1. Computer Abstractions and Technology

1.1 介紹

計算應用的種類與它們的特性

- 個人型計算機
    - 最為人熟知的計算型式
    - 強調以低成本提供單一使用者不錯的效能
    - 能執行其他公司的軟體
- 伺服器
    - 大型主機、迷你計算機及超級電腦的現代版
    - 通常經由網路來使用
    - 伺服器強調其可靠度，其當機的代價通常高於單一使用者的個人電腦
- 嵌入式計算機
    - 計算機中最大的一個族群
    - 在應用與效能的範圍也最廣
    - 嵌入式計算系統是設計來執行單一應用或一組相關的應用
    - 嵌入式應用通常有其獨特的應用需求以及最低效能和嚴格的成本與與功耗限制

1.2 計算機架構的八大理念

- 配合摩爾定律作設計
    - 該定律指出IC容量於每18至24年即加倍
- 用抽象化來簡化設計
    - 計算機架構師與程式師均需想出使自己更有產能的方法，否則其花費在設計上的時間將有如資源量隨摩爾定律而膨脹般遽增。
    - 使用抽象化，可提供一個簡潔的模型予上方各層
- 使經常的情形變快
    - 經常的情形往往也比較少見者為單純且易於改良
    - 需透過仔細的實驗與測量以求確保
- 經由平行性提升效能
- 經由管道處理提升效能
- 經由預測提升效能
    - 在一些情況下假設你的預測已經夠準確並且由錯誤的預測中回復的機制不會太昂貴的話，平均而言猜測結果後就劇以開始繼續作下去會比等到確之結果才更快。
- 記憶體的階層
    - 程式設計師希望記憶體要容量大，速度快且低廉
    - 記憶體的速度會影響效能、容量又限制可以處理問題的規模、而且目前計算機價格的主要部份來自於記憶體的成本。
    - 可以用階層式的不同記憶體來滿足這些互相牴觸的要求，其方式是將最快最小和每位元單價最貴的記憶體至於頂層，而最慢最大和每位元單價最廉價的記憶體置於底層。
- 經由冗餘提升可靠性
    - 計算機不只快，還要可靠。
    - 物理裝置難免失效，於是我們以加入額外組件以便失效發生時接手及幫助檢出失效情況的方式使系統成為可靠。

1.3 你的程式之下

- 每一個計算機中最重要的有兩類：作業系統與編譯器
- 作業系統將使用者應用與硬體關聯上，並提供各種服務與管控的功能
    - 處理基本輸出入的動作
    - 配置儲存體與記憶體
    - 在多個應用同時使用計算機時提供有保護的分享
- 編譯器執行另一重要功能：將高階語言如C、C++、Java或Visual Basic所寫的程式翻譯成硬體可執行的一群指令。
- 程式將一道以符號表示的組合語言翻譯成二進形式的機器語言，稱為組譯器。
- 程式師應將他們的生產力，以及他們可以更清楚思考，歸功於高階程式語言以及可以將以其編撰的程式翻譯成組合指令的編譯器。
- 高階程式語言的好處
    - 他們使用英語文字與算術表示法，容許程式師以更自然的語言思考，也使得程式看起來更像文字而非晦澀難懂的符號表
    - 更好的程式師產生力，軟體開發領域裡少有的共識之一，就是當程式開發時如果能採用可以用更少行來表達一個思想的語言，則所需時間較少
    - 程式不須與開發他們時所使用的計算機相關，此乃因為編譯器與組譯器可將高階語言程式翻譯成任何計算機的二進指令。

1.4 覆蓋之下

- 所有計算機內的硬體都執行以下的基本功能：輸入資料、輸出資料、處理資料以及儲存資料。
- 計算機的兩種關鍵組件是輸入裝置以及輸出裝置
- 有些裝置比如無線網路可同時提供計算機輸入與輸出
- 存資料的安全地方
    - 計算機內的記憶體是揮發性記憶體，一旦失去電力就會忘記
    - 光碟並不因你切斷電力而失去所記錄的影片，因此其為一種非揮發性記憶體
    - 我們以名詞主記憶體來代表保存正在執行中的數據及程式的揮發性記憶體
    - 我們以名詞次記憶體來代表保存執行之間的數據及程式的非揮發性記憶體
- 有網路的電腦的優勢
    - 通訊：資訊可在計算機間以高速交換
    - 資源分享：與其每個計算機擁有自己的輸入/輸出裝置，同一網路上的計算機可共享之
    - 非本地存取：連結遠距離的計算機，使用者即可使用遠方的計算機

1.5 建構處理器與記憶體的技術

1.5.1 科技趨勢

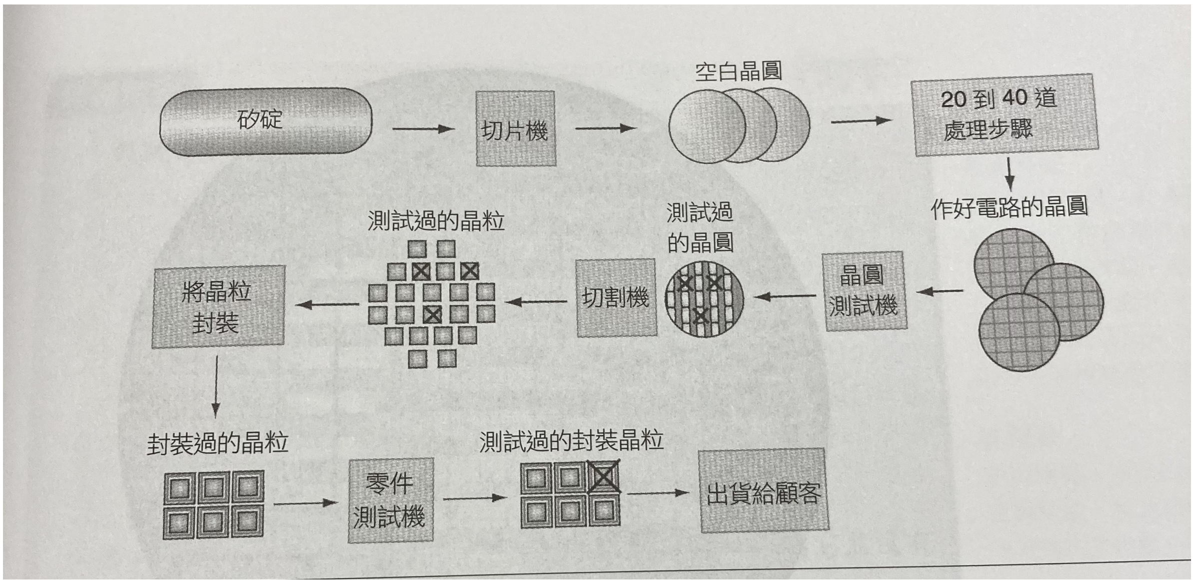
- 電子產品逐漸發展中
- 追求效能增加與價格減少
- 效能與價格比越來越高

年代	一些科技	效能與價格比
1951	真空管	1
1965	電晶體	35
1975	積體電路	900
1995	超大積體電路	2400000
2013	極大積體電路	250000000000

1.5.2 半導體科技

- 半導體由矽組成
- 加上導體、絕緣體、開關改變他的性質

1.5.3 積體電路的製造



- 良率：一個晶元上好晶粒占所有晶粒數量的百分率。

1.5.4 良率

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die Area}/2))^2}$$

面積與不良率的非線性關係

- 晶圓圓片價格與面積是固定的
- 不良率取決在於製造的流程
- 晶片損毀區域大小取決於架構與電路設計

1.6 效能

1.6.1 回應時間與流通量

回應時間：要花多少時間能夠完成工作

流通量：在一個時間單位內，總共能夠完成多少工作

1.6.2 時間

定義時間：

- 完成一個工作總共需要多少時間，包含硬碟與記憶體存取，輸入輸出的動作，OS...
- 可能包含其他程式在多執行緒的執行時間
- 包含很多因素

CPU的時間：

- 被稱作「CPU的執行時間」或者「CPU的時間」
- 通常還會細分成「對於OS的CPU時間」與「對於使用者程式的CPU時間」

CPU的表現：對於一個單一的程式，使用者的CPU時間。

1.6.3 相關表現

定義表現：$\text{Perfomance} = \dfrac{1}{\text{Execution Time}}$

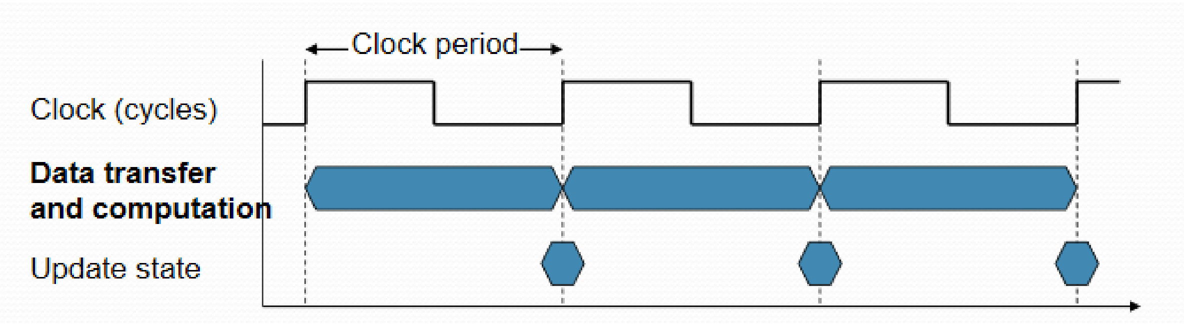
若我們說x比y快n倍，則

$$n = \frac{\text{Perfomance } p_x}{\text{Perfomance } p_y} = \frac{\text{Execution time } e_y}{\text{Execution time } e_x}$$

要精確的確定一個程式的執行時間不是一件簡單的事情

我們通常都是「預估」一個程式的執行時間

1.6.4 CPU的時脈



時脈週期：一個時間週期的時間，例如$250\text{ps} = 2.5 \times 10^{-10}\text{s}$

時鐘頻率：一秒鐘的時脈次數，例如$4.0\text{GHz} = 4000\text{MHz} = 4 \times 10^{9}\text{Hz}$

$$\text{CPU時間}：\text{CPU Clock Cycles} \times \text{Clock Cycles Time} = \frac{\text{CPU Clock Cycles}}{\text{Clock Rates}}$$

表現會隨著「減少一個時脈週期的時間」、「增加時鐘頻率」進步

通常來說，設計硬體的人會在這兩者之間衡量取決。

Example

Computer A: 2GHz clock, 10s CPU Time

設計一個電腦B，目標是6s CPU Time，且clock cycle增加1.2倍

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2\text{Clock Cycles}_A}{6s}$$

$$\text{Clock Cycles}_A = \text{CPU Times}_A \times \text{Clock Rates}_A = 10s \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times 20 \times 10^9}{6s} = 4\text{GHz}$$

1.6.5 指令執行時間

時間單位：依照使用者的看法，通常來說是秒。

CPU時間：電腦利用固定頻率的時脈來執行指令，並且決定事件發生的時間。

執行時間以cycle為單位。

1.6.6 程式執行時間

CPU的執行程式的時間：

$$\text{Clock Cycle for program} \times \text{Clock Cycle Time} = \frac{\text{Clock Cycles for Program}}{\text{Clock Rate}}$$

CPI：Clock cycles per Instruction，也就是一個指令所需的時脈週期

對於程式的CPU的時脈週期：程式的指令數 x CPI

1.6.7 指令數量與CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycles Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rates}}$$

決定程式指令數量的要素：決定於程式、指令集架構與編譯器

平均一個指令的時脈週期：決定於CPU，也被不同種類的指令所影響

Example

A電腦：$\text{Cycle Time} = 250\text{ps}, \text{CPI} = 2.0$

B電腦：$\text{Cycle Time} = 500\text{ps}, \text{CPI} = 1.2$

一樣的指令集架構，哪一個比較快速，然後大概需要多少時間？

在A電腦，一個指令通常需要執行$250 \times 2.0 = 500\text{ps}$

在B電腦，一個指令通常需要執行$500 \times 1.2 = 600\text{ps}$

故A電腦比B電腦快。

A電腦比B電腦快$\frac{600}{500} = 1.2$倍

1.6.8 更多關於CPI的部分

不同的指令類別，使用不同數量的時脈

$$\text{Clock Cycles} = \sum_{i=1}^{n}(CPI_i \times IC_i)$$

計算加權平均的CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n}(CPI_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}})$$

Example

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

若Sequence 1的$IC = 5$，則$\text{Clock Cycles} = 2 \times 1 + 2 \times 1 + 3 \times 2 = 10$

所以$\text{Avg. CPI} = \frac{10}{5} = 2$

若Sequence 2的$IC = 6$，則$\text{Clock Cycles} = 4 \times 1 + 1 \times 2 + 1 \times 3 = 9$

所以$\text{Avg. CPI} = \frac{9}{6} = 1.5$

1.6.9 對於CPU表現的總結

$$\text{CPU Times} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycles}}$$

表現通常依賴於：

1. 演算法(影響指令數量與可能影響CPI)
2. 程式語言(影響IC與CPI)
3. 編譯器(影響IC與CPI)
4. 指令集架構(影響IC與CPI以及時脈)

1.7 功耗壁障

- 對於CMOS，能耗可以這樣表示：
  - $\text{Energy} \propto \frac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Clock Rate}$

    其中Energy是能量，Capacitive load是電容性負載，Voltage是電壓。Clock Rate是時脈。
- 降低電壓會使得電晶體漏電太多，有如水龍頭無法關緊，故降低電壓可以有效降低能量，但不適用於現代的方法。

2. Instructions: Language of the Computer

2.1 介紹

- 計算機語言中的單字稱為指令，而它的字彙稱為指令集。

2.1.1 MIPS運算元

名稱	舉例	註解
32個暫存器	`$s0-$s7,` `$t0-$t9，$zero，` `$a0-$3，$v0-$v1，$gp，` `$fp，$sp，$ra，$at`	快速存取資料的地方。在MIPS中，資料必須要在暫存器裡才能執行運算，暫存器$zero恆等於0，而暫存器$at則是保留給組譯器來處理值很大的常數。
2^{30}記憶體字組	`Memory[0], Memory[4]` `...` `Memory[4294967292]`	在MIPS裡，記憶體只能由資料傳輸指令來存取。MIPS使用位元組位址，所以連續的字組位置相差4。記憶體裡儲存資料結構，陣列和溢出暫存器(spilled register)

2.1.2 MIPS組合語言

指令	舉例	意義	註解
加法	`add $s1,$s2,$s3`	`$s1=$s2+$s3`	三個暫存器運算元
減法	`sub $s1,$s2,$s3`	`$s1=$s2-$s3`	三個暫存器運算元
加立即值	`addi $s1,$s2,20`	`$s1=$s2+20`	加上常數
載入字組	`lw $s1, 20($s2)`	`$s1=Memory[$s2+20]`	字組由記憶體載入至暫存器
儲存字組	`sw $s1, 20($s2)`	`Memory[$s2+20] = $s1`	字組由暫存器儲存至記憶體
載入半字組	`lh $s1, 20($s2)`	`$s1 = Memory[$s2+20]`	半字組由記憶體載入至暫存器
載入無號半字組	`lhu $s1, 20($s2)`	`$s1 = Memory[$s2+20]`	無號半字組由記憶體載入至暫存器
儲存半字組	`sh $s1,20($s2)`	`Memory[$s2+20] = $s1`	半字組由暫存器儲存至記憶體
載入位元組	`lb $s1,20($s)`	`$s1=Memory[$s2+20]`	位元組由記憶體載入至暫存器
載入無號位元組	`lbu $s1,20($s2)`	`$s1=Memory[$s2+20]`	無號位元組由記憶體載入至暫存器
儲存位元組	`sb $s1,20($s2)`	`Memory[$s2+20]=$s1`	位元組由暫存器儲存至記憶體
載入連結的字元組	`ll $s1,20($s2)`	`$s1=Memory[$s2+20]`	作為不可分割的（記憶體與儲存器內容）交換中第一部分的載入字元組
條件式儲存字元組	`sc $s1,20($s2)`	`Memory[$s2+20]=$s1;$s1=0或1`	作為不可分割的（記憶體與儲存器內容）交換中第二部分的載入字元組
載入上半部立即值	`lui $s1, 20`	`$s1=20*65536$`	載入常數至較高的16位元
及	`and $s1,$s2,$s3`	`$s1=$s2 & $s3`	三個暫存器運算元；逐位元的及運算
或	`or $s1,$s2,$s3`	`$s1=$s2 | $s3`	三個暫存器運算元；逐位元的或運算
反或	`nor $s1,$s2,$s3`	`$s1=~($s2 | $s3)`	三個暫存器運算元；逐位元的反或運算

指令	舉例	意義	註解
及立即值	`andi $s1,s2,20`	`$s1 = s2 & 20`	暫存器與常數做逐位元的及運算
或立即值	`ori $s1,s2,20`	`$s1 = s2 & 20`	暫存器與常數做逐位元的或運算
邏輯左移	`sll $s1,s2,10`	`$s1 = $s2 << 10`	左移常數個位元位置
邏輯右移	`srl $s1,s2,10`	`$s1 = $s2 >> 10`	右移常數個位元位置
若等於則分支	`beq $s1,$s2,25`	若 `($s1==$s2)` 則前往 `PC+4+100`	等於測試：PC相對的分支
若不等於則分支	`bne $s1,$s2,25`	若 `($s1!=$s2)` 則前往 `PC+4+100`	不等於測試：PC相對的分支
若小於則分支	`slt $s1,$s2,25`	若 `($s2<$s3)` ，`$s1=1`; 否則 `$s1=0`	小於比較，用於beq, bne
無號若小於則設定	`sltu $s1,$s2,$s3`	若 `($s2<$s3)` ，`$s1=1`; 否則 `$s1=0`	無號數的小於比較
若小於立即值則設定	`slti $s1,$s2,20`	若 `($s2<20)` ，`$s1=1`; 否則 `$s1=0`	小於某常數的比較
若無號小於立即值則設定	`sltiu $s1, $s2, 20`	若 `($s2<20)` ，`$s1=1`; 否則 `$s1=0`	無號數的小於某常數的比較
跳躍	`j 2500`	前往10000	跳至目的位置
透過暫存器跳躍	`jr $ra`	前往 `$ra`	用於switch敘述、程序返回
跳躍並連結	`jal 2500`	`$ra=PC+4 前往 10000`	用於程序呼叫

2.1.3 Opcodes Table

Opcode bitfields						
add	000000	rs	rt	rd	00000	100000
sub	000000	rs	rt	rd	00000	100010
addi	001000	rs	rt	imm		
lw	100011	rs	rt	offset		
sw	101011	rs	rt	offset		
lh	100001	rs	rt	offset		
lhu	100101	rs	rt	offset		
sh	101001	rs	rt	offset		
lb	100000	rs	rt	offset		
lbu	100100	rs	rt	offset		
sb	101000	rs	rt	offset		
lui	001111	rs	rt	imm		
and	000000	rs	rt	rd	00000	100100
or	000000	rs	rt	rd	00000	100101
nor	000000	rs	rt	rd	00000	100111
andi	001100	rs	rt	imm		
ori	001101	rs	rt	imm		
sll	000000	rs	rt	rd	sa	000000
srl	000000	rs	rt	rd	sa	000010
beq	000100	rs	rt	offset		
bne	000101	rs	rt	offset		
slt	000000	rs	rt	rd	00000	101010
sltu	000000	rs	rt	rd	00000	101011
slti	001010	rs	rt	imm		
sltiu	001011	rs	rt	imm		
j	000010	target				
jal	000011	target				

