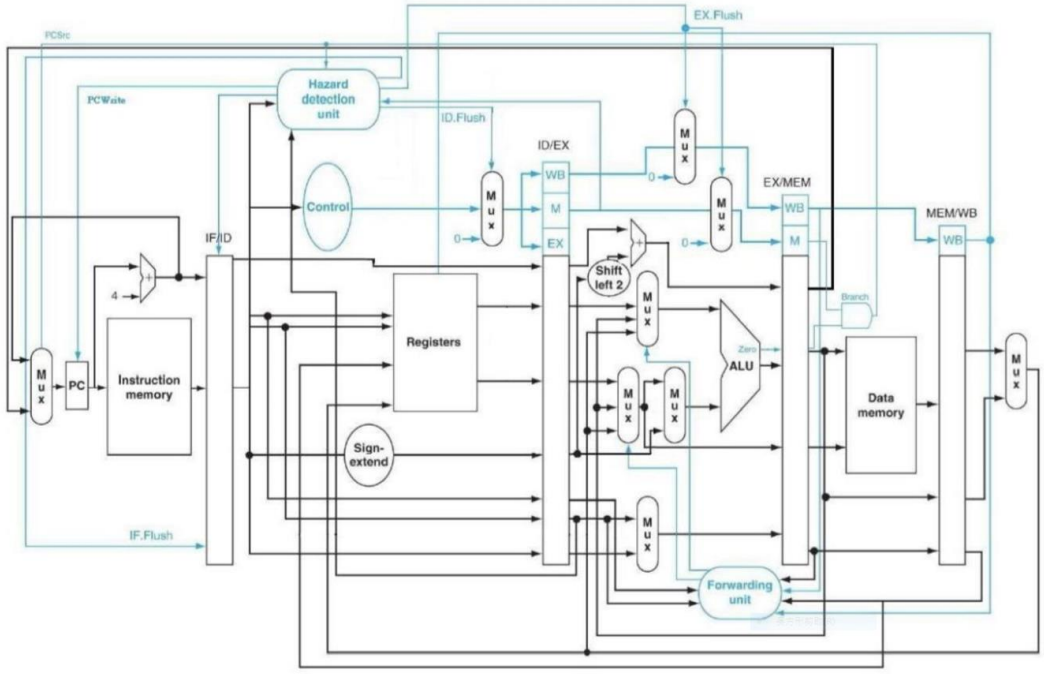
Computer Organization
Lab5

Architecture diagrams:

使用題目上的架構



Hardware module analysis:

這次的主要目的是加了 Forwarding Unit 和 Hazard Detection Unit 來解決 data hazard.

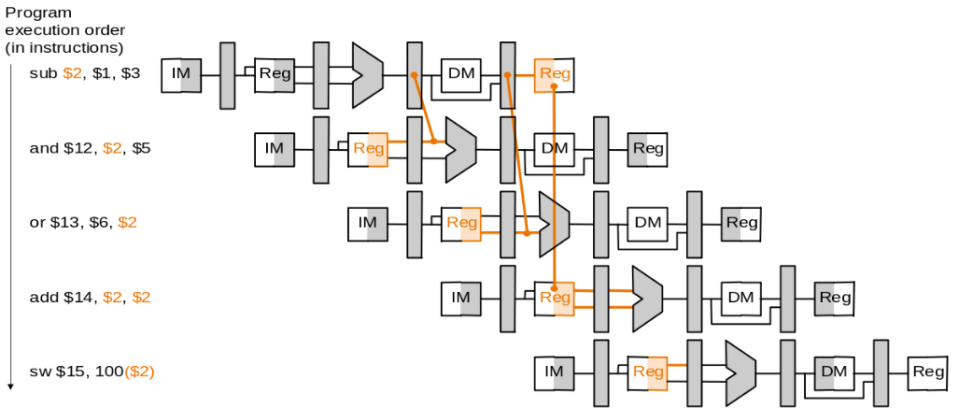
此 lab 解決的 hazard 有三種

1. R-type data hazard
2. Load-use data hazard
3. Branch hazard

Forwarding Unit 主要進行 data 的 forwarding(duh)
Hazrd Detection Unit 則是進行 stall 與 flush 的動作

