**Digital System Design Final Project**

B04901060 電機三 黃文璁
B04901042 電機三 許耀中
B04901067 電機三 陳博彥

---

目錄



---

## 1. Baseline

### 1) 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：

**a. 在 5 個 MIPS pipeline stage 中的哪個階段處理 jump 相關指令**

**b. Jump 後要對 IF 進行 flush**

**c. JR 和 JALR 需要得到正確的 register file 值，故要使用 forwarding**

**d. 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。

**a. 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。

**b. IF flush**

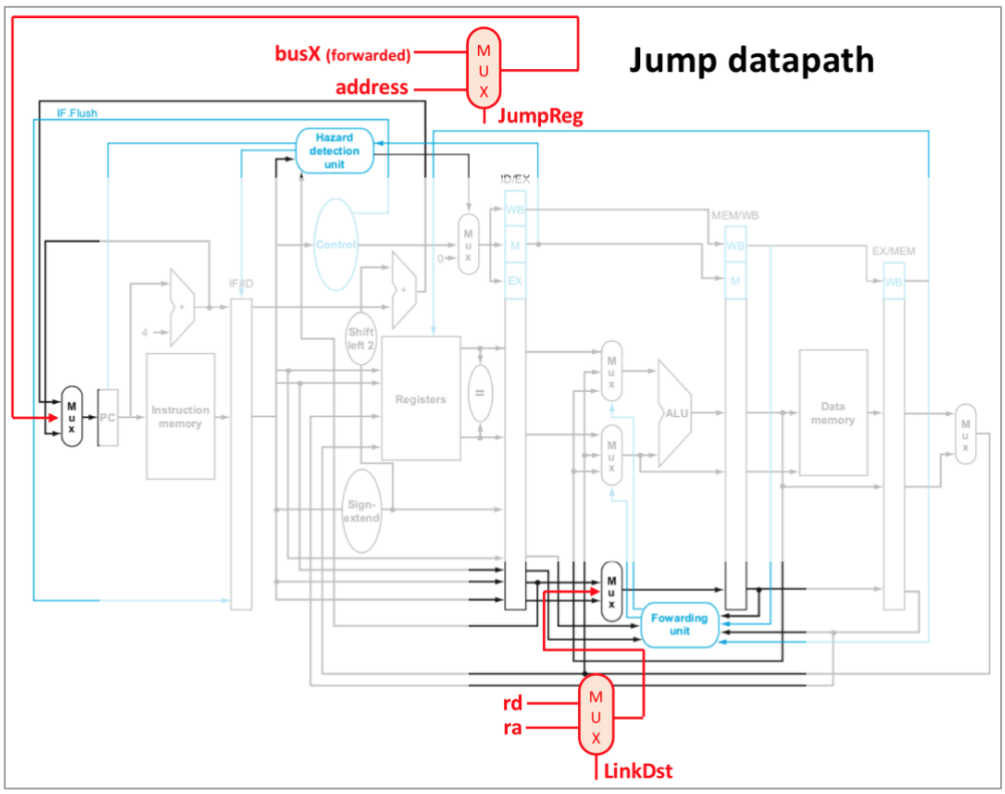
由於我們在 ID 才處理 jump，故 IF 會多讀到下一個 address 的指令，但實際上不應該執行，所以我們需要 flush 掉這個指令，當 ID 級的 control unit 發現這個指令為四種 jump 之一，就會給 flush 訊號到 IF 級。實作上也很簡單，將傳給 IF 級的 instruction 設為 `32'b0` 即可。

