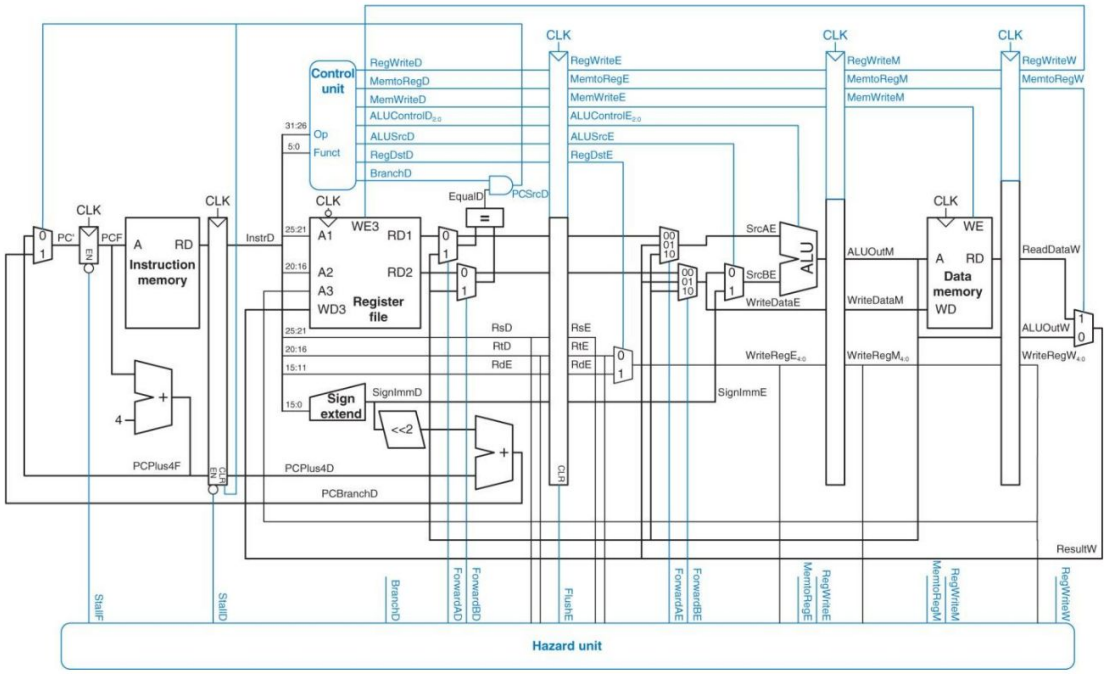
DSD Final Report (MIPS)

B05901013 張景皓, B05901014 吳靖平

Part 1: Baseline



---

1. Jump-related instructions (j/jr/jal/jalr)

j destination	PC = address * 4
jal destination	$ra = PC; PC = address * 4
jr $rs	PC = $rs
jalr $rt $rs	$rt = PC; PC = $rs

- Jump-related instructions的解碼發生在ID stage，也表示同時間IF stage所讀取的指令為PC+4而非jump所指到的PC。所以凡是讀到j/jr/jal/jalr等指令，IF/ID的gate所存取的值必須要flush，重新讀取新指令(jump destination)，如此才不會執行到錯誤的指令：

```
//IF_ID is stalled to refetch new instruction when nop_FD is HIGH
assign nop_FD  = (Jump[1] ^ Jump[0]) ? 1'b1 : 1'b0; // j/jal: Jump = 2'b01;
jr/jalr: Jump = 2'b10
```

- jr/jalr和其另外兩個指令不同的地方是由$rs來決定下個PC。既然從registers讀值，那就會有registers沒有及時更新的hazard，所以同樣也需要forwarding unit的輔助。

- jal/jalr皆需要link reg(寫回registers)，所以還需要另外寫一個MUX決定write register address:

```
assign WriteReg_E = (RegDst_E == 2'b00) ? rt_E : ((RegDst_E == 2'b01) ? rd_E :
5'd31); //$ra == $31
```

因為MIPS有pipeline的關係，所以不能在ID stage馬上把位址寫入registers，我們把決定位址的MUX寫在EX stage，位址會傳到WB stage再寫回ID stage的registers。

2. Branch (beq/bne)

這次branching除了HW3的beq之外還多了bne，不過其實兩種指令本來就是一體兩面，實作方法如下:

```
assign equal = (read_data1_i == read_data2_i);
assign Branch_o = (Brncheq_i & equal) | (Brnchne_i & ~equal); //beq: Brncheq_i = 1’b1;
bne: Brnchne_i = 1’b1;
```

無論beq或是bne為真，PCSrc會被拉高，branching address會傳至PC以讀取下個正確指令:

```
assign PCbranch_W = PCSrc_D ? PCbranch_D : PCplus4_F;
assign PC_i_W = (Jump_D == 2'b00) ? PCbranch_W : ((Jump_D == 2'b01) ? Jumpaddr_D :
read_data1_forwarded_D);
```

3. Forwarding

```
assign ForwardAD = (RegWrite_IE & (RegisterRd_IE != 5'b0) & (RegisterRd_IE ==
RegisterRs_FD)) ? 2'b11 : (RegWrite_EM & (RegisterRd_EM != 5'b0) & (RegisterRd_EM ==
RegisterRs_FD)) ? 2'b10 : (RegWrite_MW & (RegisterRd_MW != 5'b0) & (RegisterRd_MW ==
RegisterRs_FD)) ? 2'b01 : 2'b00;

assign ForwardBD = (RegWrite_IE & (RegisterRd_IE != 5'b0) & (RegisterRd_IE ==
RegisterRt_FD)) ? 2'b10 : (RegWrite_MW & (RegisterRd_MW != 5'b0) & (RegisterRd_MW ==
RegisterRt_FD)) ? 2'b01 : 2'b00;

assign ForwardAE = (RegWrite_EM & (RegisterRd_EM != 5'b0) & (RegisterRd_EM ==
RegisterRs_IE)) ? 2'b10 : (RegWrite_MW & (RegisterRd_MW != 5'b0) & (RegisterRd_MW ==
RegisterRs_IE)) ? 2'b01 : 2'b00;

assign ForwardBE = (RegWrite_EM & (RegisterRd_EM != 5'b0) & (RegisterRd_EM ==
RegisterRt_IE)) ? 2'b10 : (RegWrite_MW & (RegisterRd_MW != 5'b0) & (RegisterRd_MW ==
RegisterRt_IE)) ? 2'b01 : 2'b00;
```

