Digital System Design Final Project

B04901060 電機三 黃文璁

B???????? 電機三 許耀中

B???????? 電機三 陳博彥

**目錄**

* **Baseline**

1. Jump instructions
2. Branch instruction
3. Forwarding
4. Stall controlling
5. Cache
6. Critical path optimization

* **Extensions**

1. Branch prediction
2. L2 cache
3. Multiplier and divider

* **AT values**

1. **Baseline**
2. **Jump instructions**

　　在這次MIPS的實作中，我們需要實作和jump相關的指令共有四個，分別為：J、JAL、JR、JALR。實作這些指令時分別會遇到的問題主要有：

1. **在5個MIPS pipeline stage中的哪個階段處理jump相關指令**
2. **Jump後要對IF進行flush**
3. **JR和JALR需要得到正確的register file值，故要使用forwarding**
4. **JAL和JALR都需要link，但目的地不一定相同**

　　以下分別討論這些問題的處理方式。

1. **Which stage?**

　　觀察四個指令的特性，可以發現J和JAL和當前register file的值無關，故實際上可以在IF stage就完成program counter的jump，但考慮到JAL需要寫入register file，可能會使control稍微複雜一點，故我們實作時仍統一將四個jump指令在ID階段處理，缺點是會讓J和JAL指令多一個cycle才能完成，但實作上單純不少。

1. **IF flush**

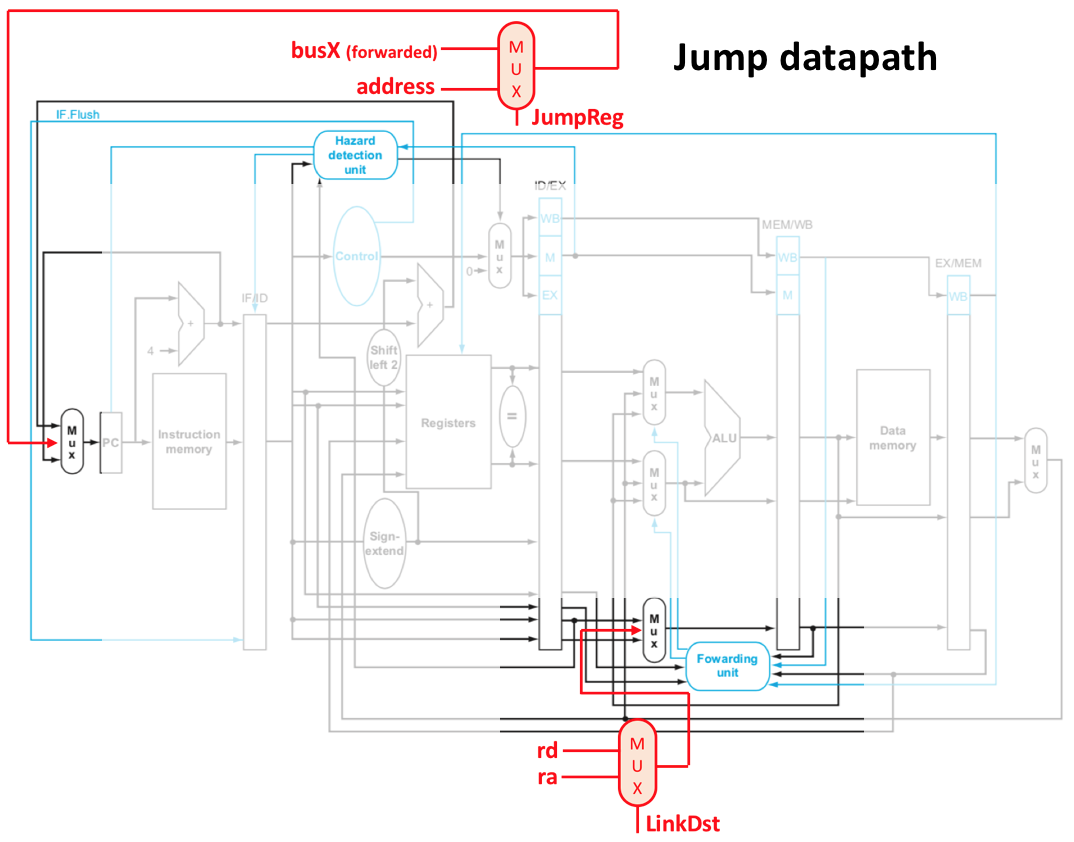
　　由於我們在ID才處理jump，故IF會多讀到下一個address的指令，但實際上不應該執行，所以我們需要flush掉這個指令，當ID級的control unit發現這個指令為四種jump之一，就會給flush訊號到IF級。實作上也很簡單，將傳給IF級的instruction設為32'b0即可。

1. **JR/JALR should use forwarded values**

　　由於JR/JALR要根據register值來決定jump address，故需要正確的register值，由於pipelined MIPS會有hazard的問題，故需要使用一個forwarding unit來解決，這部分會在後文中提到。