**c. 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。

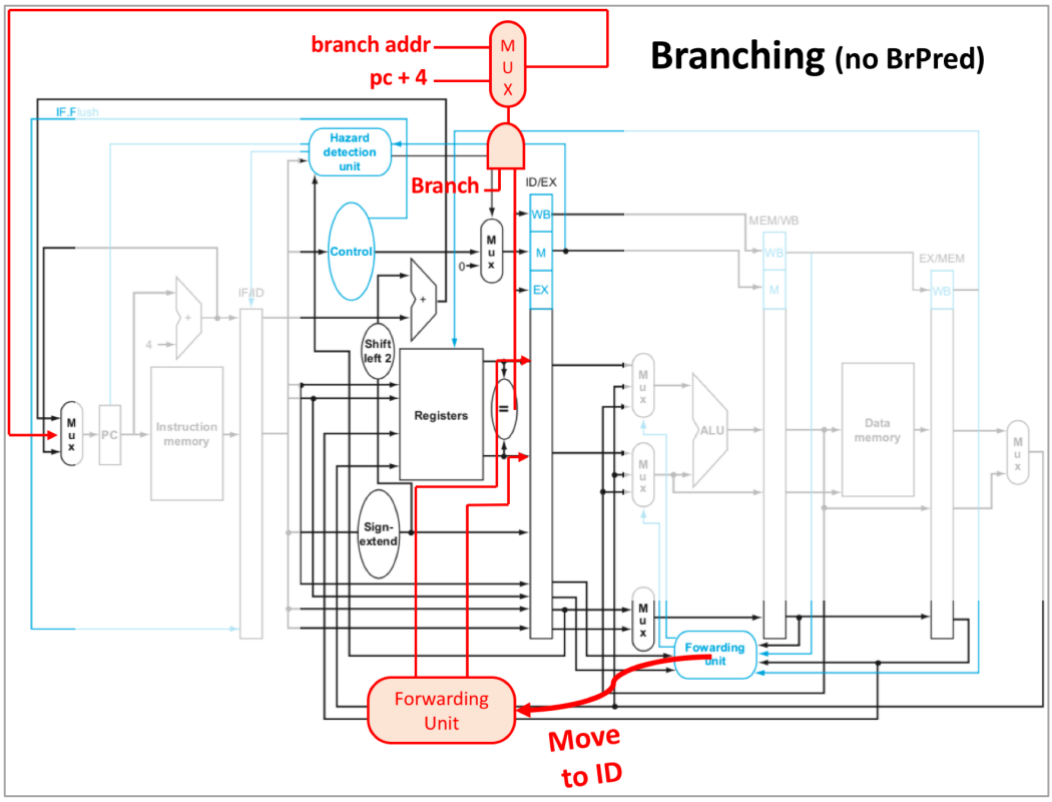
**d. Link destination of JAL/JALR**

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：




**2) 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：




**3) 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：

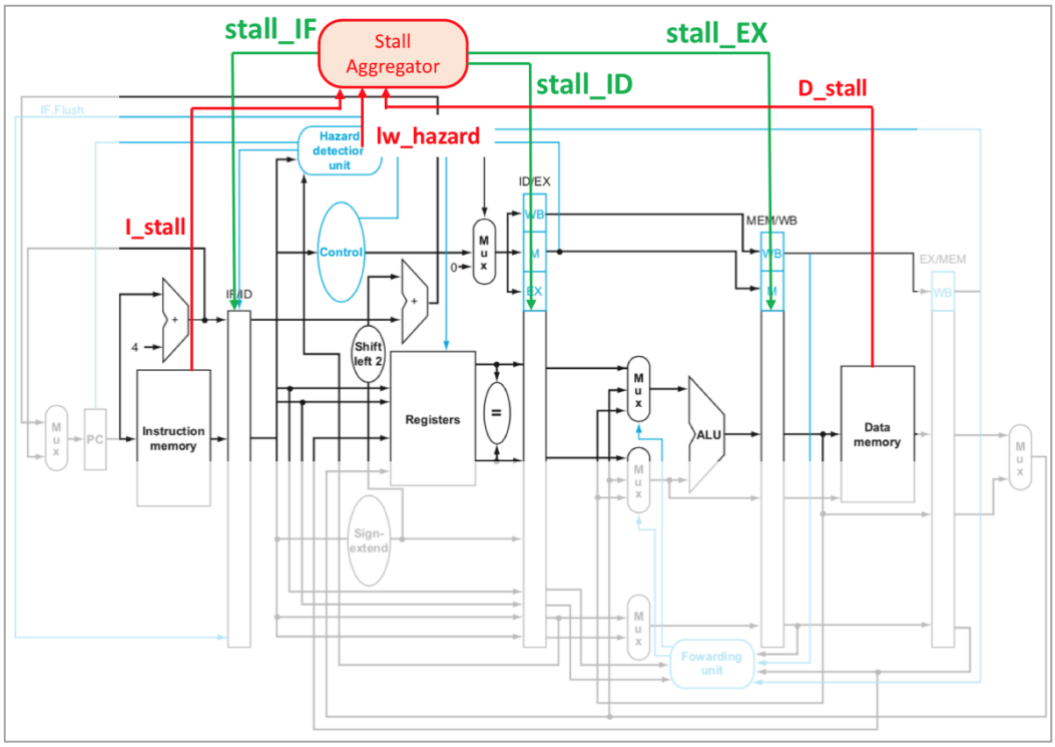
```
ForwardEX_X  = EX_RegWrite  & (EX_RW!=0)  & (EX_RW==ID_RX);
ForwardEX_Y  = EX_RegWrite  & (EX_RW!=0)  & (EX_RW==ID_RY);
ForwardMEM_X = MEM_RegWrite & (MEM_RW!=0) & (MEM_RW==ID_RX);
ForwardMEM_Y = MEM_RegWrite & (MEM_RW!=0) & (MEM_RW==ID_RY);
ForwardWB_X  = WB_RegWrite  & (WB_RW!=0)  & (WB_RW==ID_RX);
ForwardWB_Y  = WB_RegWrite  & (WB_RW!=0)  & (WB_RW==ID_RY);
```

**4) 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：

a. **ICACHE stall**
   Flush IF
b. **load-use hazard**
   Stall IF/ID, flush ID
c. **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：




**5) Cache**

在現代的處理器中，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

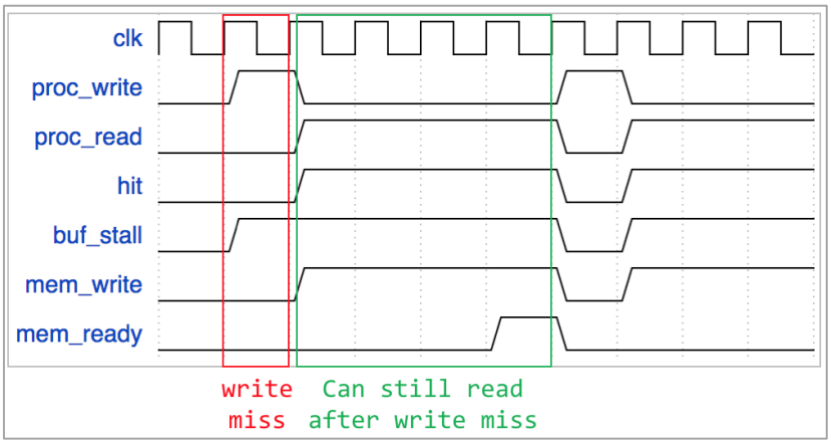
- **DCACHE**
  1) **Mode**: Read/write
  2) **Size**: 32 words
  3) **Placement**: Direct mapped
  4) **Write policy**: Write back + write buffer

