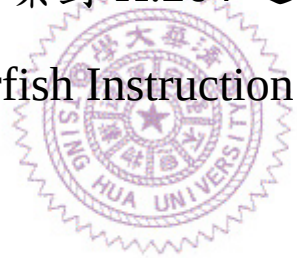
國　立　清　華　大　學

碩　士　論　文

題目：Starfish 指令集對 H.264 之指令使用率的探討

Study of Starfish Instruction Set Usage for H.264



系 所 別：資訊工程學系碩士班

學號姓名：9562557 曾鈞澤 (Jyun-Ze Zeng)

指導教授： 黃婷婷 博士 (Dr. TingTing Hwang)

中華民國九十七年八月

致　謝

即將要碩士畢業，也將告別學生身份了，在這一路上，要感謝的人太多了，就還是容我慢慢道來吧。

首先感謝我的指導教授—黃婷婷教授，攻讀碩士的這兩年中，婷婷教授提供我很大的自由去學習各方面的知識，並在最後的畢業論文上給予充分的指導，且於口試時間的安排上，當我並未確實告知教授暑期實習的時間，教授在告誡我這是不對的情況下，仍幫忙我把後續事情安排好，讓我得以順利完成口試與實習，真的很感謝教授對我的幫忙與兩年來的指導。

其次是感謝我的母親。先父在我國中二年級時便因癌症過世，之後完全靠母親工作讓我和弟弟能繼續讀書，如今碩士畢業，總算能讓母親卸下擔子了，也希望自己未來工作之餘，仍有時間多陪伴母親。

再來是從大學專題到研究所，一路陪伴著的夥伴：介俊、PP 以及林公。感謝你們的照顧，無論是課業上的互相討論，亦或是系運上共同奮鬥奪冠軍的歡笑；從剛進研究所的馬力歐賽車，到離別前的星海爭霸；雖然很可惜沒有一起去 ICCAD 參加比賽，但我會記得這兩年來的點點滴滴；很高興有你們的陪伴，希望未來有機會在一起打球。

感謝研究所的學長姐：熊爸、MC、Rider、PY、賢德，以及最美麗的雯大；謝謝熊爸大助教，風趣穩健，無論在課業還是各項雜事上，一直指點著我們 (除了球場之外)；感謝 MC 的加入，讓報告 Paper 的輪值上又多了一人；感謝 Rider，跟我們分享各種趣事；感謝 PY，在我實習時帶我去吃下午茶；感謝賢德，在報告論文時分享先進製程上的各種東西；感謝雯大，是妳的美麗照耀了整個實驗室。

還有 CT 實驗室的夥伴：Ortiz、阿福、小本、阿達；感謝你們在球場上以及 BBO 上的陪伴，我們擁有共同的兩年回憶；感謝各位大學同學：標哥、Punk、landen、dowlin…很多很多人，是你們陪我渡過美好的大學時光，在球場上、迎新營、資工營、系學會、畢業旅行，在各種活動中，彼此的歡笑淚水、辛苦愉悅，各種回憶，我將刻在心中，不忘…。

回憶連結著彼此，希望未來的故事仍然有你們，感謝曾經遇過的每個人，因爲有你們，才有現在的我，謝謝。

Abstract

As the multimedia technology development, there are more and more video/audio standards. In order to encode or decode different video/audio standards in one system, DSP (Digital Signal Processor) is a good choice to do video/audio process.

The speed of DSP is influenced much by the hardware architecture and the efficiency of compiling instructions, and both of them are decided when defining instruction set. In this paper, we profile the instruction usage with different compilers to analyze why some instructions are nearly used in H.264. We use the instruction set and compilers of Starfish DSP, which is developed by TsingHua University and ChiaoTung University to do our experiment.

Study of Starfish Instruction Set Usage for H.264

Student: Jyun-Ze Zeng
Advisor: TingTing Hwang

Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 300

目錄

圖片索引

表格索引

摘要

隨著多媒體技術的發展，影音格式的種類也越來越多，爲了能夠處理多種影音格式，使用 DSP (Digital Signal Processor)便成了最佳選擇。

除了硬體的時脈頻率(Clock frequency)之外，DSP 的執行速度與其硬體架構，以及編譯器能否有效率的編譯程式有很大的關係，而上述兩點皆受到指令集(Instruction Set)很大的影響，在本篇中，將特別討論指令集與編譯器之間的關係。

Starfish 是由清華大學、交通大學多位教授一起開發的低功率 DSP，從指令集、編譯器，到整顆 DSP，皆是由兩校的教授學生共同完成；在本篇中，將以 Starfish 爲例子，分析不同編譯器對程式使用率的影響，以及執行 H.264 時爲何有某些指令的使用率不高的情況發生。

第一章

簡介

隨著現代人的生活品質日益提升，人們觀看各類影片、音樂等多媒體的需求也逐漸增加，對於各類娛樂越來越重視，新一代 3G 手機也把影音多媒體的傳輸播放列爲重要特性之一 [1]，可見影音處理的需求是越來越重要；對於各類專門處理影音的主機而言，一般的 CPU (Central Processing Unit)屬於廣泛目的(general purpose)的使用，對於影像處理等需要高度數學運算的處理並沒有優勢，爲了加速多媒體影音的執行，這些播放平台—尤其是手機—需要一個可以專門處理多媒體播放的處理器，而這也是 DSP (Digital Signal Processor)之所以產生的原因。

DSP 是專門處數位訊號的處理器，和 CPU 相同的是需要編譯器等軟體方面的協助來將應用程式轉換成處理器所能接受的機械語言 [12]；然而相對於應用廣泛的 CPU 而言， DSP 定義了大量關於數學運算方面的指令，甚至是專門用在某些常用函式的指令 [2] [13]，來加快程式的執行，使得 DSP 能快速處理特定的應用程式，如各種影音編碼、解碼器等等，雖然各種影音的編碼、解碼器彼此之間都有些差異，但核心常用的函式仍然類似，使用 DSP 能方便的處理這方面的運算；而若將這些應用程式作成 ASIC (Application-specific Integrated Circuit) [3] [14]，會因爲程式之間的差異性，使得無法以一個 ASIC 來包含所有的應用程式，只能針對某個應用程式來做成對應的 ASIC，雖然速度比使用 DSP 還要快上許

多，但卻缺乏了使用彈性。因此選擇 DSP 作爲多媒體資料的處理器，可以獲得比一般 CPU 還快的影像處理速度，也比選用 ASIC 來的有彈性，可以對使用不同編碼的影音資料做解碼，這在目前行動寬頻的世代，許多平台如手機經常用來接收各種影音來說，是很方便的。

DSP 的執行速度，除了和硬體的運算頻率(cycle per second)有直接的關係之外，和指令集也有關係；一般而言，可以使用的指令越多，就可以用越少的指令來執行程式，但這還要考慮到編譯器(Compiler)編譯程式的效率，以及增加指令所需要付出的代價；Starfish 是由清華大學及交通大學共同研發的一款 DSP [4]，從定義指令集開始，到編譯器、模擬器(Simulator)，到硬體的實現，都是由兩校的教授學生共同完成，在這篇論文中，將以 Starfish 作爲研究對象，討論編譯器對指令使用效率的影響，並以 H.264 [5] [11] 作爲測試程式，分析指令使用率，討論執行 H.264 時，爲何某些指令的使用率特別低。