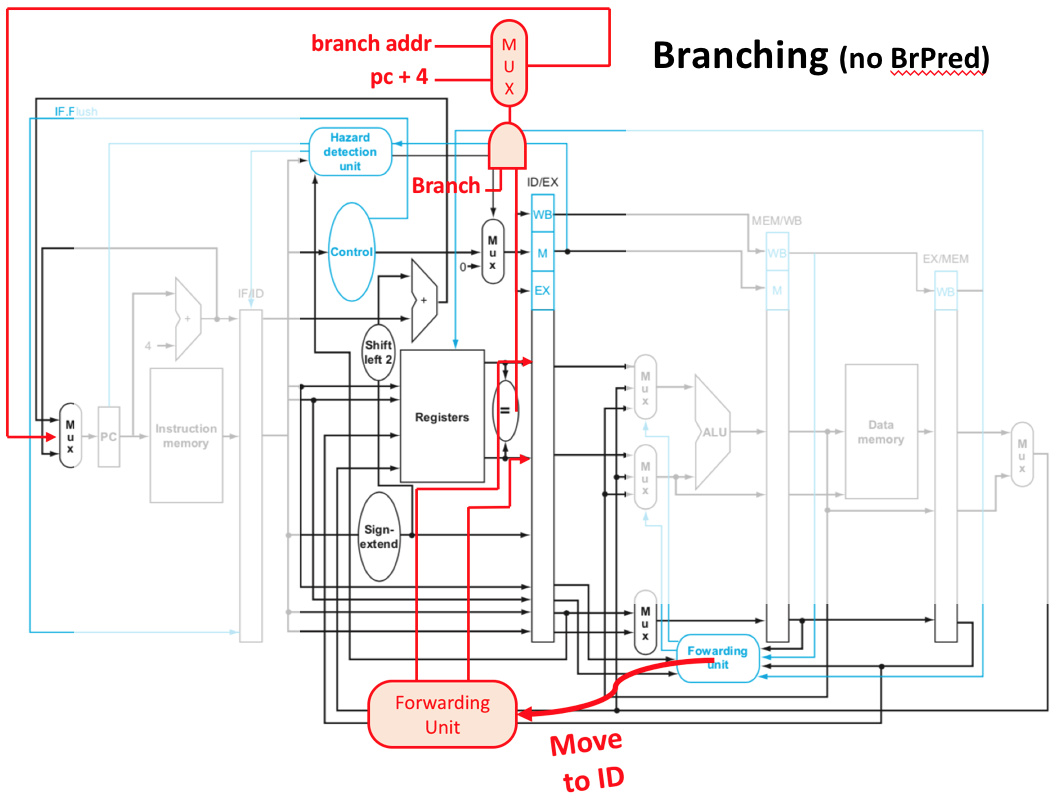
1. **Link destination of JAL/JALR**

　　JAL和JALR需要將address寫回register file，但課本上的設計沒有考量到這兩個指令，故需要加入額外的control signal和MUX來實現。我們使用額外的兩個控制訊號LinkRA和LinkRD來決定write back的目的地。整體來說，jump相關指令會增加的MUX如下圖：



1. **Branch instruction**

　　這次實作的branch指令只有一個BEQ，由於我們也在ID級處理BEQ，故基本上處理方式和前面的jump指令相同，同時由於BEQ也需要用到register file的值，故同樣要使用forwarding。為了方便起見，我們將EX、MEM、WB三級的值都forward到ID級，而不像課本forward到EX級。包含BEQ的架構改變如下：



1. **Forwarding**

　　Forward的條件基本上和課本相同，只不過多了從EX級forward到ID級的條件，所有條件如下：

　　ForwardEX\_X = EX\_RegWrite & (EX\_RW!=0) & (EX\_RW==ID\_RX);

　　ForwardEX\_Y = EX\_RegWrite & (EX\_RW!=0) & (EX\_RW==ID\_RY);

　　ForwardMEM\_X = MEM\_RegWrite & (MEM\_RW!=0) & (MEM\_RW==ID\_RX);

　　ForwardMEM\_Y = MEM\_RegWrite & (MEM\_RW!=0) & (MEM\_RW==ID\_RY);

　　ForwardWB\_X = WB\_RegWrite & (WB\_RW!=0) & (WB\_RW==ID\_RX);

　　ForwardWB\_Y = WB\_RegWrite & (WB\_RW!=0) & (WB\_RW==ID\_RY);