以下進行 hazard 的整理

1. R-type hazard



上圖來自網路大神的筆記

https://hackmd.io/@8bFA57f7SRG-K7AAT8s62g/ryv1NT3S?type=view

從圖清楚的看到 R-type 的兩種 hazard

第一種的條件為 IF/ID.RegRt(Rs) = EX/MEMRegRd

第二種為條件為 IF/ID.RegRt(Rs) = MEM/WB.RegRd

總之原因是寫入的目標與下個指令讀取的來源相衝突

加入 forwarding 後可以不用 bubble 解決衝突，code 的條件判斷如下

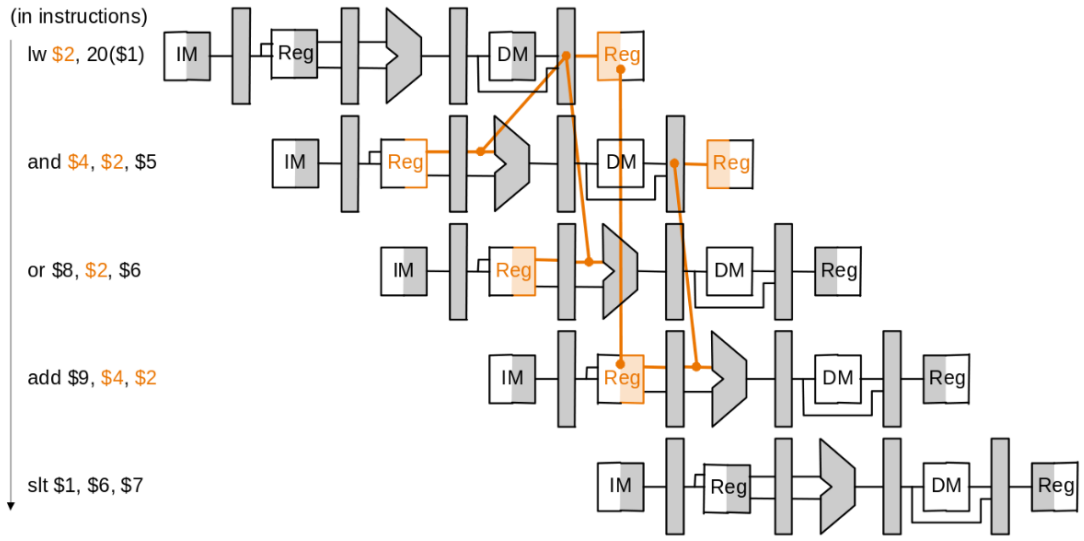
需要 Forwarding

```
if(    EXMEM_WB[1] && EXMEM_Rd != 0 && EXMEM_Rd == IDEX_Rs)
                        Rs_forward <= 2'b01;

if(    EXMEM_WB[1] && EXMEM_Rd != 0 && EXMEM_Rd == IDEX_Rt)
                        Rt_forward <= 2'b01;

if(MEMWB_WB[1] && MEMWB_Rd != 0 && MEMWB_Rd == IDEX_Rs)
                        Rs_forward <= 2'b10;

if(MEMWB_WB[1] && MEMWB_Rd != 0 && MEMWB_Rd == IDEX_Rt)
                        Rt_forward <= 2'b10;
```

不需要 Forwarding

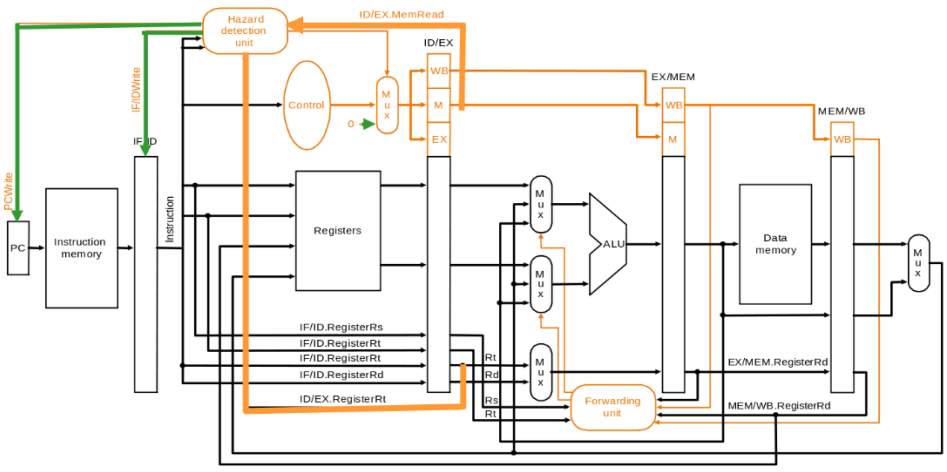
```
if(MEMWB_Rd != IDEX_Rs && EXMEM_Rd != IDEX_Rs)
                        Rs_forward <= 2'b00;

if(MEMWB_Rd != IDEX_Rt && EXMEM_Rd != IDEX_Rt)
                        Rt_forward <= 2'b00;
```