之後各章節的架構如下：在第二章中，將描述目前市面上常見的 DSP，第三章將敘述 Starfish 指令集的架構，第四章討論編譯器對指令使用率的影響，第五章分析某些指令使用率低落的原因，最後在第六章做個總結。

第二章

相關研究—常見 DSP 介紹

在市面上常見的 DSP 提供商(vendor)包括 TI (Texas Instrument)、Freescale、ADI (Analog Device)以及 Agere 等四家廠商，雖然由於半導體產業的競爭，ADI 的手機部門已經由聯發科(Media Tek, MTK)收購，Agere 也被 LSI 及 Infineon 合併，但這四家廠商的 DSP 仍具有相當的市佔率；圖 2.1 [6] 是 Forward Concepts 所公佈的 2007 年各家 DSP 提供商的市場佔有率，可以看見 TI 與 Freescale 提供的 DSP 有將近八成的市佔率；Agere 雖已被合併，但 Forward Concepts 仍將其 DSP 的占有率列出，約 7%；ADI 原先市占率最高的手機部門已被收購，在市面上只剩下通用(multipurpose)的 DSP；其它的廠商包括 NEC、NXP、Renesas 等等，也佔了 13%的市場。

表 2.1 [7] 列舉了由 ADI 生產的 ADSP-BF5xx 系列、Freescale 的 MSC71xx 系列，以及 TI 的 TMS320C64x 與 TMS320C67x 系列的 DSP 之間比較；其中第三列的 fixed/floating 是指 DSP 是處理固定位數運算(fixed point)或者是浮點數運算(floating point)。fixed point 的 DSP 因爲容易對硬體做優化，運算速度通常比較快，但若所執行的應用程式中有浮點數運算的話，需要編譯器(compiler)的協助才能處理，花費的時間反而較 floating point 的 DSP 爲多，因此可以依照應用程式的需求來決定使用哪一種 DSP；第六、七列是 BDTI(Berkeley Design Technology) [8] 的測試程式對不同 DSP 的評分，分數越高評價越好。

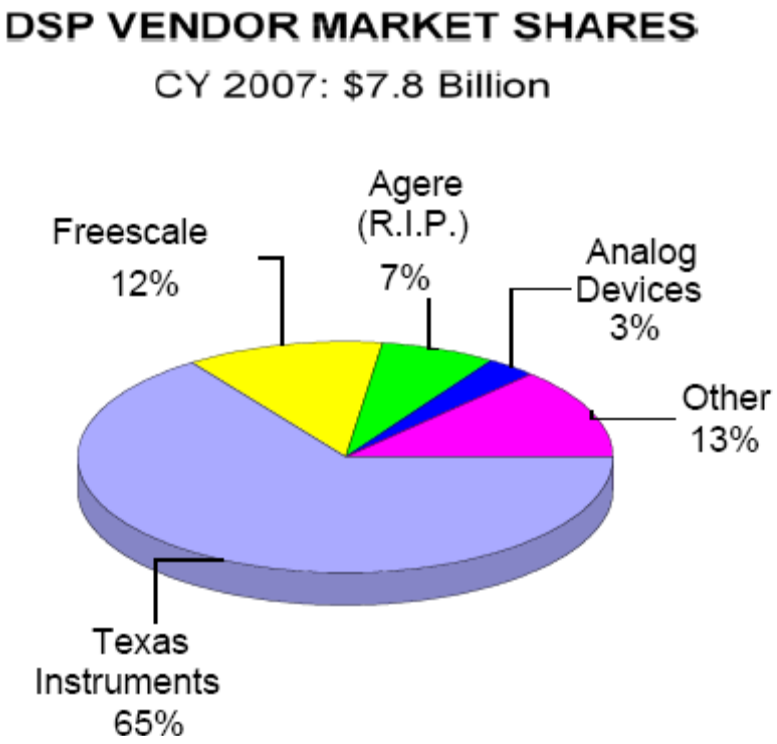


圖 2.1 各家 DSP 提供商的市場佔有率

表 2.1 不同 DSP 之間的比較

Vendor	Analog Devices	Freescale	Texas Instruments	
Family	ADSP-BF5xx (Blackfin)	MSC71xx (SC1400)	TMS320C64x/ DM64x	TMS320C67x
Floating, Fixed or Both	Fixed Point	Fixed point	Fixed point	Floating point
Instruction Width	16/32 bits	16 bits	32 bits	32 bits
Core Clock Speed	750 MHz	300 MHz	1 GHz	300 MHz
Architecture	3-way VLIW	6-way VLIW	8-way VLIW	8-way VLIW
Score of BDTImark2000	**4190**	**3370**	**9130**	**1500**
Total On-Chip Memory, Bytes	52 K–328 K	408 K–472 K	128K – 1032K	72 K–264 K
Unit Price	$5–60	$25–35	$9–198	$12–29

ADSP-BF5xx 系列的特色在於它們是低功率的 DSP，隨著運算速度的不同，所消耗的能量也不一樣；舉例來說，ADSP-BF533 在 100MHz 時，消耗的功率是 24mW，當運算速度到達 756MHz 時，消耗功率爲 644mW，因此 ADSP-BF5xx 適合向手機這方面對消耗功率敏感的應用；ADSP-BF5xx 的架構是 3-way VLIW (Very Long Instruction Word)，單位時間內可以同時執行 3 個指令，和 Freescale 與 TI 的 DSP 相比，平行度較差，但因爲執行速度可達 750MHz，因此在 BDTI 的測試程式評分中，有著不錯的評價。

MSC71xx 是另一個高效能的 DSP，雖然與 ADSP-BF5xx 系列相比，運算速度最高只有 300MHz，但 6-way VLIW 的架構，使得它在 BDTI 的測試程式中，有著不遜色於 ADSP-BF5xx 系列的分數，在平均價格中，更是較爲低廉，這些特色使得它在低成本(low cost)的低端應用程式中佔有優勢；MSC71xx 系列是以 Starcore 的 SC1400 DSP 爲基礎而設計的，Starcore 組織是由 Freescale、Agere 以及 Infineon 爲了和 TI 競爭所共同創立的 DSP 研究公司，雖然最終在激烈競爭下被迫解散，但其員工仍轉移給 Freescale、Agere 及 Infineon， 而 Freescale 也在 Starcore 所開發的 DSP 如 SC140、SC1400 等爲基礎，繼續研發自己系列的 DSP。

TMS320C64x 系列是目前 TI 最高效能的 fixed point DSP，它是針對需要高效處理的應用程式所開發，如 3D 影像處理、網路數據處理等，在高達 1GHz 的運算速度，8-way VLIW 的架構下，TMS320C64x 系列能有效處理這些程式，當然平均價格也較其他 DSP 高上不少；TI 開發另一系列的 DSP—TMS320C67x—與 C64x 系列的差異在於它是 floating point 的 DSP，因此運算速度較 C64x 系列低上許多。