2.2 計算機硬體的運作

- 每一道MIPS算術指令只能執行一種運算且永遠一定使用三個變數。
- 若我們要相加四個變數a, b, c, d，則可以寫成以下

```
add a, b, c
add a, a, d
```

2.3 計算機硬體的運算元

- MIPS架構中暫存器的大小是32位元；一群群的32位元如此經常地被使用，因此它們也在MIPS架構中被稱為字組(word)。
- 很大數量的暫存器單純地由於其內部的電子訊號必須傳遞更遠而需時更久、可能導致時脈週期時間增加。

2.3.1 記憶體運算元

- 程式語言具有內含單一數據元素的簡單變數，如以上各例所見；然而也有較複雜的資料結構—陣列(array)與結構(structurue)。

  該等複雜資料結構可包含較計算機中所具有的暫存器數目。

- MIPS指令的算術運算僅在暫存器上運作；因此MIPS也必須具有能在暫存器及記憶體間轉移資料的指令。

  該類地指令稱為資料轉移指令。

- 存取記憶體字組時，指令必須提供記憶體的位址。
- 記憶體是一個很大的一維陣列，並且由0開始的、稱為位置者作為其索引。
- 傳統上稱呼將記憶體中的值複製至暫存器中的資料轉移指令為載入。載入指令的格式包含運作名稱、後接將被載入值的暫存器、之後是用以存取記憶體的另一個常數以及另一個暫存器。

  記憶體位址由該常數加上第二個暫存器的內容而得。

- 實際的MIPS該指令稱為lw，代表載入字組(load word)

Example - 編譯有一個運算元在記憶體中的賦值敘述

假設A為含有100個字組的陣列，且編譯器已如前述將g與h存於`$s1`以及`$s2`中。另設陣列的開始位址，或稱基底位置，存於`$s3`中。

編譯下述C賦值敘述：

```
g=h+A[8]
```

先把`A[8]`轉移至暫存器中：`lw $t0, 8($s3)`

再將其與`h`相加，存至`g`：`add $s1, $s2, $t0`

Example - 編譯使用到讀取與儲存的賦值敘述

假設變數h的值已被編譯器存至暫存器`$s2`，且base address陣列A已被編譯器存至暫存器`$s3`，求下列的C語言編譯後的MIPS組語敘述。

```
A[12] = h + A[8]
```

```
ls $t0, 48($s3)   # 取得A[12]的值，並且放到暫存器
ls $t1, 32($s3)   # 取得A[8]的值，並且放到暫存器
add $t0, $s2, $t1 # h與A[8]做相加
sw $t0, 48($s3)   # 存回A[12]的記憶體中
```

2.3.2 常數或immediate運算子

- 程式很常在運算子中使用常數，例如增加index來讀取陣列的下一個元素。

Example - 用load word來做常數相加

假設`$s1 + AddrConstant4`是常數4的記憶體位址。

```
lw $t0, AddrConstant4($s1)  #$t0 = 常數4
add $s3, $s3, $t0           #$s3 = $s3 + $t0 = $s3 + 4
```

Example - 用add immdiate來做整數相加

我們可以用addi來直接做整數的相加，省略掉讀取或暫存的部分。

```
addi $s3, $s3, 4           #$s3 = $s3 + 4
```

2.4 有號數與無號數

與數位邏輯相同，MIPS中用的32位元數字，最左邊的數字代表有號或者無號，針對這個對二進制運算後轉成10進制。

2.5 指令呈現在電腦的方式

- 暫存器很常被指令所提及，因此我們可以透過指令的編號，來將指令表達成一連串的數字。

Example - 將MIPS組語翻譯成機器碼

將以下的MIPS組語翻譯成機器碼。

```
add $t0, $s1, $s2
```

我們可以翻譯成

0	17	18	8	0	32
MIPS指令形式	第一個指令元素	第二個指令元素	目標暫存器	沒有用到的指令格	MIPS指令形式
000000	10001	10010	01000	00000	100000

2.5.1 MIPS欄位

- MIPS欄位有各自的名子，方便我們去討論。

R-format

op	rs	rt	rd	shamt	funct
這個指令的運算類別	第一個指令元素	第二個指令元素	目標暫存器	左移/右移數量	函數

I-format

op	rs	rt	constant or address
這個指令的運算類別	第一個指令元素	第二個指令元素	常數或暫存器的地址

Example - 翻譯MIPS組語翻譯至機器語言

若base address陣列A存於暫存器`$t0`，變數h存於暫存器`$s2`中，且翻譯以下的C語言至機器語言。

```
A[300] = h + A[300];
```

首先我們要先把C語言轉成MIPS指令

```
lw $t1, 1200($t0)add $t0, $t1, $s2sw $t1, 1200($t0)
```

則接下來將其轉為R-format或者I-format

OP	RS	RT	RD	ADDR/SHAMT	FUNC
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

再將每一個欄位轉成二進制

100011	01001	01000	0000010010110000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000010010110000		

2-6 邏輯運算子

Logical Operation	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
bit-by-bit NOT	~	~	nor

Example - 將MIPS組語翻譯至機器語言

若我們假設`$s0`為9_{ten}，翻譯以下MIPS組語至機器語言，並求出`$s0`的值。

```
sll $t2, $s0, 4
```

我們要使用R-format中的shamt來代表移位的格數，故

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

接著再將其翻譯成長度為32的二進制碼。

op	rs	rt	rd	shamt	funct
000000	00000	10000	01010	00100	000000

也就是

```
0000 0000 0001 0000 0101 0001 0000 0000
```

也因為左移4格，故原先的

```
$s1 = 0000 0000 0000 0000 0000 0000 0000 1001
```

變成了

```
$t1 = 0000 0000 0000 0000 0000 0000 1001 0000
```

故`$t1`的值為144。

2-7 能作決定的指令

- MIPS組合語言包含了兩個作決定的指令，兩者均與if且含有go to的敘述類似。
    - ```
      beq register 1, register 2, L1
      ```

      若register 1內含的值與register 2內含的值相同，則接下來執行被標記為L1的敘述。
    - ```
      bne register 1, register 2, L1
      ```

      若register 1內含的值與register 2內含的值不同，則接下來執行被標記為L1的敘述。
- MIPS組合語言包含有一暫存器跳躍(jump register, jr)指令，意為無條件跳躍至暫存器內容所示之位址處。
- 通常來說，我們會用I-format來構造bne/beq指令
    - opcode就是bne、beq的opcode
    - rs與rt用來比較兩個的暫存器
    - immediate用來儲存word address
        - 指令通常都是word aligned (byte address通常都是隔4，也就是最後兩個bit永遠都是0)
        - 如果敘述達成，那麼就跳躍到$PC = (PC + 4) + (Immediate + 4)$
        - 如果敘述不達成，那麼就跳躍到$PC = (PC + 4)$
        - 如此一來，我們可以掌握大概±2^{17}個byte，來進行跳躍。
- 通常來說，我們會用J-format來構造jal或者j指令
    - J-format指令由6bits的opcode與26bits的target address所組成
    - 跳躍一樣是跳躍word address
    - 如果我們一定要指定32-bits的address
        - 我們會把位址設定在暫存器，接著用jr來進行跳躍
        - `jr $ra    #跳躍到$ra存進的記憶體地址`

Example - 將C語言編譯至MIPS碼

下列程式片段中，f、g、h、i及j為變數。若該五變數f至j對應至五個暫存器`$s0`至`$s4`，該C if敘述編譯成的 MIPS 碼為何？

```
if(i == j) f=g+h else f=g-h
```

```
bne $s3, $s4, Else          #若i≠j則跳到Elseadd $s0, $s1, $s2          #把$s1, $s2相加並且assign到$s0j Exit                        #跳躍至ExitElse: sub $s0, $s1, $s2Exit:
```

Example - 將C語言編譯至MIPS碼

設i及k在暫存器`$s3`及`$s5`中且save數列的基底位址在`$s6`中。相對於此C片段的MIPS組合碼為何？

```
while(save[i] == k){    i += 1;}
```

```
Loop: sll $t1, $s3, 2  #暫時暫存器$t1=i*4add  $t1, $t1, $s6     # t1 = save[i]的位址lw   $t0, 0($t1)       # 暫時暫存器$t0=save[i]bne  $t0, $s5, Exit    # 若save[i]≠k則前往Exitaddi $s3, $s3, 1       # i = i + 1j Loop                 # 前往LoopExit:
```

Example - 在小於時設定

```
slt $t0, $s3, $s4     # 若$s3 < $s4，則設定$t0 = 1
```

```
slti $t0, $s3, 10     # 若$s3 < 10，則設定$t0 = 1
```

Example - 有號與無號的比較

若`$s0`內含二進制數字

```
1111 1111 1111 1111 1111 1111 1111 1111
```

及暫存器`$s1`內含二進制數字

```
0000 0000 0000 0000 0000 0000 0000 0001
```

則下列兩指令執行後暫存器`$t0`及`$t1`的值各為何？

```
slt  $t0, $s0, $s1
sltu $t0, $s0, $s1
```

```
slt  $t0, $s0, $s1  #-1         < 1, $t0 = 1
sltu $t0, $s0, $s1  #4294967295 > 1, $t0 = 0
```

Example - 邊界檢測捷徑

若`$s1` >= `$t2`或者`$s1`為負，則分支至IndexOutOfBounds

```
sltu $t0, $s1, $t2
beq  $t0, $zero, IndexOutOfBounds
```

2.8 Supporting Procedures in Computer Hardware

2.8.1 Procedure Calling Step

1. 把參數放進變數暫存器(`$a`)
2. 跳到函數標籤(jal ProcedureLabel)
3. 讀取函數的資料
4. 運行函數的運算
5. 把結果丟到`$v`暫存器
6. 回去呼叫的原點(jr `$ra`)

2.8.2 Register Usage

- `$a0` - `$a3`: 變數暫存器
- `$v0`, `$v1`: 結果暫存器
- `$t0` - `$t9`: 暫時暫存器
- `$s0` - `$s7`: 儲存暫存器(Callee可以儲存或複寫，Caller在Caller做回傳後使用)
- `$gp`: 靜態資料的全域指標暫存器
- `$sp`: 堆疊指標暫存器
- `$fp`: 堆疊框指標暫存器
- `$ra`: 回傳地址暫存器

2.8.3 Procedure Call Instructions

- 用來呼叫函數的指令：`jal` (jump and link)
  - 用法：`jal ProcedureLabel`
  - 把$PC + 4$的位置放到暫存器`$ra`上
  - 跳到目標位址
- 用來回去原函數的指令：`jr`
  - 用法：`jr $ra`
  - 複製`$ra`暫存器的值到程序計數器PC上

2.8.4 Callee的權利，Caller的責任

- Callee的權利
  - 可以自由的使用V/A/T暫存器
  - 可以認定參數有被正確的傳遞
  - 為了Callee的權利，Caller必須要負上一定的責任，在有必要的時候把值傳至指定的暫存器
- Caller的責任
  - 回傳地址：`$ra`
    - 這樣函數呼叫時會複寫`$ra`
  - 變數：`$a0`, `$a1`, `$a2`, `$a3`
    - 如果在函數運行完後，我們還依然要用到這些變數的話
  - 暫時暫存器：`$t0` - `$t9`
    - 如果在函數運行完後，我們還依然要用到這些暫時暫存器的值的話
  - 回傳值：`$v0`, `$v1`

        - 如果在函數運行完後，回傳的值有在這些暫存器上的話

2.8.5 Caller的權利，Callee的責任

- Caller的權利
    - 可以自由的使用`$s`暫存器，不用擔心被callee所覆寫
    - 可以認定回傳的結果有被正確的傳遞
    - 為了Caller的權利，Callee必須要負上一定的責任，也就是在Callee執行完後，把S暫存器復原

2.8.6 函數呼叫的溝通

- Callee的義務
    - 如果用到了`$s`，或者有很大的local structure，那麼就下拉`$sp`來讓我們之後能對暫存器做還原。
        - `addi $sp, $sp, -12`
    - 如果用到了`$s`，那麼就先把原先的值做備份。
        - `sw $s0, 8($sp)`
    - 從`$a0 ~ $a3`來拿取函數的參數。
    - 執行函數的指令
    - 如果不是void，那就把要回傳的值丟到`$v0, $v1`。
    - 函數結束後，把前面的兩個步驟反向操作來還原。
    - `jr $ra`
- Caller的義務
    - 下拉`$sp`來還原暫存器。
        - `addi $sp, $sp, -28`
    - 儲存`$ra`的值，因為`jal`會影響它的值
        - `sw $ra, 24($sp)`
    - 如果函數呼叫結束之後，我們還會用到`$a, $t, $v`的值的話，我們把這些值壓到stack，或者從`$s`作複製。
        - `sw $t0, 20($sp)`
    - 把前面四個變數丟到暫存器`$a0~$a3`，額外的變數將會被丟到stack，也就是`$a4 = 0($sp)`
    - `jal`到想要的函數
    - 反向操作前面三個步驟。

An Example of Passing Argument

```
int doh(int i, int j, int k, int l, int m, char c, int n){  return i+j+n}
```

```
doh: lw  $t0, 4($sp)     add $a0, $a0, $a1     add $v0, $a0, $t0     jr  $ra
```

Leaf Procedure Example

```
int leaf_example(int g, h, i, j){   int f;   f = (g + h) - (i + j);   return f;}
```

- 因為有四個變數，所以第四個變數會被壓到stack上。
- 因為有local變數f，所以這個部分會被壓到stack上。
    - Callee的義務
- 回傳值到`$v0`

```
leaf_example: addi $sp, $sp, -4           sw   $s0, 0($sp)      # 把$s0存到stack上，這是callee的義務            add  $t0, $a0, $a1                  add  $t1, $a2, $a3              sub  $s0, $t0, $t1              add  $v0, $s0, $zero  # 運行程式     lw   $s0, 0($sp)                    addi $sp, $sp, 4      # 還原$s0，這是callee的義務           jr   $ra              # 回傳
```

2.8.7 非末端程序

- 非末端程序的舉例：一個副程式呼叫自己，或者呼叫別的副程式
- 對於巢狀呼叫，caller必須要儲存這些東西到stack：
    - 它的回傳地址`$ra`
    - 在呼叫之後，所有的變數暫存器`$a`，暫時暫存器`$t`，還有它的回傳值`$v`
    - 在呼叫之後還原回去

Non-Leaf Procedure Example

```
int fact(int n){    if (n < 1){     return 1;   }else{      return n * fact(n-1);   }}
```

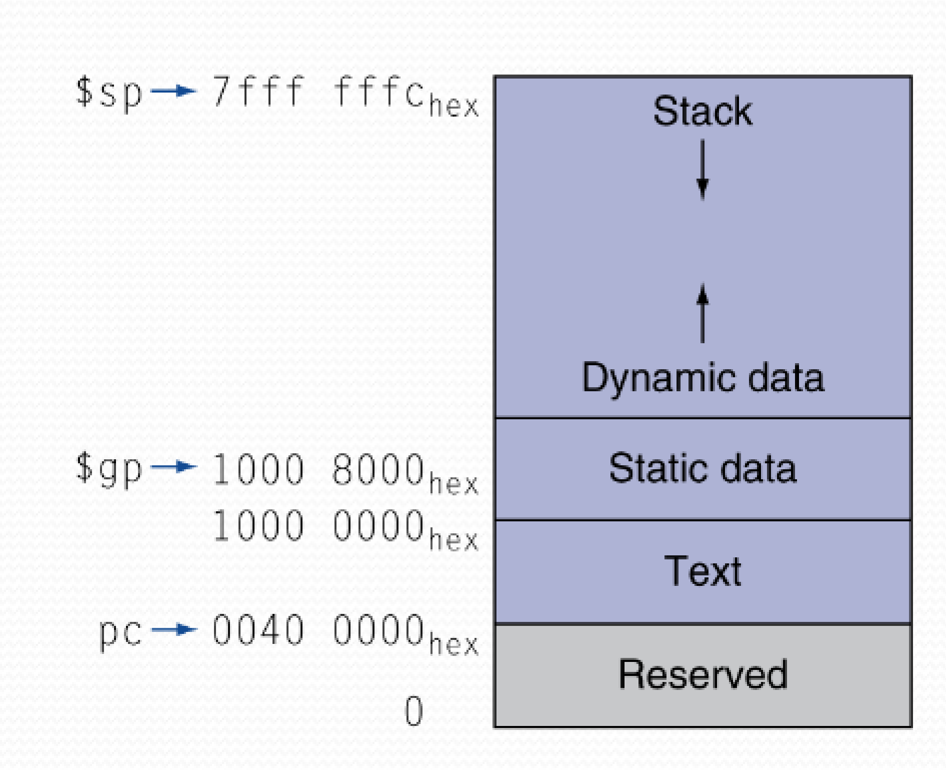
- 變數n在`$a0`
- 結果在`$v0`

```
fact:     addi $sp, $sp, -8     #讓stack儲存兩個東西(caller的義務)      sw   $ra, 4($sp)      sw   $a0, 0($sp)      slti $t0, $a0, 1      #如果$a0小於1，那$t0就會被設置成0      beq  $t0, $zero, L1   #如果$t0不等於0，那就會跳到L1      addi $v0, $zero, 1     #如果是0的話，那就把return 1加回去      addi $sp, $sp, 8      #把stack前面兩個東西pop掉     jr   $ra              L1:  addi $a0, $a0, -1     #實作n-1     jal  fact   #跳到fact，並且將$ra設置成下面的(PC+4)      lw   $a0, 0($sp)      #恢復之前的$a0      lw   $ra, 4($sp)      #恢復之前的$ra      addi $sp, $sp, 8      #把stack前面兩個東西pop掉      mul  $v0, $a0, $v0   #把$a0跟$v0做相乘      jr   $ra
```

2.8.8 Stack的區域資料

- 區域資料會被callee所分配
- Procedure Frame($fp)：用來給編譯器去控制stack的儲存

2.8.9 記憶體的布局



- Text: 程式碼
- Static data: 用來放全域變數
    - 例如一些static的變數、靜態記憶體array、字串也會放在這裡
    - `$gp` 用來設置初始的記憶體位置，讓我們可以很快速的找到靜態的資料
- Dynamic data: heap
    - 一些動態陣列，或者Java的new實作類別
- Stack
    - 自動儲存空間