2. Load-use hazard



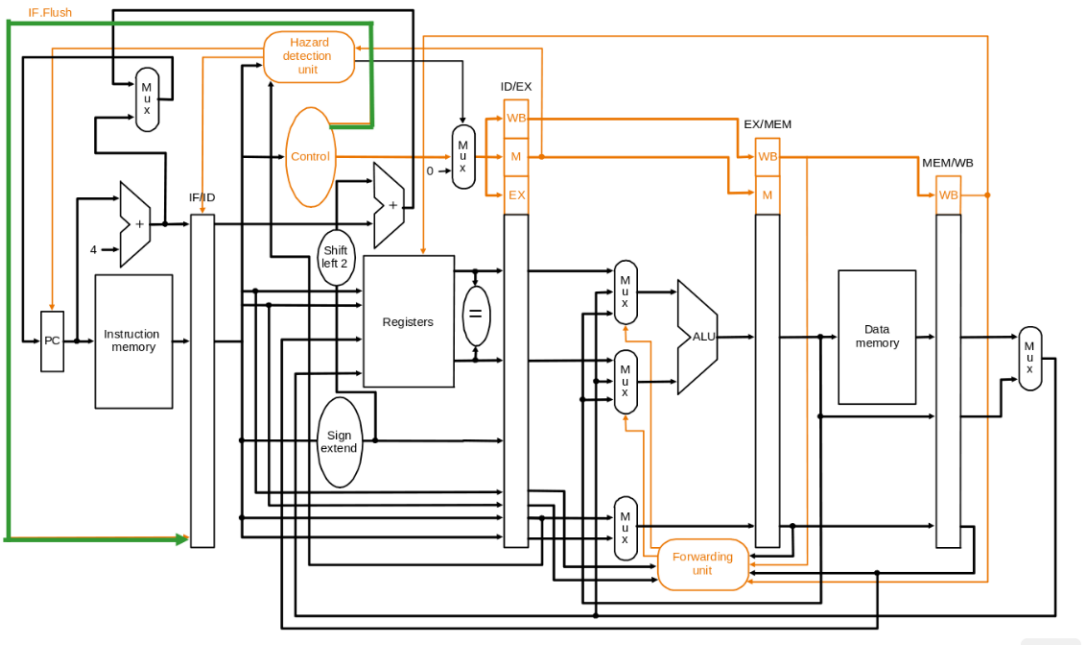
上圖可以看到就算使用 forwarding 也無法解決 data hazard
所以 load-use hazard 無法避免的需要 stall 一個 cycle
Stall 這動作是由 hazard detection unit 進行的，有兩條控制線分別連到 Program Counter 和 IF\ID.Reg，在 load-use 發生時控制線會停止兩個 reg 的寫入達到 stall 的作用，以下為 load-use 發生的條件

```
if(     (IFID_RsRt[9:5] == IDEX_Rt || IFID_RsRt[4:0] == IDEX_Rt) &&
        IDEX_Rt != 0 &&
        M[1]
 )
                IDFlush <= 0;   //load use  不需要 flush
                EXFlush <= 0;
                IFFlush <= 0;
                PCWrite <= 0; //  暫停 PC 寫入
                IFIDWrite <=0;
```



3. Branch Hazard

因為要等比較結果才能知道要不要 branch 所以真到要 branch 時會讀到不需要的指令，hazrd detection unit 可以 flush 調不需要的指令



