

● Pipeline 利用了指令間的平行度, 增加了 throughput

指令層級的平行度稱作 ILP (Instruction-Level Parallelism)

更高階的有: Program-level parallelism \Rightarrow OS 分派 process 給不同的 CPU core

Thread-level parallelism \Rightarrow OS 分派 thread 給不同的 CPU core

增加 ILP 的方法有 =:

①. Increase pipeline depth (superpipeline) (Stage 數變多)

- 1. Speedup 上升 ($\because \text{Speedup} = \frac{N \times S \times T}{[(S-1) + N] \times T} \div S (\because N \gg S)$)
- 2. Difficulty to balance stages 上升
- 3. Hazard penalty 上升, Hazard 個數也變多

②. Multiple issue: 每個 Stage 可執行多個指令, 造成指令執行率上升 (CPI < 1)

要解決 2 個問題: (IPC \uparrow)

- 1. 包裝指令到 issue slot 中
- 2. 處理 data hazard & control hazard

而 Multiple issue processor 有 2 種 implementation 方法:

- ①. Static multiple issue \Rightarrow 利用 Compiler, 在 Compile time 時就完成 pack instr. & handle hazard
- ②. Dynamic multiple issue \Rightarrow 可以在 run-time 時才處理, 但需利用硬件, processor 來做 pack instr. & handle hazard (Superscalar)

● Speculation (軟、硬皆可)

找到指令間潛在的 ILP 方法就是利用 Compiler or Processor 來猜指令的性質, 才能更佳地排成更有效的順序, 或解決指令間 dependency

①. beg 指令可猜 taken or not taken, 如同前面的 branch delay

改善指令分派率

②. sw - lw 指令可猜不存在 data hazard, 可將 lw 移動到 sw 前

Speculation 機制中需要:

- 1. 確認猜測是否正確
- 2. 修復猜錯造成影響

Speculation 的實現可以分為

- Compiler (static)
- Processor (dynamic)

I. Compiler 來做 Speculation

step: ①. 使用猜測重排指令順序 (有可能猜錯)

②. Insert check code to check the correctness of speculation

③. 提供修補程式


II. Processor 來做 Speculation

①. 猜測重排指令順序

②. 將猜測結果存到一個 buffer 中

③. 若正確: 將 buffer 內容寫到 Reg, MEM
若否: flush buffer, 重新執行正確順序

圖示:

①.  : superpipeline

②.  : multiple issue pipeline

* Multiple issue pipeline 除了 register 間的 dependency
還會有 data memory 的 dependency

③

sw	r10, 41(\$S3)	如果 16+\$S3 = 64+\$S5
lw	\$t1, 16(\$S5)	的話, 表示寫入, 讀取

相同位址 DM 內容
會有 hazard

\because MIPS pipeline 中同一
stage 只有 MEM stage
在做讀或寫
 \therefore 不會有此 hazard

Static Multiple Issue 之指令集架構

- MIPS-64 (軟體)
- VLW (軟體)
- IA-64 (EPIC) (軟, 硬體)

I. MIPS-64

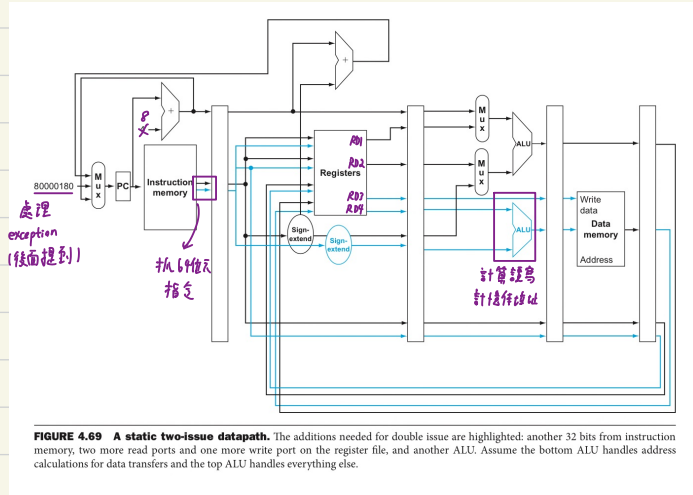
ISA 同前面學的 MIPS-32, 差別在可一個 stage 可執行 2 個指令 (two-issue)

但有限制此指令包為: 一個要是 R-type 或 branch 指令, 一個要是 lw or sw 指令, 若找不到不具 dependency 的指令, 則用 nop 插入指令包
 一個 Cycle 分發 2 個指令需要 fetch 64 位元的 instruction (2 個 32 bit 指令)