3. Arithmetic for Computers

加法跟減法

在二進制作加法、減法跟在十進制作加法差不多概念，就是從右加或減到左

Example 1

嘗試用二進制加法把6_{ten}與7_{ten}加起來。

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}}
\end{array}
$$

Example 2

嘗試用二進制減法把7_{ten}減去6_{ten}

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
\end{array}
$$

或者你可以用補數來做減法，先把6_{ten}取補數然後加起來

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
\end{array}
$$

溢位

若一個有號數用32個bit所儲存，則做相加或相減可能會使有號數結果超過32個bit，超出變成33個bit

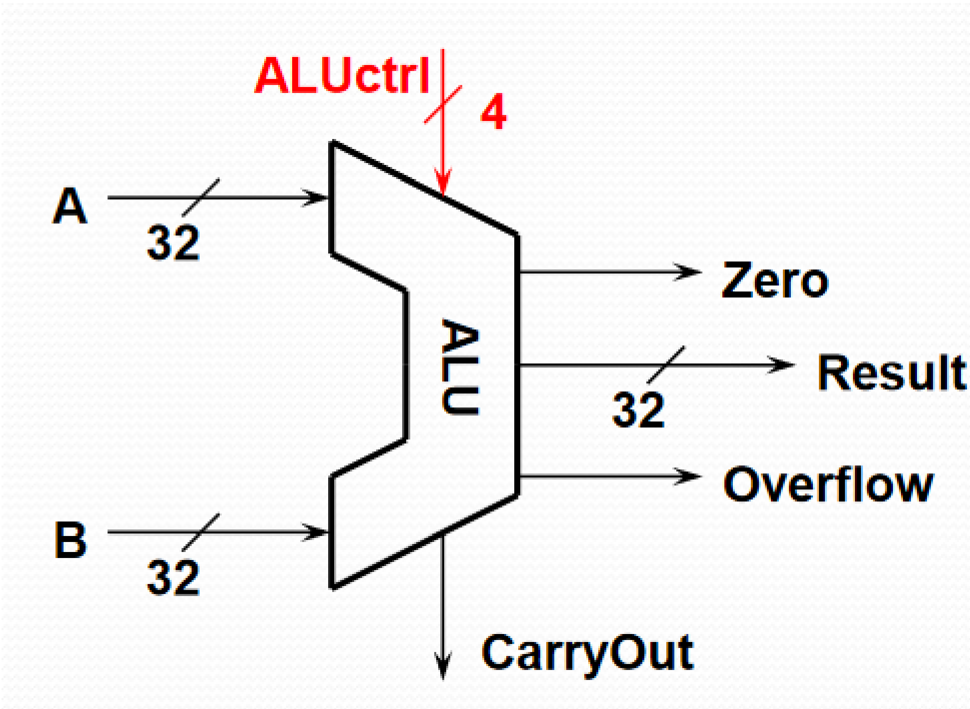
在這種情況，我們稱作為溢位，溢位會使最後相加的結果是錯誤的，見以下表格。

	A	**B**	**結果(R)**
A+B	$A \geq 0$	$B \geq 0$	$R < 0$
A+B	$A < 0$	$B < 0$	$R > 0$
A-B	$A \geq 0$	$B < 0$	$R < 0$
A-B	$A < 0$	$B \geq 0$	$R > 0$

在MIPS中提供了兩種運算子。

- add, addi, sub，皆為有號數加法、減法，會在溢位發生時給予異常的結果
- addu, addiu, subu, 皆為無號數加法、減法，會比有號數加法多一個bit，在有號數溢位發生時，顯示正常的結果

ALU的功能介紹



- 輸入端A：輸入一個32位二進制數字
- 輸入端B：輸入一個32位二進制數字
- ALUctrl：輸入一個4位數，用來設定運算子，例如0010就是ADD
- Zero：輸出0代表結果不是0，輸出1代表結果為0
- Result：輸出一個32位二進制數字，代表運算的結果
- Overflow：輸出1代表結果溢位，輸出0代表結果沒有溢位
- CarryOut：輸出1代表進位，輸出0代表沒有進位

ALUctrl 表格

ALU控制用的二進制數字	函數
0000	ADD
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

設計ALU的指南

Trick 1 - 分而治之

處理and、or之類的東西，其實我們可以一個bit一個bit做運算。

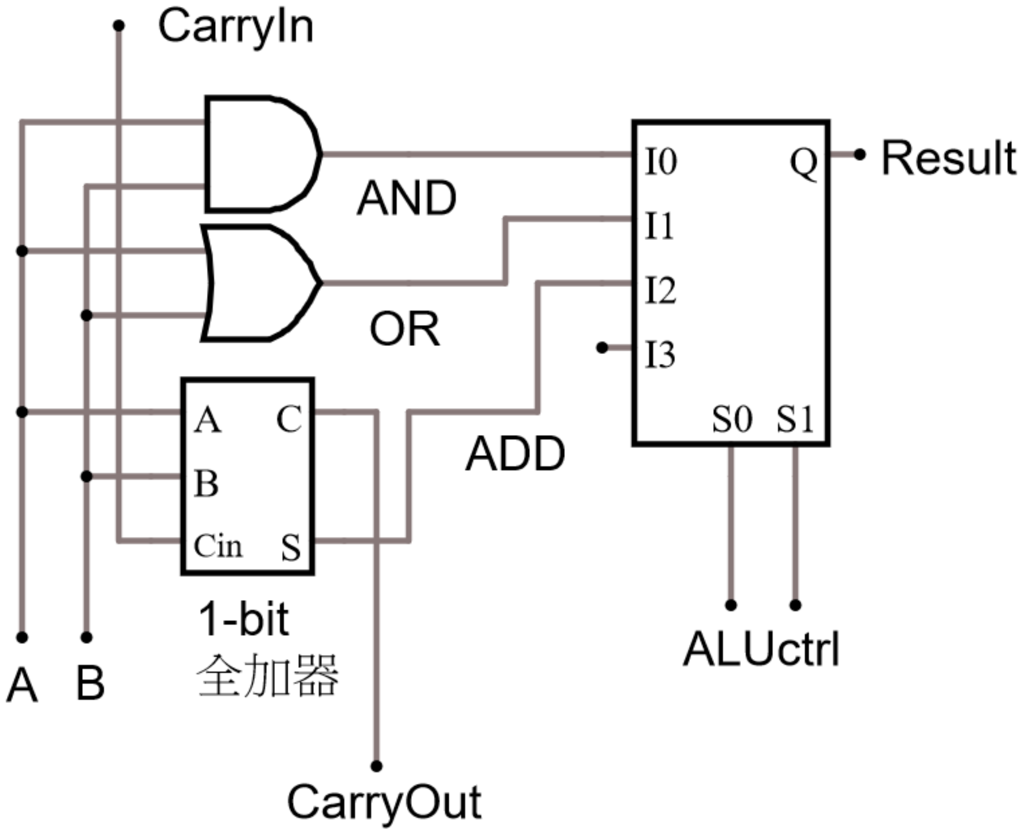
但是像是加法要處理進位的部分，所以我們的結果沒辦法單純一個一個算。

所以我們可以用32個ALU，把加法的部分拆成32個bit，接著把ALU串在一起就能得到加法的結果了。

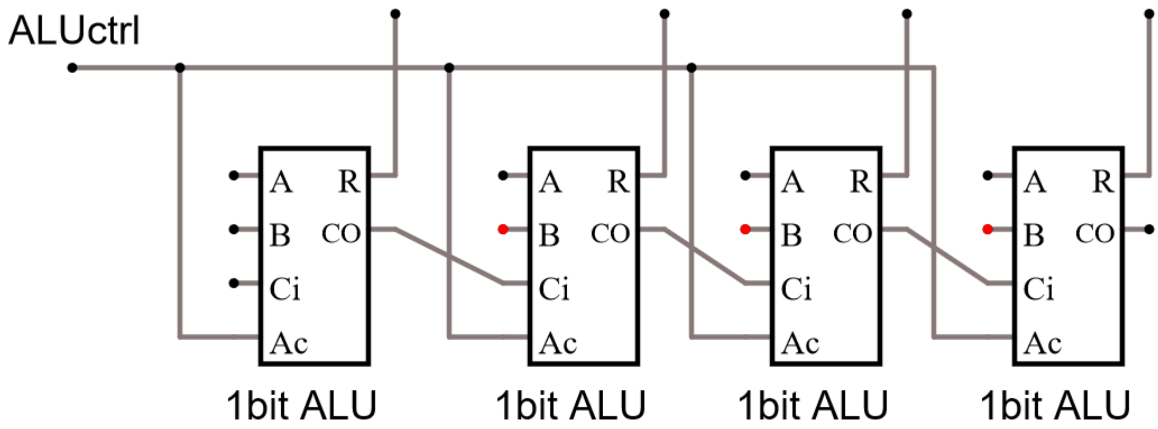
Trick 2 - 完成部分的問題後延伸

要實作slt、xor，我們可以先弄出add、or、and、sub，再來實作slt、xor的部分。

實作1-bit ALU

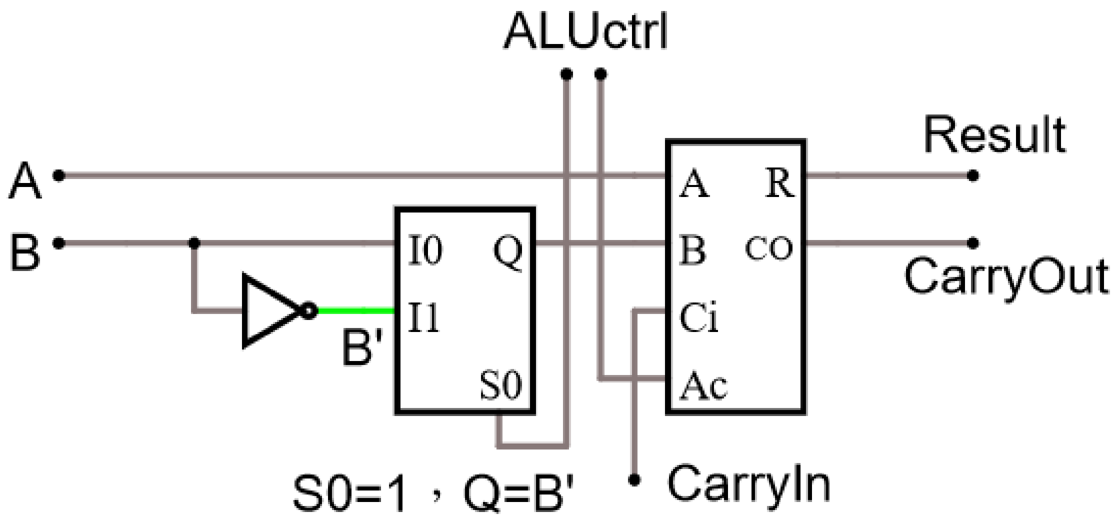


實作4-bit ALU



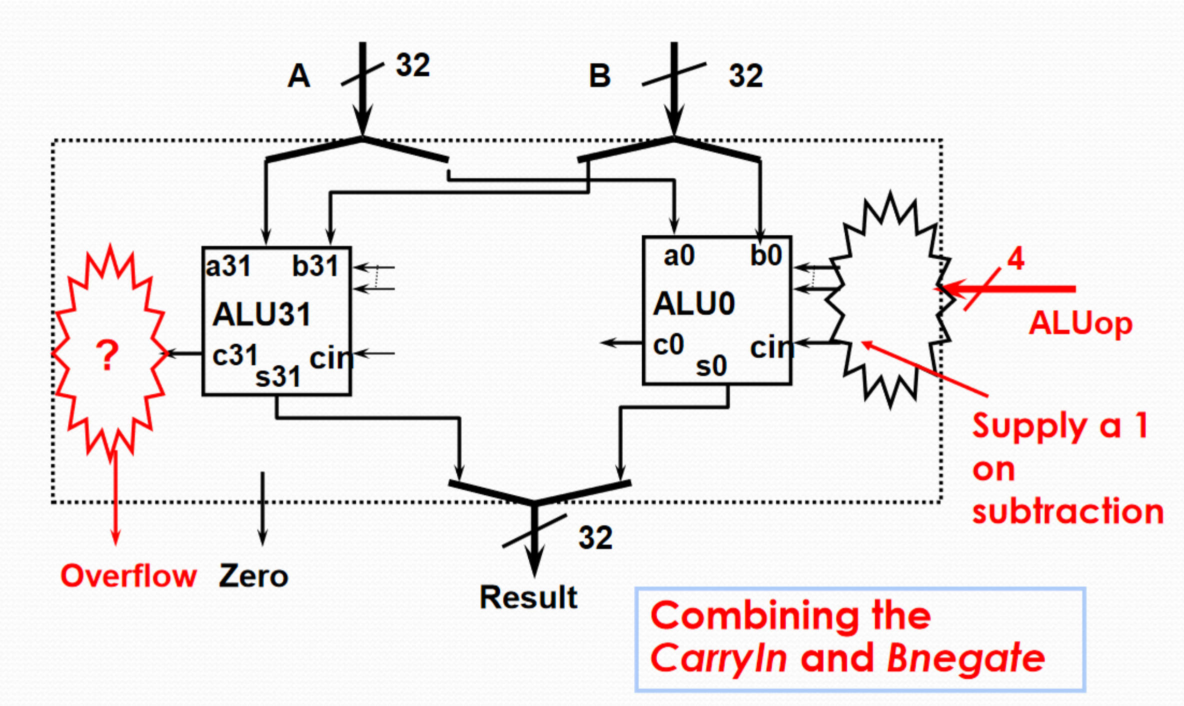
實作ALU Subtract

利用補數，A+(-B) = A-B



實作32-bit ADD串接加法

由於加法與減法皆為二補數加法、減法，所以對於MSB、LSB的部份我們要稍微做一點修改。



MSB：

判斷有沒有Overflow

LSB：

減法的部份會是1補數，那為了要轉成二補數，所以我們得要加1

這個1的部分不可能會無中生有，在剛剛我們有講到若S0=1，則信號會輸出B'

我們可以用這個S0的信號接到Cin上，這樣就能有+1的部分了。

實作溢位偵測

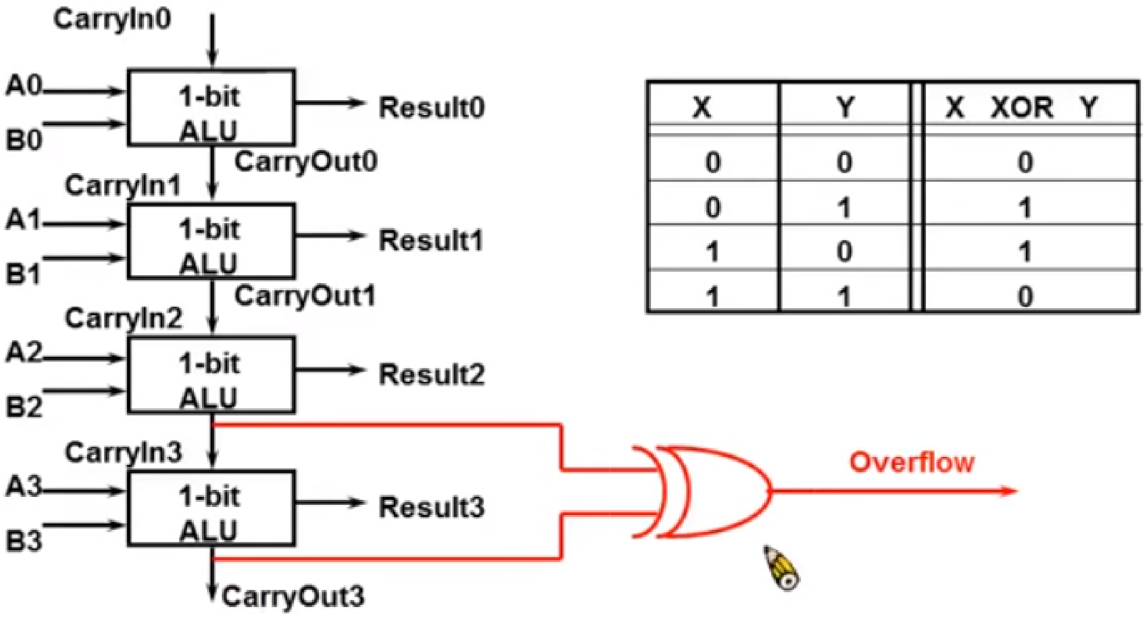
$$\text{Overflow} = \text{CarryIn[n-1]} \oplus \text{CarrayOut[n-1]}$$

如果最高位元的CarryIn是0，然後CarryOut是1

或者最高位元的CarryIn是1，然後CarryOut是0

就會發生溢位

因此在實作的部分，我們只需要將CarryIn與CarryOut取XOR即可。



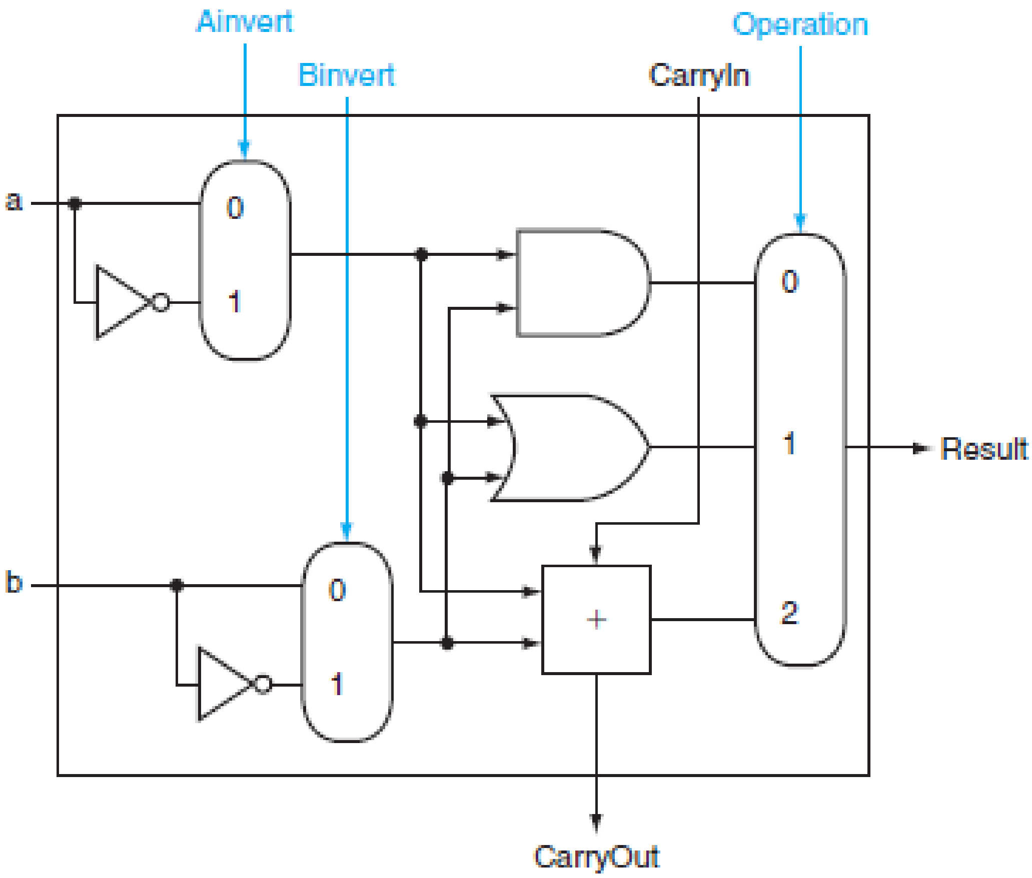
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