1. **Stall**

　　在這次的實作中主要有三種情況會發生stall，可以用一個controller來判斷是那些stage需要stall或flush，分別如下：

1. **ICACHE stall**

Flush IF

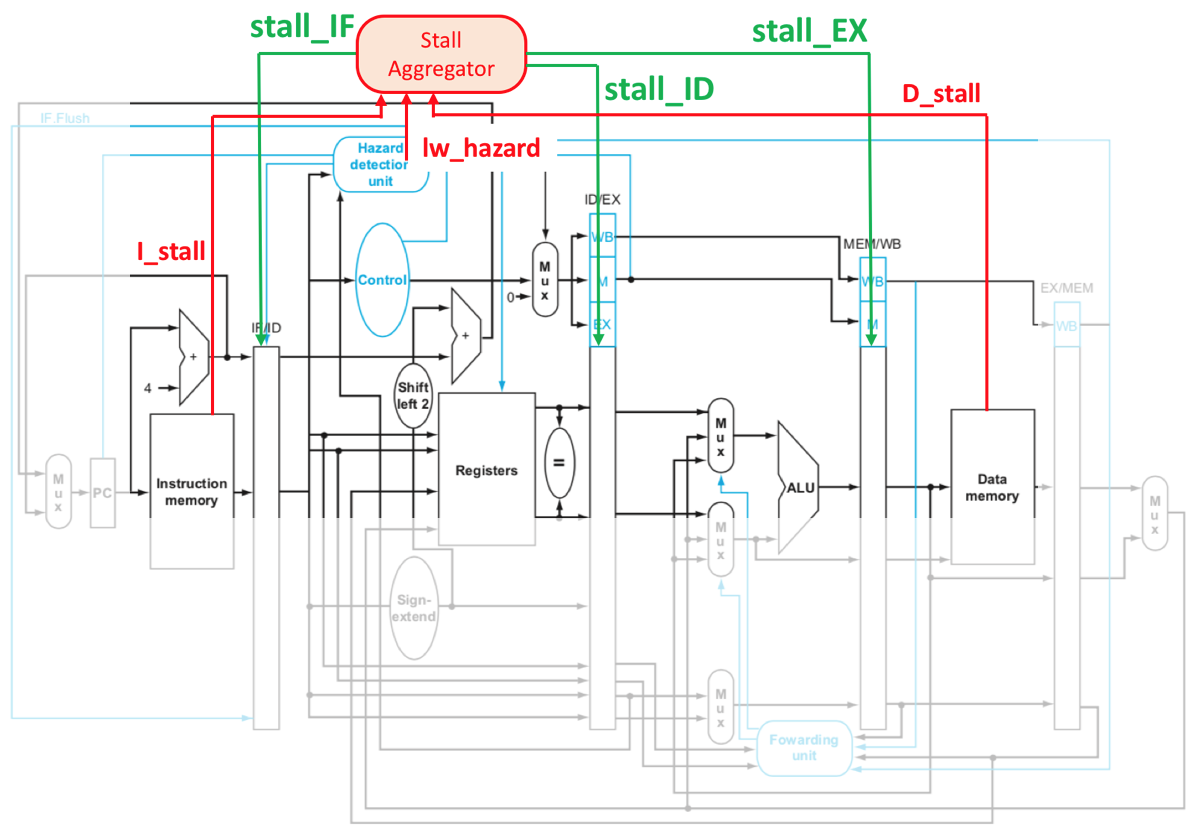
1. **load-use hazard**

Stall IF/ID, flush ID

1. **DCACHE stall**

Stall IF/ID/EX stage

於是我們新增了一個controller來控制各級的stall，如下圖：



1. **Cache**

　　在現代的處理其中，快取一直是很重要的一環，所以在這份報告中我們也會對cache和L2 cache進行較多討論。首先是這次baseline所使用的cache規格：

* **ICACHE**

1. **Mode**: Read-only
2. **Size**: 32 words
3. **Placement**: Direct mapped

* **DCACHE**

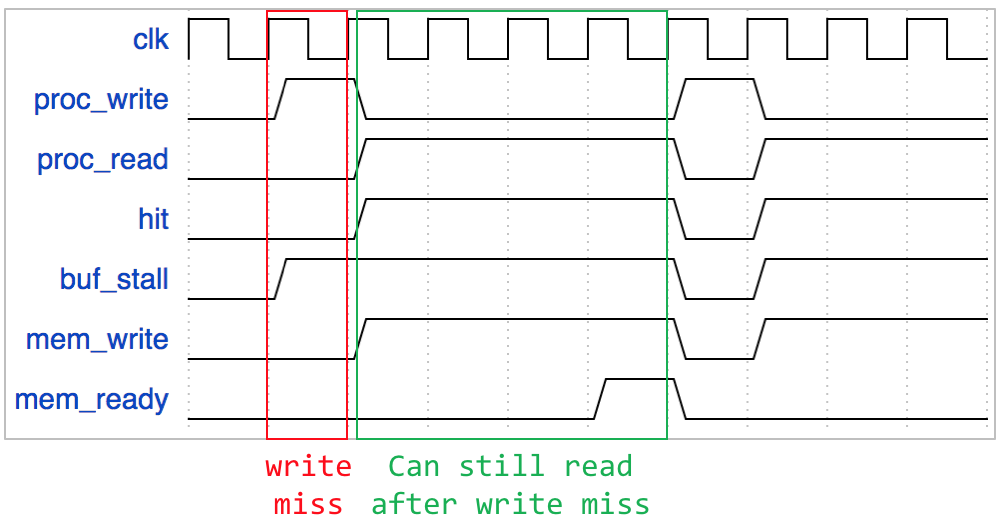
1. **Mode**: Read/write
2. **Size**: 32 words
3. **Placement**: Direct mapped
4. **Write** **policy**: Write back + write buffer