第三章

Starfish 指令集介紹

Starfish 是由清華大學以及交通大學合力研發的低功率(low power)DSP，主要執行的應用程式爲 MP3 及 H.264 等多媒體方面的處理；它採用台積電 90nm 製程，3-way VLIW 的架構，運算速度最少爲 180MHz，最大特色是低功率，平均每百萬赫茲消耗 0.11 毫瓦(0.11mW/MHz)，市面上較低功率的的 DSP 如 ADI 的 ADSP-BF533 所消耗的功率至少在 0.24mW/MHz 以上，仍較 Starfish 高出一倍以上；Starfish 大致的規格如表 3.1。

表 3.1 Starfish 的規格列表

	Status
Process	90nm CMOS LP
Architecture	3-way VLIW
Instruction Width	16 / 32 bit
Gate Count	250K
Speed(MHz)	285(TT), 180(SS)
Power	0.11mW/MHz
Demo Applications	-MP3 -JPEG2000 -H.264 decoder(Baseline Profile, QCIF@30fps)

在指令集方面，Starfish 原先定義了 12 類指令，後來爲了增進 H.264 的執行效率，又新增關於 H.264 解碼器中的 interpolation 和 de-blocking filter 方面的指令[9][10]，表 3.2 列出 Starfish 指令集中各類指令及相關說明。

除了一般處理器必要的指令如 Load / Store、Move 之外，在 Arithmetic Operations 中還定義了乘加(Multiply and Accumulate)方面的指令，這使得 Starfish 容易處理矩陣乘法等需要大量乘加運算的程式；Starfish 也定義了關於影像處理方面的指令—Video Pixel Operations，以加快執行 H.264 等影像編解碼器的速度，在之後的章節中會特地爲這方面的指令作分析。

表 3.2 Starfish 中指令種類與說明

指令種類	說明
Program Flow Control	包括 Jump、If…else…等控制程式執行的指令
Load / Store	對 Memory 做存取的指令
Move	對 Register 的值做搬移的指令
Stack Control	對 Stack 做存取(push / pop)的指令
Control Code Bit Management	Control Code Bit 儲存兩個值比較之後的結果，是用來輔助 If…else 等指令
Logic Operations	關於 And、Or 等邏輯運算的指令
Bit Operations	關於 Bit 運算的指令
Shift / Rotate Operations	關於 Shift / Rotate 的指令
Arithmetic Operations	關於加減法、乘加等數學運算的指令
Video Pixel Operations	和影像處理有關係的指令
Vector Operations	同時對 Register 的 HI/LO 兩部份做運算的指令
Parallel Instructions	將兩個以上，最多三個可以同時執行的指令所組合而成的非常規指令
New Instruction	針對影像處理中 Interpolation 及 De-Blocking filter 做特殊運算的指令

而 Vector Operations 是一些同時對 Register 的前 16 個位元(high half-register)及後 16 個位元(low half-register)做運算的指令，包括加減法、乘加法以及取最大、最小值等常見的 ALU(Arithmetic Logic Unit)運算；使用 Vector 方面的指令可以同時處理二筆資料，增加指令的效率，但爲了實現這方面的指令，硬體方面需要付出一些成本，需要兩個 ALU 才能同時處理這些運算。

Starfish 是 3-way VLIW 的架構，最多可以同時處理三個指令，指令集中的平行指令(parallel instruction)部分便是爲此而定義的；Starfish 指令長度爲 16 和 32 bit，一個 32 bit 與二個 16 bit 的指令可以組合成一個平行指令，通常是一個 ALU 或是 Video 方面的指令搭配上二個 Load / Store 方面的指令，這些在指令集中都有詳細說明。

指令集中定義了哪些指令會影響硬體的設計，過多的指令只會徒耗硬體成本而不會增加執行效率，在之後的章節中會分析各類指令使用率，討論爲何在執行 H.264 時某些種類指令的使用率會如此之低，並探討編譯器對指令使用的影響。

第四章

編譯器對指令使用的影響

第一節 概論

指令集定義好之後，仍需靠編譯器才能將高階語言編譯成機械碼，故編譯器能否有效率的編譯程式，對執行速率有直接的影響；下列將以 Starfish 作爲例子，討論不同版本的 Starfish 編譯器對全部指令以及平行指令使用率的影響。

在下列的實驗中，是以 Starfish 的模擬器(Simulator)，對編譯出來的 H.264 decoder 做模擬，使用的測試檔是 H.264 3 frame 的檔案，編譯器版本分別是 CC34_O3、CC41_O3_DSPlib 以及 CC41_O3_DSPlib_Alg，其中 CC34 表示由 gcc 3.4 改編成的編譯器，O3 代表編譯時打開所有 gcc 支援的優化特性，DSPlib 和 Alg 則都是人工所撰寫的 library；下列實驗都是以這三種版本的編譯器做比較。

第二節 程式使用率與所使用指令

將經過編譯後的程式，把指令由使用次數高到低依序累加，可以得到圖 3.1；從圖 3.1 中可以觀察到兩件事；

1. 80—20 理論，80%的程式執行著 20%的指令，以 CC34_O3 的編譯器爲例，共使用約 150 個指令，但程式 80%的時間只執行其中 20 多個指令，只佔總使用指令的 16%；而從總指令數來說，Starfish 指令集共有 769 個指令，更是只佔了不到 5%，對 H.264 decoder 而言，大部分指令是不常使用的。

2. 在圖 3.1 中，CC41_O3_DSPlib_Alg 版本的編譯器，在同樣的程式使用率下，使用的指令種類比其它版本的編譯器多，這是因爲沒有 library 時，編譯器是使用一些基本的指令來描述函式，使用指令種類少且效率較低；當某些函式時常被執行，以人工的方式寫成 library，使用指令種類較多且效率較高；許多指令集中所定義的指令，都是要寫成 library 才能有效率的使用。