我們最初的寫法是參照講義的內容，但是後來發現判斷條件會不夠完整，應該在EX/MEM/WB每個階段之間都要加上forwarding　datapath。以ForwardAD來

說，ID/EX、EX/MEM、MEM/WB都有forwarding的判斷條件使data可以從各個stage直接送回ID stage。

4. **Stall**

Baseline設計中stall的處理是交由hazard unit負責，主要有下列時機會需要:

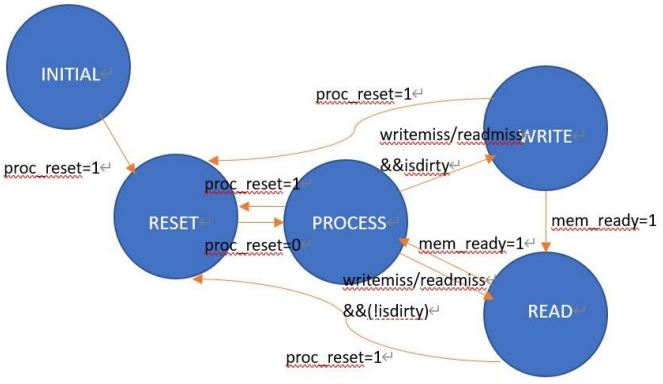
- **CacheRead & ID/EX.Rt == IF/ID.Rs**: stall PC & IF/ID, flush ID/EX
- **ID/EX.Rt == IF/ID.Rt**: stall PC & IF/ID, flush ID/EX
- **Cache stall** (read/write miss): stall PC and all pipeline gates
- **j/jr/jal/jalr**: flush IF/ID

5. **Cache**

- I Cache:
  - Mode: read-only(ICACHE_ren = 1'b1, ICACHE_wen = 1'b0)
  - Size: 32 words (8 blocks * 4 words)
  - Approach: direct mapping
  - Write policy: write back
- D Cache:
  - Mode: read/write available
  - Size: 32 words (8 blocks * 4 words)
  - Approach: direct mapping
  - Write policy: write back

	area (um^2)	testbench clock (ns)	operation time (ns)	cycle	sdc (ns)
Direct mapping	**249159**	**4.57**	**8369.94**	~1832	3
2-way associative	258395	6.01	10905	**~1818**	3.5

我們是直接使用HW4寫過的cache。Direct mapping與2-way associative兩個版本都有合成，但2-way合成出來結果不甚理想，連cycle = 6(ns)都過不了，因此最終決定使用direct mapping的版本作為Baseline MIPS的cache。下圖是cache的FSM:



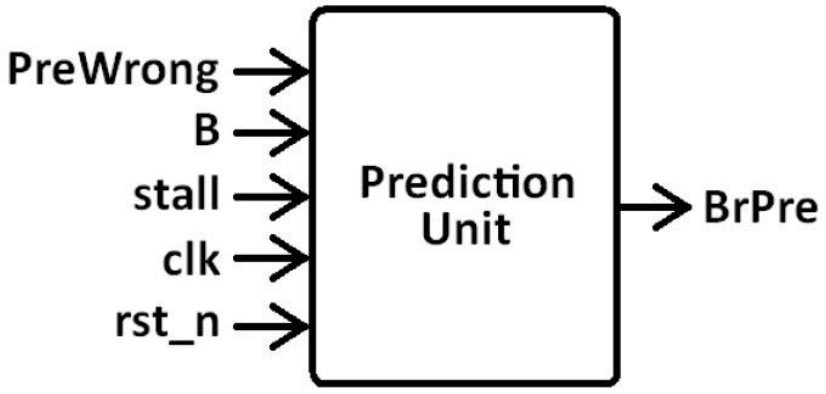
Part 2: Extention

---

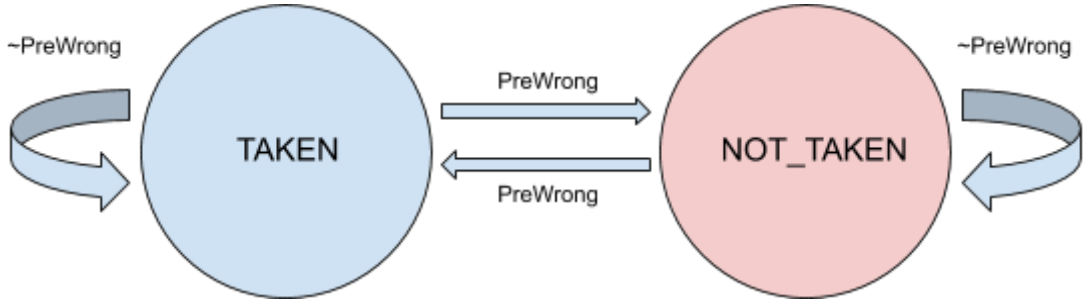
1. **Branch Prediction**

- (1) 基本架構：按照pdf說明裡的設計，將計算branch address的硬體移到了IF stage，並且增加了prediction unit。此prediction unit的I/O設計參考了說明文件裡的架構，如下圖(表)所示:

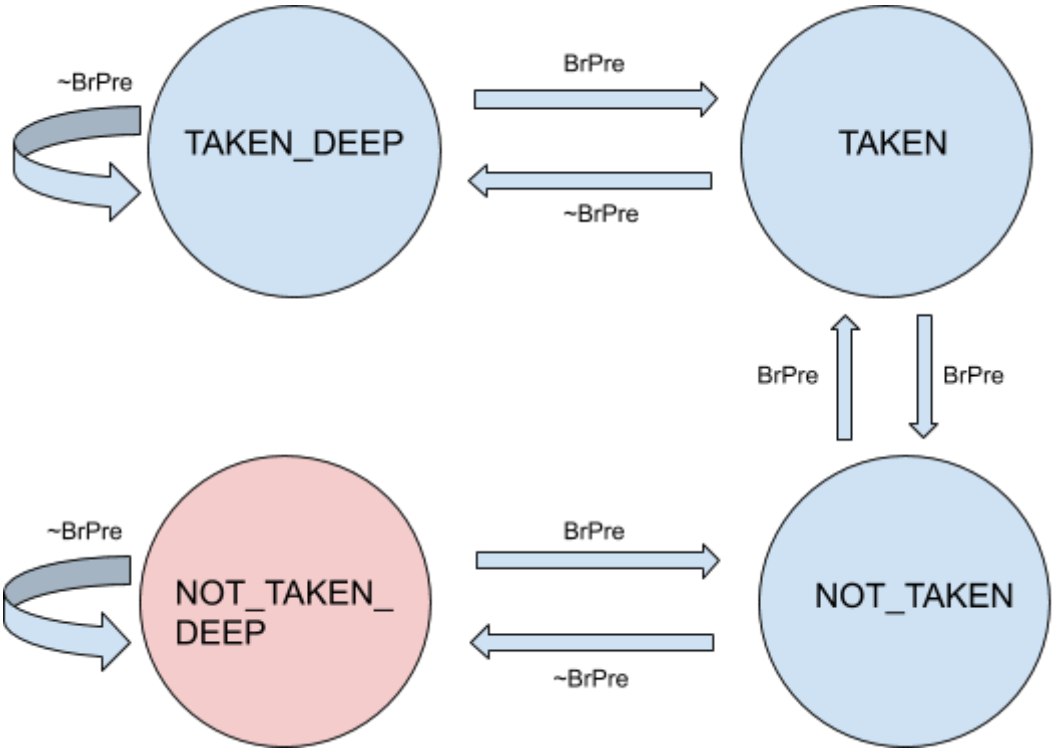
name	I/O	width	discription
PreWrong	I	1	若為high表示前一次預測結果錯誤
B	I	1	若為high表示現在的instruction是beq或bne
stall	I	1	若為high則內部的state不更新
clk	I	1	clock
rst_n	I	1	active-low reset
BrPre	O	1	預測結果，high表示branch taken



- (2) Prediction unit FSM：我們測試了兩種prediction unit，分別是1-bit predict和2-bit predict。兩者的差別是1-bit predict只要一出錯就會改變預測結果，而2-bit predict則需要連續出錯兩次才會改變。粉紅色的state為起始state。



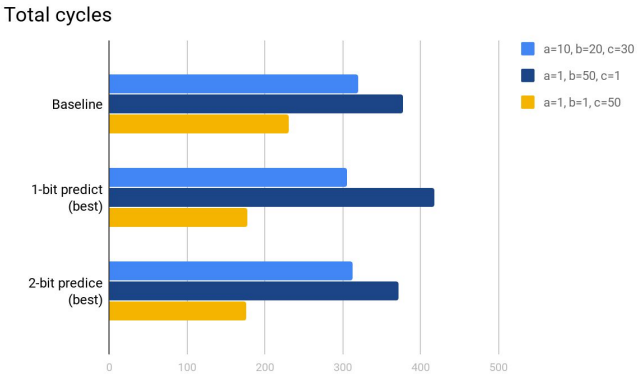
▲1-bit predict



▲2-bit predict

- (3) RTL模擬結果與分析：我們嘗試了三種測資，一種是預設值(a=10，b=20，c=30)，另一種是幾乎由interleave branch組成(a=1，b=50，c=1)，最後一種則是幾乎由always branch組成。下表為使用Baseline、1-bit predict和2-bit predict的結果，單位為total cycles。(註：在我們的設計中，測資結束後至少要有一行nop才能正常結束，因此我們有在測資中加上一行nop，有跟助教確認過是可以的)

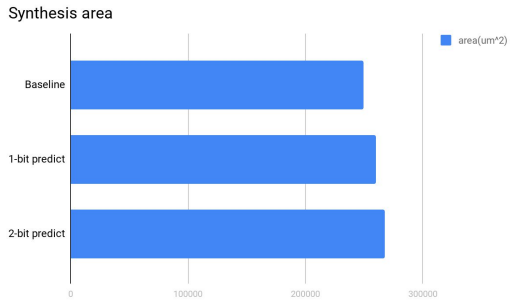
Model	a=10, b=20, c=30	a=1, b=50, c=1	a=1, b=1, c=50
Baseline	320	377	230
1-bit predict	**305**/306	419/418	178/177
2-bit predict	313/313/314/315	426/372/371/**371**	184/178/**176**/**176**