以下分別針對幾個設計細節來介紹：

**a. 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：



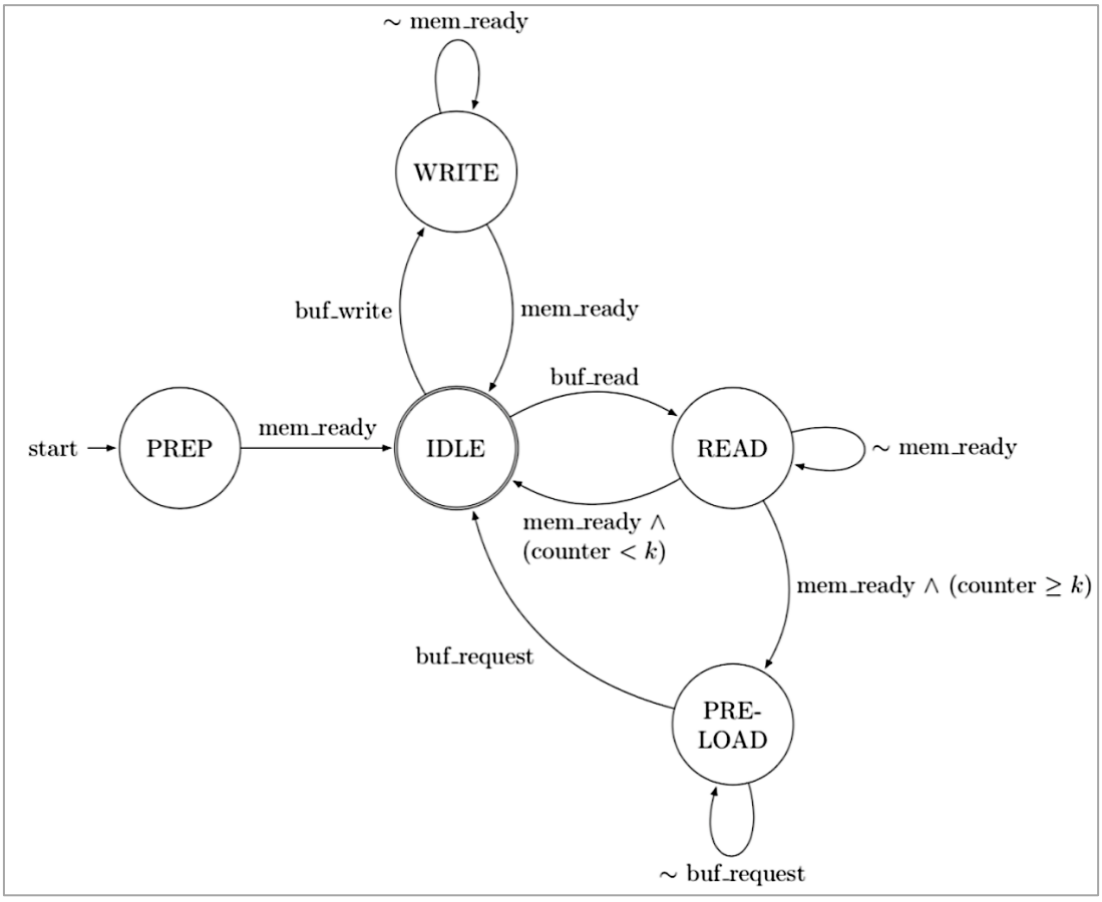

利用這個技巧，假如沒有快速連續的 read/write miss 的話，可以完全將 memory 在寫入後等待 `mem_ready` 的 latency 隱藏起來。

**b. 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：

(a) **Without RP**: 6164 cycles (miss penalty: 4 cycles)
(b) **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：

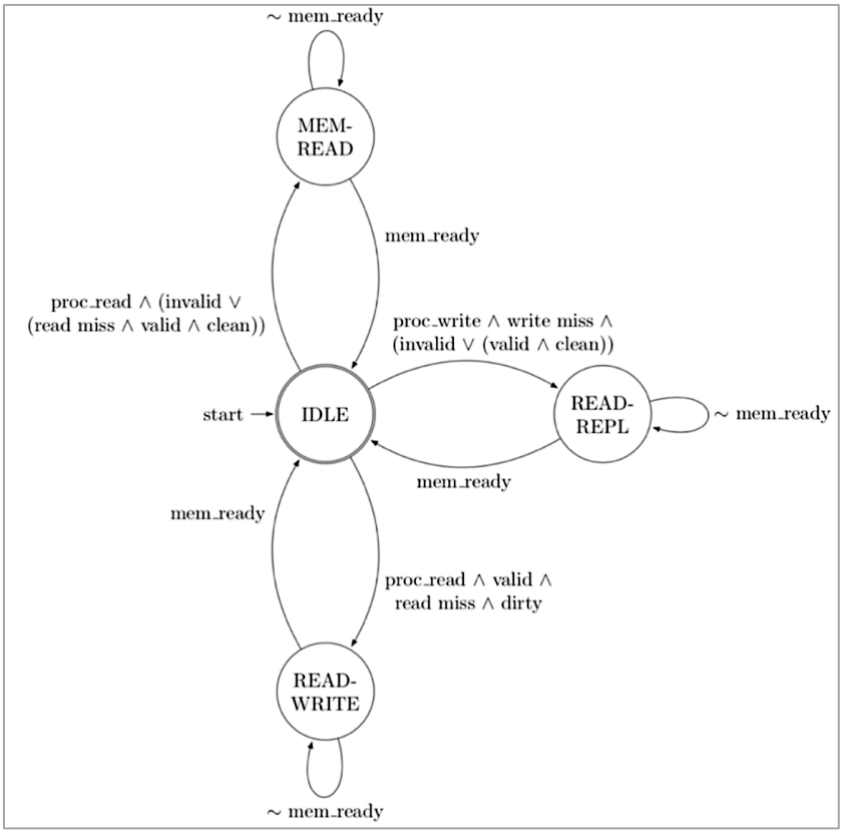
(a) **Without RP**: 2065 cycles
(b) **With RP**: 2074 cycles (+0.4%)

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。
2. **Final 使用的 `slow_memory.v` 比作業四的 `memory.v` 還來得慢。**