實作NOR運算子

$$\overline{A+B} = \overline{A} \cdot \overline{B}$$

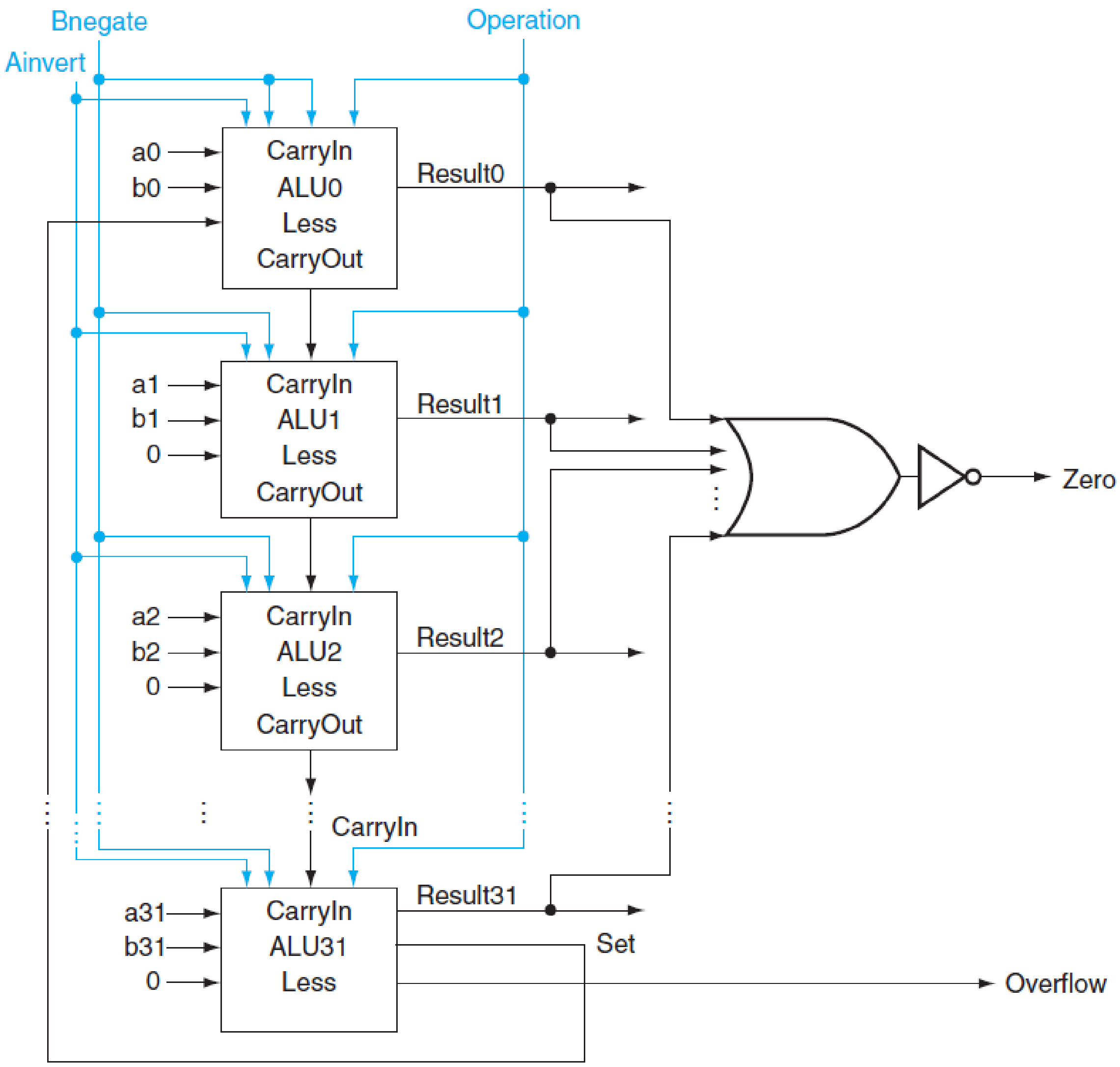
所以其實我們可以把ALUCtrl理解成，第一個bit為A取不取反相，第二個bit為B取不取反相

第三與第四個bit則為控制AND、OR、ADD的多工器Select。



實作零檢測

我們只需要把Result全部連到一個NOR上，若Result存在一個1，則必使NOR=0。



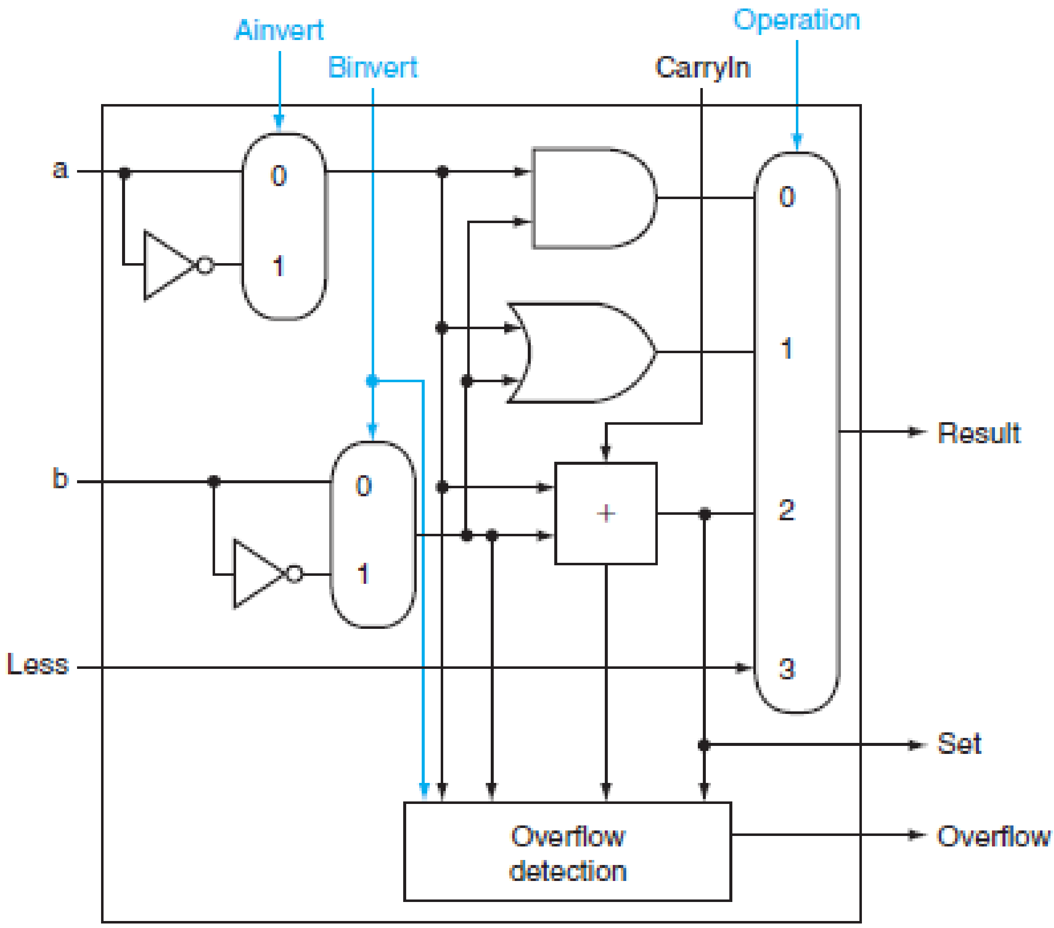
實作SLT運算子

如果$A < B$，那麼第一個暫存器的LESS應該要等於1

已知如果$A < B$，那麼經過減法之後，他的第31個ALU sign-bit應該要是1，所以已知SET=1。

所以我們把第1~第30個ALU的LESS全部設成0。

把SET往前拉到第0個ALU的LESS上，這樣只要對ALU0下ALUctrl=0111就能夠知道SLT的結果了。



濂波進位加法器ALU

可以注意到CarryIn，CarryOut的部分是串接的。

這個部分可能會導致Delay很長，從LSB到MSB會經過很多個Gate，因此最好的方法是平行處理加法。

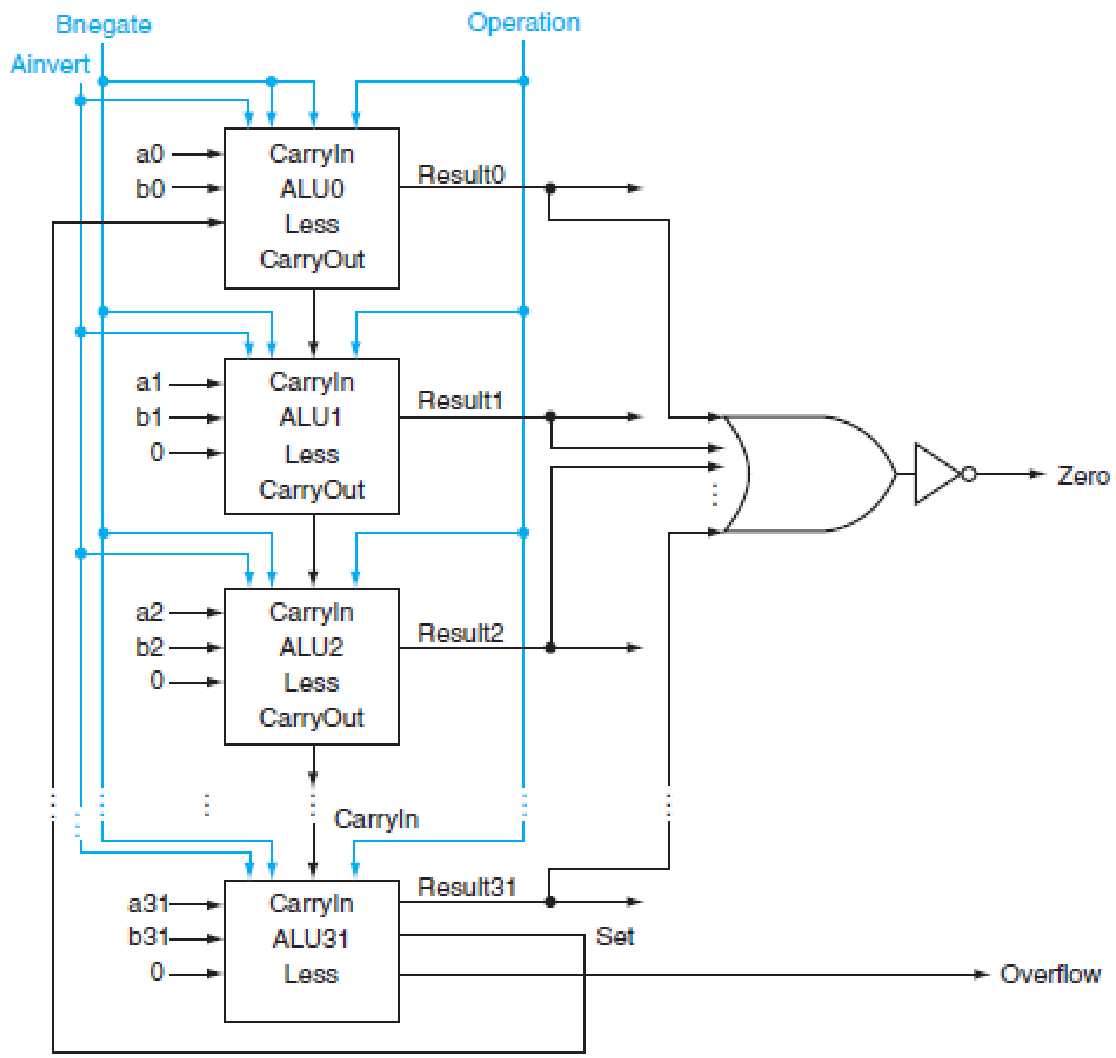


FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

Carry Look-ahead加法器

我們希望能夠讓每個Bit的Carry不用仰賴前面的ALU所產生的結果，就能夠知道每個Bit的Carry。

透過加法器的進位真值表

Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

我們可以知道，$\text{CarryOut} = (X \times \text{CarryIn}) + (Y \times \text{CarryIn}) + (X \times Y)$

因此，我們可以知道，
$\text{CarryIn}_1 = \text{CarryOut}_0 = (X_0 \times \text{CarryIn}_0) + (Y_0 \times \text{CarryIn}_0) + (X_0 \times Y_0)$

$\text{CarryIn}_2 = \text{CarryOut}_1 = (X_1 \times \text{CarryIn}_1) + (Y_1 \times \text{CarryIn}_1) + (X_1 + Y_1)$

$= (X_1 \times X_0 \times Y_0) + (X_1 \times X_0 \times \text{CarryIn}_0) + (X_1 \times Y_0 \times \text{CarryIn}_0)$

$+(Y_1 \times X_0 \times Y_0) + (Y_1 \times X_0 \times \text{CarryIn}_0) + (Y_1 \times Y_0 \times \text{CarryIn}_0) + (X_1 \times Y_1)$

因此我們要求CarryIn_2，只會需要$X_0, X_1, Y_0, Y_1, \text{CarryIn}_0$，而不用透過前一個ALU的運算結果

即使前一個ALU再複雜，我們只需要這些值就能算出Carry了，就能有效的解決串接所造成的Delay問題。

Carry Look-ahead的名詞

接著我們定義兩個新的名詞：

Generate Carry at Bit i: $G_i = A_i \times B_i$

Propagate Carry at Bit i: $P_i = A_i + B_i$

因此，上面的式子我們可以寫成

$\text{CarryIn}_1 = \text{CarryIn}_0 \times (P_0) + (G_0)$

$\text{CarryIn}_2 = \text{CarryIn}_1 \times (P_1) + (G_1)$

$= [\text{CarryIn}_0 \times (P_0) + (G_0)] \times (P_1) + G_1$

$= P_1 P_0 \text{CarryIn}_0 + P_1 G_0 + G_1$

$\text{CarryIn}_3 = \text{CarryIn}_2 \times P_2 + G_2$

$= [P_1 P_0 \text{CarryIn}_0 + P_1 G_0 + G_1] \times P_2 + G_2$

$= P_0 P_1 P_2 \text{CarryIn}_0 + P_1 P_2 G_0 + G_1 P_2 + G_2$

透過上面的觀察，我們可以知道，如果$\mathrm{CarryIn}_i$要進位，則

G_{i-1}必須要為true (i.e. $A_{i-1} \times B_{i-1} = 1$)

或者G_{i-2}必須要為true，且P_{i-2}必須要讓G_{i-2}來通過(i.e. P_{i-2}為true)

或者G_{i-3}必須要為true，且$P_{i-2} \times P_{i-3}$必須要為true

或者G_{i-j}必須要為true，且$P_{i-2} \times P_{i-3} \times \ldots \times P_{i-j}$必須要為true

或者我們使用Cin_0，條件是$P_{i-2} \times P_{i-3} \times \ldots \times P_0$必須要為true

Carry Look-ahead的延遲

已知

Generate Carry at Bit i: $G_i = A_i \times B_i$，delay = 1

Propagate Carry at Bit i: $P_i = A_i + B_i$，delay = 1

所以

$\mathrm{CarryIn}_1 = \mathrm{CarryIn}_0 \times (P_0) + (G_0)$，總共是3個delay

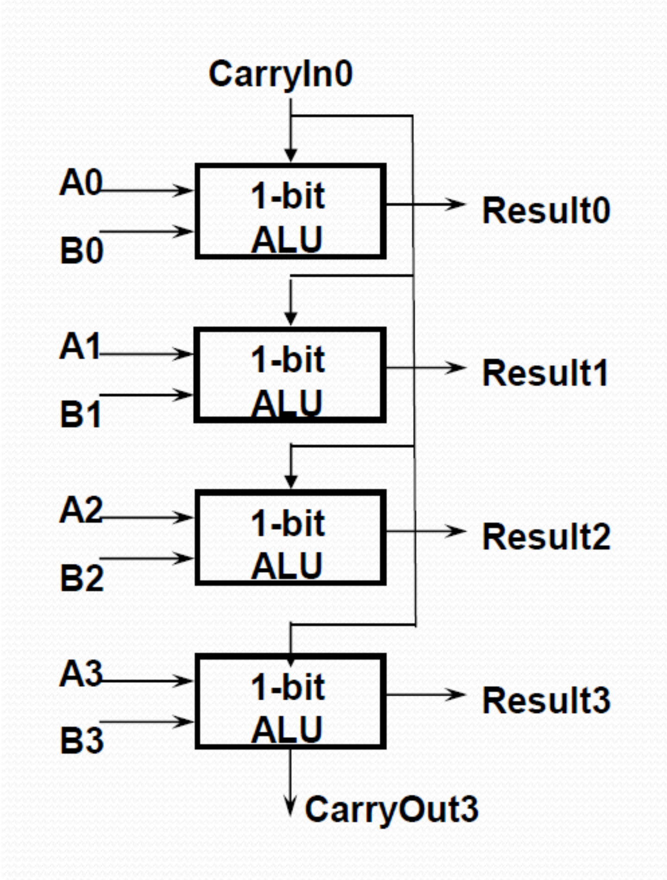
$\mathrm{CarryIn}_2 = P_1 P_0 \mathrm{CarryIn}_0 + P_1 G_0 + G_1$，總共是3個delay

$\mathrm{CarryIn}_3 = P_0 P_1 P_2 \mathrm{CarryIn}_0 + P_1 P_2 G_0 + G_1 P_2 + G_2$，總共是3個delay。

所以$\mathrm{CarryIn}$的delay幾乎為常數。

Carry Look-ahead的電路

實務上，我們不把Propagate建出來。



建立完整的Carry-Lockhead很貴，而且當CarryIn_{31}的時候，電路會非常非常長，會拉低效能。

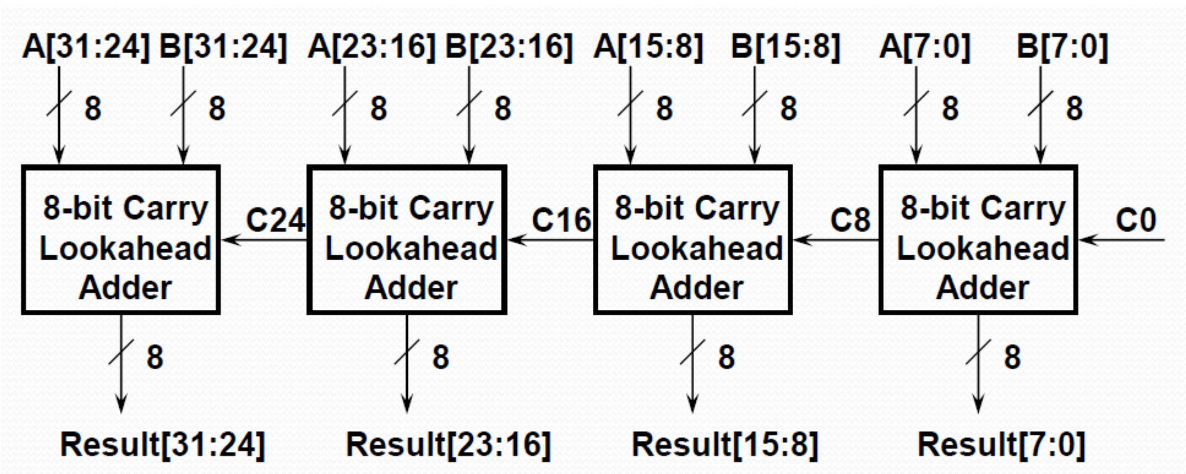
所以我們會用兩種方法解決這個問題：

「cascaded carry look-ahead adder」與「multiple level carry look-ahead adder」

Cascaded Carry Look-ahead Adder

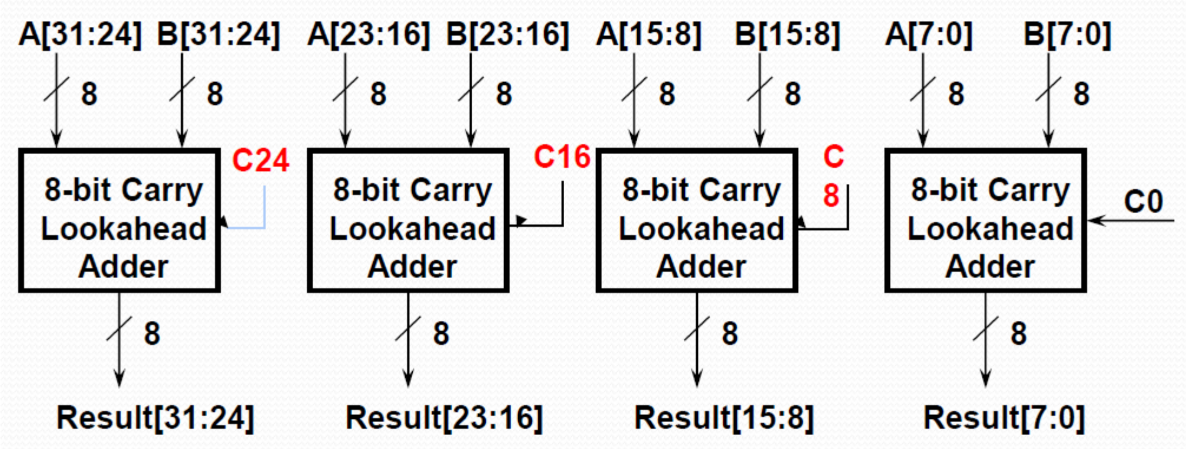
把這一堆的Carry Lockahead adder串接起來，本身來說8-bit Carry Lockahead Adder進行Carry Lock-ahead。