以下分別針對幾個設計細節來介紹：

1. **Write back + write buffer**

　　這次MIPS所使用的slow\_memory.v會傳回mem\_ready訊號，但只會維持一個cycle，若我們讓他在背景同時寫入而不理會的話會比較難設計state來得知目前記憶體是否可用，所以會需要一個write buffer，專門處理寫入，write buffer的重點是會將mem\_ready轉換成stall的方式輸出到cache，所以cache可以很容易知道目前是否能進行寫入。

　　以下波形展示了這個概念：



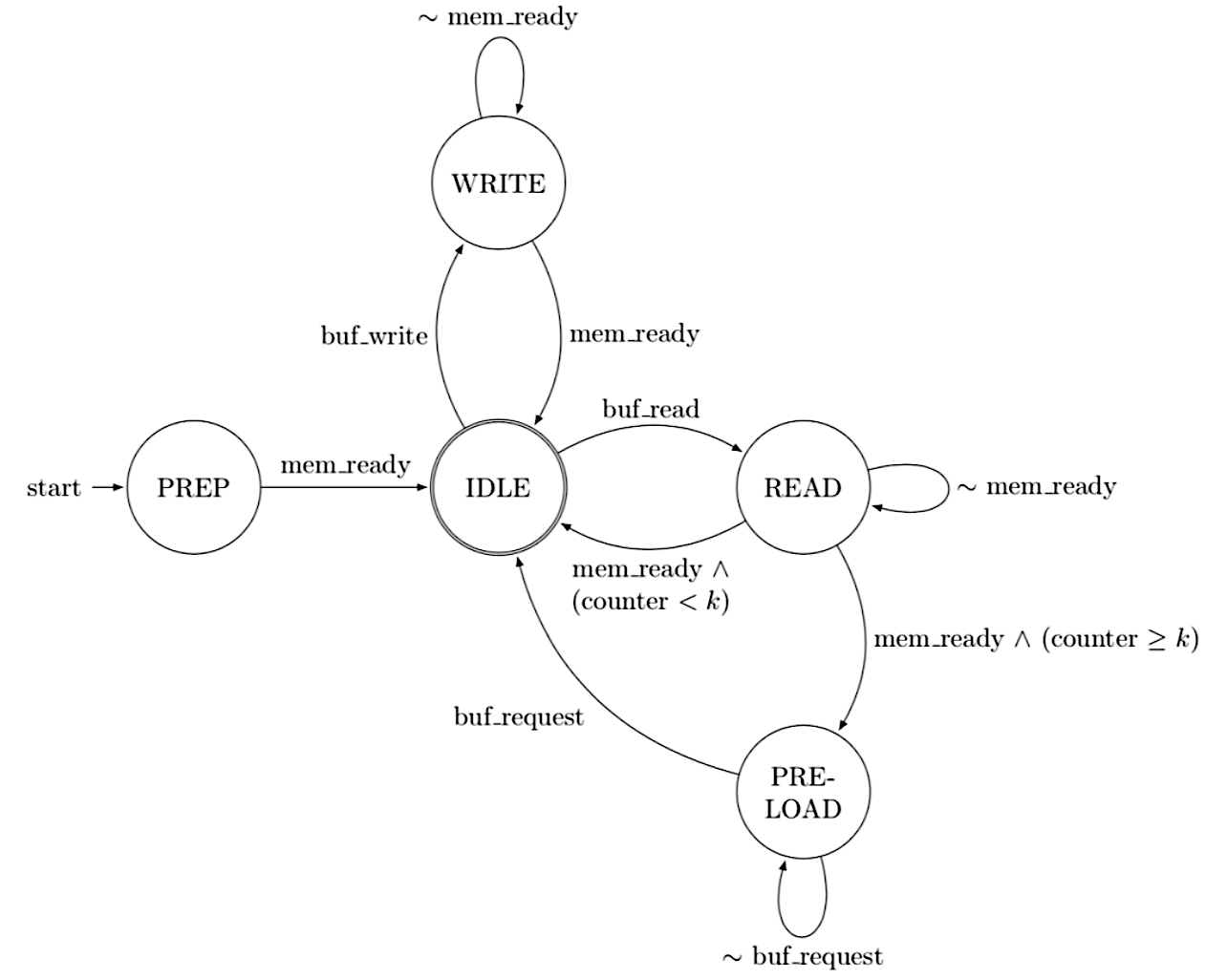
　　利用這個技巧，假如沒有快速連續的write miss的話，可以完全將memory在寫入後等待mem\_ready的latency隱藏起來。

1. **Read preloader (RP)**

　　讀取操作則不像寫入可以被隱藏起來，但還是有些技巧可能可以提升效能，例如在作業四的循序讀取時，可以設計一個FSM，使得：

*當連續讀取k個連續的地址後，預先讀取下一個地址。*

這個概念若畫成FSM的話，如下（FSM使用LaTeX+TikZ繪製）：



值得注意的是，這個方法在HW4的效果非常顯著，比較如下：

1. **Without RP**: 6164 cycles (miss penalty: 4 cycles)
2. **With RP**: 4172 cycles (-32.3%) (miss penalty: **0.2** cycles)

