**Final\_project \_MIPS\_CPU**

**DCS121\_109511314\_鄭旭恩**

1. **MIPS\_CPU\_Introduction**

MIPS（Microprocessor without Interlocked Pipeline Stages）架構是一種經典的精簡指令集（RISC）架構，具有高效、簡潔和易於實現的特點。以下是對MIPS架構CPU的詳細介紹：

**MIPS指令集結構**

MIPS架構最初由於1981年在加州大學柏克萊分校開發。它的設計遵循精簡指令集計算機（RISC）的設計理念，目標是將指令的執行時間降低到最小，通過精簡指令集和高度優化的硬件實現來實現高性能和高效能。MIPS指令集被設計成固定大小的32位元長度，大多數指令在執行時具有相同的執行時間。主要的指令類型包括：

* **算術運算和邏輯操作**：包括加法、減法、乘法、除法等基本算術操作。
* **數據轉移**：用於將數據從存儲器讀取到寄存器或將數據從寄存器寫入存儲器。
* **控制流**：支持條件分支、無條件分支和跳轉指令，例如分支相等（beq）、無條件跳轉（j）、加載字（lw）和存儲字（sw）等。

**MIPS架構特點**

1. **固定長度指令**：所有指令均為32位元，簡化了硬件設計和指令解碼。
2. **單一週期執行**：大多數指令在單個時脈週期內完成執行，有助於實現高速執行。
3. **五級流水線**：典型的MIPS處理器具有五級流水線，包括指令提取(IF)、指令解碼(ID)、執行(EX)、存儲器訪問(MEM)和寫回(WB)階段。
4. **通用寄存器**：MIPS架構有32個通用寄存器（$0 - $31），所有算術運算和數據操作都是在這些寄存器之間進行的。

**Hazard:**

1. **Structural hazard:** 這是指某個指令由於硬體資源無法同時提供所需的組合指令，導致該指令無法在預定的時鐘周期內執行。這種風險通常可以在設計階段提前避免，以確保硬體資源的合理分配。
2. **Data Hazard**：當一個指令需要的數據在執行時尚未準備好，導致該指令無法在正確的時鐘周期內執行。這種風險是動態產生的，需要使用Data forwarding等方式來解決。
3. **Control hazard：**也稱為branch hazard。當pipeline中的一個指令的執行依賴於前面的分支指令結果，而該分支指令的執行方向未知時，可能導致流水線停滯等待分支結果。這是一種動態產生的風險，可以通過branch prediction來減少其對性能的影響。

**Solving Method:**

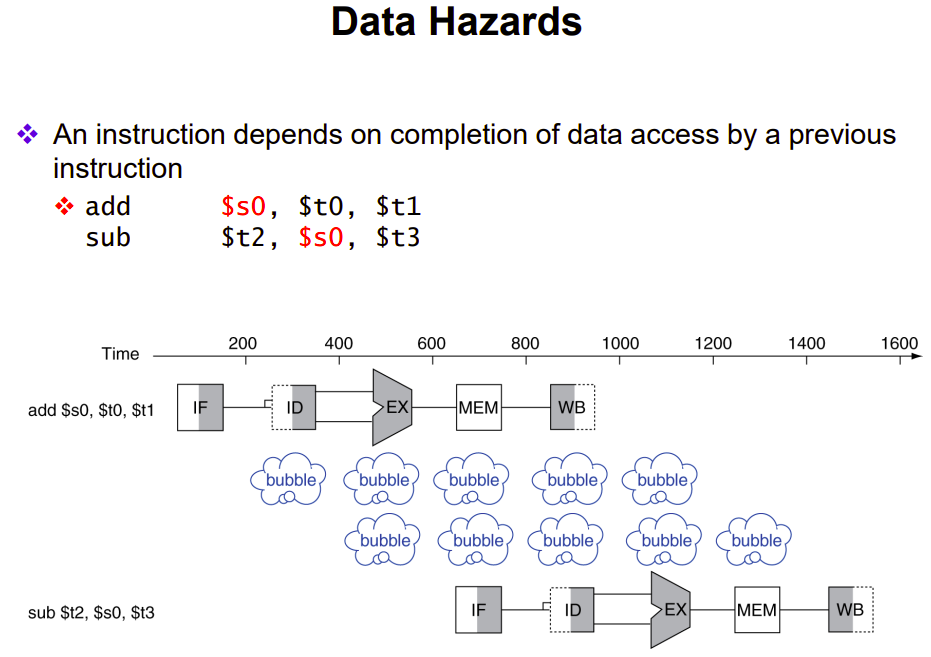
1. **Pipeline stall**：也稱為“Bubble”。這是一種為了解決”Hazard”而啟動的停滯機制。通過暫停流水線中的指令執行，以等待所需資源或數據的到來，從而解決指令間的衝突。
2. **Forwarding:**這是一種通過從內部資料送到下一個指令進而解決Data hazard的方法，而不是等待數據從可見的寄存器或內存中到達。這可以在不引入停滯的情況下，直接將上一個指令計算出的結果轉發給需要該數據的後續指令，提高流水線的效率。
3. **Branch prediction**：這是一種通過假設分支的一個給定結果來解決Branch hazard的方法，並基於這個假設繼續執行，而不是等待確定實際的分支結果。通過預測分支的方向，減少流水線因等待分支結果而引起的停滯，從而提高指令執行效率。
4. **Problem finding:**