各指令間也有在 forwarding path, 每 lw 指令可以 forward 寫入暫存器內容給 R-type 指令



要實現 two-issue 的話, 主要功能元件的 port 需更改, datapath 如下:



上面做 R-type, 下面做 lw, sw

注意: Register file 的寫入 port 跟讀出 port 增加

另外: ∵ EX stage 中 ALU 要用作 R-type & lw, sw 指令使用

∴ 共有 2 個 ALU

PC 改成 PC + P, 一次跳 2 個指令

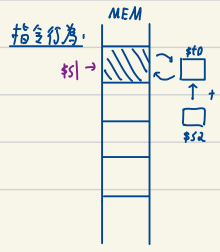
Static two-issue pipeline code scheduling 方式

給予:

```

loop: lw $t0, 0($s1)
      adda $t0, $t0, $s2
      sw $t0, 0($s1)
loop control { addi $s1, $s1, -4
              bne $s1, $0, loop
    
```

設要以 MIPS-64 重排指令, 並避免 pipeline stall
 而 Control hazard 由 HW 解決的話
 ∴ (1,2) 有 load-use 需隔一個 clock cycle
 (2,3) 有 data hazard, 不能在同一指令包
 (4,5) 有 data hazard



clock cycle	R-type or bne	lw or sw
1	<u>nop</u>	lw \$t0, 0(\$s1)
2	addi \$s1, \$s1, 4	<u>nop</u>
3	addu \$t0, \$t0, \$s2	<u>nop</u>
4	bne \$s1, \$0, loop	sw \$t0, 4(\$s1)

要注意的是, \therefore 將 addi 重排至會使到 \$s1 的 sw 之前

至於 sw 使用 \$s1 內容時, 已經減退 4

\therefore sw 指令中的 offset 要更改為 4

但這種方式中 4 個指令包下, 有 3 個是含 nop 的 (ILP 不好)

這是因為, $\left\{ \begin{array}{l} 1. \text{指令數目太少} \\ 2. \text{太多的 data hazard (True hazard 無法改)} \end{array} \right.$

利用 for loop 的 loop unrolling 將指令數上升, 使指令可排為更有效率的指令句

將 loop body 展開, $\text{for } i=1 \text{ to } 20$ $\xrightarrow{\text{unrolling}}$ $\left. \begin{array}{l} s = s+1 \\ s = s+2 \\ \vdots \\ s = s+20 \end{array} \right\}$ 轉成 machine code 時, 無 loop control instruction

\therefore 執行時間更好, 但空間上較差

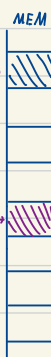
要儲更多的 instr.

eg $\left\{ \begin{array}{l} \text{loop: lw } \$t0, 0(\$s1) \\ \quad \text{addu } \$t0, \$t0, \$s2 \\ \quad \text{sw } \$t0, 0(\$s1) \\ \text{loop control } \left\{ \begin{array}{l} \text{addi } \$s1, \$s1, -4 \\ \text{bne } \$s1, \$0, \text{loop} \end{array} \right. \end{array} \right.$

設 for loop 的 index 為 4 的倍數

則可將 loop body 展開 3 次

lw \$t0, 0(\$s1)	
addu \$t0, \$t0, \$s2	
sw \$t0, 0(\$s1)	
lw \$t0, 4(\$s1)	
addu \$t0, \$t0, \$s2	
sw \$t0, 4(\$s1)	
lw \$t0, 8(\$s1)	
addu \$t0, \$t0, \$s2	
sw \$t0, 8(\$s1)	
lw \$t0, 12(\$s1)	
addu \$t0, \$t0, \$s2	
sw \$t0, 12(\$s1)	
addi \$s1, \$s1, -16	
bne \$s1, \$0, loop	



但這裡發現所有指令都用到了 \$t0 暫存器

然而, 不同 loop body 間的暫存器內容應不存在 dependency 才對, 但 \therefore 重複的 \$t0 使用, 會造成 Compiler 無法

彈性做 Code Scheduling

這種情形稱作: anti-dependence / name dependence

原因來自 Reg Name 重複使用

解決 Antidependence 方法為: 展開時重新命名暫存器 (Register Renaming)