**c. 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：

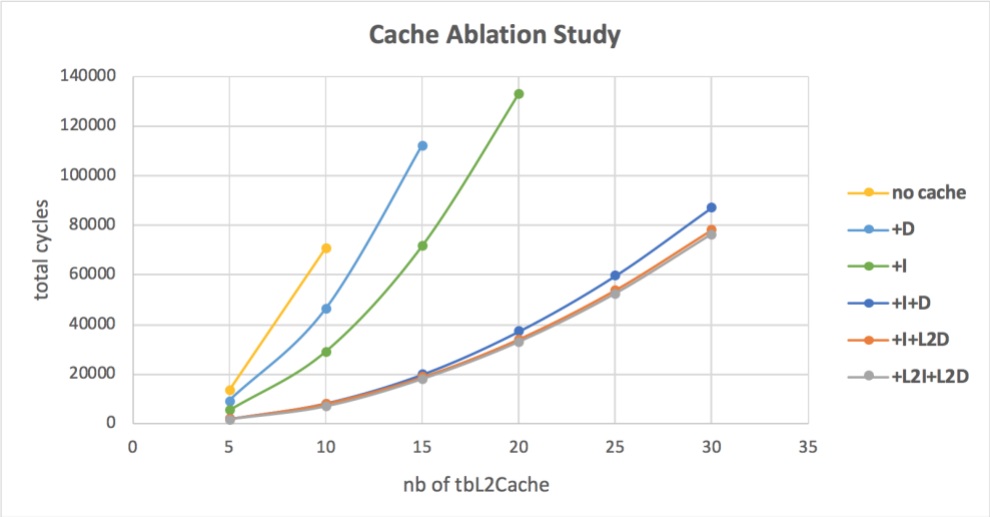


**d. Cache ablation study**

比較 ICACHE、DCACHE 的有無對於整體效能的影響，如下表：

| nb | no cache | +D | +I | +I+D | +I+L2D | +L2I+L2D |
|---|---|---|---|---|---|---|
| 5 | 13187 | 9148 | 5231 | 1830 | 1830 | 1601 |
| 10 | 70483 | 46059 | 28942 | 8130 | 7991 | 6832 |
| 15 | | 111895 | 71577 | 19725 | 18686 | 17797 |
| 20 | | | 133012 | 37068 | 33972 | 32753 |
| 25 | | | | 59495 | 53757 | 52208 |
| 30 | | | | 87050 | 78045 | 76166 |

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：

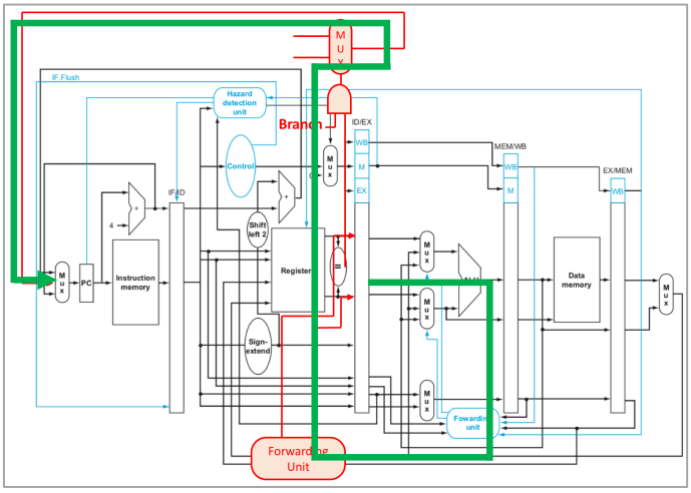



首先觀察沒有任何 cache 的版本，注意到在 nb=10 的時候其所需要的 cycle 數足足有加上 ICACHE 和 DCACHE 版本的 9 倍左右，是非常大的差異，故可以顯示出 cache 的重要性。

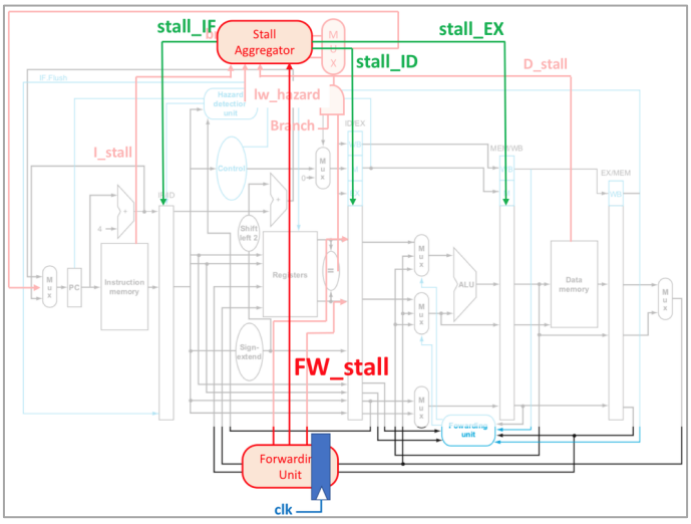
另外比較的是 ICACHE 和 DCACHE 的重要性，由於在這次程式中 data memory 的讀寫並不算特別多，又因為每個 cycle 都需要進行一次 instruction fetch，故單獨比較的話 ICACHE 影響比 DCACHE 來得大。可以從 +I 和 +D 的曲線觀察。但值得注意的是 +D 和 +I 效能仍然遠低於同時使用兩者，也就是圖表中的 +I+D，故可以得到結論：和 memory 讀寫有關的部分都應該加上 cache。至於 +L2I 和 +L2D，也就是 L2 cache 相關的比較則請參考 extensions 的討論。

**e. 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：**

| | original | crit. break | |
|---|---|---|---|
| total cycles | 2065 | 3262 | +58.0% |
| synth clock cycle | 3.6 | 2.5 | -30.6% |
| synth area | 306717 | 334398 | +9.0% |