注意到有read preloader的情況下，read miss penalty幾乎為0，也就是在作業四的大量循序讀取條件下，read preloader的效果非常好，但是在這次baseline的程式中則不是如此，比較如下：

1. **Without RP**: 2065 cycles
2. **With RP**: 2074 cycles (+0.4%)

可以發現由於read preloader的一些overhead，導致有read preloader時的效果反而降低，我們認為主要原因有兩個：

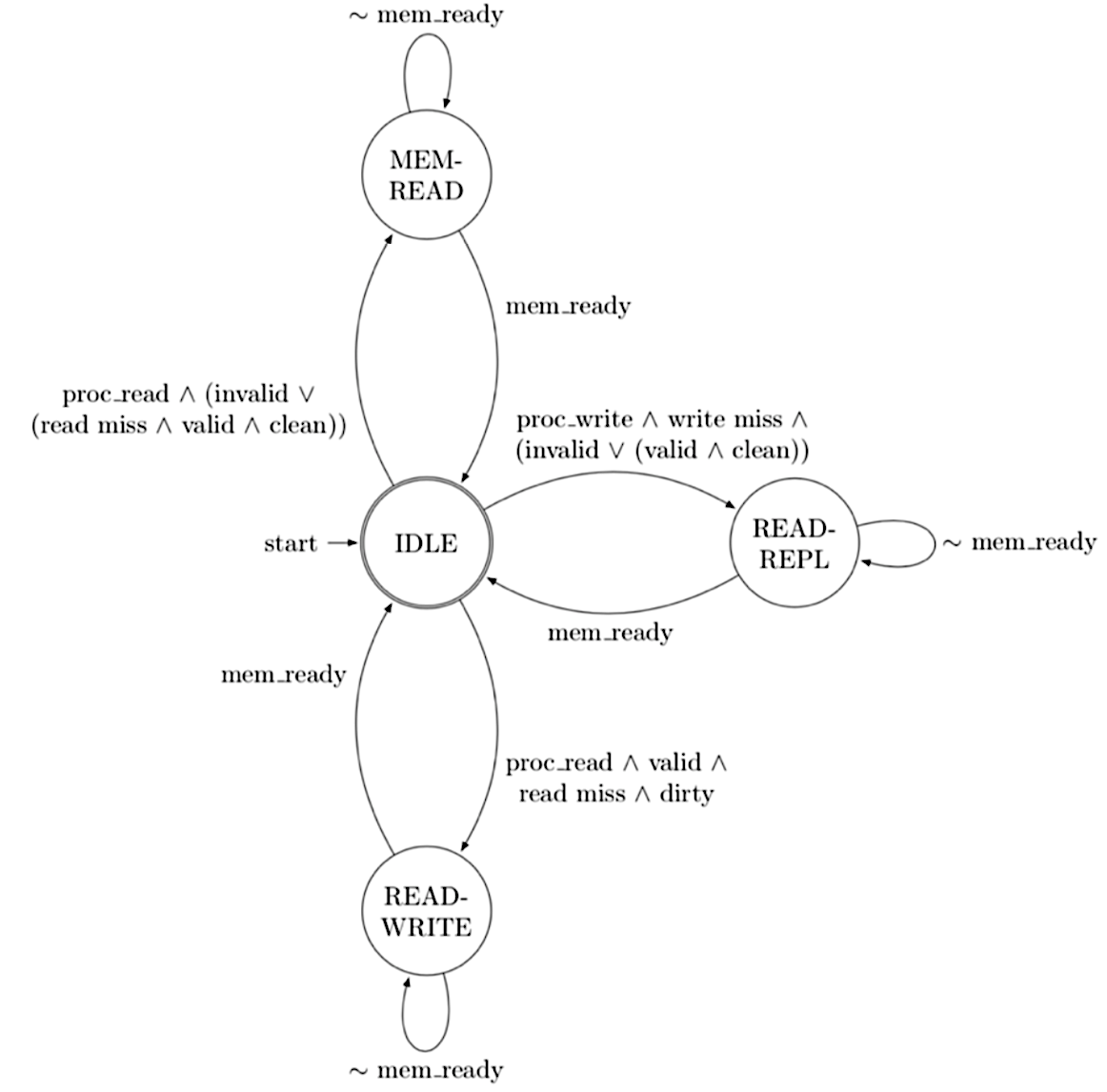
1. **程式很小，ICACHE幾乎不會有read miss**

經過測試，baseline程式的read miss rate大約為0.9%，故幾乎不會有read miss，也就不需要經常從記憶體讀值。

1. **Final使用的slow\_memory.v比作業四的memory.v還來得慢。**
2. **Cache FSM Design**

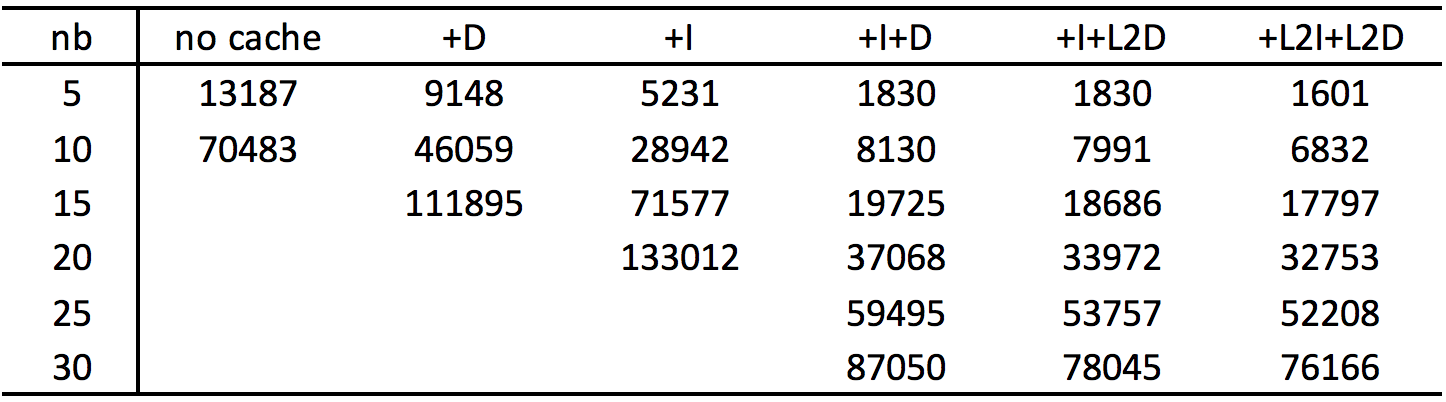
　　這次使用的cache對MIPS而言是Mealy machine，所以當hit時，可以直接以combinational的方式得到值，而不用等待下一個cycle，另外stall也為combinational。優點是cycle數較少，缺點則是可能會有glitch產生，且可能讓critical path變長。