再次附上網路大神的精美示意圖(不過他沒有 flush 後面的東西)

Finished part:

CO_P5_test_1.txt 測試結果

```
Register=============================================================

r0=         0, r1=         16, r2=        256, r3=          8, r4=         16, r5=          8, r6=         24, r7=         26

r8=         8, r9=         1, 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=         0, r30=         0, r31=         0


Memory=============================================================

m0=         0, m1=         16, m2=         0, m3=          0, m4=         0, m5=          0, m6=         0, m7=          0

m8=         0, m9=         0, m10=         0, m11=         0, m12=         0, m13=         0, m14=         0, m15=         0

r16=        0, m17=        0, m18=         0, m19=         0, m20=         0, m21=         0, m22=         0, m23=         0

m24=        0, m25=        0, m26=         0, m27=         0, m28=         0, m29=         0, m30=         0, m31=         0
INFO: [USF-XSim-96] XSim completed. Design snapshot 'TestBench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:06 . Memory (MB): peak = 989.145 ; gain = 0.285
```

自己測試 beq 的結果
我用 lab4 的測資去改的
執行的指令如下

00100000000000010000000000000101 (addi, $1 = 5)
00100000000000010000000000000101 (addi, $2 = 5)
00010000001000100000000000000001 (beq, $1 =$2?)
10001100000001010000000000000111
00100000000000010000000000000011
00100000000000010000000000000100
00100000000000010000000000000101
00100000000000010000000000001111

即將 branch

```
Pipeline Registers==============================================================

IFID: 0000000000000000000000000001010000100000000000010000000000000011

IDEX: 1101000001000000000000000000000000000001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001110101000000000

EXMEM: 00100000000000000000000000000000100001000000000000000000000000000000000000000000000000000000000000010100010

MEMWB: 10000000000000000000000000000000000000000000000000000000000000010100010

RST: 1

Branch: 1, 1
```

執行 Flush 可以看到 IFID 變成 0

```
Pipeline Registers==============================================================

IFID: 0000000000000000000000000000000000000000000000000000000000000000

IDEX: 0000000000000000000000000000000000000101000000000000000000000000000000000000000000000000000000000000000000000101000000000000000000000000001100001000000000

EXMEM: 00000000000000000000000000000000101100000000000000000000000000000000000111000000000000000000000000000001010

MEMWB: 00000000000000000000000000000000000000000000000000000000000000000000010

RST: 1

Branch: 0, 0
```

Problems you met and solutions:

1. **Forwarding Hazard 條件問題**
   原本以剩下條件用 else 就好，但其實這樣會造成問題，所以我還是把不會發生 hazard 的條件列出來，詳細條件如前面 hardware analyses 說的

2. **Hazard Detection Unit 條件問題**
   ((IFID_RsRt[9:5] == IDEX_Rt || IFID_RsRt[4:0] == IDEX_Rt) && IDEX_Rt != 0 && M[1])前面那段要刮起來不然會判斷錯誤
3. **Load-use hazard stall**
   因為需要 pipeline register 可以暫停寫入，所以稍微修改了 pipline register，加了條 write enable 控制線
4. **Forwarding Unit 造成 Don't Care Term**
   在用 Lab4 的測資測試時，發現 addi 指令會造成 Don't care term，結果會不正確，原本以為是 Forwarding Unit 的條件判斷錯誤造成的，結果是眼殘接錯 WB register，往前抓到前一個 step 的(不知為改正前 Lab5 結果還是對的@@)

5. **Beq 指令結果不對，flush 之後 reg 呈現 Don't care**
   原本以為是 Flush 的問題，後來將相關線路的數值印出來才發現 ALUSrc MUX 的第二個 source (branch pc)接錯線，接上之後 beq 正常運行但 flush 會錯誤，後來發現 rst_i 的值平常是 1，所以要 flush 的話 pipreg 的 rst 值是

   (rst_i & !IFFlush)

   而不是

   (rst_i | IFFlush)

Summary:

好險有網路大神的筆記，不然可能寫不太出來，雖然期末考爆掉了，不過藉由這次的 lab 終於把課堂上所教的 hazard 都搞清楚了，雖然寫 lab 很花時間但很好玩，不過還是希望明年不要再寫一次 rrr