\Rightarrow $\left\{ \begin{array}{l} \text{lw } \$t0, 0(\$s1) \\ \text{addu } \$t0, \$t0, \$s2 \\ \text{sw } \$t0, 0(\$s1) \\ \text{lw } \$t1, 4(\$s1) \\ \text{addu } \$t1, \$t1, \$s2 \\ \text{sw } \$t1, 4(\$s1) \\ \text{lw } \$t2, 8(\$s1) \\ \text{addu } \$t2, \$t2, \$s2 \\ \text{sw } \$t2, 8(\$s1) \\ \text{lw } \$t3, 12(\$s1) \\ \text{addu } \$t3, \$t3, \$s2 \\ \text{sw } \$t3, 12(\$s1) \\ \text{addi } \$s1, \$s1, -16 \\ \text{bne } \$s1, \$0, \text{loop} \end{array} \right.$

clock cycle	R-type or bne	lw & sw
1	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)
2		lw \$t1, 4(\$s1)
3	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)
4	addu \$t1, \$t1, \$s2	lw \$t3, 12(\$s1)
5	addu \$t2, \$t2, \$s2	sw \$t0, 0(\$s1)
6	addu \$t3, \$t3, \$s2	sw \$t1, 4(\$s1)
7		sw \$t2, 8(\$s1)
8	bne \$s1, \$0, loop	sw \$t3, 12(\$s1)

註: addi \$s1, \$s1, -16 放在第一行才會是最佳 case

但後面用到 \$s1 的要補回 16

若 index 為 3 的倍數, 即 loop body 展開 2 次

$\left\{ \begin{array}{l} \text{lw } \$t0, 0(\$s1) \\ \text{addu } \$t0, \$t0, \$s2 \\ \text{sw } \$t0, 0(\$s1) \\ \text{lw } \$t1, 4(\$s1) \\ \text{addu } \$t1, \$t1, \$s2 \\ \text{sw } \$t1, 4(\$s1) \\ \text{lw } \$t2, 8(\$s1) \\ \text{addu } \$t2, \$t2, \$s2 \\ \text{sw } \$t2, 8(\$s1) \\ \text{addi } \$s1, \$s1, -12 \\ \text{bne } \$s1, \$0, \text{loop} \end{array} \right.$

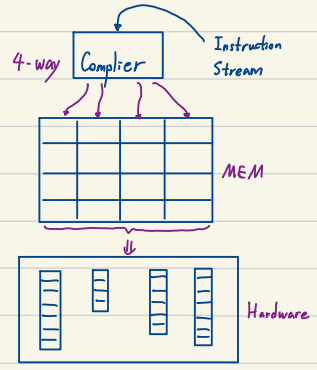
clock cycle	R-type or bne	lw & sw
1	addi \$s1, \$s1, -12	lw \$t0, 0(\$s1)
2		lw \$t0, 8(\$s1)
3	addu \$t0, \$t0, \$s2	lw \$t0, 4(\$s1)
4	addu \$t1, \$t1, \$s2	sw \$t0, 12(\$s1)
5	addu \$t2, \$t2, \$s2	sw \$t1, 8(\$s1)
6	bne \$s1, \$0, loop	sw \$t2, 4(\$s1)

若 index 为 2 的倍数, 即 loop body 展开 1 次

```
lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)
lw $t1, 4($s1)
addu $t1, $t1, $s2
sw $t1, 4($s1)
addi $s1, $s1, -8
bne $s1, $s0, loop
```

clock cycle	R-type or beg	lw & sw
1	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)
2		lw \$t0, 4(\$s1)
3	addu \$t0, \$t0, \$s2	
4	addu \$t1, \$t1, \$s2	sw \$t0, 0(\$s1)
5	beg \$s1, \$s0, loop	sw \$t1, 4(\$s1)

VLIW:



II. Very Long Instruction Word (VLIW)

- 樣利用 Compiler 來做 packing instruction, hazards handling

Packing Instruction 方法为: 把一群指令分類到 issue packet (很長, 執行很多動作指令)

可將 issue packet 視為一個單獨指令

VLIW 的優缺點:

優: ① 硬體較簡單, 擴充性更好 (要增加 issue 個數只需加功能單元即可)

缺: ① Compiler 複雜度上升, Compile time 增長 (∵ 要分析 data stream, 找到沒有 dependency 的指令們才能放到 issue packet)

② Hazard 只能用 Stall 解決 (沒有 forwarding path)

③ Object Code 不相容

④ ∵ VLIW 的基本執行單位很大, 需要很高的 bandwidth

⑤ Code bloat (要做 loop unrolling, program 所需空間变大)

↳ Code size explosion

↳ "The Itanium approach turned out to that the wished-for compilers were basically impossible to write" — Donald Knuth

III. Intel IA-64 架構 (軟硬皆有)

IA-64 為一 Reg-to-Reg 的 RISC ISA, 但支援利用 Compiler 開運的平行度, 又稱作 Explicitly Parallel Instr. Computer (EPIC)

IA-64 可用 Software-based & Hardware-based 兩種方式來 Packing Instr.

