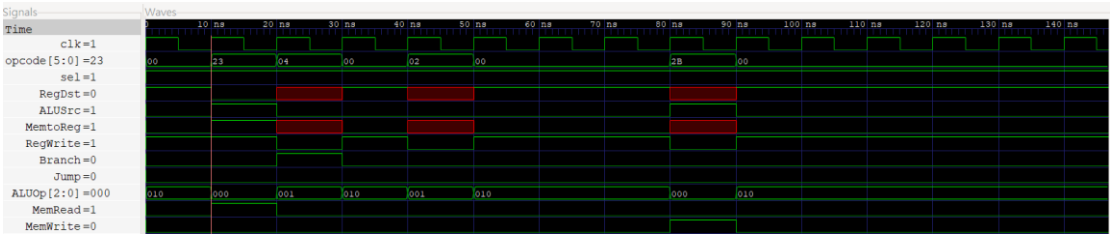
4. Register File



指令執行的解碼階段中，Register File 有個 RegWrite 訊號 Clock 訊號，控制資料的寫入，RN1 與 RN2 對應的分別是 5-bit 的 rs 與 rt，也就是欲讀取之暫存器編號；WN 為欲寫入之暫存器編號；WD 為欲寫入暫存器之資料；RD1 與 RD2 則分別是 rs 與 rt 暫存器所讀取之資料。

以 30ns 時為例，可以看到正緣觸發且 RegWrite 訊號為 1，意味著此刻是可寫入的狀態，並且在 40ns 時 WD 有了新的資料(00000100)，若與 Data Memory 的波形圖一起看，能夠看到同樣在 30ns 時，rd 輸出了 00000100。

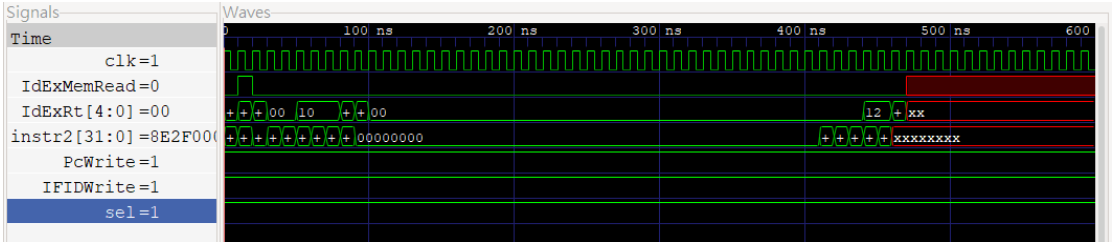
5. Control Unit



在 32-bit 的 opcode 中會由 6-bit 的 op 來決定該指令的功能，並依此發送相對應的控制訊號。

以第一道指令為例，由於在此 opcode 是以 16 進制表示，若將 23 轉為二進制可以看作 100011，則可得知是為 lw 指令。依照執行 lw 所需的控制訊號，會分別將 RegDst、Branch、Jump 以及 MemWrite 設為 0；ALUSrc、MemtoReg、RegWrite 以及 MemRead 設為 1，並且 ALUOp 設為 000。

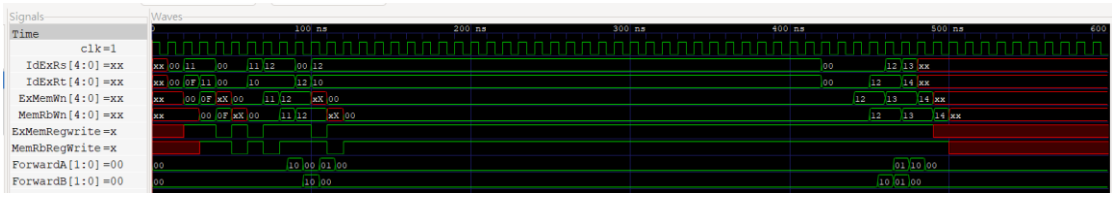
6. Hazard Detection



根據特定的訊號(idEXMemRead)以及暫存器(IdExRt、instr2 所擷取出的 rs 和 rt 暫存器)判斷是否發生 hazard，並且輸出相對應的訊號。

若 PCWrite、IFIDWrite 以及 sel 皆輸出 1 的訊號，即表示沒發生 hazard，因此從該波形圖可看出期間並沒有發生 hazard。

7.Forwarding



根據特定的訊號 ExMemRegwrite 以及 MemRbRegWrite 以及暫存器 IdExRs、IdExRt、ExMemWn、MemRbWn，判斷是否發生 hazard 並輸出相對應的訊號。

由波形圖的 ForwardA 與 ForwardB 可看出 EX hazard 與 MEM hazard 的發生。

五、心得

陳詩云：本來以為這次 project 經過期中作業後會更順利，但好像也沒那麼順利，一開始要研究如何把 single_cycle 的電路切成 pipeline，好不容易接好線之後又遇到 data hazard 的問題。我覺得這次作業除了要活用課堂所學還要非常細心，只要一個 input 或 output 寫錯就會造成後面指令執行錯誤，雖然這次作業有點複雜，不過可以學到很多東西，除了更了解 verilog 的語法，也更理解指令的執行路徑。

翁千涵：透過這次的期末作業，讓我以實作的方式更加了解課堂上的內容，雖然不是負責撰寫程式碼，但在寫書面報告與繪圖時，試著將所理解的內容轉換成文字，以便釐清思緒。在撰寫的過程十分燒腦，考驗邏輯是否能正確運作，除了理解程式碼之外，還需要將其結合課堂所學的理論觀點。有了這次的歷練讓我收穫很多，也期許自己能更加進步。

侯孝璇：在本次的專題裡應用了一學期以來所學習所有知識，在過程中遇到不理解的地方需要重複翻閱課程講義，在這反覆查詢間漸漸熟悉起課程內容，也算是作到複習的作用。現在也不由得再次感嘆電路設計的難度，元件透過導線的銜接都是相依的，又是平行處理，又要考慮效能等問題，經過多方思考與設計後才能獲得一個完整的電路。過程是有點痛苦的，但至少是有收穫的。

六、未來展望

透過本次期末專題之 Pipelined CPU Design 為基礎，面對相關作業時，能夠應用所學，設計並撰寫更佳的電路。

組員分工：
10820109 陳詩云 程式撰寫
10820119 翁千涵 書面報告與繪圖
10820130 侯孝璇 書面報告與繪圖