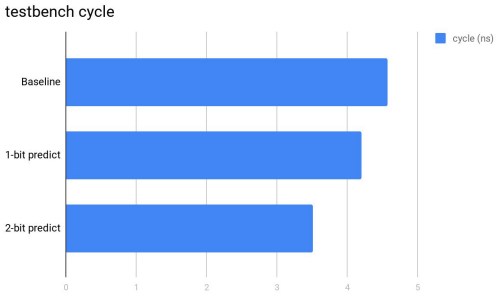


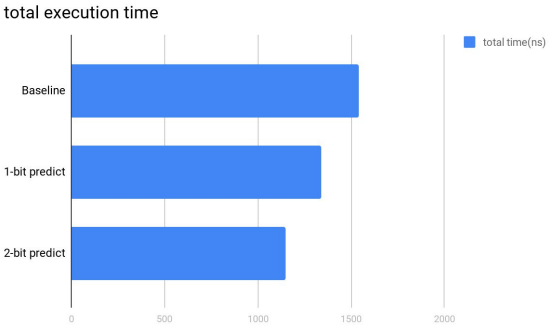
其中，1-bit predict中兩個數值分別為起始state是NOT_TAKEN/TAKEN的結果，而2-bit predict中四個數值分別為起始state是NOT_TAKEN_DEEP/NOT_TAKEN/TAKEN/TAKEN_DEEP的結果。可以發現，在三者之中interleaved表現最好的是Baseline，最差的是起始state為NOT_TAKEN_DEEP的2-bit predict和不論起始state為何的1-bit predict，與課堂上教的和我們預期的相符。在BrPred的測資中，interleaved的部分對於動態的branch prediction是很不利的，尤其是1-bit的版本，因為它每次都會預測錯誤並改變預測結果。相對的，對於Baseline(相當於always not taken)來說，在interleaved的部分每兩次才會預測錯一次，表現自然較好。因此第三組測資(always taken)就是為了看到Branch prediction效果而設計的。在第三組測資中Baseline的表現就不如有prediction的版本，因為全都是branch taken的狀況，符合我們想要看到的結果。

- (4) gate-level合成結果與分析(i)：分別合成完1-bit predict和2-bit predict後，對預設值測資(a=10, b=20, c=30)進行模擬所得到的結果如下：

Model	sdc cycle(ns)	area(um^2)	tb pass cycle(ns)	total execution time(ns)
Baseline	3	**249159**	4.57	1541
1-bit predict	3.1	259936	4.2	1341
2-bit predict	3	267800	**3.51**	**1149**



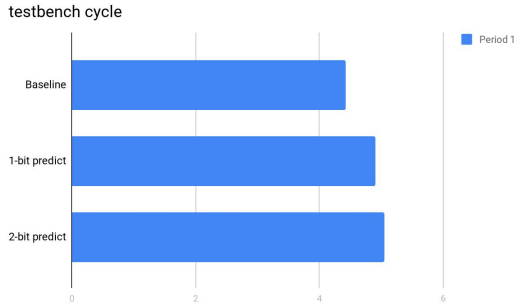


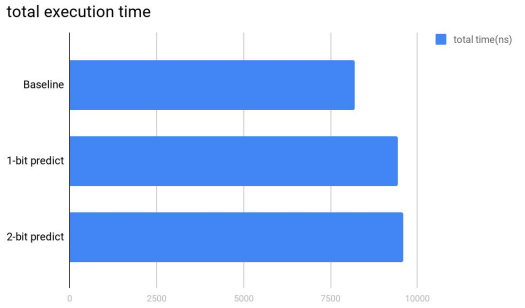


出乎我們意料之外的，2-bit能通過的cycle比1-bit和Baseline小上非常多，且面積也沒有增加太多(約7.5%)。原本我們認為，把計算branch的那些硬體都移到IF stage應該會讓stage之間不平均，使得cycle更大。然而，從這樣的結果我們可以分析，原本IF和ID應該就有不平均的問題，把branch的硬體往前移反而有助於平衡兩者。至於為何1-bit的結果跟2-bit差這麼多，只能認為是合成上有甚麼條件讓design compiler合出跟2-bit完全不一樣邏輯的電路。另一方面，從面積的增加幅度來說合乎我們的預期。2-bit的複雜度大於1-bit，1-bit又大於Baseline，這點在面積上完全反應出來。

- (5) gate-level合成結果與分析(ii)：接著我們用hasHazard的預設測資做了測試，結果如下:

Model	sdc cycle(ns)	area(um^2)	tb pass cycle(ns)	total execution time(ns)
Baseline	3	**249159**	**4.43**	**8187**
1-bit predict	3.1	259936	4.91	9443
2-bit predict	3	267800	5.05	9596





顯然Baseline的結果是最好的，但cycle三者的pass cycle都比BrPred的測資大。我們認為這是因為BrPred測資中沒有jal、jr和jalr的指令，因此某些data path沒有被用到，cycle也因此可以壓到更低。而在hasHazard測資中，幾乎所有指令都有，因此構造較複雜的2-bit predict自然會有較大的cycle，掩蓋掉理論上cycle數應該要較小的優勢。最後是各個model的critical path，在baseline中是在D_cache的讀取，在1-bit predict中是從PC的output到IF/ID的input，也就是說branching的部分是critical path。在2-bit predict中，critical path是從ID/EX中的register_rt到PC的input，我們認為這應該是因為jr/jalr的指令再加上forwarding讓整體路徑相對變得最長所導致。

2. L2 Cache

- (1) 基本架構：做成unified cache，規格為64個block，每個block有四個word的2-way associative cache。Write policy使用write back。所謂unified是指從I_slow_memory和D_slow_memory進來的資料會共用這些blocks，因此每個block除了原本的data、tag、valid、dirty外還需要多加上source來代表這筆data是從哪個memory來的。因為要同時支援兩個cache的存取，原本cache的I/O除