以下為這次Cache的FSM，大致和作業四使用的相同：

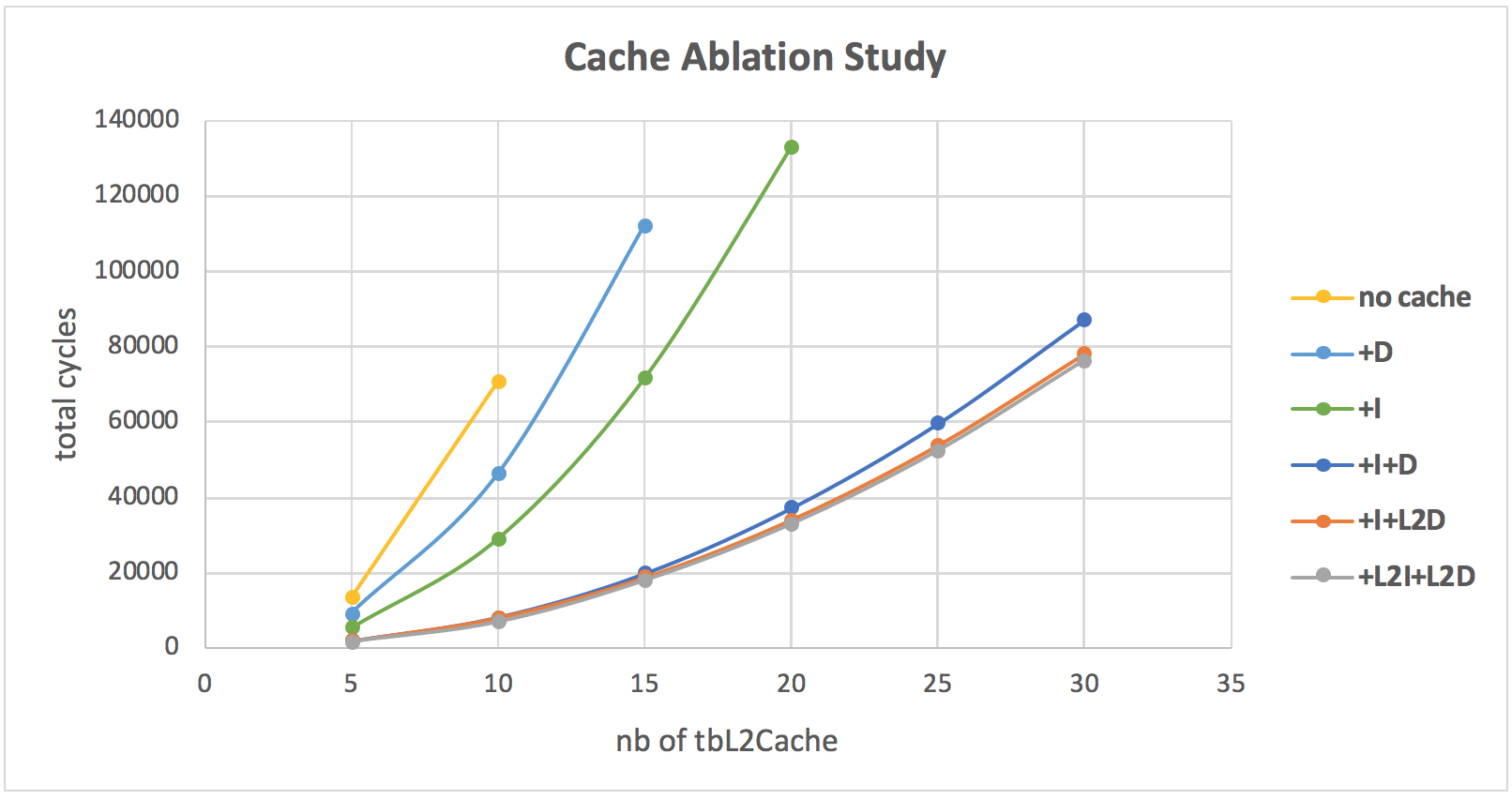


1. **Cache ablation study**

　　比較ICACHE、DCACHE的有無對於整體效能的影響，如下表：



　　表中使用了L2Cache的testbench generator，生成nb=5～30的程式進行比較，結果使用RTL模擬的cycle數。