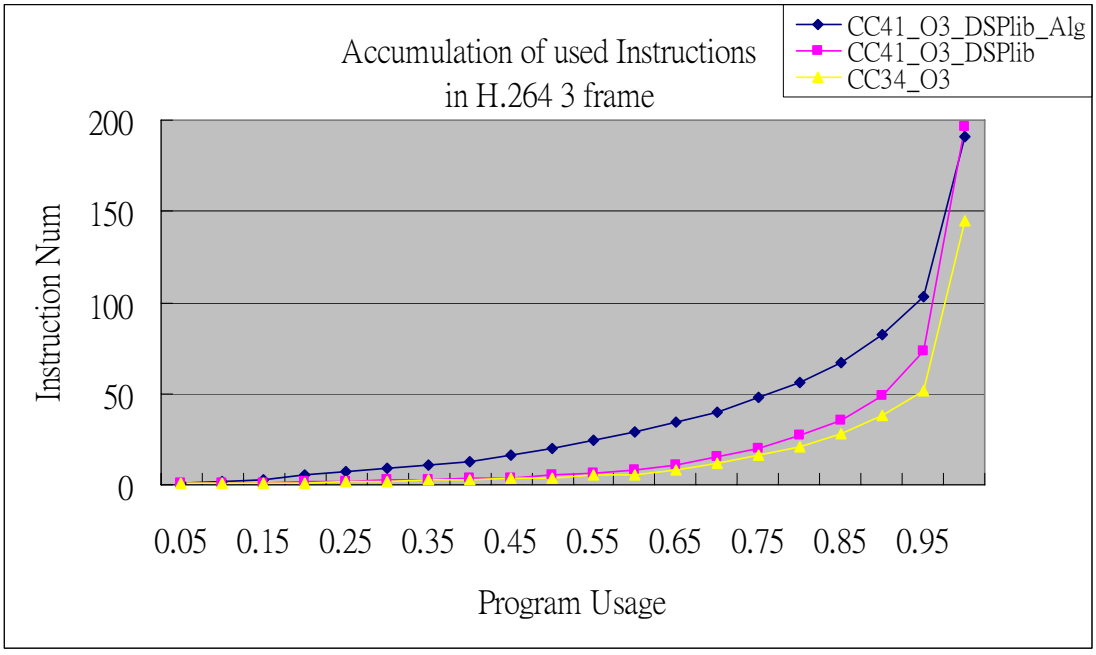


圖 4.1 指令使用數對程式使用率在不同編譯器下之累積圖

第三節 不同編譯器對各類指令使用數的影響

圖 3.2 是不同的編譯器對各類指令使用的次數關係，可以看到不論是哪一類的指令，在加入 library 之後，都減少了使用次數，而 CC41_O3_DSPlib_Alg 版本的編譯器，因所使用的 library 較多，總指令數只有 CC34_O3 的 7%。

圖 3.3 是不同的編譯器，各類指令所佔的百分比；在最原始的編譯器 CC34_O3 中，使用了許多 Shift/Rotate 以及 Arithmetic 方面的指令；而加入了 library 之後，Load/Store 方面的指令使用率大幅提升，這是因爲爲了加快執行速度，在 library 中經常使用查表法來取代原本的運算，而 vector 方面的指令使用率增加，也減少了其它一般指令的使用量。

從圖 3.2 與圖 3.3 中可以看到，library 的使用，能有效率的減少總指令數，提升指令的執行效率，讓編譯器不再重複使用少數的指令來編譯；而 Load/Store 指令的使用率增加，也告訴我們需要針對記憶體管理(Memory Management)做有效的處理。

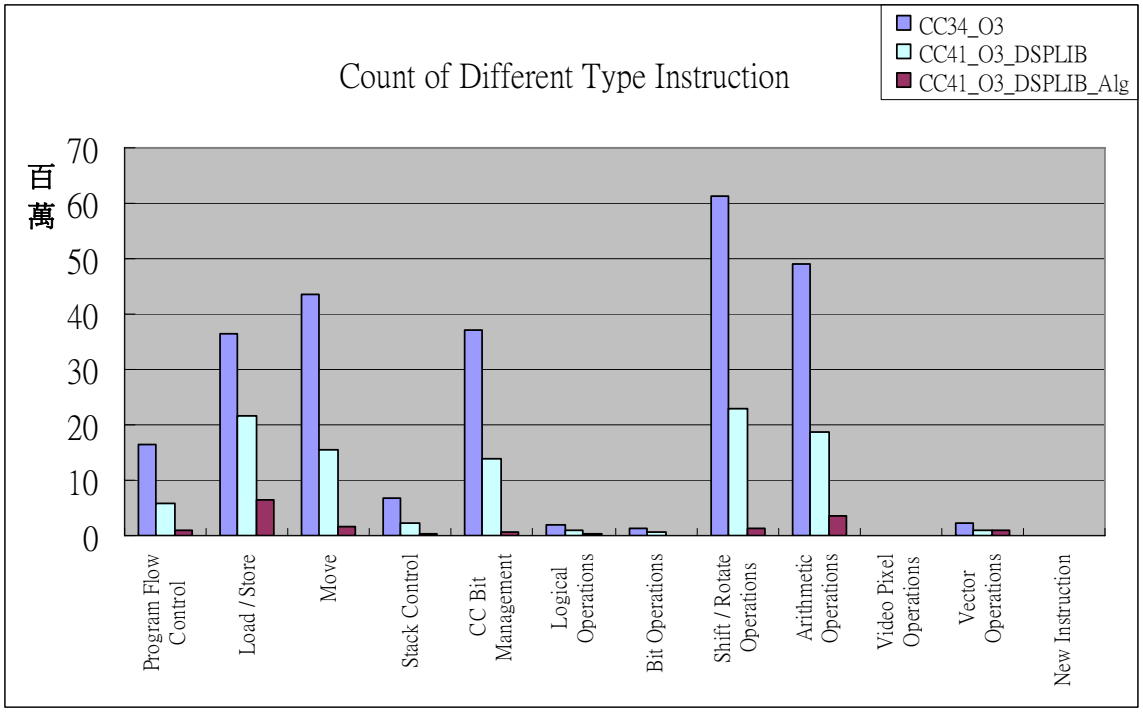


圖 4.2 不同編譯器下各類指令的總使用數

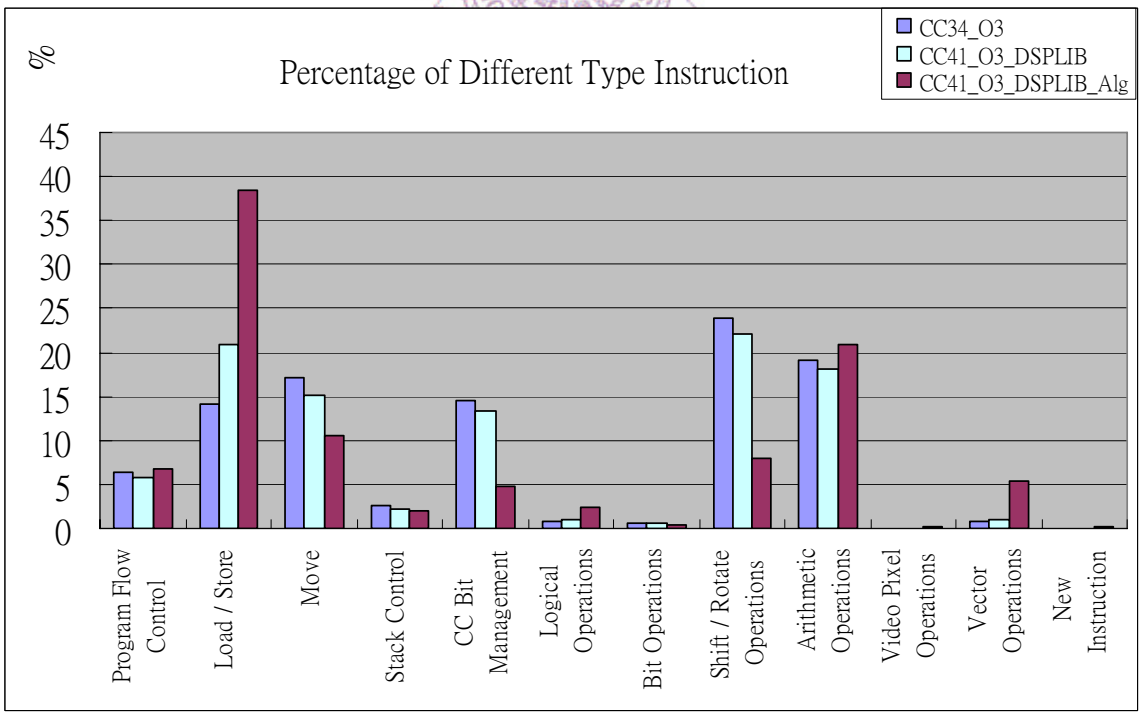


圖 4.3 不同編譯器下各類指令使用率

第四節 不同編譯器下平行指令的使用率

所謂的平行指令(parallel instruction) 是指一個 cycle 內同時執行數個指令，Starfish 中最多可以同時執行 3 個指令(3-way parallel instruction)，圖 3.4 是比較不同編譯器使用平行指令的情況。