以下為再實作pipeline MIP中遇到的幾個問題，包括衝突(Hazard)以及解決方式。以下舉例原先的第四個指令: 4. `0011402A`，經過解析後為: slt $8, $0, $17`。然而接著的下一個指令為: 5. `1100000C`，解析後為:`beq $8, $0, $12`。

這類指令在這次大量出現，因為本身是做bubble sort的功能。這些指令會同時遇到兩種hazard，分別是Data hazard以及 Branch hazard。在這次專題中，使用不同方法去解hazard並且加以比較。

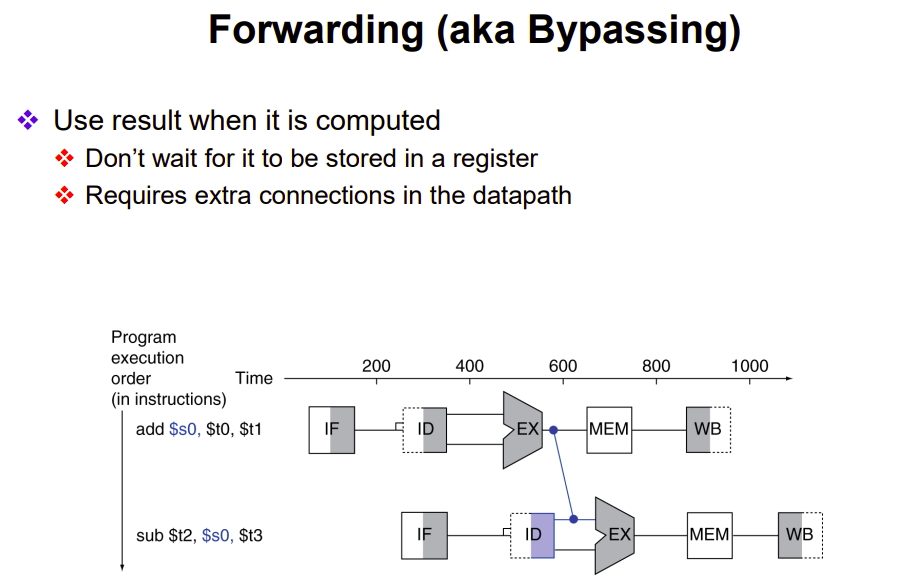
**Insert bubble:**

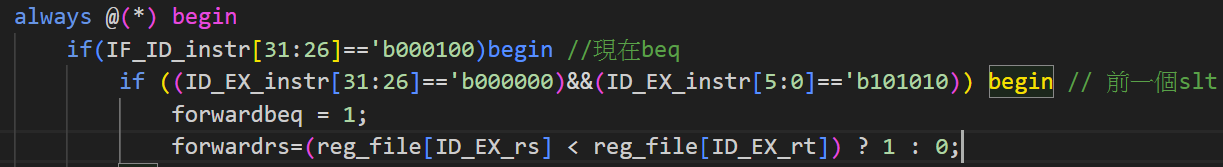
首先的做法是在各個會出現hazard的地方後方加入bubble。這可以很簡單的避免pipeline的延遲特性。使下一個指令要存取寄存器的值的時候，上一個指令已經將正確的值寫回寄存器了。這個做法實現了pipeline CPU。然而，這個也衍生一個很大的問題，也就是大量instruction數量爆增，使得cycle也跟著暴增。如下圖解法。



**Data forwarding:**

Data hazard的部分是在slt 判斷寄存器rt[$0]的大小是否小於寄存器rd[$17]的值，接著再將1或0寫入寄存器rs[$8]的時候，下一個指令beq需要存取寄存器rs[$8]的值。在指令還未寫回寄存器的時候，下一個指令已經讀取到尚未改變的值，這個就是所謂的Data hazard。這會造成所有後續的bubble sorting 都出錯。因此在這裡我們使用data forwarding ，在EXE階段計算完的當下就把值送到到下一個指令beq使其可以讀取到正確的值。如下圖所示。