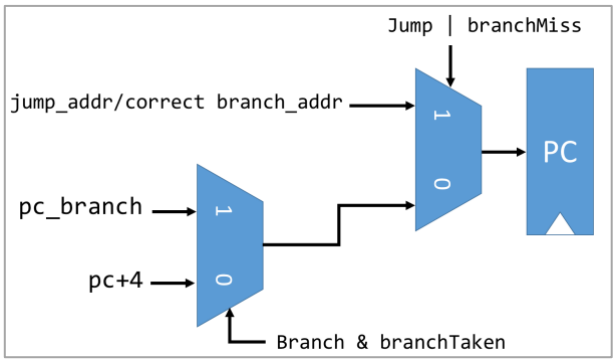
note: using `compile` to synthesize

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知，對結果的 AT 值並沒有改進，故不採用。

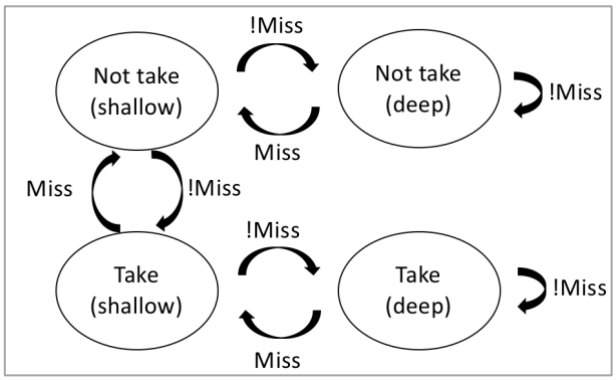
## 2. Extensions

### 1) Branch Prediction

Branch prediction 基本上就是要預測 branch taken/not taken，若預測錯誤的話則需要回覆成正確的地址，若以 MUX 來表示 branch address 的 data path 如下圖：




實作上我們用來預測 branch 的方法為簡單的 2-bit saturating counter，其 FSM 如下，基本概念為連續兩次成功或失敗，才會使 branch predictor 的預測改變：




以下實驗比較的是調整三個 tb generator 參數，在 RTL 模擬下的 cycle 數，我們同時比較了 1-bit 和 2-bit counter 的差別：

| (a, b, c) | no BrPred | BrPred (2b) | | BrPred (1b) | |
|---|---|---|---|---|---|
| (100, 0, 0) | 458 | 458 | +0.0% | 458 | +0.0% |
| (0, 100, 0) | 663 | 660 | -0.5% | 759 | +14.5% |
| (0, 0, 100) | 877 | 775 | -11.6% | 775 | -11.6% |

其中 a, b, c 三個參數分別為：不使用 BEQ，交錯的 BEQ，以及 BEQ。

在不使用 BEQ 的情況下顯然效能不會改變，從實驗結果也可以看到結果完全相同。在交錯的情況下可能會使 FSM 的預測結果一直為錯，也就是在兩個 shallow state 之間切換，故有可能導致較差的效能，這個現象在 1-bit saturating counter 的情況更為明顯，由表中可以觀察出 1-bit 在這個情況下反而讓效能變得更差。最後是總是 BEQ 的情況，在這種情況下 1-bit 和 2-bit 都能有效減少 cycle 數，大約減少了 11% 左右，顯示 branch prediction 帶來的效益。

- 合成

  將包含 BPU 的 MIPS 進行合成，和同樣在 4ns 下合成的 baseline 進行比較，面積分別為：

  **BrPred**: 307333 ($um^2$)

  **Baseline**: 294923 ($um^2$)

  可以注意到兩者其實相差不大，故在 branch 很頻繁的程式中，使用 branch prediction 效益不錯，但以這次 baseline 來說，branch prediction 的效果並不明顯，total cycle 比較如下：

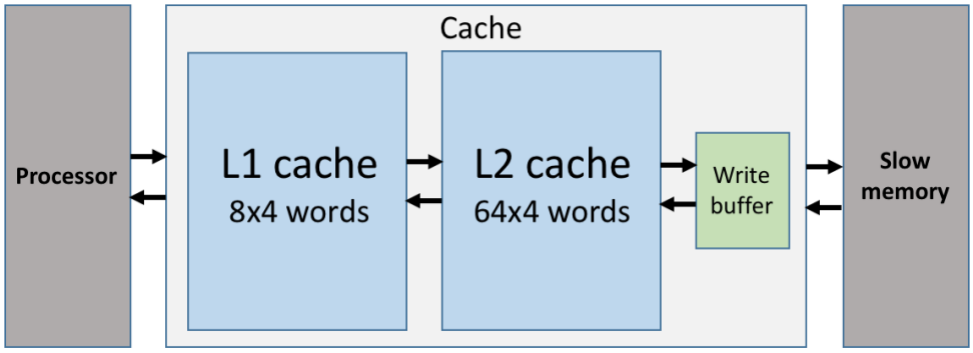
  **BrPred**: 2144 (cycles)

  **Baseline**: 2143 (cycles)

  故在 baseline 中我們不使用 branch prediction。

**2) L2 Cache**

L2 cache 的實作比較簡單，因為 cache 本身的實作和 L1 是幾乎一樣的，只不過大小和 I/O 需要做一點微調，整體的 block diagram 如下：




其中 write buffer 在前面已經提到，L2 cache 的其他規格如下：

- **Size**: 64 words
- **Placement**: direct mapped
- **Write policy**: write back + write buffer

比較 L1 和 L2 的 block 差異：

L1 cache(8 blocks)

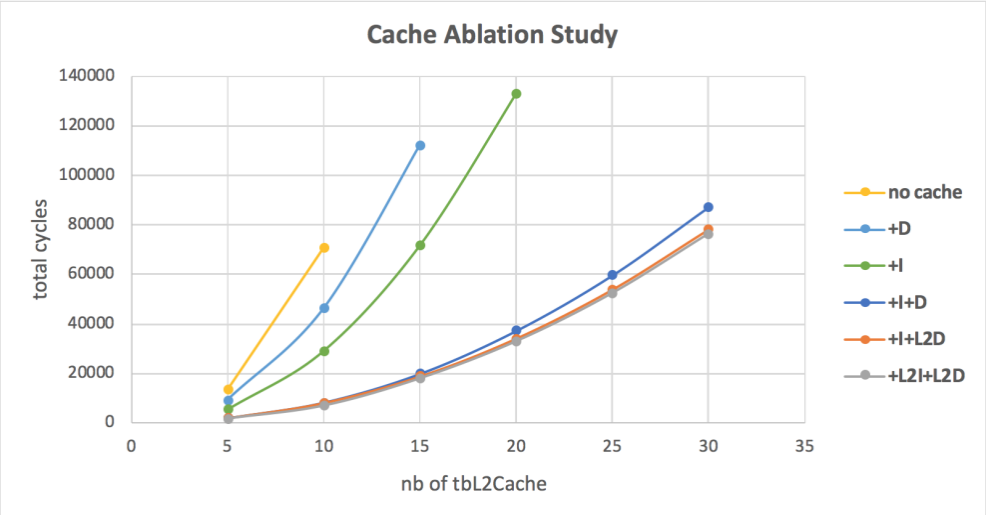
| Valid | Dirty | Tag | Data |
|---|---|---|---|
| 1b | 1b | 25b | 32*4b |