了clock跟reset外都必需有兩套以分別對應兩個cache和memory。之所以做成2-way associative是因為考慮到I cache和D cache同時存取的情況，這種情況下若是做成direct mapped的話100%會需要stall，若做成2-way則有機會避免。由於無法事先知道這種強況發生的機率高低，我們認為還是做成2-way的比較好。下表是我們設計的L2 cache的I/O，其中name為紅字者表示永遠為0，在合成時應會被DC優化掉，但為了不失一般性，在RTL我們仍予以保留。

name	I/O	width	discription
clk	I	1	clock
proc_reset	I	1	active-high reset
proc_read_I	I	1	I cache的讀取訊號
proc_write_I	I	1	I cache的寫入訊號
proc_addr_I	I	28	I cache的讀取/寫入地址
proc_rdata_I	O	128	根據proc_addr_I讀出的data
proc_wdata_I	I	128	I cache要寫入L2 cache的data
proc_stall_I	O	1	相當於I cache的mem_ready
mem_read_I	O	1	對slow_memory_I的讀取訊號
mem_write_I	O	1	對slow_memory_I的寫入訊號
mem_addr_I	O	28	讀取/寫入slow_memory_I的地址
mem_rdata_I	I	128	從slow_memory_I讀出來的data
mem_wdata_I	O	128	寫入slow_memory_I的data
mem_ready_I	I	1	slow_memory_I傳來的mem_ready
proc_read_D	I	1	D cache的讀取訊號
proc_write_D	I	1	D cache的寫入訊號
proc_addr_D	I	28	D cache的讀取/寫入地址
proc_rdata_D	O	128	根據proc_addr_D讀出的data
proc_wdata_D	I	128	D cache要寫入L2 cache的data
proc_stall_D	O	1	相當於D cache的mem_ready
mem_read_D	O	1	對slow_memory_D的讀取訊號

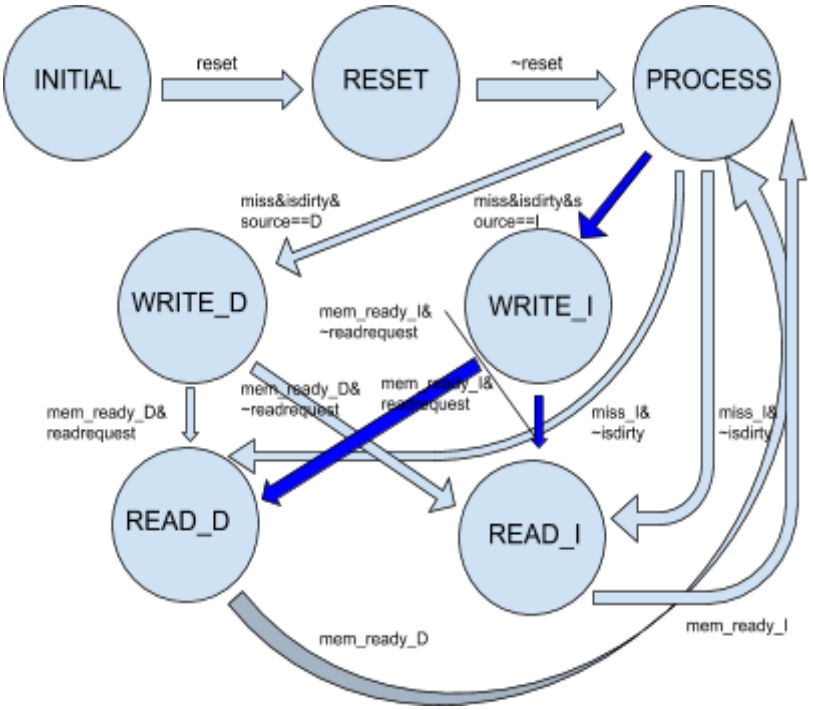
mem_write_D	O	1	對slow_memory_D的寫入訊號
mem_addr_D	O	28	讀取/寫入slow_memory_D的地址
mem_rdata_D	I	128	從slow_memory_D讀出來的data
mem_wdata_D	O	128	寫入slow_memory_D的data
mem_ready_D	I	1	slow_memory_D傳來的mem_ready

- (2) replacement policy：關於replacement policy的部分，大致是採用簡單的LRU (last recent used)，在2-way架構下每個block只要一個bit就可以輕鬆的辦到。然而，因為是unified cache，有時會有I cache跟D cache同時存取的情況發生，這時LRU bit就要特別指定更新條件。在我們的L2 cache架構中，若發生這種情況，一律讓存有D memory data的slot成為LRU slot。這麼做的理由為，一般而言instruction是一行一行往下讀取，被洗掉的話下一次要重新讀取的機率很高，因此選擇存取順序隨機性比較大的D memory data當作優先被洗掉的data。

```
if(proc_addr_D[BLOCK_WIDTH-1:0]==proc_addr_I[BLOCK_WIDTH-1:0])begin
        block_LRU[proc_addr_D[BLOCK_WIDTH-1:0]] <=
        (proc_addr_D[27:BLOCK_WIDTH]==block_tag[proc_addr_D[BLOCK_WIDTH-1:0]][TAG_WIDTH
        -1:0])?1'b0:1'b1;//treat D read hit as LRU
end
```

實務上，會頻繁從L2 cache讀取同一個block內資料的情況並不常見，因為L1 cache裡面已經有空間儲存data，且L2 cache的block number較大，所有的access都擠在同一個block的機率也較低。因此針對LRU的這個小設計只是預防萬一。

- (3) FSM：如下圖所示，共7個state，扣除INITIAL和RESET後共有五個state，分別代表hit、兩個memory的read和write。其中，miss代表來自D_cache或I_cache的miss，isdirty代表被allocate的block中的data是已經寫過的，source則是代表被allocate的block中data的來源。readrequest是一個1 bit的register，它負責儲存miss的來源以便在執行完WRITE後回到正確的READ state。值得一提的是，在設計L2 cache時，我們將兩個cache視為同等的地位，但事實上I_cache是一個read only的cache，也就是說不可能會有來自I_memory的data被汙染，因此圖中深藍色箭頭的path事實上是不可能發生的。