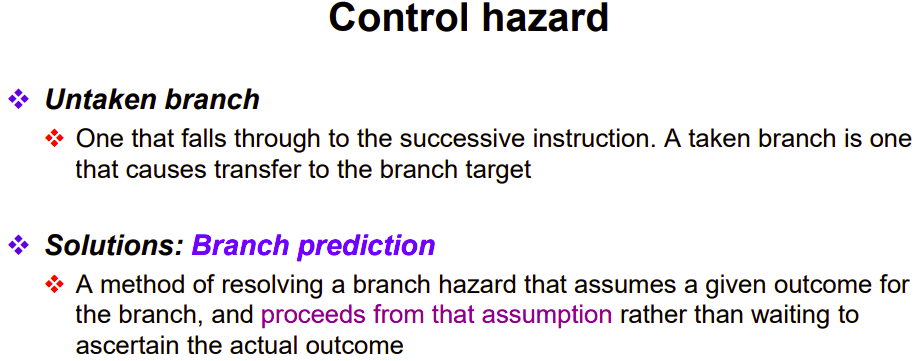


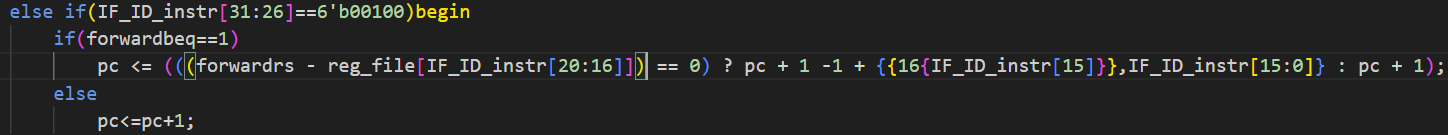


Data forwarding 解決 slt 及 beq 之間的data hazard問題。

**Branch prediction:**

接著為control hazard 也叫做 branch hazard。在前面我們為了讓所有instruction pipeline的程度皆為五個cycle。因此我們也將beq原先只有三個cycle就可做完的指令在後面塞入兩個nop延長到五個cycle。但這也就意味著每個指令跳轉之前會多兩個錯誤的指令，導致一步錯步步錯。因此首先的解決方法為在beq後面無論如何都先加入兩個空的bubble。但這也導致instruction 的數量大增加。導致cycle time 大量增加。接著我將部分改成在第IF\_ID 這個pipeline的時候就就做branch prediction，接著直接跳轉，等於只需要在後方插入一bubble即可。大量減少cycle。如下段的比較所示，降低了3000~4000個 cycle。這也是我認為最需要先處理的hazard部分。因為在指令中大量的在這幾個指令間跳動。因此解決這個便可以減低很多的cycle。這部分的解決方法如下圖。

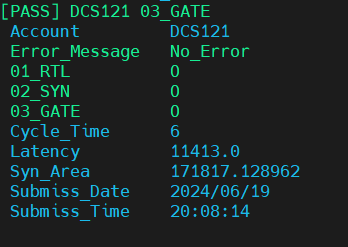
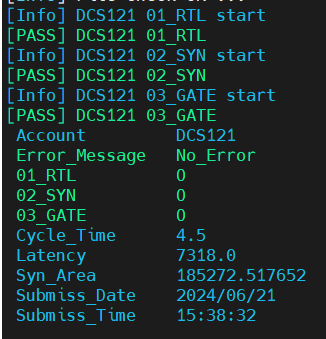