L2 cache(64 blocks)

| Valid | Dirty | Tag | Data |
|---|---|---|---|
| 1b | 1b | 22b | 128 b |

注意到 L2 cache 的 data 部分並不需要區分 words，因為這次使用的 memory 其 data width 為 128bit。接著我們進行實驗來比較這次有哪些部分使用 L2 cache 的效益，以及 L2 cache 大小對效能的影響。

## a. Cache ablation study (L2)




詳細數據參考前文的表格，可以總結出幾個結論：

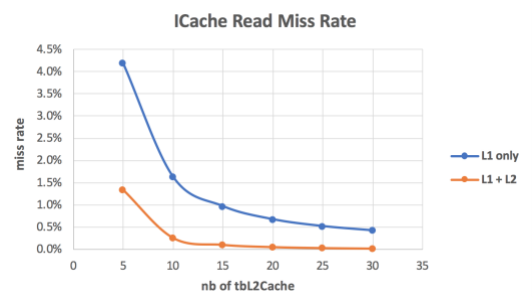
**(a) 使用 L2 DCACHE 大概能讓 cycle 數減少 10%。**

**(b) 使用 L2 ICACHE 則相對來說效能提升不大。**

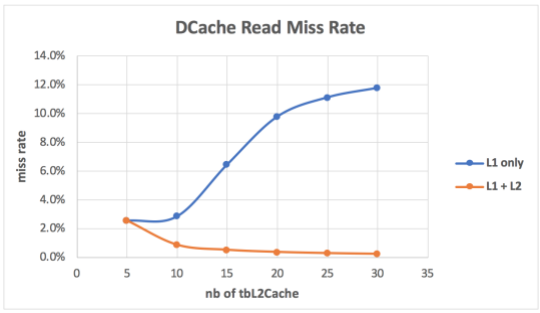
以下從 miss rate 的角度來分析這個現象。我們調整 tb L2Cache 的 nb 參數，對 64 words 的 L2 cache 進行分析。

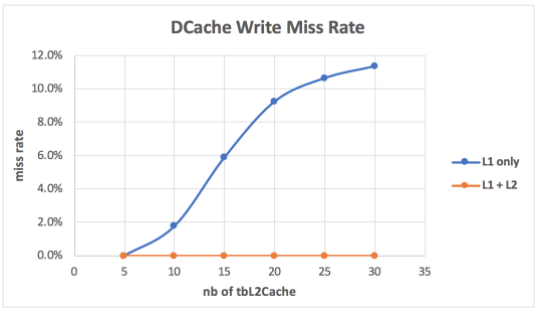
## b. Cache miss rate

| | I | L2I | D | | L2D | |
|---|---|---|---|---|---|---|
| nb | read miss rate | read miss rate | read miss rate | write miss rate | read miss rate | write miss rate |
| 5 | 4.2% | 1.3% | 2.6% | 0.0% | 2.6% | 0.0% |
| 10 | 1.6% | 0.3% | 2.9% | 1.8% | 0.9% | 0.0% |
| 15 | 1.0% | 0.1% | 6.4% | 5.9% | 0.5% | 0.0% |
| 20 | 0.7% | 0.1% | 9.8% | 9.2% | 0.4% | 0.0% |
| 25 | 0.5% | 0.0% | 11.1% | 10.6% | 0.3% | 0.0% |
| 30 | 0.4% | 0.0% | 11.8% | 11.4% | 0.2% | 0.0% |




可以觀察到 ICACHE 的 miss rate 隨著 nb 增加 miss rate 卻逐漸降低，考慮到這次產生的程式，這是由於 nb 雖然增加，但程式的總行數相同，約 50 條指令，故 ICACHE 讀取的 address 範圍不變，且指令行數和 ICACHE 大小相差不大，故 miss 的次數不會增加很多，但執行的指令次數則增加很多，造成 ICACHE miss rate 逐漸下降。由於指令行數不多，故 L2 ICACHE 效益不大。

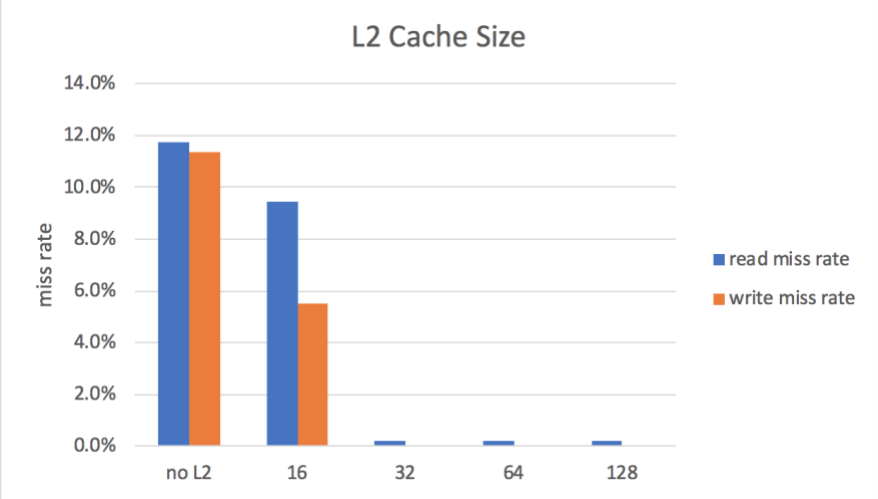



DCACHE 則有較大的 address 範圍，且會隨著 nb 增加而增加，故從 DCACHE 的 read miss rate 可以觀察出沒有 L2 的情況下，read miss rate 逐漸上升，但使用 L2 後則夠大，可以使得 miss rate 逐漸降低。寫入的情況則更極端，不使用 L2 的情況下 miss rate 也逐漸上升，使用 L2 時 write miss rate 則保持為 0，這是由於 L2 cache 夠大，使得寫入操作可以完全在 cache 內完成，而不用真正寫回 memory。