圖 3.4 中，加入越多的 library，使用越多的平行指令，在沒有 library 的情況下，編譯器很難自動產生平行指令；但即使加入 library，Starfish 使用平行指令的效率仍不高，以 CC41_O3_DSPLib_Alg 版本的編譯器爲例，3-way 的平行指令只佔整體不到 1%，大部分是 2-way 平行指令，甚至還有許多 1-way 的平行指令。

平行指令的使用率仍有提升的空間；扣除掉 1-way 的平行指令後，平行指令使用率最多不高於整體的 5%，但若要提高指令使用效率，加入更多的 library，使用更多的平行指令是最快的方式，

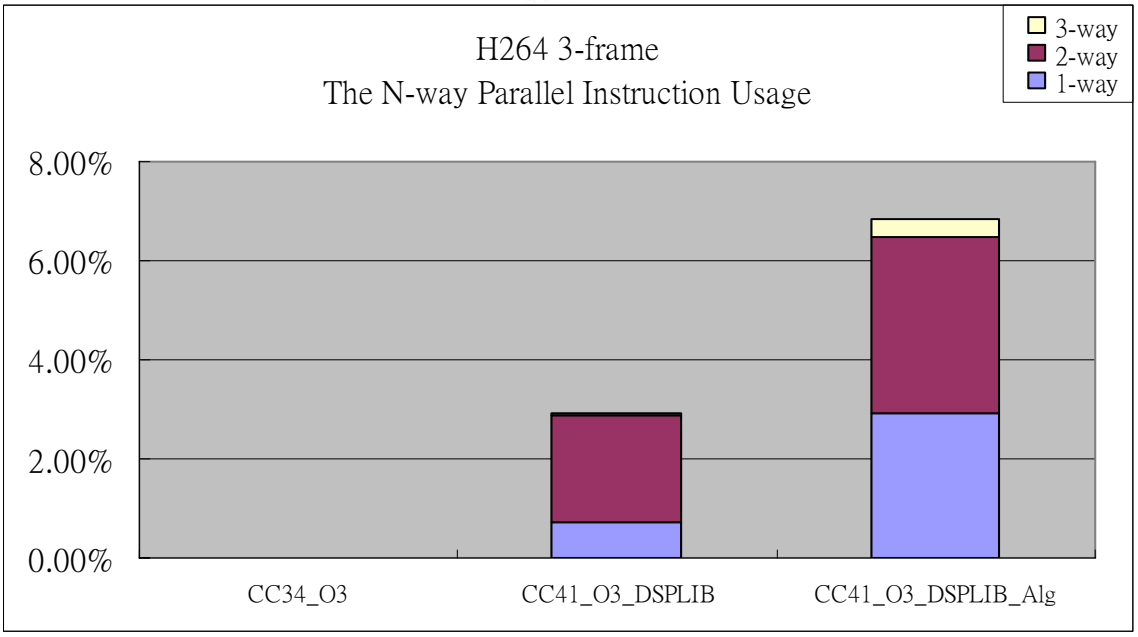


圖 4.4 不同編譯器下平行指令使用率

第五節 結論

指令集中大部分的指令並不會被使用到，以 Starfish 指令集爲例，總共 788 個指令在 H.264 3 frame 的測試中，只使用了不到 200 個指令，而大多數時間都在執行其中的數十個指令；而所使用的 library 越多，用到的指令也越多。

編譯器加入 library 之後，編譯出來的程式中，Load/Store 指令所佔的比例增加，Shift/Rotate 以及 Conditional Bit 方面的指令使用率大幅減少，也比較不再集中使用某些指令；指令使用效率的提升，大幅減少執行時間，但從平行指令的使用情況來說，執行效率仍有進步的空間。

以 H.264 的例子而言，平行指令的使用率約 7%，但這之中大多是 2-way 及 1-way 的平行指令，3-way 的平行指令只用了不到 1%，仍有提升空間。

第五章

指令使用率之分析

第一節 概論

在前一個章節中提到，Starfish 指令集總共定義了 788 個指令，但執行 H.264 3 frame 的例子時，只會使用不到 200 指令；究竟哪一方面的指令是不常被使用的呢？以下是爲此所作的分析。

表 5.1 是使用 CC41_O3_DSPLib_Alg 版本編譯器，編譯 H.264 3 frame 檔案時，各類指令的指令使用數以及使用率；我們觀察發現 Arithmetic、Video Pixel，以及 Vector 方面的指令使用率是最低；在 Arithmetic 和 Vector 方面的指令中，MAC(Multiply and Accumulate)的指令佔了 62%，卻只使用了不到 3%，因此我們將重點分析 Video Pixel 和 MAC 方面的指令，來了解爲何使用率如此之低。

表 5.1 以 CC41_O3_DSPLib_Alg 編譯 H.264 時各類指令的使用率

Type	Total Instruction Num	Used Instruction Num	Usage
Program Flow Control	26	14	53.85%
Load / Store	102	57	55.88%
Move	58	13	22.41%
Stack Control	10	10	100.00%
CC Bit Management	34	20	58.82%
Logical Operations	8	4	50.00%
Bit Operations	12	6	50.00%
Shift / Rotate Operations	68	18	26.47%
Arithmetic Operations	296	34	11.49%
Video Pixel Operations	30	1	3.33%
Vector Operations	144	13	9.03%
total number	788	190	24.11%

第二節 分析 **Video Pixel** 指令的使用

2.1 Video Pixel 指令簡介

爲了加快影像處理，Starfish ISA 特別定義了一些以 byte 爲單位的指令，來加速影像編碼、解碼的動作。這些指令可以使用在某些特定的函式，如 motion compensation 中的 intra-prediction、video pixel 與 residual 的相加等等；然因其特殊性，需要另外寫 library 才能妥善的運用這些指令，而無法由 compiler 自動產生。

下列是所定義的 Video Pixel Operation 指令：

表 5.2 Video Pixel 指令列表

ALIGN8, ALIGN16, ALIGN24
DISALGNEXCPT
BYTEOP3P (Dual 16-Bit Add / Clip)
Dual 16-Bit Accumulator Extraction with Addition
BYTEOP16P (Quad 8-Bit Add)
BYTEOP1P (Quad 8-Bit Average - Byte)
BYTEOP2P (Quad 8-Bit Average – Half-Word)
BYTEPACK (Quad 8-Bit Pack)
BYTEOP16M (Quad 8-Bit Subtract)
SAA (Quad 8-Bit Subtraction-Absolute-Accumulate)
BYTEUNPACK (Quad 8-Bit Unpack)

Video Pixel 指令依照功能性，可以分成普通的以及有特殊用途的指令；普通的 Video 指令包括 ALIGN Operation、DISALGNEXCPT、BYTEPACK 以及 BYTEUNPACK，這些指令通常不會單獨使用，而是用來輔助處理其它的 Video 指令；特殊用途的指令包括 SAA、BYTEOP1P、BYTEOP2P 等指令，專門用在某些函式，如 intra-prediction 等等，以下將對這些特殊用途指令做個別介紹。