然後C8, C16, C24的方式把他做串接。



Multiple Level Carry Look-ahead Adder

稍微優化一下串接的delay。



C8, C16, C24的部分，我們可以製造出一個Super Propagate跟Super Generate。

這樣就能把他化簡成一般的Lock-ahead了。

舉個例子，已知
$\text{CarryInput}_4 = G_3 + (P_3G_2) + (P_3P_2G_1) + (P_3P_2P_1G_0) + (P_3P_2P_1P_0\text{CarryInput}_0)$

我們令$SG_0 = G_3 + (P_3G_2) + (P_3P_2G_1) + (P_3P_2P_1G_0) + (P_3P_2P_1P_0\text{CarryInput}_0)$

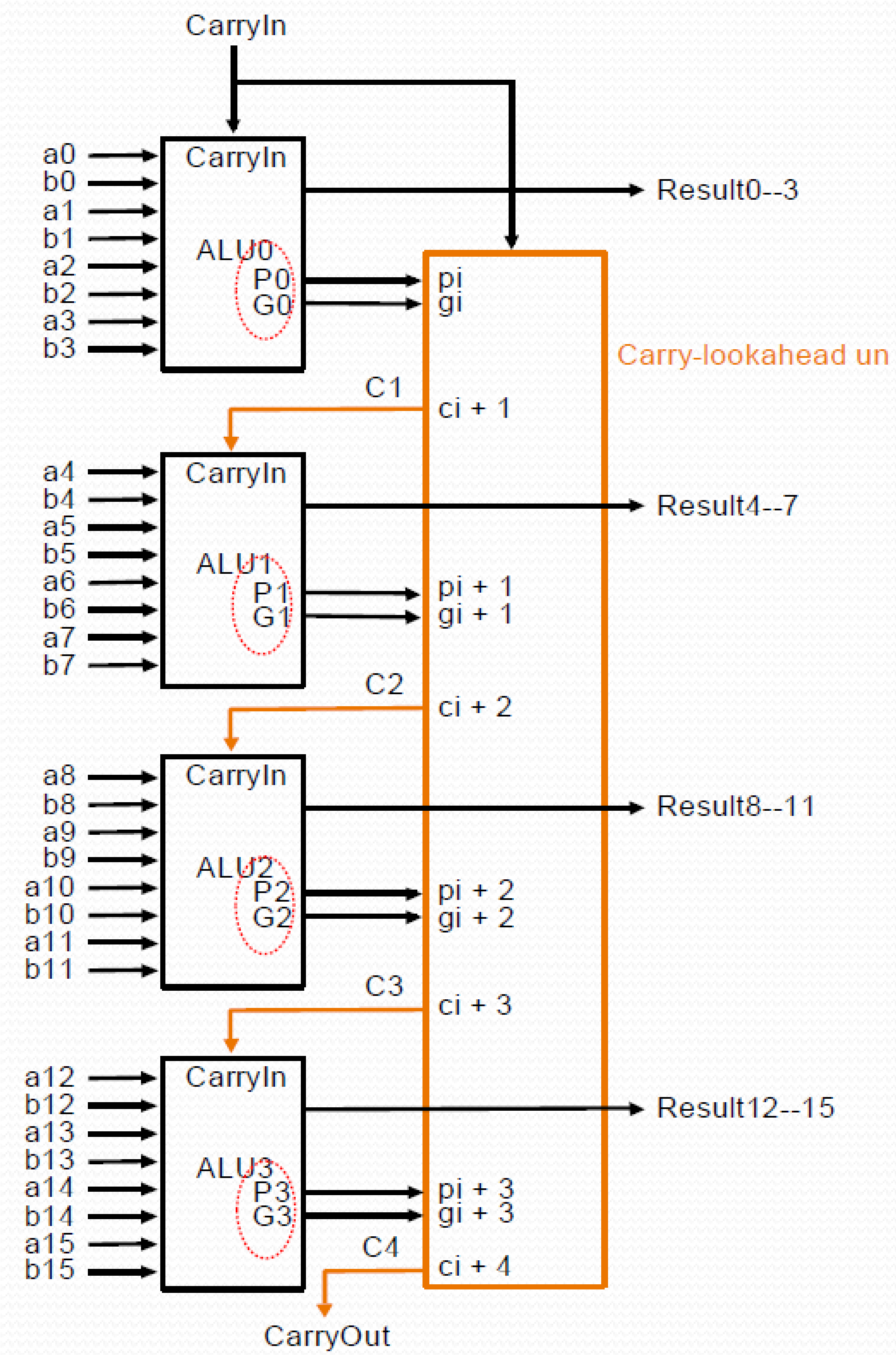
$SP_0 = P_3P_2P_1P_0$

所以$\text{CarryInput}_4 = SG_0 + SP_0 \times \text{CarryInput_0}$，Delay變成4個delay。

根據上面，我們可以知道第i個$\text{8-bit Carry Lock-ahead}$，
$\text{CarryInput}_{i+1} = SG_i + SP_i \times \text{CarryInput}_i$

就能把區塊化簡成一個高階的Carry Lock-ahead，因此每個區塊的delay都是一樣的。

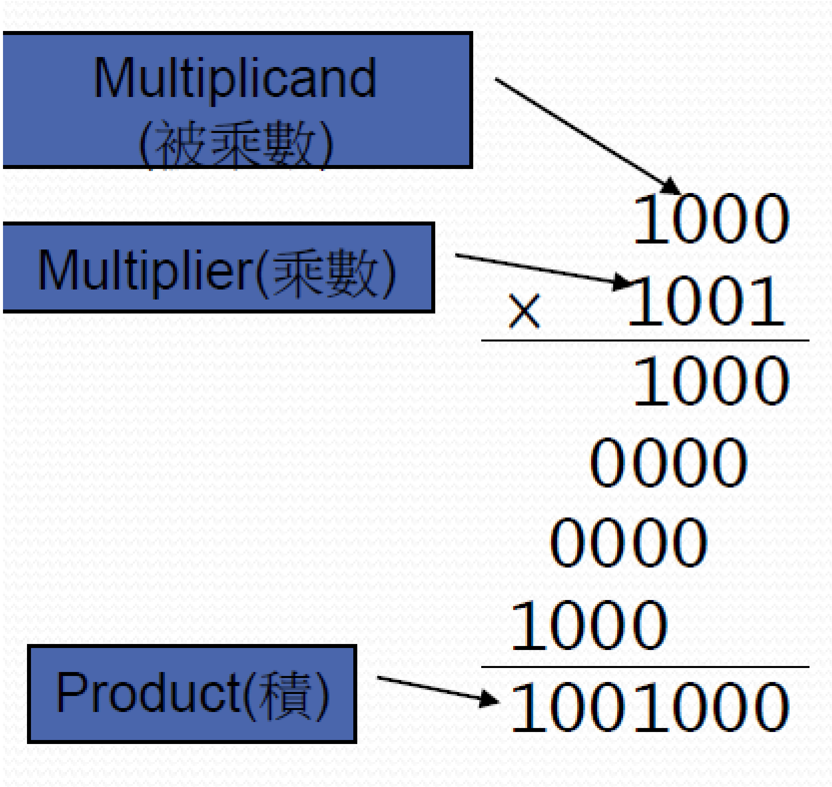
Multiple Level Carry Lookahead Adder 的實作



乘法概念

我們利用長乘法(直式乘法)來進行乘法。

例如$1000_{(2)} \times 1001_{(2)} = 1001000_{(2)}$



乘法觀察

透過上面的觀察，可以知道當n個bit乘以m個bit，結果最多會有$n+m$個bit。

再仔細觀察，我們可以知道：

令被乘數為M且乘數為m，則若$m_i=1$時，就Copy一次M加至結果，若$m_i=0$時，就加0，加完之後左移一格。

乘法在硬體上的概念

透過上面的乘法結果，我們可以知道：

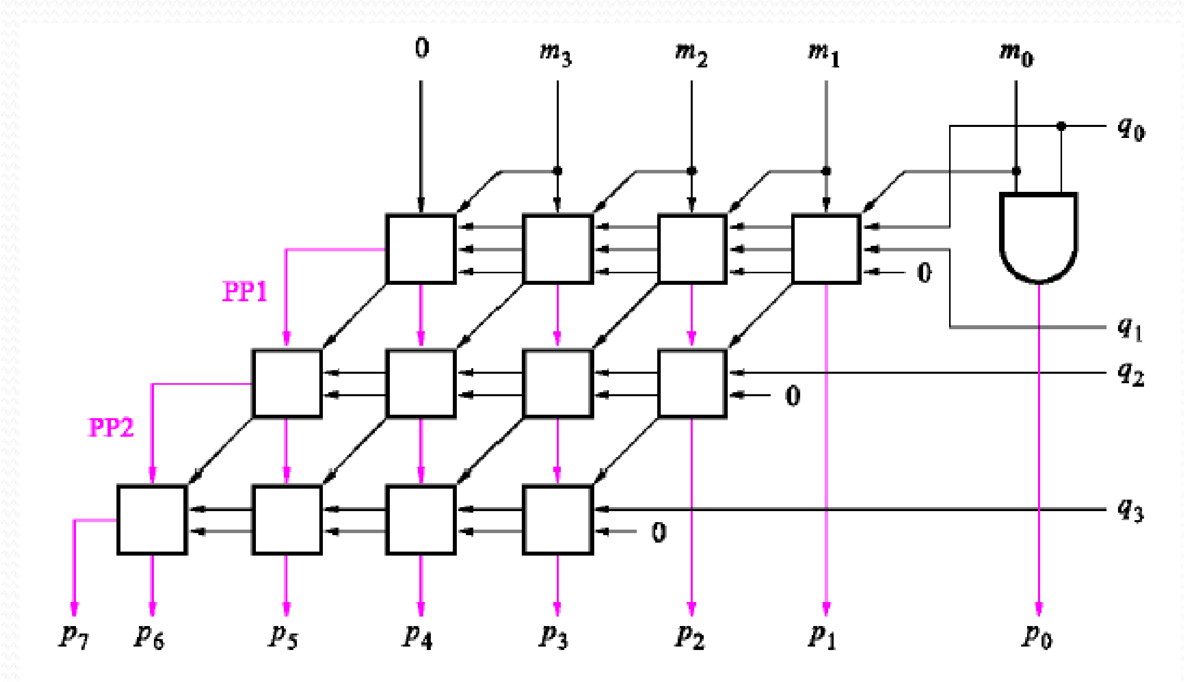
令被乘數為M且乘數為Q

第一個bit單純只有$M_1 \times Q_1$

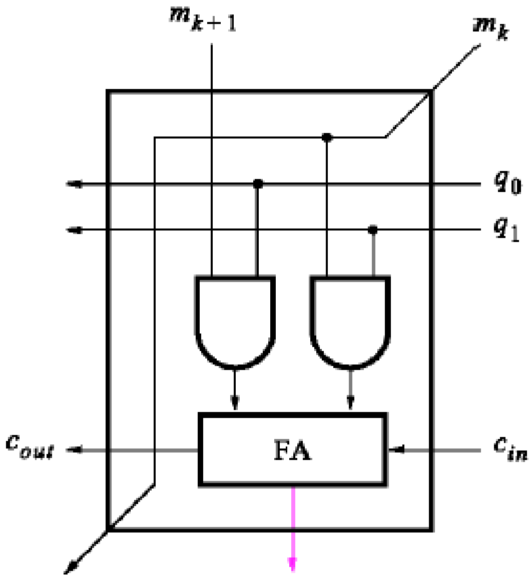
第二個bit為$Q_0 \times M_1 + Q_1 \times M_0$