其中 +D 代表使用 DCACHE，而 +I 代表使用 ICACHE，+L2D 和 +L2I 分別都是使用了L2 Cache，將會在extension的部分再進行討論，若畫成曲線則如下圖：

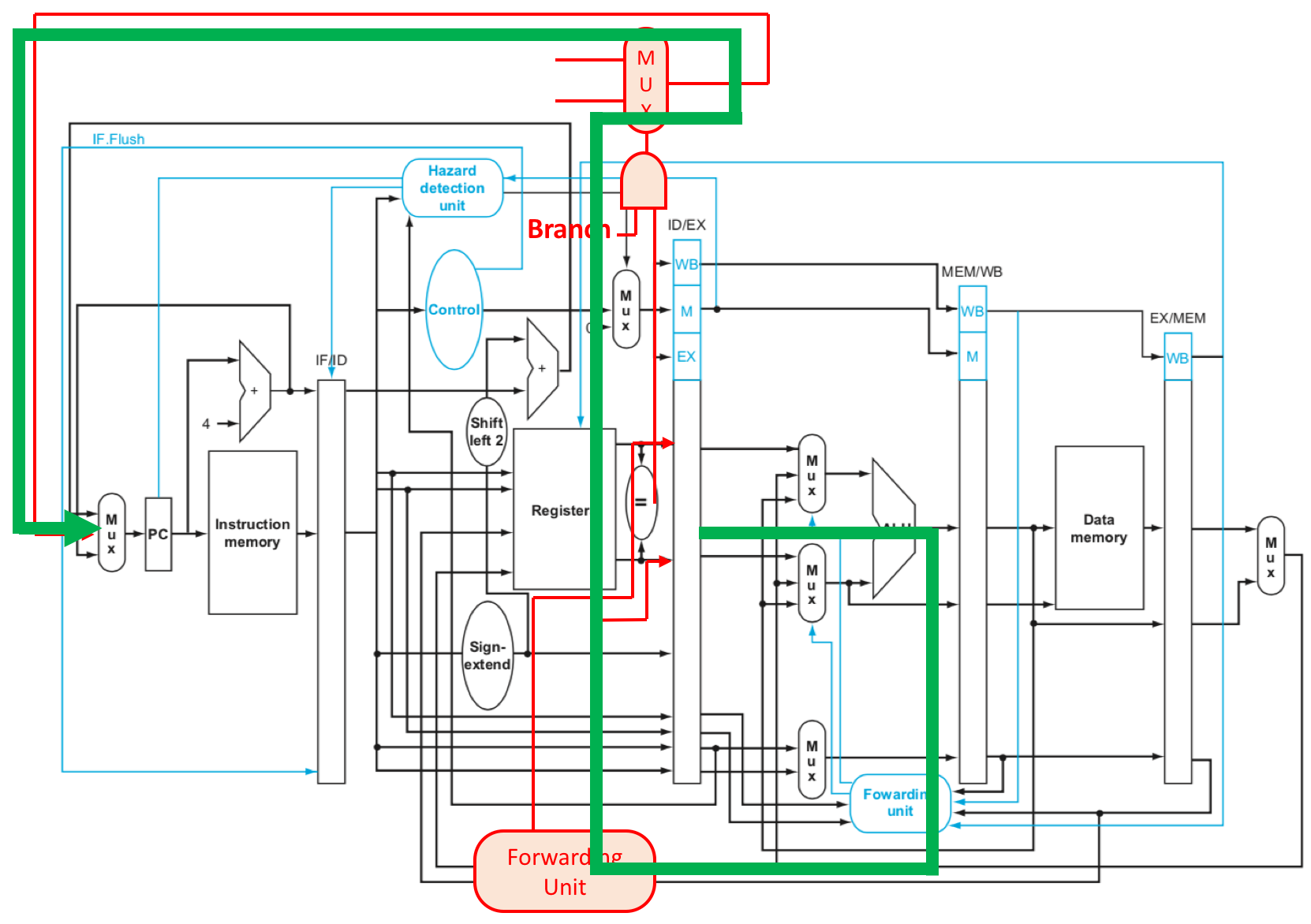


　　首先觀察沒有任何cache的版本，注意到在nb=10的時候其所需要的cycle數足足有加上ICACHE和DCACHE版本的9倍左右，是非常大的差異，故可以顯示出cache的重要性。