Branch prediction 解決 beq 跳轉延遲的control hazard問題

**Analysis of different ways to solve hazard:**

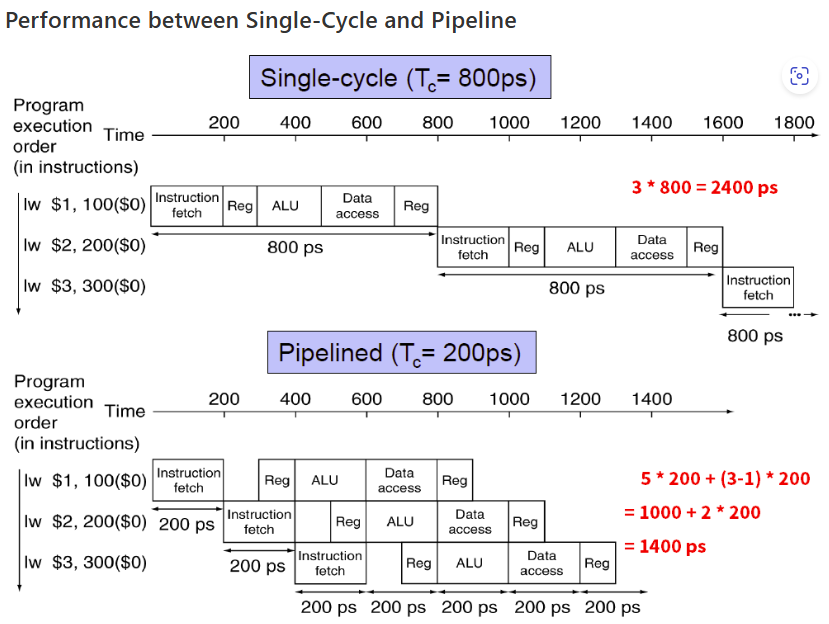
下列為不使用不同方式去解Data hazard所得出的數據比較。在左下側圖是使用在遇到各個不同的hazard的時候，在後方加入兩個bubble使得其可以不被pipeline的延遲特性影響。不過缺點便是cycle暴增，因為多了許多無用的bubble指令。接著為了解決這個問題，我先從波型圖中尋找在那些指令間出現最多的無用指令(bubble)，發現大量的slt 指令接著beq指令。而這些造成許多的無用指令。因此嘗試用data forwarding以及branch prediction 如上述2.Problem finding 的敘述。成功的壓下cycle至7318。而且同時面積並沒有增加太多。

Add bubble with data forwarding and branch prediction

|  |  |  |
| --- | --- | --- |
|  | Add bubble | Data forwarding & branch prediction |
| Cycle time | 6 | 4.5 |
| Cycle | 11413 | 7318 |
| Area | 171817 | 185272 |
| Performance | 11,765,684,526 | 6,101,192,232 |

1. **Comparison with HW04**

****

**HW04 Single Cycle MIPS:**

在HW04中是使用single cycle並且所以指令在單一個cycle中做完。也因為在同一個cycle中要執行得更多，導致cycle time的有個下限。再往下給有可能會出現timing violation。不過相較於pipeline MIPS架構的好處就是面積以及cycle可以壓到最小，因為每個指令皆在一個cycle中便可以完成，也不需要而外的硬體資源來做而外的data forwarding等等。

**Final Pipeline MIPS:**

在pipeline MIPS架構中。將同一個指令拆成五個部分，因此每個cycle所需要的時間相較於single cycle MIPS可以往下壓。不過相對的pipeline要去解決不同指令之間所出現的hazard。因此需要犧牲而外的硬體資源去做forwarding等等。加上會出現部分的bubble也會增加cycle數量。

**Analysis:**

在HW04中，我們採用了Single Cycle MIPS架構，其中所有指令在單一個時鐘週期內完成。這意味著每個指令的執行時間等於cycle time的長度，導致cycle time無法縮短。這種設計簡化了電路結構，減少了硬體資源需求，但也可能導致了高頻時脈的timing violation。