**c. L2 DCACHE size**

以下為 tbL2Cache nb=20 的 miss rate 比較，調整 DCACHE 的大小：




可以注意到其實在 L2 cache 大小為 32x4 時，miss rate 就可以幾乎為 0，故其實以這次使用的 tb 來說不需要使用到 64x4 而只要 32x4 即可。

- 合成

對使用 L2 cache 的 MIPS 進行合成，使用 cycle 為 5ns，面積比較如下：

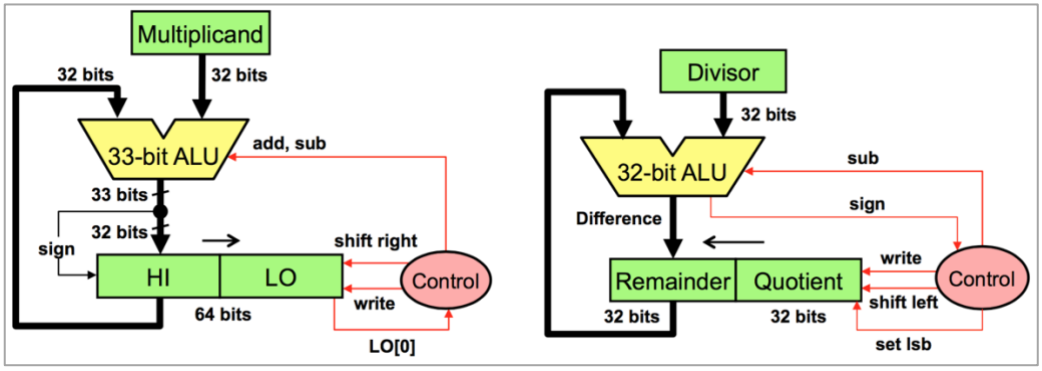
**No L2**: 282703 ($um^2$)

**L2**: 914091 ($um^2$)

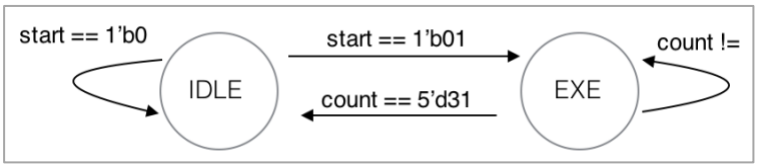
可以注意到面積增加很多，且對 baseline 使用的 AT 評分來說並沒有進步，故 baseline 也不採用 L2 cache。

### 3) Multiplier and Divider

本次實作的乘除法器使用 iterative approach，支援負數運算，架構和講義相同，如下圖：




乘法和除法都使用 32 個 cycle 來完成計算，兩者的 FSM 如下：




其實狀態為 IDLE，ALU 在接受到乘除法的 opcode 時，會產生 start 訊號，並將它傳給乘/除法器，乘/除法器在接到 start 訊號時，會將 stall 設為 1，將 counter 設為 0，並跳至 EXE 狀態，之後每個 clock counter 會加一，當 counter 為 31 時，乘/除法的數值已經被計算好，則會跳回 IDLE state 並將 stall 設回 0。此時 $HI 和 $LO 也會存有正確的值，直到下一次計算。

以下比較直接使用 Verilog 的 arithmetic operators (* / %)，和上述的 iterative 方法的結果差異：

| | area (um²) | cycles | clock (ns) | total time (ns) |
|---|---|---|---|---|
| iterative | 326755 | 541 | 6.1 | 3300.1 |
| arithmetic | 738854 | 188 | 19.4 | 3647.2 |

注意到從 RTL 模擬的角度來看，使用 arithmetic operator 顯然能有更小的 cycle 數，但是在合成後觀察總時間則會發現，使用 arithmetic 由於會合成出巨大的 combinational circuit，故只能以遠慢於 iterative approach 的 clock 合成，這次實驗中可以發現 clock 慢了超過三倍，結果是總執行時間反而為 iterative 更快。結論是在 Verilog 中的電路設計要考慮到合成，故儘量避免使用複雜的 arithmetic operator。

# *DSD Final Project Scores*

1. Baseline

(1) Area: ($um^2$)

截圖:

```
Number of ports:                         4014
Number of nets:                         23466
Number of cells:                        19744
Number of combinational cells:          15078
Number of sequential cells:              4647
Number of macros/black boxes:               0
Number of buf/inv:                       2829
Number of references:                       7

Combinational area:            144747.482378
Buf/Inv area:                   22030.554472
Noncombinational area:         145640.310276
Macro/Black Box area:               0.000000
Net Interconnect area:        2797727.592407

Total cell area:               290387.792654
Total area:                   3088115.385061
```

**290388 ($um^2$)**

(2) Total Simulation Time (hasHazard testbench): (ns)

截圖:

```
FSDB Dumper for IUS, Release Verdi_N-2017.12, Linux, 11/12/2017
(C) 1996 - 2017 by Synopsys, Inc.
*Verdi* FSDB WARNING: The FSDB file already exists. Overwriting the FSDB file
 may crash the programs that are using this file.
*Verdi* : Create FSDB file 'Final.fsdb'
*Verdi* : Begin traversing the scope (Final_tb), layer (0).
*Verdi* : Enable +mda dumping.
*Verdi* : End of traversing.
-------------------------- Simulation FINISH !!--------------------------
==============================================================================

 \(^o^)/ CONGRATULATIONS!!  The simulation result is PASS!!!

==============================================================================
Simulation complete via $finish(1) at time 8741400 PS + 0
```