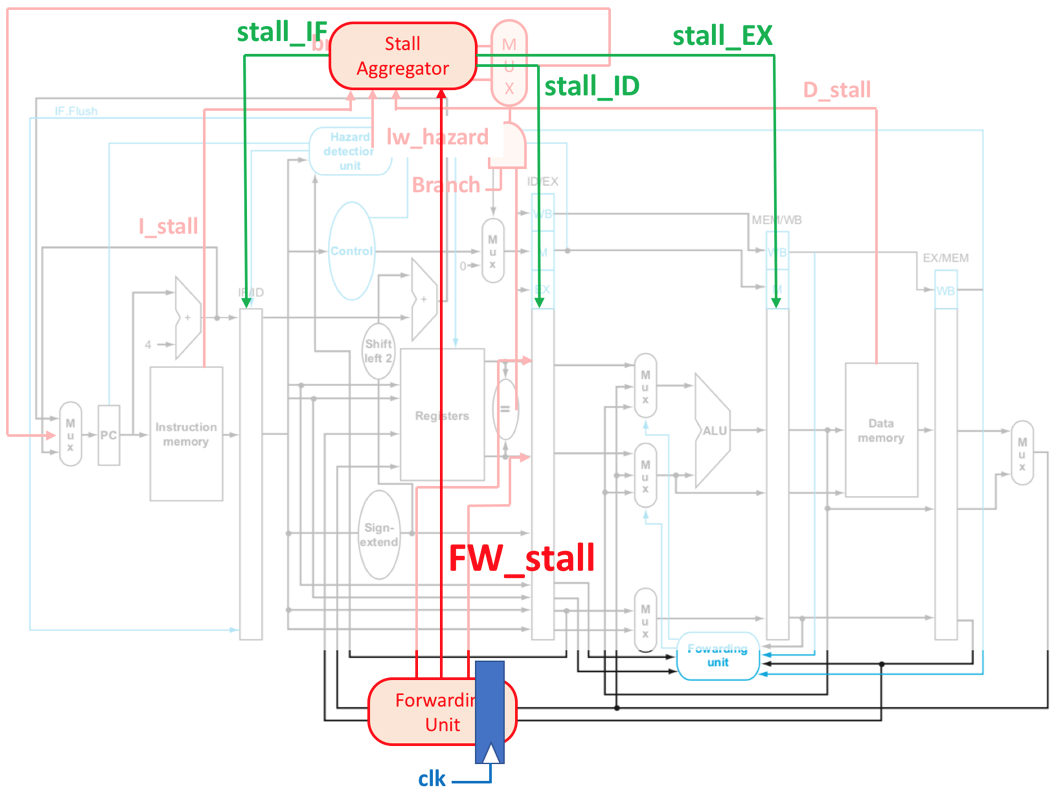
　　另外比較的事ICACHE和DCACHE的重要性，由於在這次程式中data memory的讀寫並不算特別多，又因為每個cycle都需要進行一次instruction fetch，故單獨比較的話ICACHE影響比DCACHE來得大。可以從 +I和 +D的曲線觀察。但值得注意的是 +D和 +I效能仍然遠低於同時使用兩者，也就是圖表中的 +I+D，故可以得到結論：和memory讀寫有關的部分都應該加上cache。至於 +L2I和 +L2D，也就是L2 cache相關的比較則請參考extensions的討論。

1. **Critical path optimization**

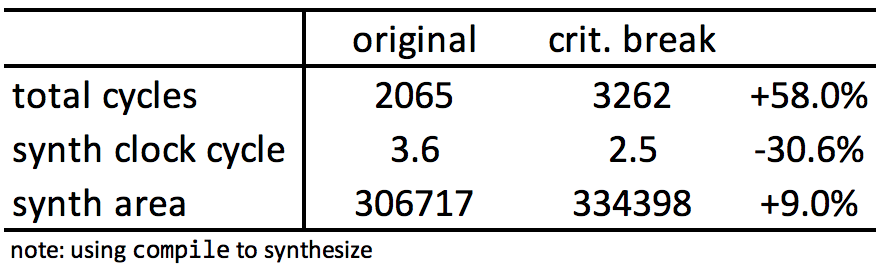
　　合成後可以大略得知這次design的critical path，如下圖：



　　試著將critical path切開，方法為在Forwarding unit的輸出加上一級register，這樣雖然必須晚一個cycle才能得到forwarded value，但可以讓critical path變短，修改後需要額外的stall control，架構如下：



**結果比較：**



　　其中synth clock cycle是合成時可以達到slack=0的最小cycle。可以發現critical path確實變短了，也就是能以更小的cycle合成。但是這次baseline沒有採用這種做法，是由於從上表中可以得知，對結果的AT值並沒有改進，故不採用。

1. **Extensions**