第三個bit為$Q_1 \times M_2 + Q_2 \times M_1 + Q_3 \times M_0$

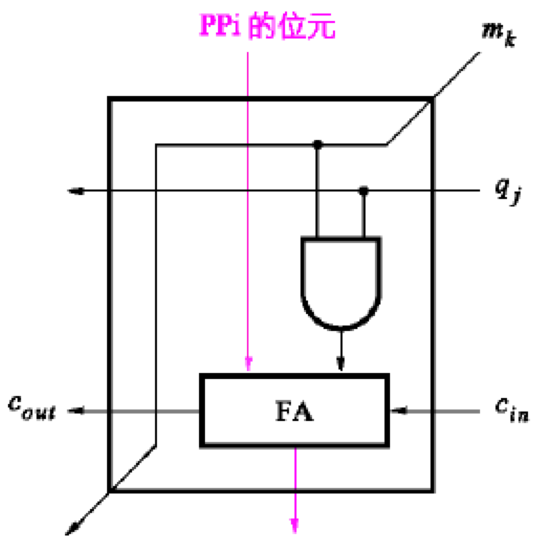
依照這個概念，我們可以畫出這一張圖



其中

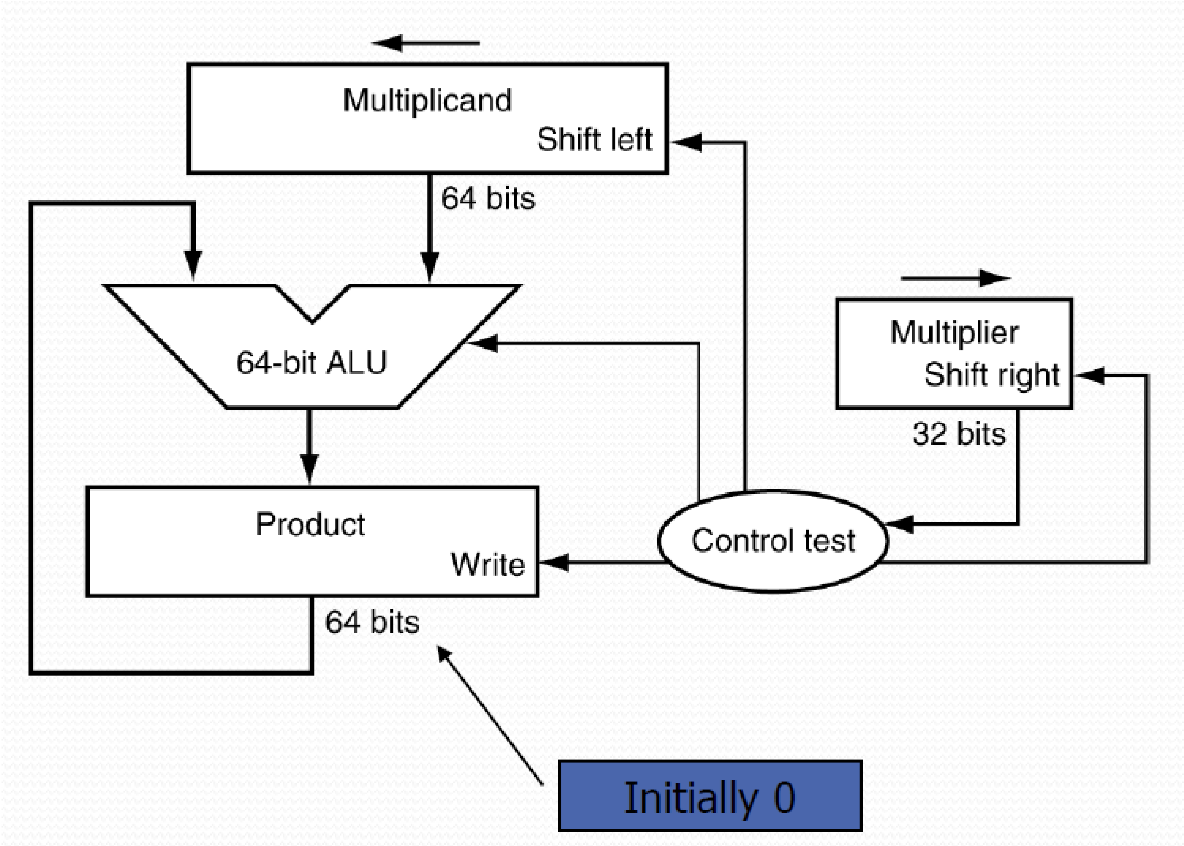


(b) 最上列的區塊

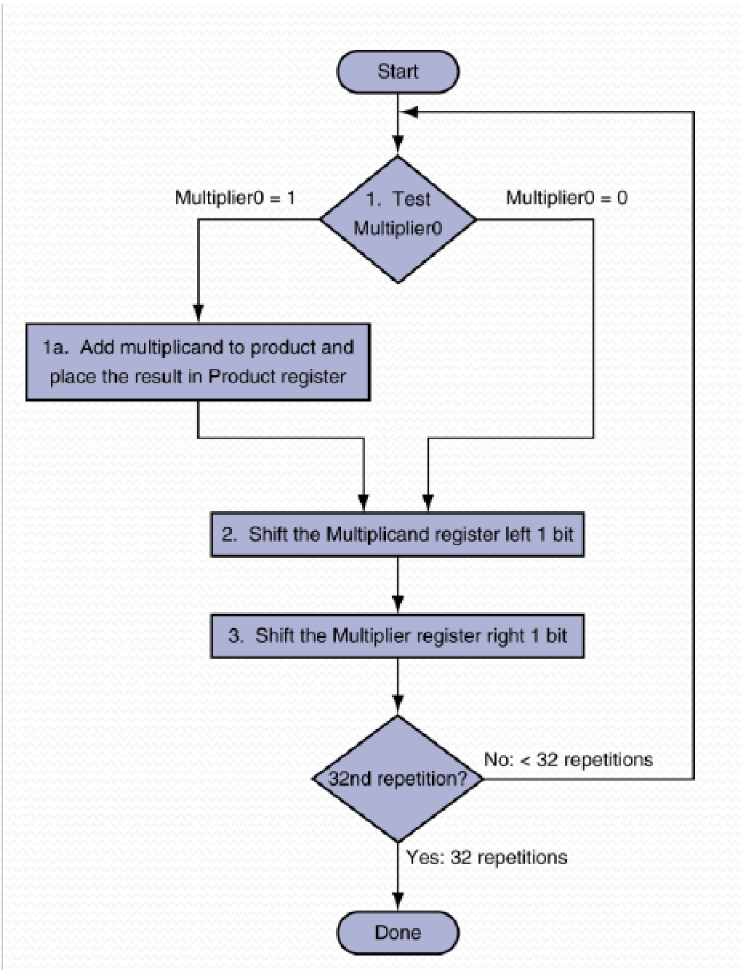


(c) 下兩列的區塊

因此，我們可以回來理解這一張圖



運作原理可以歸納成這一張圖



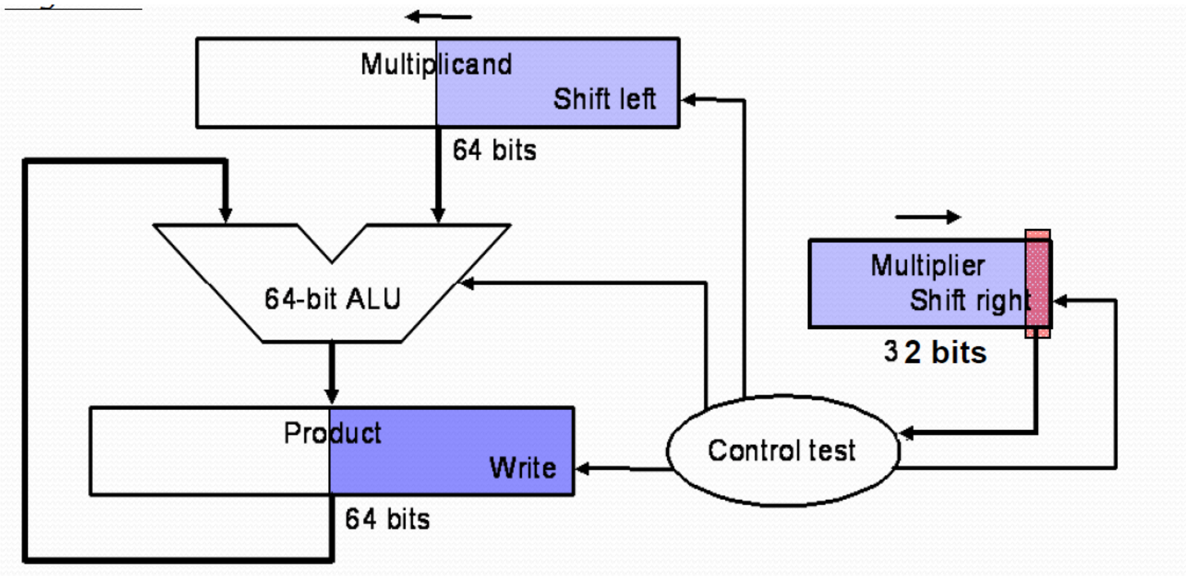
如果被乘數的LSB=1，就把乘數與暫存在Product的前一個結果做相加。

接著把被乘數右移，乘數左移，確定是不是第32位，如果是就跳出，否就回去第一個步驟。

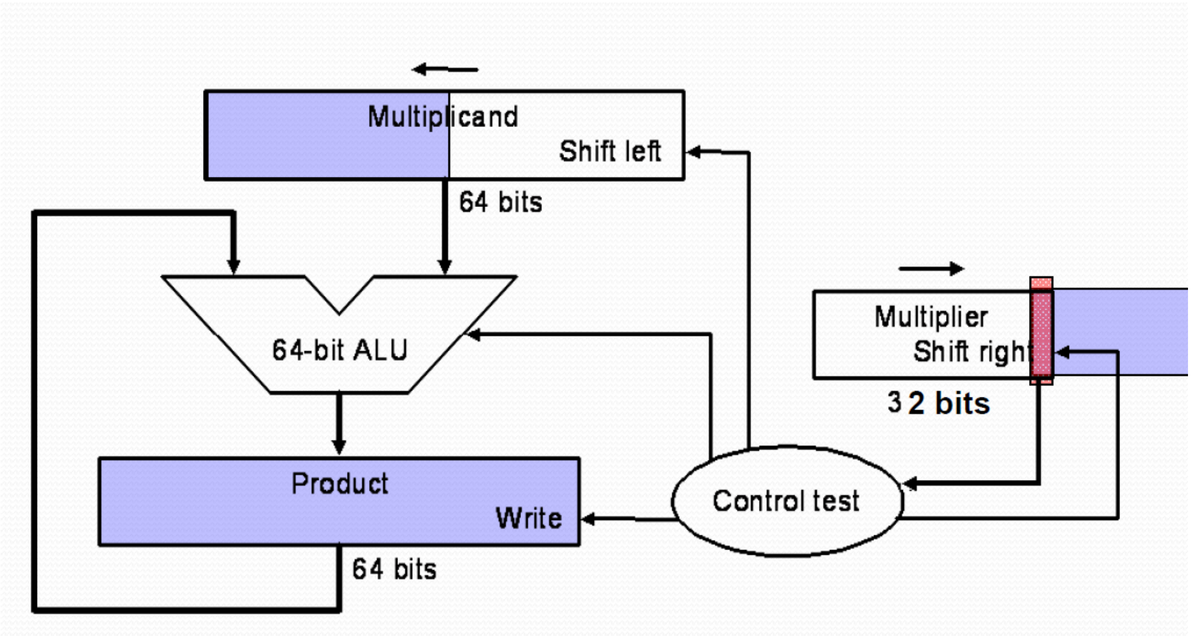
乘法在硬體上的實作：版本1

製作一個32bit的乘法器，我們需要一個64bit的乘數暫存器，32bit的被乘數暫存器，64bit的ALU，以及64bit的乘法結果暫存器。

原因是乘數每做一次步驟就要往左推一次，所以64bit才能存取32bit的乘數乘法過程所產出的0。



做完乘法後，會長得像這樣



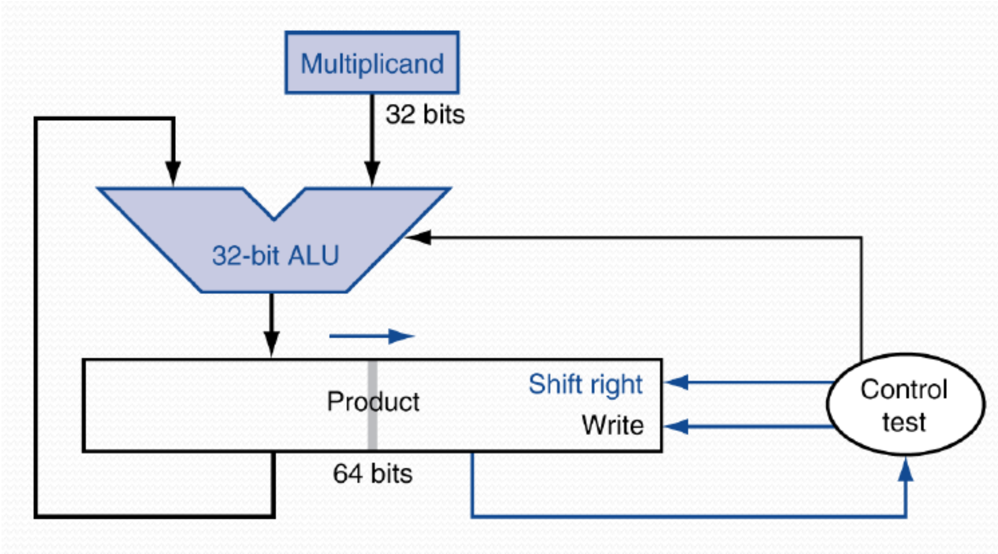
觀察一下這個做法，可以知道

1. Multiplicand很佔空間，一半以上都是0
2. 每做一次乘法大概需要花上快100個$Clock$，左移與右移加上加法以及一些細碎的東西，$32 \times 3 + start \approx 100$
3. 要把Product結果存起來很浪費空間

也許我們可以

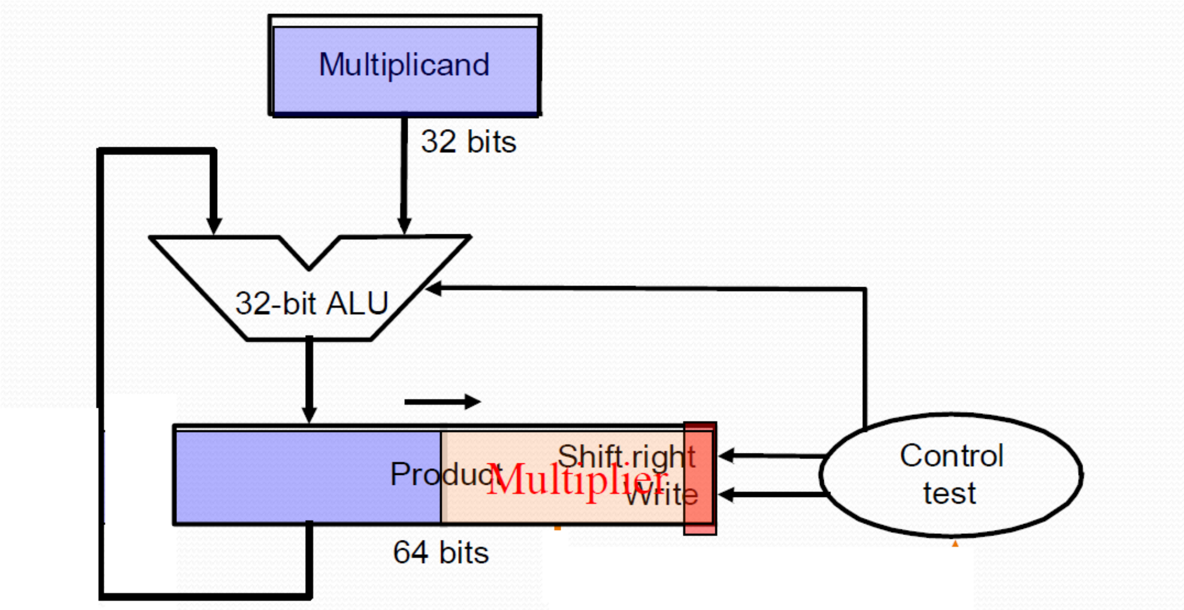
1. 取代乘數左移，被乘數右移的部分
2. 把乘數與Product弄成一塊暫存器

乘法在硬體上的實作：版本2



我們用一個32bit的乘數暫存器，32bit的ALU，以及一個64bit的Product暫存器，這個Product暫存器可以寫入以及右移

這個Product包含了右半部Mulitiplier與左半部Product暫存器，用Hi/Lo控制，加起來有64bit，省去了Multiplier原先的32bit。



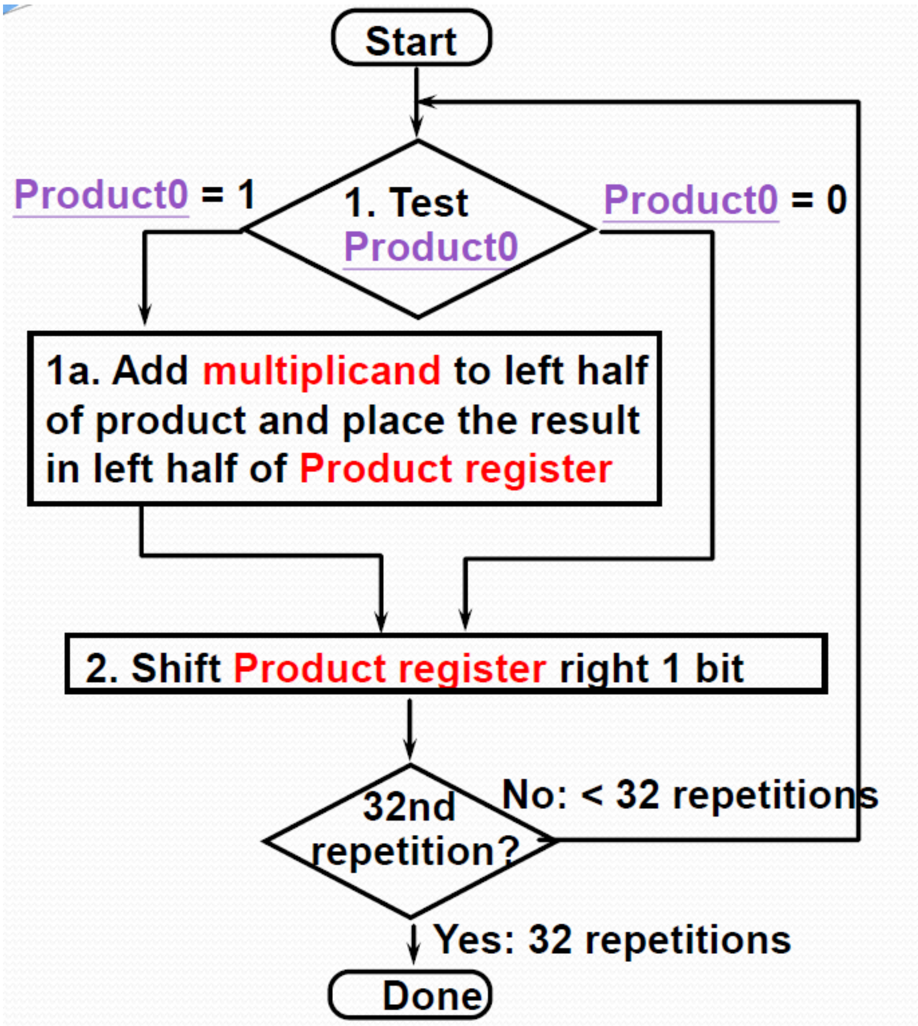
運算的方式如下

```
已知結果的左半邊0用不到(因為都是0)，被乘數只需要用到第一位的數字
所以被乘數如果第1個bit已經被用過了，那就沒用了，即使右移也沒關係

那麼如果我們把結果左移的話，被乘數就會把第1個bit給pop，結果就會往左一格
例如原先已經加完的結果暫存器為0110 1011，乘數為0110，我們可以知道結果的部分為5~8格的bit，乘數的部分為1~4格的bit
那麼我們將整個結果暫存器右移一格，得到0011 0101，乘數為0110，因為被乘數的第1位是1，所以5~8格加上0110
得到1001 0101，此時的結果是第4~8格，也就是10010

原因是，在前面的時候我們將整個暫存器右移一格，所以結果所佔的格數變成了左邊的4~8格
已知結果左邊bit的0不影響，所以右移這個動作本身不影響結果，只是結果的格數多一格最左邊的bit
那麼如果我們在原來的位置加上乘數，就變相等於在乘數的右邊多加上了一個0，就達成了Version 1的左移部分
```

因此如果我們執行到最後，則結果所佔的格數就是1~8格，此時被乘數的bit均被pop掉

根據上面，我們可以得出一個結論
如果被乘數的第一位是0，那什麼都不要動，否則在左半邊的bit加上乘數
接著右移一格，然後回去再運行一次步驟，直到所有的被乘數bit均被pop

因此，演算法可以表示成以下這張圖



舉個例子，$0010_2 \times 0011_2$

Example: 0010 x 0011

Multiplicand	Product	
0010	0000 0011	(S, 1)
	0010 0011	(1a)
0010	0001 0001	(2, 1)
	0011 0001	(1a)
0010	0001 1000	(2, 1)
0010	0000 1100	(2, 1)
0010	0000 0110	(2, D)

觀察這個做法可以知道，數與乘法的暫存器合併在一起了，所以能夠節省很多空間

有號乘法的做法

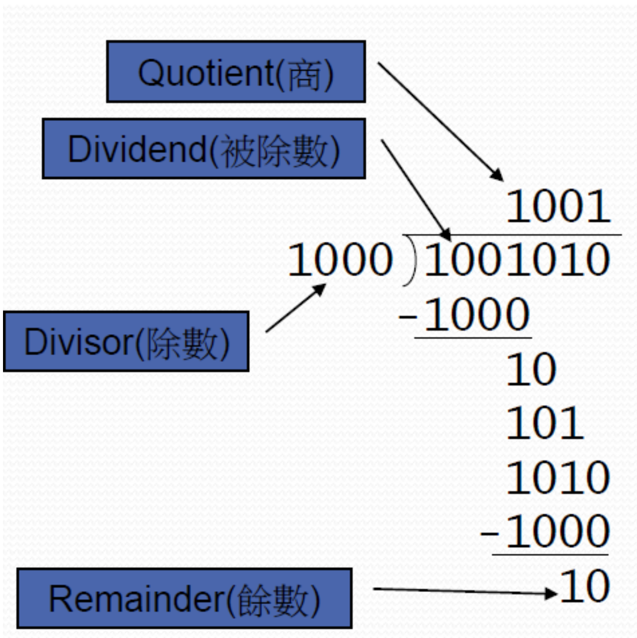
```
考慮一下有號乘法要怎麼做：

我們先把所有的數字由負轉成正，然後接下來考慮數字的signed bit乘起來結果會是如何
如果乘起來是正的，那把所有數字由負轉乘正，再相乘後本身的結果就是正確答案，如同(-7)*(-7)=7*7=49
但是如果乘起來是負的，那我們就要把數字再加上負的部分就是正確解答，某種程度上來說就是取二補數

所以，我們可以得出一個小小的結論：如果signed bit是0，那最後一位其實就不用做處理了，是1再做處理

那麼，我們來考慮怎麼計算二補數，先從簡單的1001轉換成2補數開始，先反向得到0110，然後再+1得到0111
因此這樣我們可以從1001的signed bit知道這個數字是負的，然後從0111得到這個數字是7，因此是-7
那麼計算模式就是：-(2^3)+0*2^2+0*2^1+2^0 = -7

因此，我們可以直接把最後的signed bit用減的部分減掉(i.e. 取二補數然後用加的)
如果signed bit本身就是0，那就不用去理會它了
```

更快的乘法做法

1. 我們可以建立Wallace tree，用很多個ALU串接，然後Carry傳到下一級來做進位，這樣64個bit會需要64個ALU
2. 銀彈法則，利用multiple adders，要花費超級多ALU，好處是多個乘法運行可以指令管線化。