① Instruction group: Compiler 將沒有 data dependency 指令裝在一起成為 Instr. group

② Bundle: 每個 bundle 有 3 個指令, 長度為 91 bit, 又有 5 個位元的 template field, bundle 寬度共 128 bits

Template field 會擇用到哪個功能單元

另外, IA-64 提供 Predication 來做猜測, 消除 branch, 減少 Control Hazard

Predication 可消除 if-then-else statement, 但無法處理 for loop, while loop ∴ 可用迴圈展開

```

if (P) S1; else S2  →  MIPS      bne (P), Else   IA-64
                        S1                      (P) S1
                        j Exit                  (~P) S2
                        Else: S2
                        Exit:

```

若無 Predication 將條件指令取代 branch 指令, 會增加 Control Hazard 個數

Dynamic Multiple Issue Processor: (Superscalar)

一樣需先利用 Compiler 做指令排程, 來消除 dependency, 增加指令分派率

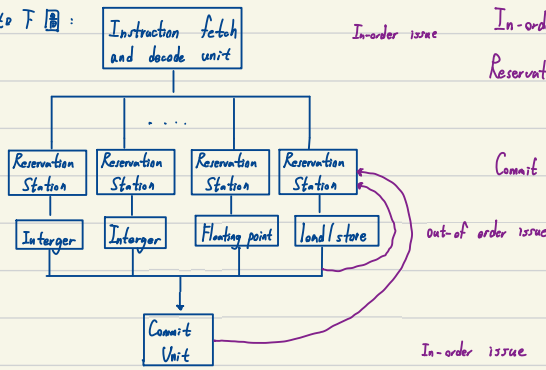
再由 Hardware 確認執行有無錯誤

課本只提到 dynamic pipeline scheduling 這一技術, 用作找到指令在同一个 clock 中執行, 且避免 hazard 和 stall 的發生

Dynamic Pipeline Scheduling

Pipeline 會被分作三個主要單元: Instruction fetch & issue, multiple functional unit, commit unit

如下圖:



In-order issue: 照指令順序去擷取指令, 解碼, 保留指令間的 dependency

Reservation Station: 每個 functional unit 會有對應的 buffer, 用作儲存運算元及運算結果和運算結果

Commit Unit: 將運算結果暫存, 當安全時, 才寫回 Register file 或 MEM

裡面的 buffer 稱作 Reorder buffer

Commit Unit 同樣得照指令順序將結果寫回 Register file 或 MEM

可視 dynamic pipeline scheduling 為分并一個程式的資料流結構, Processor 會以不改變資料流前提下, 以某種順序執行指令
透過讓指令擷取 & 解碼和將結果寫回 Reg. file 時採 in-order issue 方式, 但執行時採 out-of-order 的方式來實現

committed: when an instruction is guaranteed to complete.

Out-of-order execution 需解決之 data hazard

∴ 指令現在不照順序執行, 會有下面了種 data hazard 出現

- 1. True data dependency ⇒ RAW
- 2. Anti dependency ⇒ WAR
- 3. Output dependency ⇒ WAW

I. Read after write:

add \$1, \$2, \$3
sub \$4, \$1, \$1

II. Write after read

add \$1, \$2, \$3
sub \$2, \$5, \$4

III. Write after write

add \$1, \$2, \$3
sub \$1, \$2, \$3

設 pipeline 設計成前面指令的暫存器寫入
可能發生在後面指令的暫存器讀取後
就會出錯

設 pipeline 設計成前面指令的暫存器讀取
可能發生在後面指令的暫存器寫入後
就會出錯

設 add 的寫入發生在 sub 的寫入後
會出錯

然而 antidependency 和 output dependency 只是因為 Programmer 重複使用 Reg 造成的 Storage Conflict

可利用 Reg renaming 來消除, 用 free register 替換來解決

$$\begin{array}{lcl} \text{eg} & & \\ \left(\begin{array}{l} R3 = R3 \times R5 \\ R4 = R3 + 1 \\ R3 = R5 + 1 \end{array} \right. & \xrightarrow{\text{Reg Renaming}} & \begin{array}{l} R6 = R3 \times R5 \\ R4 = R6 + 1 \\ R7 = R5 + 1 \end{array} \end{array}$$

● 各个技巧或元件來提升 ILP 為 software or hardware based? S: 軟, H: 硬, B: 皆有

1. Branch Prediction: B, 如 delayed branch 是利用 Compiler 在做, 而 HW 亦可利用 BHT, BTB 來達成
2. Multiple issue: B, 有 Static 和 Dynamic
3. VLIW: S
4. Superscalar: H
5. Dynamic-scheduling: H
6. Out-of-order execution: H, 用大量功能單元來實現
7. Speculation: B
8. EPIC: B, 有 instruction group & bundle
9. Reorder buffer: H
10. Reg renaming: B
11. Predication: S