**8741 (ns)**

(3) Area*Total Simulation Time: ($um^2$ * ns)

**2.538 * $10^9$ ($um^2$ x ns)**

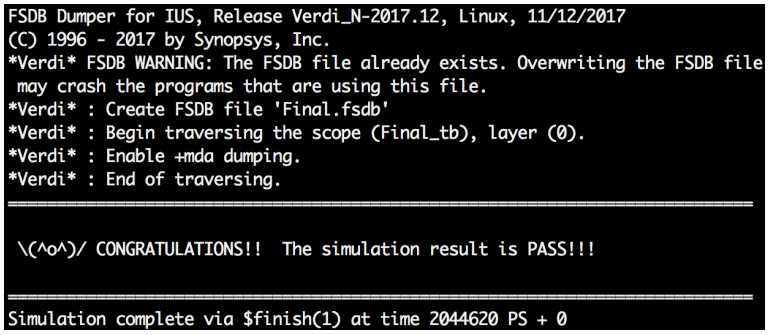
(4) Clock cycle for post-syn simulation: (ns)

**4.08 (ns)**

2. BrPred

(1) Total execution cycles of I_mem_BrPred:

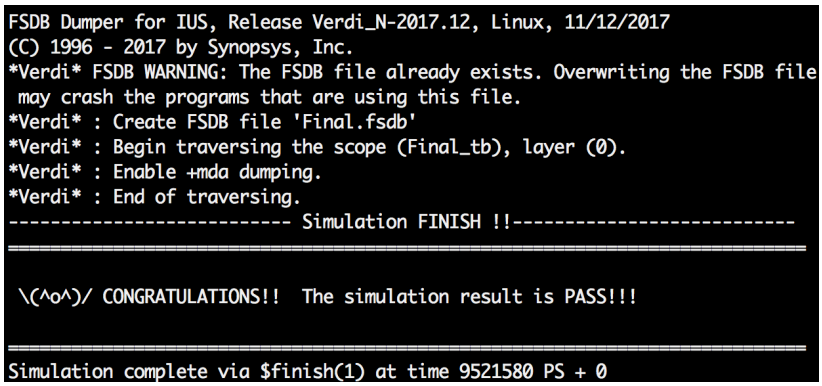
截圖:




2045 ns / 4.44 = **460 (cycles)**

(2) Total execution cycles of I_mem_hasHazard:

截圖:



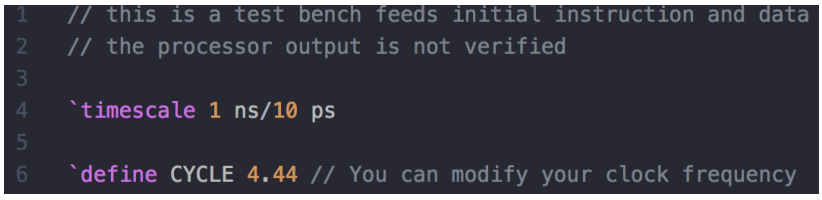

9522 / 4.44 = **2145 (cycles)**

(3) Synthesis area of BPU(Total area of BrPred minus baseline design, two design clock cycle need to be same): ($um^2$)

**12410 ($um^2$)**

(4) Clock cycle for post-syn simulation: (ns)

截圖:



```
1  // this is a test bench feeds initial instruction and data
2  // the processor output is not verified
3
4  `timescale 1 ns/10 ps
5
6  `define CYCLE 4.44 // You can modify your clock frequency
```

**4.44 (ns)**

3. L2Cache

(1) Avg. memory access time: (ns)

**6.46 (ns)**

(2) Total execution time: (ns)

截圖:

```
FSDB Dumper for IUS, Release Verdi_N-2017.12, Linux, 11/12/2017
(C) 1996 - 2017 by Synopsys, Inc.
*Verdi* FSDB WARNING: The FSDB file already exists. Overwriting the FSDB fil
 may crash the programs that are using this file.
*Verdi* : Create FSDB file 'Final.fsdb'
*Verdi* : Begin traversing the scope (Final_tb), layer (0).
*Verdi* : Enable +mda dumping.
*Verdi* : End of traversing.
-------------------------- Simulation FINISH !!--------------------------
================================================================================

 \(^o^)/ CONGRATULATIONS!!  The simulation result is PASS!!!

================================================================================
Simulation complete via $finish(1) at time 207348150 PS + 0
```

207348 / 6.1 = **33991 (cycles)**

(3) Clock cycle for post-syn simulation: (ns)

**6.1 (ns)**

4. MultDiv

(1) Total synthesis area: ($um^2$)

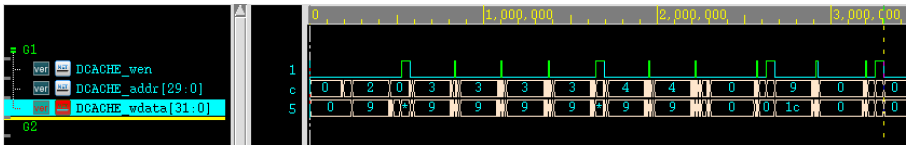
截圖:

```
Number of ports:                        5070
Number of nets:                        26883
Number of cells:                       21851
Number of combinational cells:         16405
Number of sequential cells:             5418
Number of macros/black boxes:              0
Number of buf/inv:                      3455
Number of references:                      9

Combinational area:            164849.790415
Buf/Inv area:                   24690.380075
Noncombinational area:         161904.796623
Macro/Black Box area:               0.000000
Net Interconnect area:        3050055.562897

Total cell area:               326754.587038
Total area:                   3376810.149935
```

**326755 ($um^2$)**

(2) Total execution time: (ns)

截圖:




**3300 (ns)**

(3) Minimum clock period: (ns)

截圖:

```
1  // this is a test bench feeds initial instruction and data
2  // the processor output is not verified
3
4  `timescale 1 ns/10 ps
5
6  `define CYCLE 6.1 // You can modify your clock frequency
```

**6.1 (ns)**