2.2 分析 H.264 中 Video Pixel 指令的使用情況

雖然 Video Pixel Operation 是專門用在影像處理方面的指令，但我們在 H.264 的例子上卻只使用了 BYTEPACK 這一個指令，這是因爲我們並沒有把這些指令用在 library 中，下列將分析 SAA、BYTEOP1P、BYTEOP2P 以及 BYTEOP3P 這四個指令，來討論爲何沒有將這些指令寫進 library 中，並討論在哪些情況下，能較方便的使用這些指令。

2.2.1 BYTEOP1P (Quad 8-Bit Average - Byte)

- 語法

  dest_reg = BYTEOP1P (src_reg_0, src_reg_1);

- 指令說明

將 src_reg_0 和 src_reg_1 中每個 byte 的值兩兩相加取平均，如圖 4.1 所示，平均值可以選擇無條件捨去 (truncation)，或是預設的四捨五入(round)；與 BYTEOP1P 很接近的指令是 BYTEOP2P，差別在於 BYTEOP2P 是取四個值的平均，因此這二者僅列出 BYTEOP1P 來說明。

Source Registers Contain

	31.....24	23.....16	15.....8	7.....0
aligned_src_reg_0:	y3	y2	y1	y0
aligned_src_reg_1:	z3	z2	z1	z0

Destination Registers Receive

	31.....24	23.....16	15.....8	7.....0
dest_reg:	avg(y3, z3)	avg(y2, z2)	avg(y1, z1)	avg(y0, z0)

圖 5.1 BYTEOP1P 指令說明

BYTEOP1P 可以用在 interpolation 或 intra-prediction 中，相對應的 pseudo-code 以及 assembly code 如下：

表 5.3 BYTEOP1P 指令的 Pseudo code

Pseudo-code – C language of BYTEOP1P

```
For (i=0;i<4;i++){
      Dest[i] = (Src1[i] + Src2[i] + 1) >> 1;
}
```

Assembly code – with BYTEOP1P, byte allocation

```
MNOP     || R0 = [I0++];   || R2 = [I1++];
R7 = BYTEOP1P(R1:0, R3:2);

[I2++] = R7;
```

因爲 BYTEOP1P 最後在處理值的時候，預設是四捨五入(round)，因此 Src1 和 Src2 相加之後不需要再加 1，而若使用一般的加法指令，則需要多做此處理，這在後面的比較中可以看到。

在上述的 assembly code 中，是假設每個 pixel 只需要 1 byte 的空間，因此在讀進資料後可以直接執行 BYTEOP1P，不需做額外的處理，但是當 pixel 的大小不只 1 byte 的時候，使用 BYTEOP1P 就不是那麼有效率；Starfish 使用的 H.264 測試檔—H.264 JM Software—每個 pixel 的大小是 2 byte，以下是當每個 pixel 需要 2 byte 的空間時，使用與不使用 BYTEOP1P 時，兩者的 assembly code，以此比較兩者的執行效率。

表 5.4 使用 BYTEOP1P 指令的 Assembly code

Assembly code – with BYTEOP1P, half-word allocation

```
MNOP     || R4.L = W[I0++]     || R5.L = W[I1++];
MNOP     || R4.H = W[I0++]     || R5.H = W[I1++];
MNOP     || R6.L = W[I0++]     || R7.L = W[I1++];
MNOP     || R6.H = W[I0++]     || R7.H = W[I1++];

R2 = BYTEPACK(R4, R6);
R0 = BYTEPACK(R5, R7);
R1 = BYTEOP1P(R1:R0, R3:R2);
(R3, R2) = BYTEUNPACK R1:R0 (R);

W[I2++] = R2.L;
W[I2++] = R2.H;
W[I2++] = R3.L;
W[I2++] = R3.H;
```

表 5.5 不使用 BYTEOP1P 指令的 Assembly code

Assembly code – without BYTEOP1P, half-word allocation

```
R4 = 0x00010001;

MNOP                 || R0.L = W[I0++]     || R1.L = W[I1++];
MNOP                 || R0.H = W[I0++]     || R1.H = W[I1++];
R6 = R0 +|+ R1 (S);  || R2.L = W[I0++]     || R3.L = W[I1++];
R6 = R6 +|+ R4 (S);  || R2.H = W[I0++]     || R3.H = W[I1++];

R6 = R6 >> 1 (V);
R7 = R2 +|+ R3 (S);
R7 = R7 +|+ R4 (S);
R7 = R7 >> 1 (V);

W[I2++] = R6.L;
W[I2++] = R6.H;
W[I2++] = R7.L;
W[I2++] = R7.H;
```

◆ 使用 BYTEOP1P 與否的比較

在表 5.4 和表 5.5 中，I0 和 I1 都是 pixel 的位址，當 pixel 大小是 2 byte，要使用 BYTEOP1P 這類 byte operation 時，需要把讀進來的資料做 BYTEPACK，運算完後也要做 BYTEUNPACK，才能把資料存入正確的位址。

在 pixel 大小是 2 byte，卻還要使用 BYTEOP1P 這類 byte operation 時，必需在 pixel 的值不會超過 255 的情況下，否則在執行 BYTEPACK 時，會發生偏差，以至於結果錯誤。

不使用 BYTEOP1P 指令時，可以用 vector 方面的指令來加速運算，雖然將 pixel 相加之後尚需執行加 1 的運算，但透過平行指令的處理，整體運算時間與使用 BYTEOP1P 時並無太大的差別。

使用 BYTEOP1P 指令，當 video pixel value 的大小不是 1 byte 的時候，原先 3 個指令可以完成的事情，此種狀況下需 12 個指令，指令的效率比 pixel 的大小是 1 byte 時差很多；而不使用 BYTEOP1P 的話，可以用 vector 方面的指令來減少指令數，需要 13 個指令數來完成，二者的效率相差彷彿。

◆ 小結

Pixel 的大小會影響到指令如何使用，若 pixel 大小是 1 byte，BYTEOP1P 可以有效率的使用；若 pixel 大小是 2 byte，使用 BYTEOP1P 就不是那麼有效率，可以用 vector 方面的指令來取代之。

2.2.2 BYTEOP3P (Dual 16-Bit Add / Clip)

◆ 語法

dest_reg = BYTEOP3P(src_reg_0, src_reg_1) (LO)
dest_reg = BYTEOP3P(src_reg_0, src_reg_1) (HI)

◆ 指令說明

如圖 4.2，依照指令的 option，將 src_reg_0 和 src_reg_1 相對應的值加起來，將結果存入對應的 byte 中；BYTEOP3P 可以用在 motion compensation 中，把經過 intra prediction、interpolation 所算出的 pixel 值和 residual 加起來，當 pixel 大小是 1 byte，residual 大小是 2 byte 時，可以很方便的使用此指令。