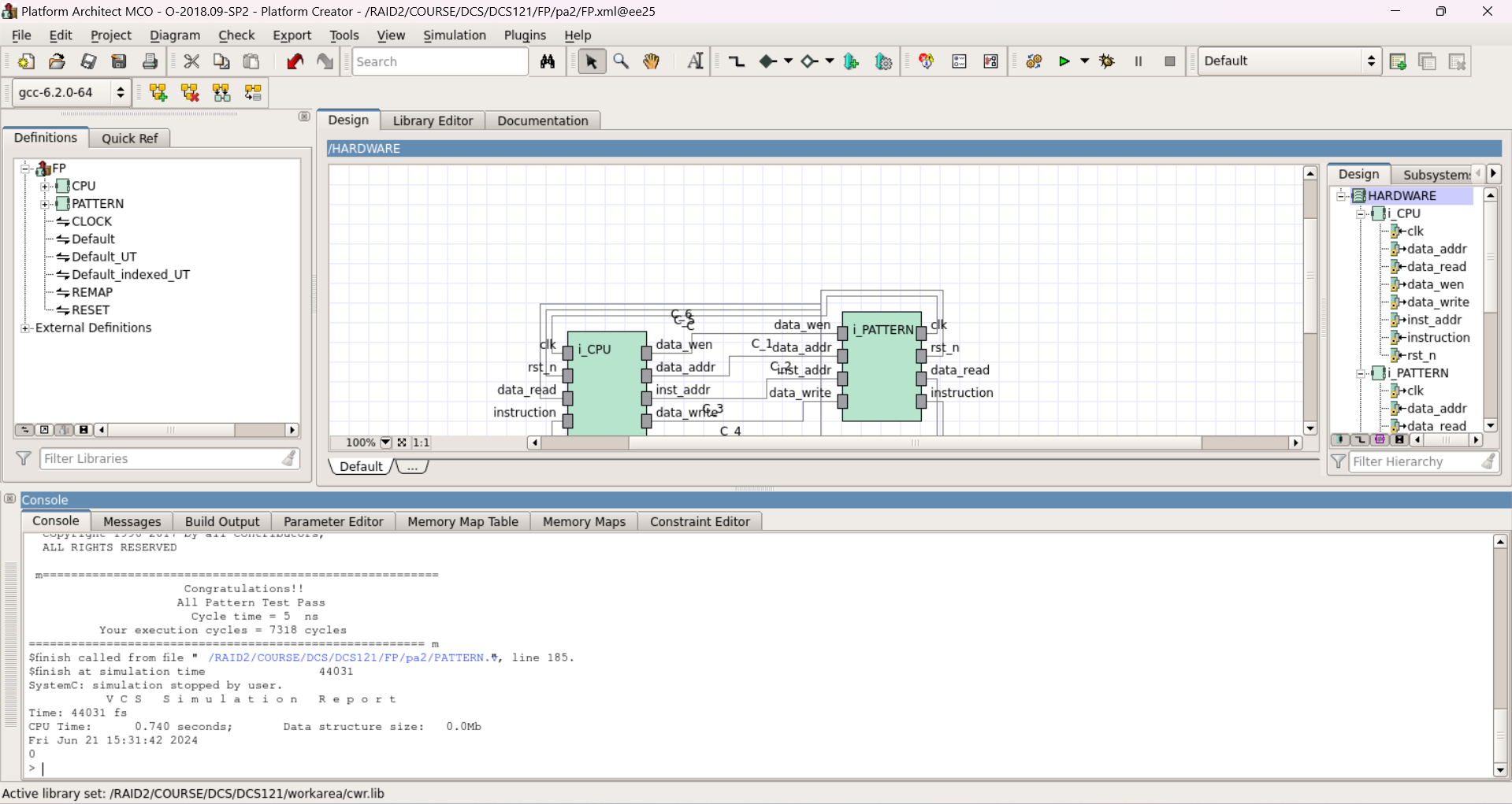
相比之下，我們在最終設計中使用了Pipeline MIPS架構，將每條指令拆分為多個階段，每個階段在單獨的cycle time內完成。這樣，每個指令的執行時間可以分散到多個cycle中，從而減少了cycle time的長度。Pipeline架構提高了指令吞吐量和整體性能，但需要額外的硬體資源來處理data forwarding 和 control hazard問題，還有部分的bubble，影響指令的執行效率。

比較這兩種架構，Single Cycle MIPS架構結構簡單，硬體資源需求少，但cycle time長，且各個指令所需時間不同，可能會因為少數需要較長處理時間的指令而使整體cycle time 都需要增加，對處理時間相對短的指令反而出現許多空等待時間，或是有可能導致在高頻下的timing violation。Pipeline MIPS架構則有更高的指令吞吐量和性能，但設計複雜，需要處理設計處理hazard的架構以及額外的硬體資源。總結來說，Single Cycle MIPS適合簡單系統，而Pipeline MIPS適合高時脈需要高效能的應用。最終設計選擇了Pipeline MIPS架構，並針對hazard進行了優化，最終達到了較好的效果。

|  |  |  |
| --- | --- | --- |
|  | HW4 | Data forwarding & branch prediction |
| Cycle time | 10 | 4.5 |
| Cycle | 5142 | 7318 |
| Area | 138438 | 185272 |
| Performance | 7,118,481,960 | 6,101,192,232 |

1. **PA**

下圖為platform architecture的實作。



1. **REFERENCE**

* [Lecture10\_Pipelined MIPS (2).pdf](file:///C:\Users\Rexx\Downloads\Lecture10_Pipelined%20MIPS%20(2).pdf)
* [Pipelining – MIPS Implementation – Computer Architecture (umd.edu)](https://www.cs.umd.edu/~meesh/411/CA-online/chapter/pipelining-mips-implementation/index.html)
* [Ch.4-3 Pipeline Processor - HackMD](https://hackmd.io/@joanne8826/HkT32O85I)
* [iT 邦幫忙::一起幫忙解決難題，拯救 IT 人的一天 (ithome.com.tw)](https://ithelp.ithome.com.tw/m/articles/10261505)