手算除法

用筆操作一下除法



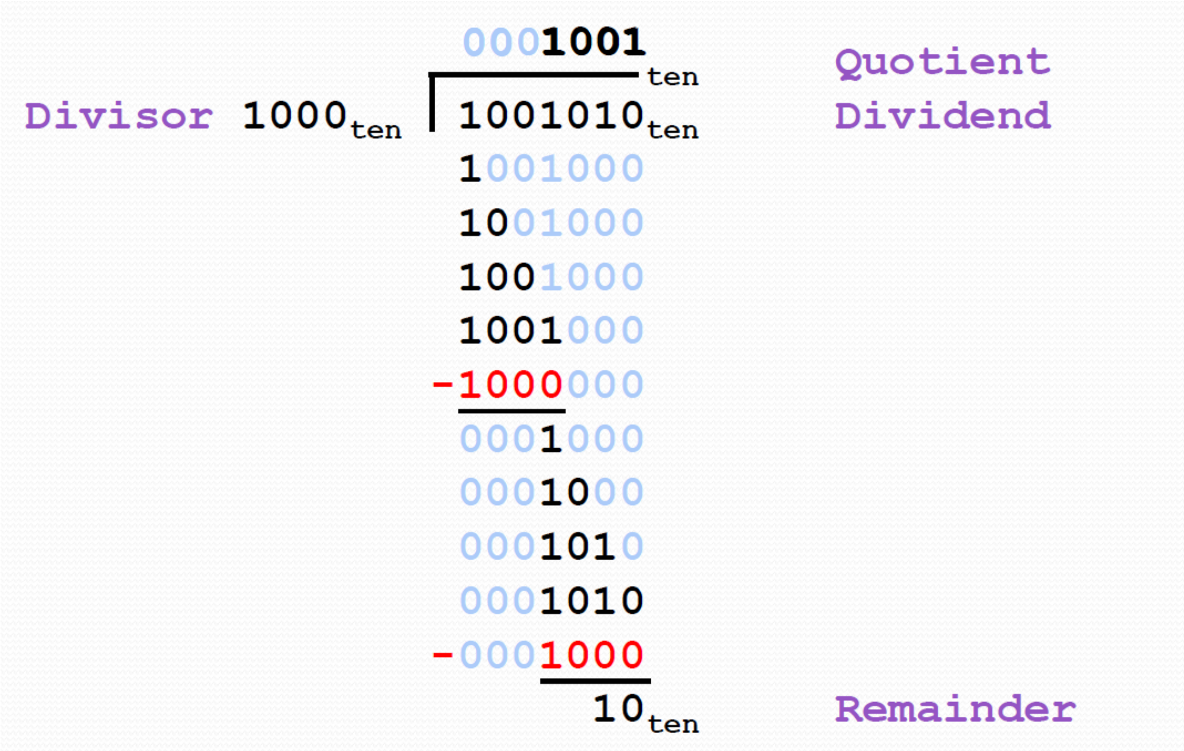
可以知道，除法是被除數連減除數的過程

我們可以知道，如果被除數小於除數，則我們在商的最右邊加上一個0，接著把下一個被除數的bit帶下來

如果被除數大於等於除數，則我們在商的最右邊加上一個1，並且把被除數減去餘數。

如果餘數是負的，那我們就把餘數再加上除數，使其變成正

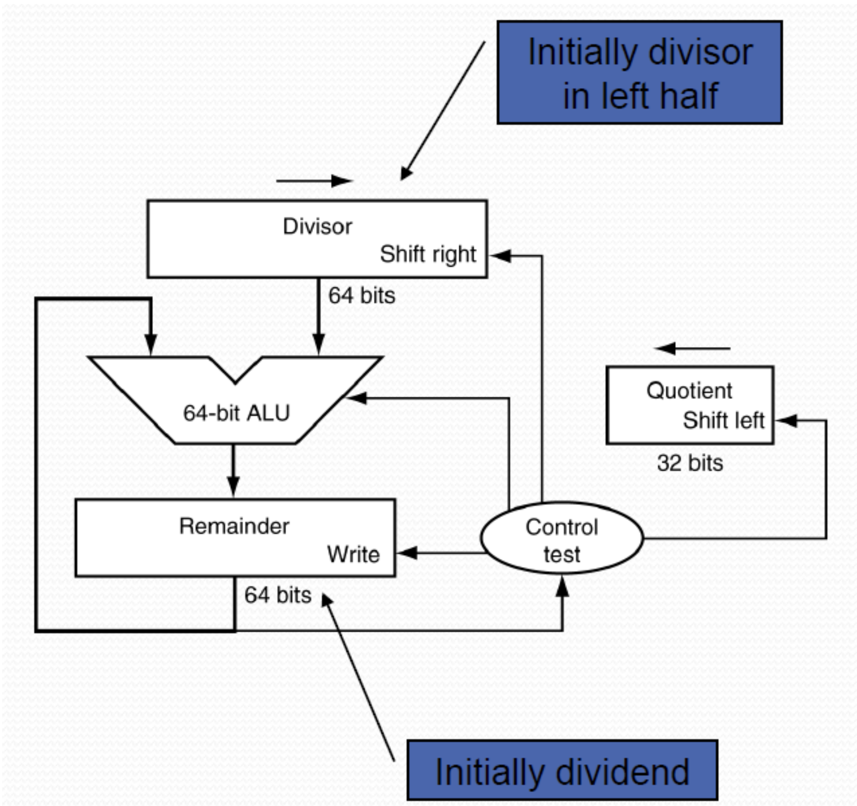
如同以下的結果。

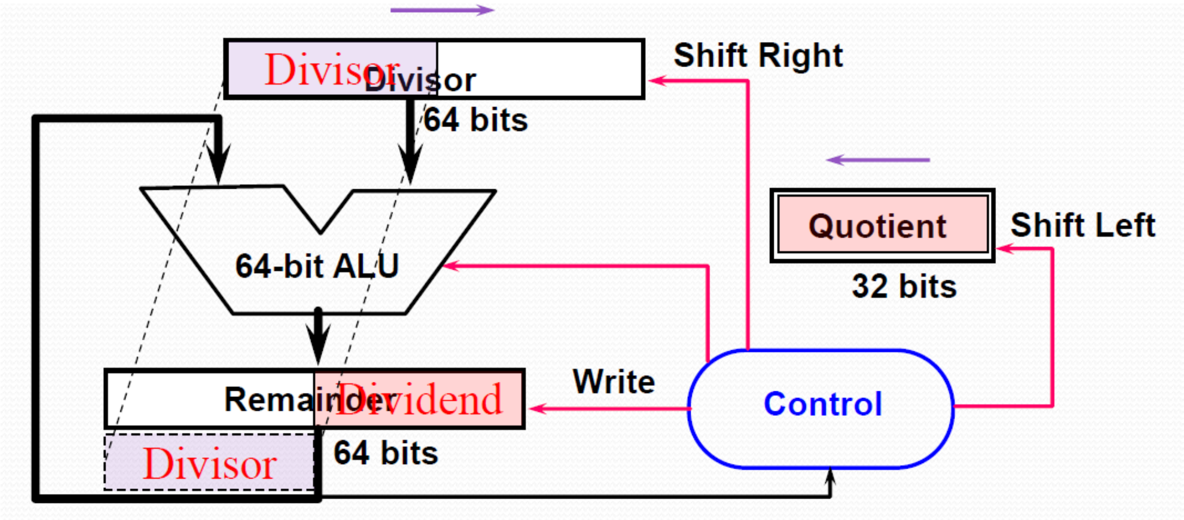


在硬體上的除法：版本1

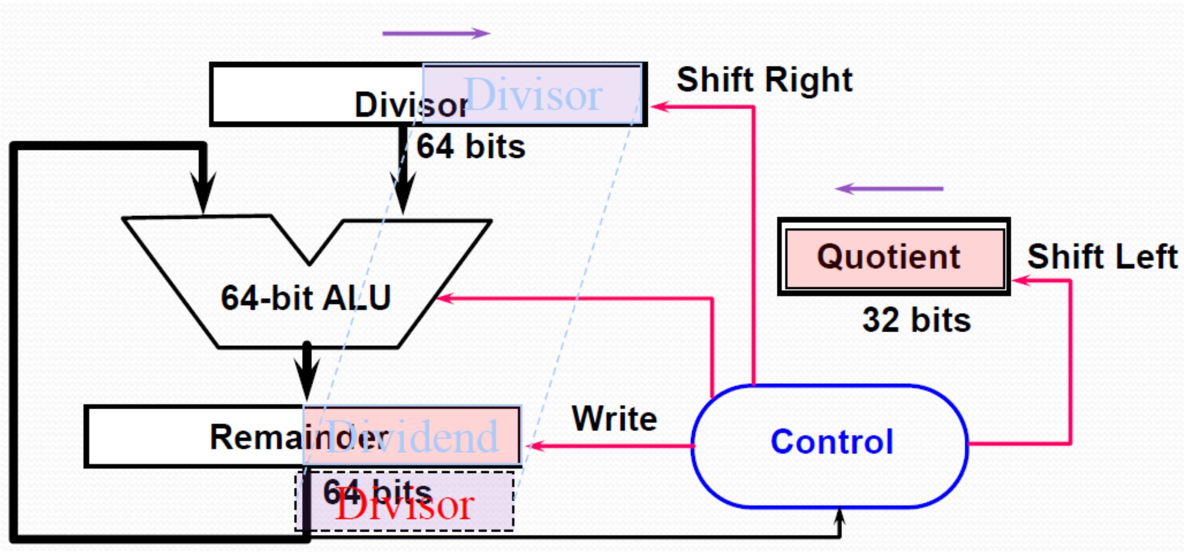
我們使用一個32bits的Quotient暫存器來儲存結果，用兩個64bits的暫存器，用來分別儲存Divisor與Remainder

以及一個64-bit的ALU，而divisor的部分，我們預設先分給左半邊的32bits，預設dividend的部分，我們放在Remainder暫存器上



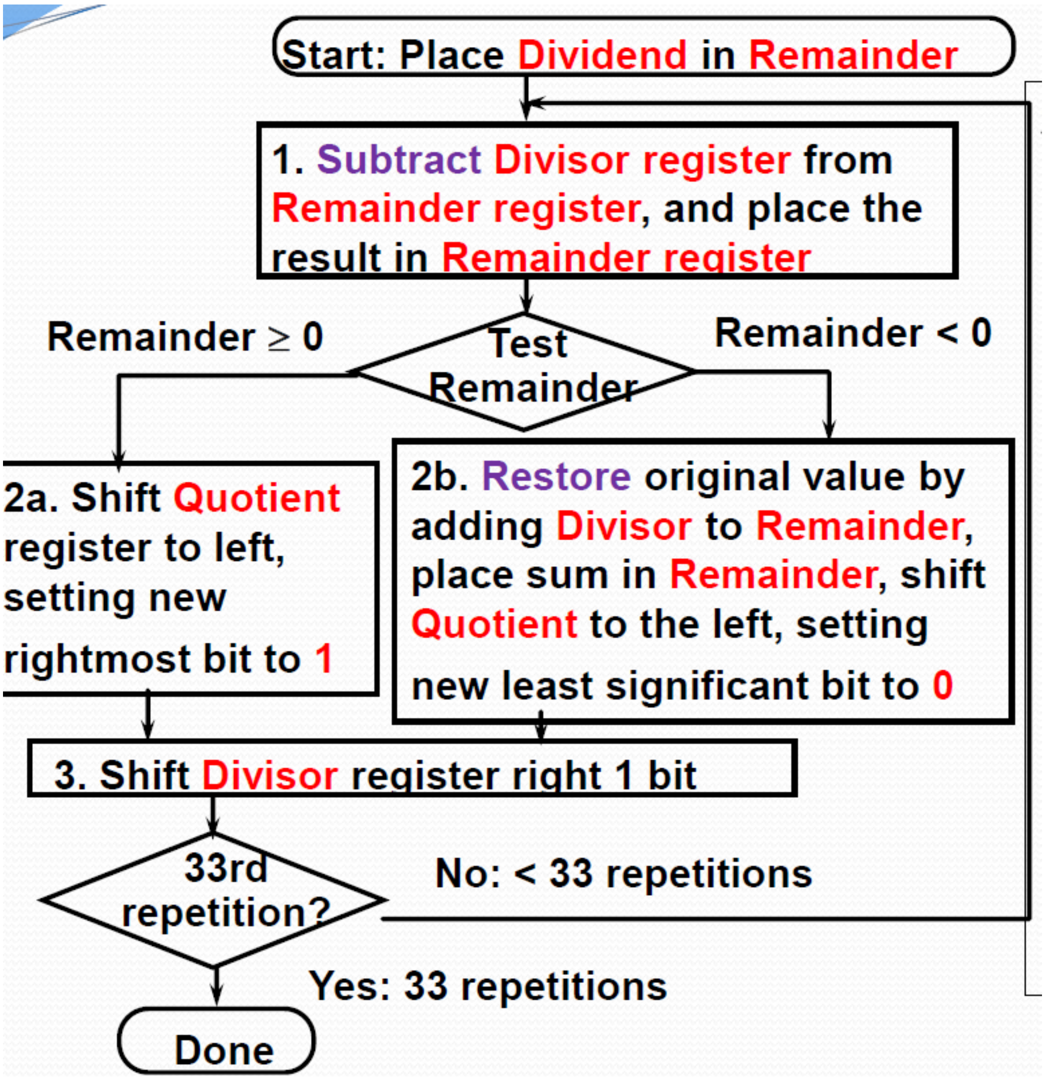


當所有的運算結束之後，Divisor的暫存器左半部會全部移至右半部。



運算想法如下：

```
第一個步驟，先把Dividend與Remainder的部分先放置好

接著每一次進行除法時，我們將Remainder register減去Divisor register

如果Remainder register大於0，則代表Remainder register > Divisor Register
這時候我們把Quotient register左移一格，並將最右邊的bit設置為1

如果Remainder register小於0，則代表Remainder register < Divisor Register
因為餘數不能是負的，所以我們把原本的餘數加回去原先的樣子，也就是加上Divisor Register
接著我們把Quotient register左移一格，並將最右邊的bit設置為0

接著我們把Divisor的部分右移一格，如果bit已經運算到第33位，則結束運算，否則繼續進行除法
```



因此，透過這個演算法，試著去運算$0111_{(2)}/0010_{(2)}$，得到以下的結果：

Example: 0111 / 0010

Quot.	Divisor	Remainder
0000	00100000	00000111 (S)
		11100111 (1)
0000		00000111 (2b)
0000	00010000	00000111 (3)
		11110111 (1)
0000		00000111 (2b)
0000	00001000	00000111 (3)
		11111111 (1)
0000		00000111 (2b)
0000	00000100	00000111 (3)
		00000011 (1)
0001		00000011 (2a)
0001	00000010	00000011 (3)
		00000001 (1)
00011		00000001 (2a)
0011	00000001	00000001 (3,D

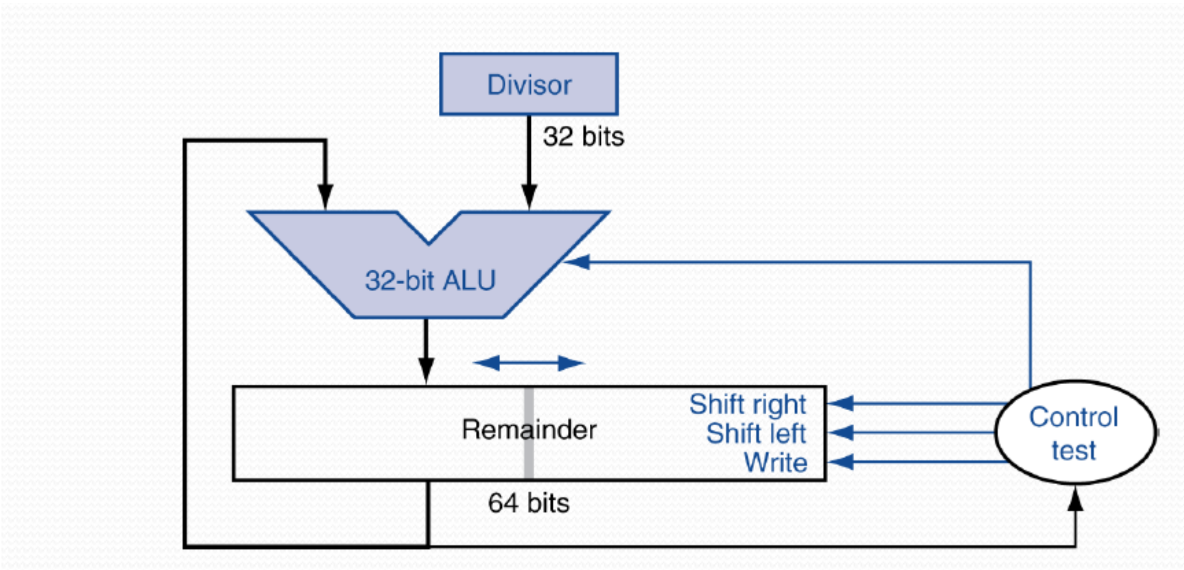
觀察版本1的模式，我們可以知道

1. Divisor在運算結束後，一半以上的空間都為0
2. 64-bit adder很浪費，用不到這麼多

所以其實我們可以考慮

1. 如同除法一樣，把位移所需要的空間進行壓縮
2. 似乎可以把結果跟Remainder的暫存器併在一起
3. 1st step cannot produce a 1 in quotient bit (otherwise quotient is too big for the register)