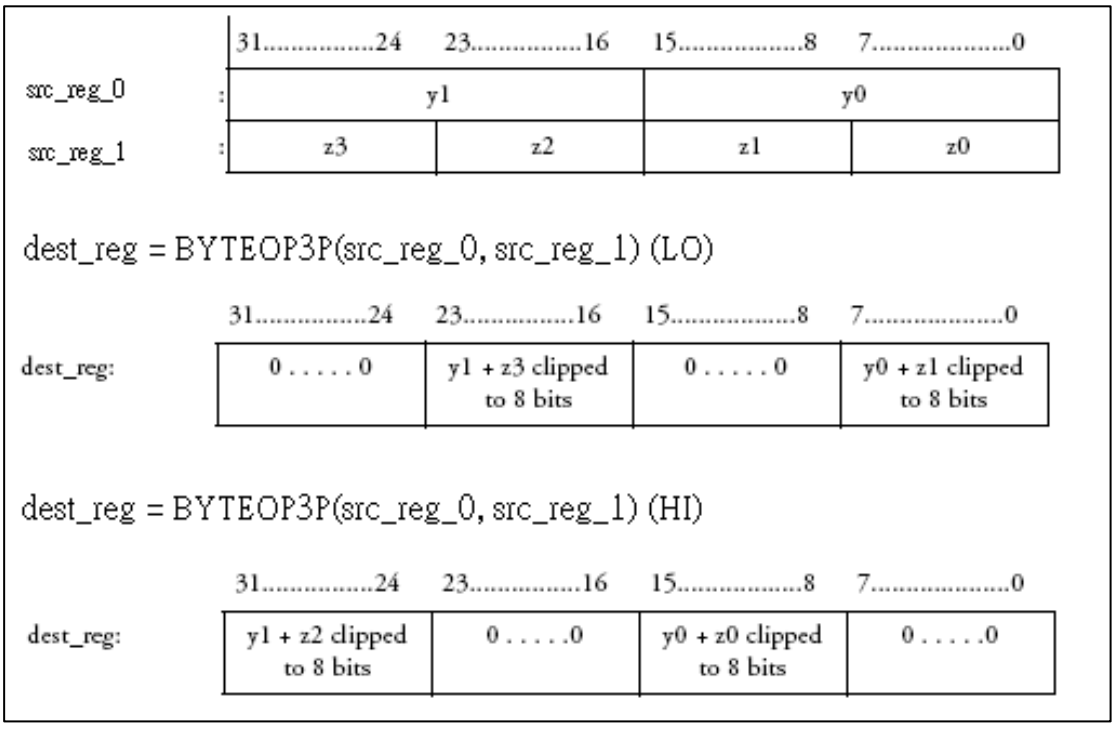


圖 5.2 BYTEOP3P 指令說明

◆ 使用 BYTEOP3P 與否的比較

表 5.6 Pixel 爲 1 byte 時使用 BYTEOP3P 指令的 Assembly code

Assembly code – with BYTEOP3P, Residual: 2 byte, Pixel: 1 byte

```
MNOP                        || R0 = [I3++]        || R2 = [I1];
R2 = R2 << 8                || R0.H = W[I3--]     || R3 = [I1++];

R3 = R3 >> 8                || R1.L = W[I3]       || I3 += 4;
R6 = BYTEOP3P(R1:0, R3:2) (LO)   || R1.H = W[I3++]    || NOP;
R7 = BYTEOP3P(R1:0, R3:2) (HI, R);
R6 = R6 + R7;
[I2++] = R6;
```

表 5.7 不使用 BYTEOP3P 指令的 Assembly code

Assembly code – without BYTEOP3P, Residual: 2 byte, Pixel: 1 byte

```
   P0 = 4;
   MNOP                     || R7 = B[P1++] (X)   || R6 = W[I3++] (X);
   LSETUP (loop_start, loop_end) LC0 = P0;
loop_start:
   R5 = R6 + R7 (S)  || R7 = B[P1++] (X)   || R6 = W[I3++] (X);
loop_end:
   B[I2++] = R5;
```

在表 5.6 中，I3 是 residual 的位址，I1 是 pixel 的位址，R1:0 的四個 half-word 儲存四個 residual，R3 等於 R2，皆存了四個 pixel 值，並透過向左和向右 shift，把相對應的 pixel 和 residual 加起來；在這個情況下需要 7 個 cycle 即可完成一個 4 x 4 區塊之 residual 和 pixel 相加的動作。

而在表 5.7 中，若不使用 BYTEOP3P，可以使用一個 loop 來減少 code size，I3 是 residual 的位址，P1 是 pixel 的位址，利用平行指令來同時完成讀取資料和相加的動作，每次運算完後馬上把結果存入輸出位址，此時共需 11 個 cycle 才能完成。

使用一般的指令，即使 residual 和 pixel 的大小有變化，程式都還是如表 5.7 所述，只是讀進的資料大小不一樣而已，但若使用 BYTEOP3P，在 residual 和 pixel 的大小不是 2 byte 與 1 byte 時，程式便無法再像表 5.6 一般精簡，表 5.8 的程式便是假設 residual 大小是 4 byte，pixel 大小 2 byte，使用 BYTEOP3P 的情況。

使用 BYTEOP3P 指令，需要 7 個指令數，不使用之，則需 11 個指令數 (3+2*4)，使用 BYTEOP3P 可以有效的減少執行時間，但這是建立在 residual 大小爲 2 byte，pixel 大小爲 1 byte 的情況；在 JM H.264 software 中，residual 大小是 4 byte(資料型態爲 integer)，pixel 大小爲 2 byte(資料型態爲 unsigned short)，在此種資料型態下，用 BYTEOP3P 很不容易來實現相對應的函式。

◆ 小結

在 residual 大小是 2 byte，pixel 大小 1 byte 的時候，BYTEOP3P 可以有效率的使用；若 residual 的大小 4 byte，pixel 大小是 2 byte 時，即使保証使用 BYTEOP3P 不會發生偏差，使用一般的指令還是比較好。

表 5.8 Pixel 爲 2 byte 時使用 BYTEOP3P 指令的 Assembly code

Assembly code – with BYTEOP3P, Residual: 4 byte, Pixel: 2 byte

```
MNOP     || R0 = [I3++]    || R4.L = W[I1++];
MNOP     || R1 = [I3++]    || R4.H = W[I1++];
MNOP     || R2 = [I3++]    || R5.L = W[I1++];
MNOP     || R3 = [I3++]    || R5.H = W[I1++];

R0 = PACK (R2.L, R0.L);
R1 = PACK (R3.L, R1.L);
R2 = BYTEPACK(R4, R5);
R3 = R2 >> 8;
R2 = R2 << 8;
R6 = BYTEOP3P(R1:0, R3:2) (LO)
R7 = BYTEOP3P(R1:0, R3:2) (HI, R);
R7 = R7 >> 8;

W[I2++] = R6.L;
W[I2++] = R6.H;
W[I2++] = R7.L;
W[I2++] = R7.H;
```

2.2.3 SAA (Quad 8-Bit Subtraction-Absolute-Accumulate)

◆ 語法

SAA (src_reg_0, src_reg_1)