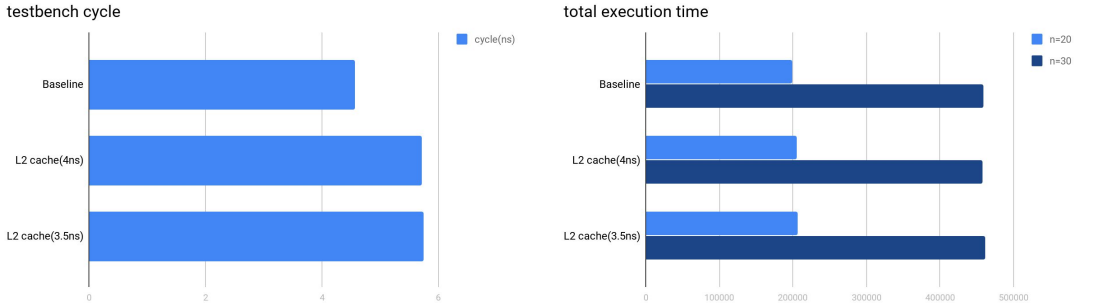
- (4) RTL模擬結果與分析：在L2 cache的RTL模擬中我們注重的是原本的I cache、D cache和L2 cache三者的miss rate。使用預設測資的模擬結果如下表(n=20/n=30)：

cache	# of access	# of miss	miss rate(%)
I cache	30545/67435	20/20	0.07/0.03
D cache	6587/14707	3144/9734	47.73/66.19
L2 cache	1310/3405	37/47	2.82/1.38

對於I cache而言，因為測資中的instruction不長，因此只要loop不把原本存好得instruction洗掉，就不容易miss。對D__cache而言，因為在SubGen的instruction中每次loop都要呼叫sw，且在OutputTestPort中也有sw的instruction，因此D cache的miss rate會這麼高也是意料之中。至於最重要的L2 cache，在n=20時有約3%的miss rate、在n=30時則有1.38%的miss rate，比起只有D cache的情況改進了非常多，而這也是L2 cache最主要的目標。另外我們也比較了baseline和加了L2 cache兩者執行L2Cache測資所花的cycle，在n=20時baseline需要花40934個cycle，加了L2 cache後只要**35914**個cycle，改進幅度約12%。至於n=30時baseline需要花93333個cycle，而加了L2 cache後只要**79993**個cycle，改進幅度約14%。

- (5) gate-level合成結果與分析：因為L2 cache合成比起baseline或其他extention項目來說要花很長一段時間，因此目前我們只合到sdc cycle為3.5(ns)的版本。

Model	sdc cycle(ns)	area(um^2)	tb pass cycle(ns)	total execution time(ns)
L2 cache	10	849752	N/A	N/A
L2 cache	5	778617	N/A	N/A
L2 cache	4	809814	5.71	205636/**457825**
L2 cache	3.5	856394	5.75	207074/461027
Baseline	3	249578	4.57	**199470**/458648



若我們單看預設測資(紫字)，那麼非常遺憾的，加上了L2 cache後表現竟然還輸Baseline。我們認為原因有二：一是2way的cache在timing上本來就滿吃虧的，在做baseline時我們也有試著用2way版的cache來做，但合出來能過的cycle比起direct mapped的大上不少，因此放棄採用。但在做unified L2 cache時基於前面提過的原因，還是選用了2way cache。事實上從上面的結果可以看到，雖然兩者的cycle差了很多(快要20%)，但總共執行時間卻不會差到太誇張(2%內)，顯示L2 cache有效的了降低miss penalty，彌補了cycle較大的劣勢。為了彰顯L2 cache的優勢，我們修改了generate中的參數讓n=30後又跑了一次(黑字)，這次L2 cache的表現就勝於baseline，與我們想要看到的趨勢相符。也就是說，雖然犧牲了大量面積與較大的cycle time，但隨著task的長度越來越長，一般來說memory的access次數也會增加，這時加了L2 cache的優勢就顯現出來了。順帶一提，面積不在我們優化的考量因素內，畢竟既然已經要加L2 cache，面積本來就會三級跳，討論起來也沒什麼意義。

3. MultDiv

本次乘除法的實作測試了三種方法，分別為: Verilog內建的arithmetic operators (i.e., *, /, %)、iterative approach以及booth encoding乘法器。

Instruction	Op/funct	Operation

mult $rs $rt	0/24	{$HI, $LO} = $rs*$rt
div $rs $rt	0/26	{$HI, $LO} = {$rs%$rt, $rs/$rt}
mfhi $rd	0/16	$rd = $HI
mflo $rd	0/18	$rd = $LO

- **支援mult, div, mfhi, mflo的MIPS基本架構**
  - **Version 1.0:** 支援乘除法指令的MIPS基本上只需要新增: control unit的decoding、$HI/$LO、ALU乘除法運算、mfhi和mflo的forwarding判斷條件。

    最原始的設計沿用baseline MIPS的架構：將$HI和$LO兩個register放在ID stage（但是其module和registers分開）、乘除法器放在EX stage。這樣的缺點是做完乘除法運算還需要3個cycle寫回去(EX -> MEM -> WB -> ID)，而且mfhi和mflo如果接續在乘除指令後面，則需要在forward unit另外增加判斷條件(e.g., ID.inst == mfhi & EX.hilo_write & MEM.hilo_write & WB.hilo_write)，對於A*T值的優化其實是不利的。

  - **Version 2.0:** 第二版改進的做法是將$HI和$LO搬至EX stage，讓ALU做完乘除法運算後可以寫入$HI/$LO（$HI/$LO是和ALU分開的）。這樣做的好處之一是可以直接用baseline MIPS的forwarding datapath:

    ```
    assign ALUOut_E = HLSrc_E ? read_HLdata_E : ALUResult_E;
    assign read_data1_forwarded_D = (ForwardA_D == 2'b11) ? ALUOut_E : (ForwardA_D
    == 2'b10) ? ALUOut_M : (ForwardA_D == 2'b01) ? Result_W : read_data1_D;
    ```

    這種架構其實是最適合用於arithmetic operator method。在ALU non-sequential的寫法下，data1/data2和hilo_write訊號會同時進到EX stage並且會在同個cycle內完成運算和寫入。

  - **Version 3.0:** 在第二版我嘗試使用iterative乘除法器時，因為該方法無法在一個cycle之內完成運算，所以勢必要有stall訊號使整個MIPS在ALU做乘除法運算時停擺。但是因為data1/data2和hillo_write都是同時進到EX stage，在ALU運算完畢時，stall變回LOW，hilo_write訊號也會跟著進到下個stage，反而使ALU的運算結果不會成功寫入$HI/$LO裡面。

    所以改良的方法是直接將$HI/$LO寫在ALU裡，如此也不必在control unit額外增加hilo_write訊號，因為ALU運算結果就是直接寫入$HI/$LO，如此也不用擔心讀到mfhi/mflo指令時$HI/$LO的值還未被更新而需要做forwarding。同樣地，baseline MIPS的forwarding datapath也不必做任何更動。

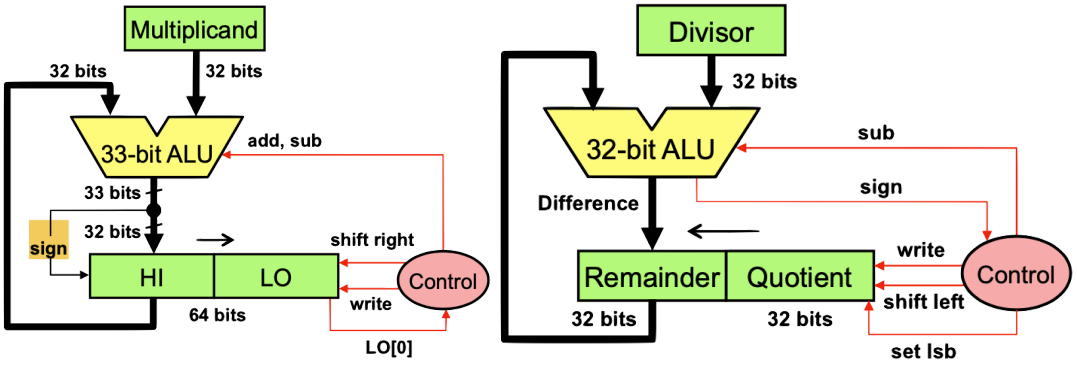
- **乘/除法器架構**

實作乘除法最簡易的方式莫過於直接使用Verilog直接支援的arithmetic operator(i.e., *, /, %) :

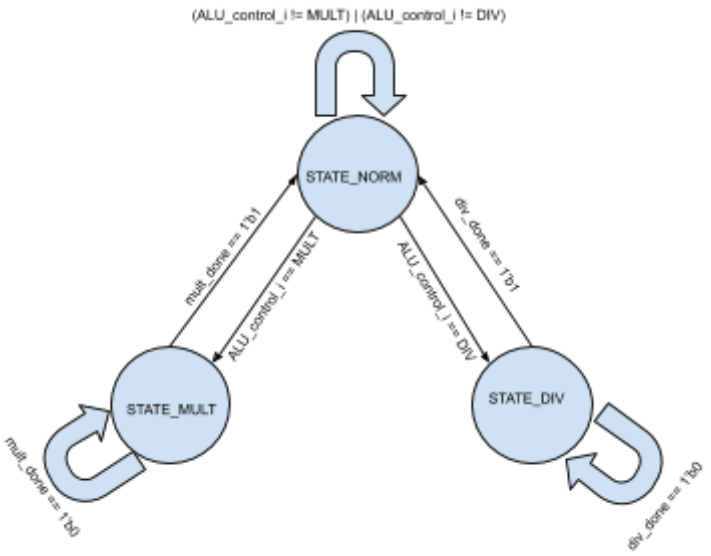
```
MULT: begin
        result_o = 32'b0;
        {HI_o, LO_o} = $signed(data1_i) * $signed(data2_i);
end
DIV: begin
        result_o = 32'b0;
        HI_o = $signed(data1_i) % $signed(data2_i);
        LO_o = $signed(data1_i) / $signed(data2_i);
end
```

該方法會大幅增加EDA tool合成的難度，更別說是32-bit的乘除了。秉持著實驗精神，我們還是有試著用Synopsys Design Compiler進行合成，試了兩次，每次約2小時還沒結束，就會遭遇工作站斷線問題。在其mapping optimization的過程可以得到合成面積約略為59萬um^2，比其他乘除法器實作方法的面積大上不只兩倍。或許arithmetic operator的cycle數可以很小，因為乘除法都是在一個cycle內完成，但是其clk一定遠大於iterative approach，連帶造成operation time也會大很多，對於A*T值的優化而言很顯然不是個最佳解。

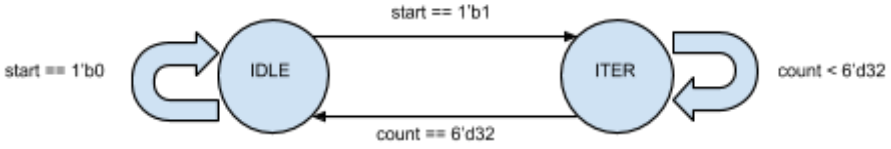
Iterative approach是現今很常見的乘除法實作方式，以shift-and-add/subtract的方式，可以支援signed binary的運算，缺點就是用時間來換取面積。其實作架構如下:



ALU的FSM如下:

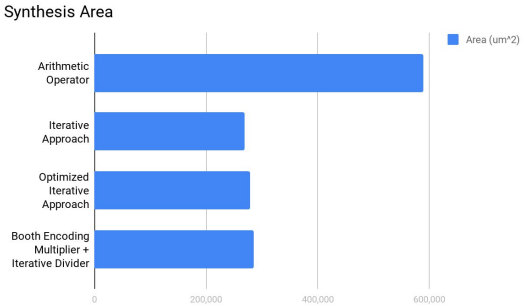


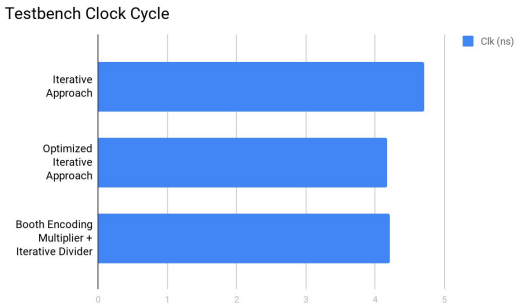
Multiplier/divider的FSM則一樣都是:

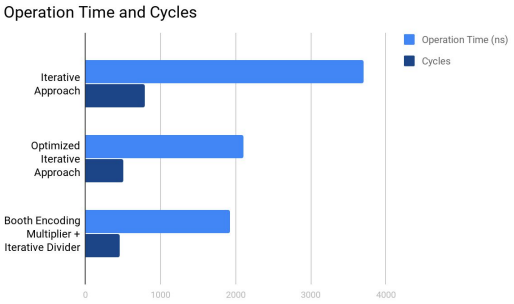


STATE_NORM為初始狀態，進行除了乘除之外的運算，當ALU_control_i == MULT或DIV時，start訊號會被拉高進到STATE_MULT或STATE_DIV，乘/除法器也會從IDLE變成ITER開始進行運算，stall也會變HIGH。重複31次後，done會被拉高回到STATE_NORM，stall變回0，CPU繼續運作，運算結果也會輸出存到$HI/$LO裡面。

	area(μm^2)	testbench clock(ns)	operation time(ns)	cycle	sdc (ns)
Arithmetic Operator (*, /, %)	~590000	N/A	N/A	N/A	10
Iterative Approach	**268584.69**	4.71	3707.55	~787	3.5
Optimized Iterative Approach	278926.95	**4.17**	2100.44	~504	3.2
Booth encoding multiplier + Iterative divider	284726.97	4.21	**1917.98**	**~456**	**3**







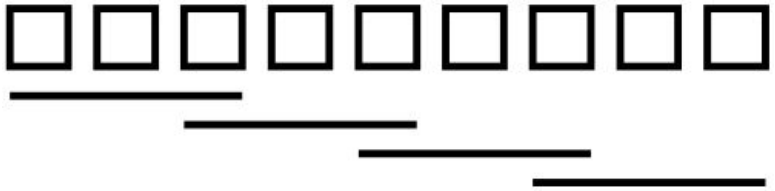
在乘除法中，有些case不必stall 32個cycle才能得到結果，例如：**被乘數/乘數中有0或1、被除數和除數相同、被除數為0、除數為1**等等，其運算結果都相當直觀，所以可以在STATE_MULT或是STATE_DIV開始的時候先加一些簡易的判斷式，若符合條件可以直接輸出結果，而不用浪費32個cycles。以乘法為例：

```
if ((multiplicand == 32'b1) | (multiplier == 32'b1)) begin
        next_done = 1'b1;
        next_product[63:32] = 32'b0;
        next_product[31:0] = (multiplicand == 32'b1) ? multiplier : multiplicand;
end else if ((multiplicand == 32'd2) | (multiplier == 32'd2)) begin //<<1
        next_done = 1'b1;
        next_product = {31'b0, (multiplicand == 32'd2) ? multiplier : multiplicand,
1'b0};
end else if (multiplier == 32'd4) begin //<<2
        next_done = 1'b1;
        next_product = {30'b0, multiplicand, 2'b0};
end else if (multiplier == 32'd8) begin //<<3
        next_done = 1'b1;
        next_product = {29'b0, multiplicand, 3'b0};
end ...
```

可以看出在這些if-else condition裡，done在下個cycle都會直接被拉高而非進到ITER state。在未優化的情況下，iterative approach大概是787 cycles，而多了判斷式的優化之後則會降至504 cycles，約是36%的跌幅。

最終我們採用的乘法器版本使用radix-4 booth encoding，除了可以支援signed-binary運算，cycle數也僅為iterative approach的一半。其運算方法為在乘數的LSB後面加一個extra bit (intially 0)，每次判斷三個bit來加減被乘數到partial sum，表格如下(以y * x為例):

X_2i+1	X_2i	X_2i-1	Operation
0	0	0	+0
0	0	1	+y
0	1	0	+y
0	1	1	+2y
1	0	0	-2y
1	0	1	-y
1	1	0	-y
1	1	1	-0



每做完一次運算後會往左兩個bit再進行下一次判斷(Verilog的實作則是把{$HI, $LO} >> 2再去看最後3個LSB)，32-bit的運算只需要16個cycle就可以完成(iterative approach一次只處理1-bit總共需要32個cycle)，而實際的模擬結果會由504 cycles變成456 cycles，再降低約10%左右，和未優化iterative approach相比則是降低了42%，面積也僅增加了6%左右，是相當划算的trade-off。

Part 3: Collaboration

B05901014 吳靖平	**B05901013 張景皓**
Baseline: cache	Baseline: other modules
Baseline: connect modules (i_MIPS)	Baseline: logic synthesis
Baseline: troubleshooting	Baseline: troubleshooting
Extension: BrPred	Extension: MultDiv
Extension: L2 Cache	