改善版本1的問題

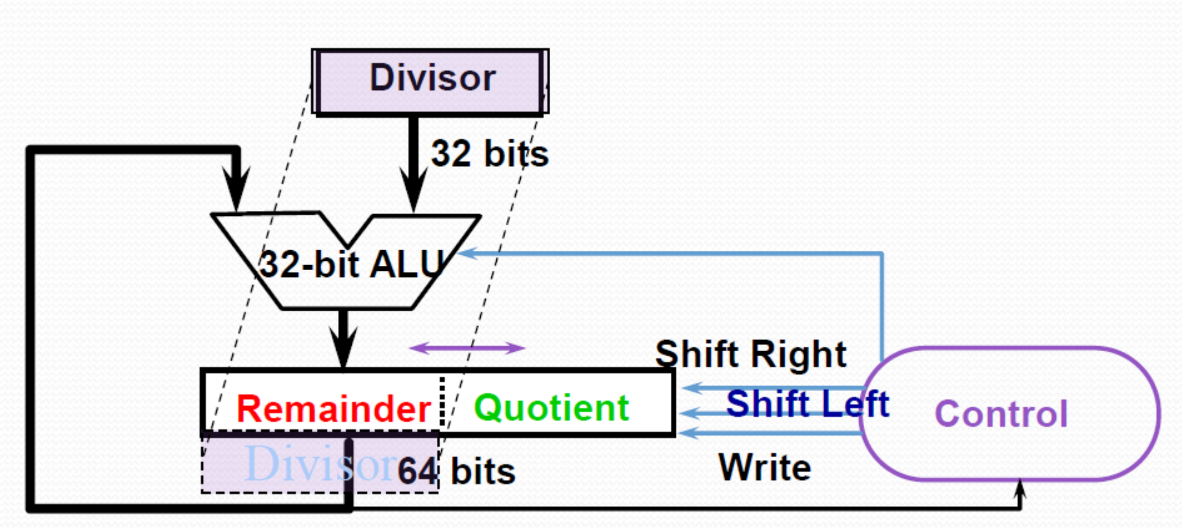


看起來很像乘法器，所以乘法器也許可以當成除法器用。

每一個Cycle做一次除法中的減法。

在硬體上的除法：版本2

電路開始時，我們將Dividend放在Remainder暫存器的右半邊

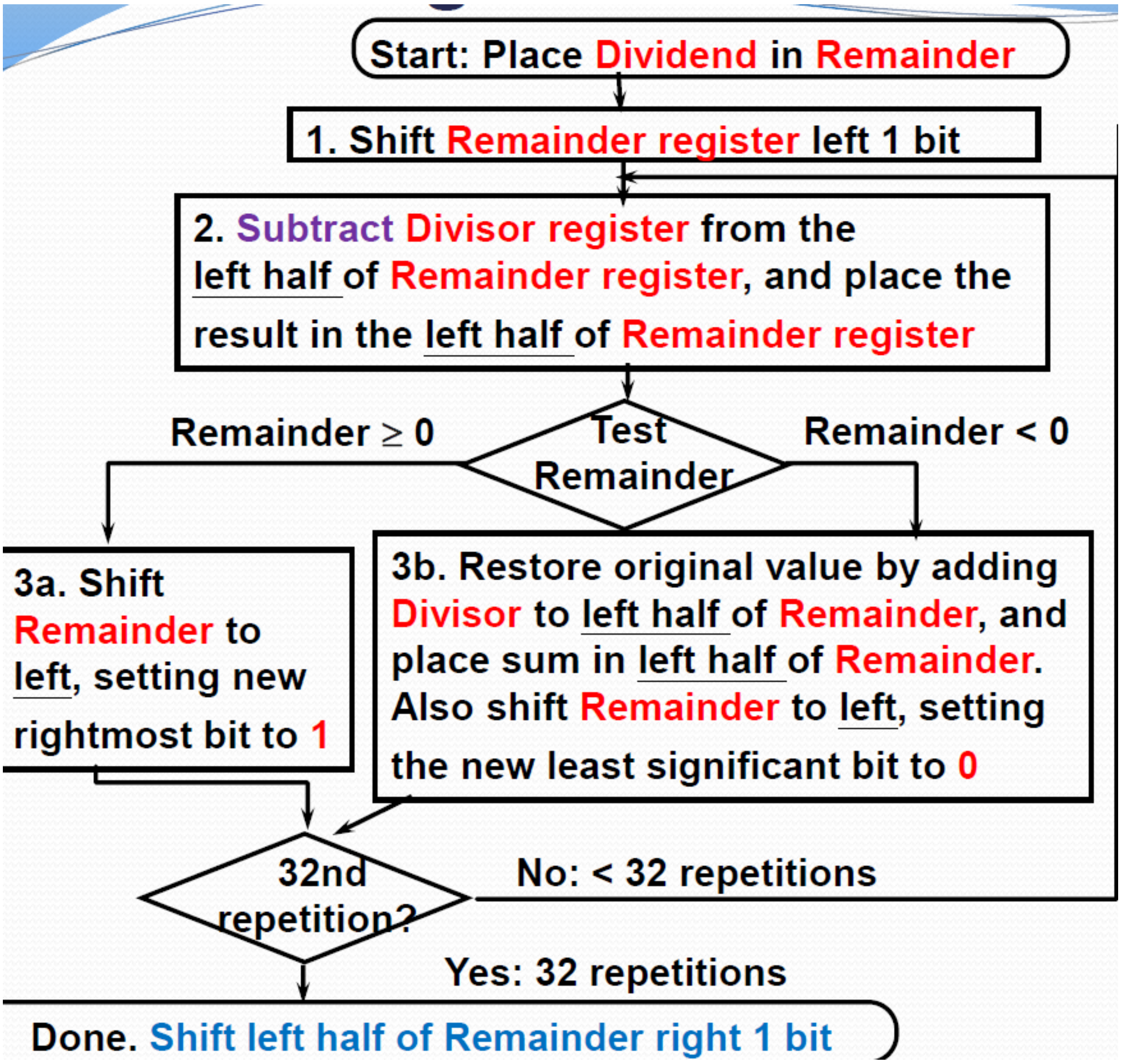


當除法運行結束之後，商會在Remainder暫存器的右半邊，餘數會在Remainder的左半邊

接著我們把Divisor的暫存器壓縮到只需要32bit的暫存器，以及原先的Quotient暫存器拋棄不用。

運算想法如下：

```
在一開始時，我們將Dividend放在Remainder暫存器的右半部。

接著進行除法，先把Remainder register左移一格，用Divisor Register減去Remainder register的左半邊暫存器
接著把結果放在Remainder register的左半邊暫存器

接著判斷Remainder register的大小，如果大於等於0，那就把Remainder register左移一格，並且設置最右邊的bit為1
如果小於0，那就把Divisor Register加回去左半邊的remainder暫存器，然後把結果存在左半邊的remainder暫存器
接著一樣把Remainder register左移一格，並且設置最右邊的bit為0。

如果bit已經處理了32個，那就結束除法，並且將Remainder整個右移一格，如果還沒就繼續除法
```



使用這個方式進行$0111_{(2)}/0010_{(2)}$，得到以下結果

Example: 0111 / 0010

Step	Remainder		Divs.
0 S	0000	0111	0010
1.1	0000	1110	
1.2	1110	1110	
1.3b	0001	1100	
2.2	1111	1100	
2.3b	0011	1000	
3.2	0001	1000	
3.3a	0011	0001	
4.2	0001	0001	
4.3a	0010	0011	
4.D	0001	0011	

有號數的除法

在手做有號數的除法時，我們會把被除數與除數換成正的，然後做除法，再把號補上去就好。

硬體上的有號數除法，我們的最終目標是讓被除數與餘數的正負一致。

如果除數與被除數的相除結果正負與商的正負不一致，那就把商做一次反相(就是乘以-1)。

這樣結果就會符合這個等式

$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$

至於Quotient會不會過大呢

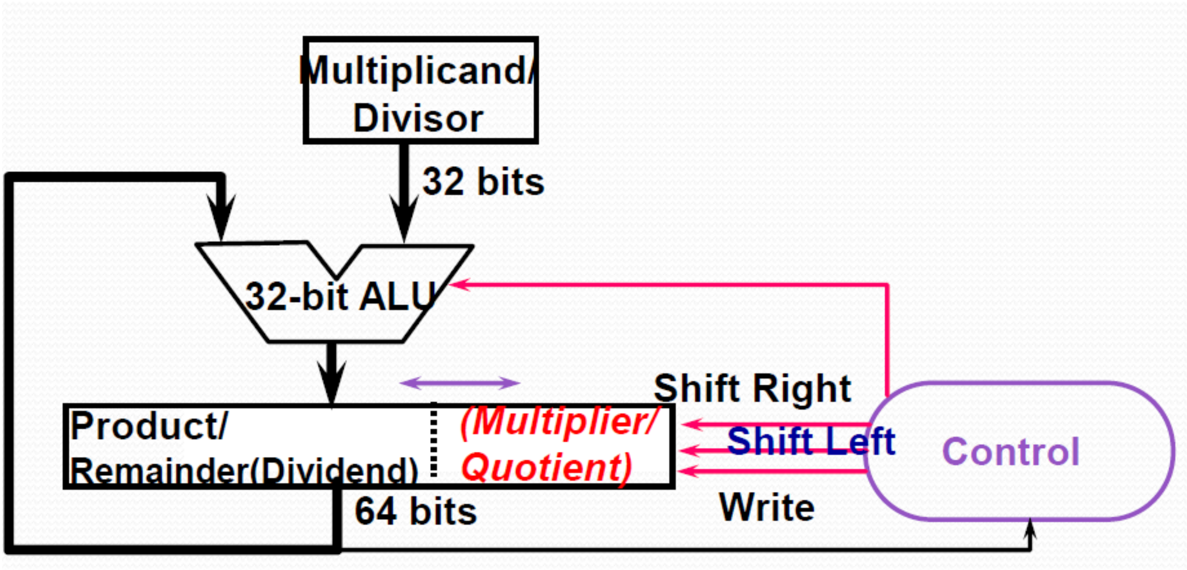
假設$\text{Divisor} = 1$，那$\text{Remainder} = 0$，因此若Dividend是一個64bits的數字，那Quotient就會是個64bits的數字。

觀察：乘法與除法

乘法與除法共用同一個硬體設備，只需要可以控制Remainder/Product暫存器的左移右移，即可把乘法器當成除法器。

硬體上的乘除法器

一個32-bit的Multiplicand/Divisor暫存器，一個32-bit的ALU，一個64-bit的Product/Remainder暫存器。



快速除法

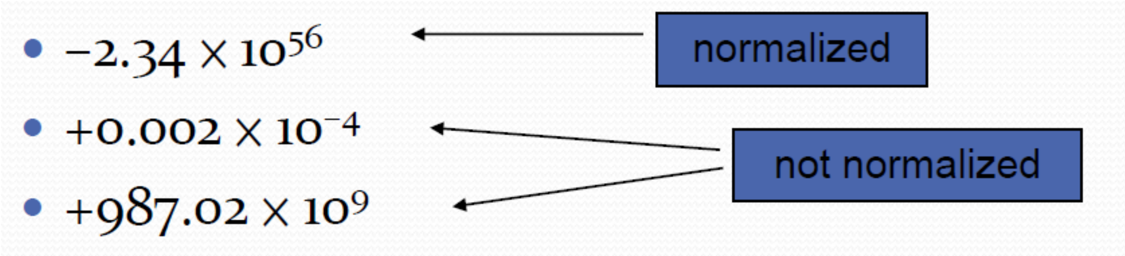
除法並不能夠指令管線化，因為remainder的正負關係。

一種快速除法：SRT divison，在每個除法步驟時，製造出其他的除數bit來達成快速除法，但依然會需要乘法的步驟。

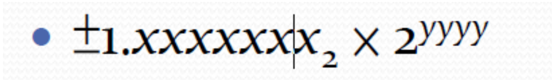
二進位浮點數

用來儲存不完整的除法的數字，例如$1/3 = 0.33333...$

用來儲存科學記號，像這些



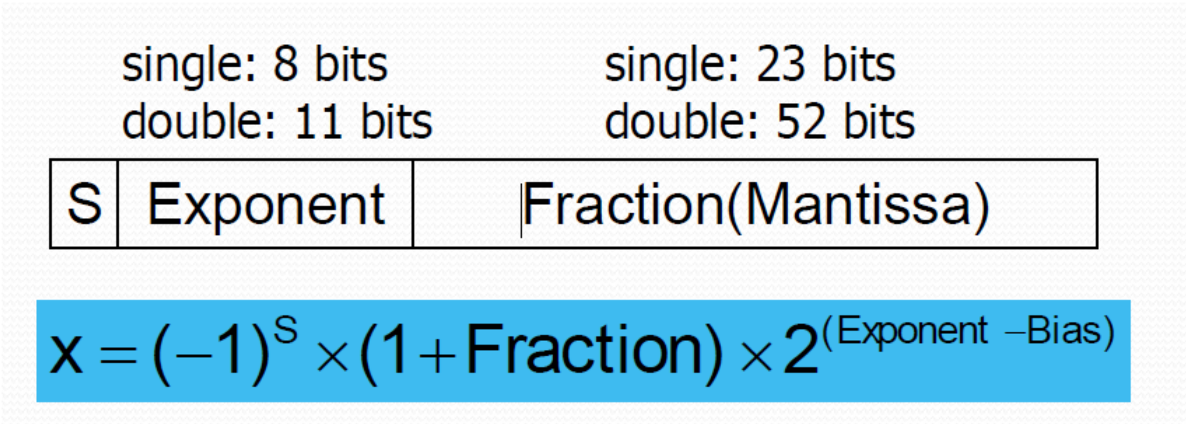
也可以拿來儲存二進制的浮點數，像



我們使用64-bit儲存Double的浮點數，使用32-bit儲存float的浮點數。

在Normalized floating point中，小數點前是一個1，則我們會說他是Normalized floating point。

二進位浮點數的呈現方式



在Normalized floating point中，我們可以知道前面的第一個數字是1，因此Fraction的部分我們直接不用呈現他，因此我們多了一個bit可以拿來做事。

但是這樣也會有一個問題，就是我們沒辦法呈現小數點前為0的結果。

而指數的部分則為Exponenet - Bias的部分是無號數，在Double的浮點數，Bias為1023，在Single的Bias為127。

浮點數的呈現範圍

考慮Single Fraction最小的呈現範圍，也就是讓Exponent = 00000001，Fraction = 00000....00000

則Exponent會是-126，結果會是$\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

考慮最大的呈現範圍，也就是讓Exponent = 11111110，Fraction = 11111....11111

則Exponent會是127，結果會大概為$\pm 2.0 \times 2^{126} \approx \pm 1.2 \times 10^{38}$

考慮Double Fraction最小的呈現範圍，也就是讓Exponenet = 00000000001，Fraction = 00000...00000

則Exponent會是-1022，結果會是$\pm 1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$

考慮Double Fraction最小的呈現範圍，也就是讓Exponenet = 11111111110，Fraction = 11111...11111

則Exponent會是1023，結果會大概為±$2.0 \times 2^{1023} \approx 1.8 \times 10^{308}$

浮點數的精確度

在浮點數中的Fraction，是個有號數。

在Single的Fraction中，精確度約為2^{-23}bits，換算大概是$23 \log 2 \approx 6$，因此在十進制中，只能精確呈現小數點後六位的數字

在Double的Fraction中，精確度約為2^{-52}bits，換算大概是$52 \log 2 \approx 16$，因此在十進制中，只能精確呈現小數點後十六位的數字

舉個例子，若我們要呈現-0.75，則$-0.75 = (-1)^1 \times 1.1_{(2)} \times 2^{-1}$

所以可以知道，$S = 1$，Fraction的精確度為$1000...00000_{(2)}$

Exponenet = -1 - Bias

如果是Single即為$-1 - 127 = -128(\text{signed}) = 126(\text{unsigned}) = 01111110_{(2)}$

如果是Double即為$-1 - 1023 = -1024(\text{signed}) = 1022(\text{unsigned}) = 01111111110_{(2)}$

因此Single為$\color{purple}{1}\color{green}{01111110}1000\ldots00_{(2)}$，Double為$\color{purple}{1}\color{green}{01111111110}1000\ldots00_{(2)}$

再舉個例子，若我們要知道11000000101000...00的結果，則

$S = 1$，$\text{Fraction} = 0100000...00_{(2)}$，$\text{Exponent} = 10000001_{(2)} = 129$

因此$x = (-1)^1 + (1 + 01_{(2)}) \times 2^{(129-127)} = (-1) \times 1.25 \times 2^2 = -5.0$

浮點數的加法

假設我們考慮四位數十進制浮點數加法。

$9.999 \times 10^1 + 1.610 \times 10^{-1}$

我們先讓小數點對齊，也就是小的指數往大的指數靠齊

$9.999 \times 10^1 + 0.061 \times 10^1$

將兩個數字做加法，也就是$9.999 \times 10^1 + 0.061 \times 10^1 = 10.015 \times 10^1$

接著把他做調整，使他符合科學記號的定義，得到1.0015×10^2

如果有需要的話，可以將他做四捨五入等調整，得到1.002×10^2

浮點數加法的硬體實作

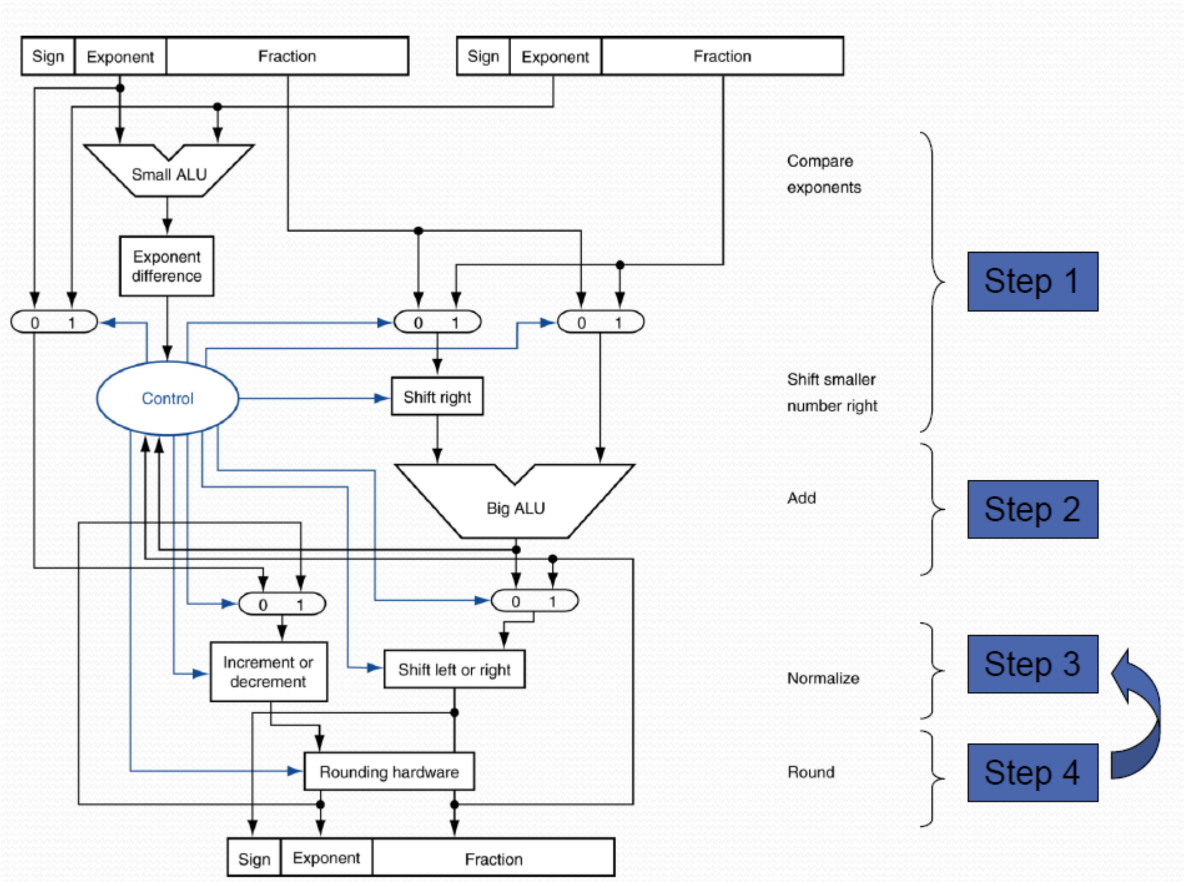
比整數加法還要複雜。

如果要在一個Clock Cycle內完成浮點數加法的部分，那一定比整數加法的耗時還要久。

如果Clock很慢，就會順勢影響到所有的指令。

所以浮點數加法通常會耗費很多個Cycle，且浮點數加法可以被管線化。

結構上大概長這樣



浮點數運算子的硬體實作

浮點運算中，乘法跟加法的複雜度差不多，但是有效位數的部分使用的是乘法器而不是加法器。

浮點數運算子的硬體實作通常做：加法、減法、乘法、除法、倒數、開根號。

且運算子通常會耗費很多個Cycle，運算子也可以被管線化。

結合律

程序的平行可能會讓運算子的位置亂跑，所以假設結合律會爛掉，就會出現以下的情況

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

Need to validate parallel programs under varying degrees of parallelism.

除法右移的問題

右移僅只限於unsigned integer，算數運算子的右移會複製sign bit.

例如$-5/4$，我們可以寫成$11111011 >> 2$，這樣會變成11111110，如果再多做幾次他會變成$-\infty$

誰會在乎精確度？

1. 對於科學來說很重要，當然現實生活中你也不會希望你的餘額少了0.00002塊錢 (除非你不care之類的)
2. Intel FDIV Bug