◆ 指令說明

SAA 指令是將 src_reg_0 和 src_reg_1 每個 byte 的值相減，取絕對值後分別存到 A0、A1 中，如下圖所示：

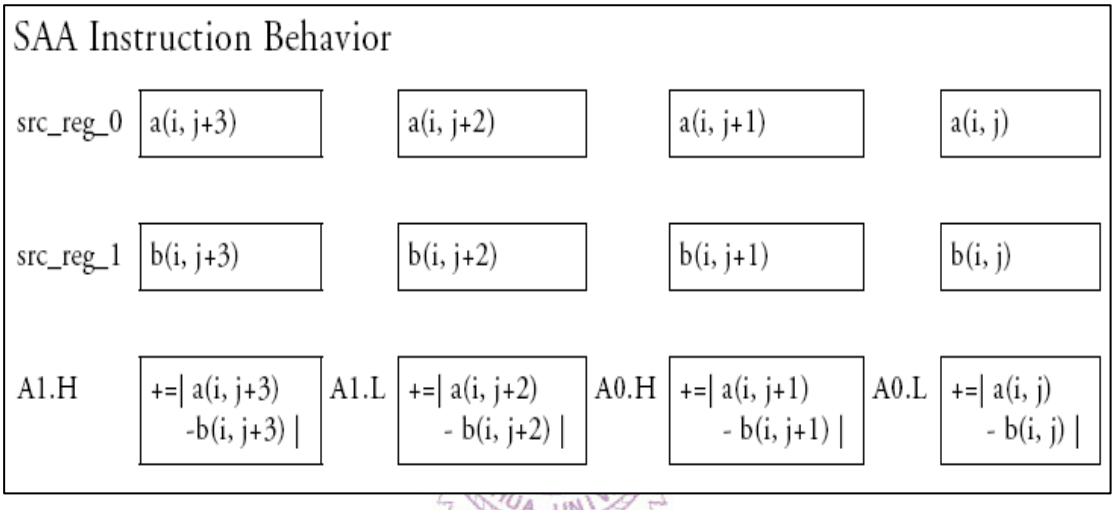


圖 5.3 SAA 指令使用說明

SAA 指令可以幫助我們實現 SAD (Sum of Absolute Difference)函式，SAD 是用來計算二個 macro block 之間的差異，以此來做影像的壓縮；它常用在影像的編碼器 (Encoder)，在做 Motion Estimation 的時候常會使用；表 5.9 是 SAD 函式的 pseudo-code。

使用 SAA 指令時，通常還會搭配另一個 Video Pixel 的指令：Dual 16-Bit Accumulator Extraction with Addition，這個指令是專門爲了 SAA 而定義的，這在之後的例子中可以看到。

表 5.9 SAD 函式的 Pseudo Code

```
Pseudo-Code – SAD function

for (i=0;i<4;i++){
        for (j=0;j<4;j++){
                result += abs(*pSrc++ - *pRef++);
        }
}
```

◆ 使用 SAA 與否的比較

SAA 指令能方便實現 SAD 函式，表 5.10 和表 5.11 是兩個 SAD 的 Assembly code，分別使用與不使用 SAA 指令，來實現上面 SAD 的函式；而在這兩個程式中，I0 代表 source picture 的位址，I1 代表 reference picture 的位址，I2 是輸出的位址；而表 5.10 的倒數第三行，將 A0、A1 的 half-word 相加的指令即是專門爲 SAA 所定義的指令：Dual 16-Bit Accumulator Extraction with Addition，可快速算出最後的累加值。

從表 5.10 與表 5.11 中可以知道，使用 SAA 指令需要 10 個 cycle，不使用 SAA 指令需要 44 個 cycle 才能完成 SAD 函式，使用 SAA 指令效率比不使用 SAA 指令提升 3 倍，但一來這是建立在 pixel 的大小是 1 byte 的時候，而 Starfish 所使用的測試檔 H.264 JM Software 每個 pixel 是 2 byte，使用 SAA 指令效果會降低；再者 SAA 指令只有在運算 Motion Estimation 時才會用到，而我們的測試資料是只有執行 H.264 解碼器的部份，因此沒有用到 SAA 的指令。

◆ 小結

在 memory 以 byte 爲單位的情況下，SAA 指令能確實降低執行編碼所需的時間，而因爲在 Starfish 的實驗中，只有執行 H.264 解碼功能，因此沒有將 SAA 指令寫成 library。

表 5.10 使用 SAA 的 Assembly code

Assembly code – with SAA, Byte Allocation

```
P0 = 4;
MNOP     || R0 = [I0++]    || R2 = [I1++];
LSETUP (loop_start_end, loop_start_end) LC0 = P0;
loop_start_end:
   SAA(R1:0, R3:2)  || R1 = [I0++]    || R3 = [I1++];

R4 = A1.L + A1.H, R5 = A0.L + A0.H;
R7 = R4 + R5;
[I2++] = R7;
```

表 5.11 不使用 SAA 的 Assembly code

Assembly code – without SAA, Byte Allocation

```
PO = 4;
MNOP     || R0 = [I0++]    || R1 = [I1++];
LSETUP (loop_start, loop_end) LCO = P0;
loop_start:
   (R3, R2) = BYTEUNPACK R1:0;
   (R5, R4) = BYTEUNPACK R1:0 (R);
   R0 = R2 -|- R4;
   R1 = R3 -|- R5;
   R0 = ABS R0 (V);
   R1 = ABS R1 (V);

   R5 = R0 +|+ R1 (V);
   R5.L = R5.L + R5.H (S);
   R6 = R5.L (X);
loop_end:
   R7 = R7 + R6 (S)  || R0 = [I0++]    || R1 = [I1++];

[I2++] = R7;
```

2.3 使用 Video Pixel 指令的需求

表 5.12 使用 Video Pixel 指令的需求

	Video Op Module	Starfish IP	Percentage
Area (gate count)	16310	231067	7.06%
Instruction Num	30	788	3.81%

Video Pixel Operation 指令所佔的面積如上表所述，佔總面積的 7.06%；指令共 30 個，佔全部指令集的 3.81%，所佔用的 op code 並不多。

面積上的 overhead 是顯而易見的，比較容易評估，還有另外一種 overhead 是屬於軟體方面的，就是這些指令是否容易使用；Video Pixel Operation 的指令因爲是設計來用在特定的函式，因此必須寫成 library 來使用之，而不同的 video standard，所需要的 library 可能也不盡相同，能否維護好所需要的 library 也是使用這類指令所需要考量的。

2.4 結論—關於使用 Video Pixel 指令

Video Pixel Operation 指令是專門用在 video operation 上，具有特殊用途，針對性較強，也因此受到的限制也較多；當 pixel、residual 的大小符合預期時，使用 Video Pixel Operation 指令會非常有效率，但反過來 pixel、residual 的大小有了變化，這些指令就不是那麼好用了。

整體而言，Video Pixel Operation 的指令是很有幫助的，但在把它們加入指令集之前要先考慮幾件事：

1. 所使用的應用程式是否需要這些指令？若今天我們只會用到 video decoder，那麼一些屬於 encoder 的指令就不需要加進來，例如 SAA，用在 motion estimation 方面的指令，就是只會在 encoder 上用到；若所使用的 decoder 因

爲 pixel 大小等其它因素，不能妥善利用這些指令，加進指令集中也只是耗費硬體資源。

2. 是否有其他指令能夠取代？以 Starfish 指令集爲例，因爲還有 vector 方面的指令可以使用，Video Pixel Operation 指令的重要性就沒那麼高了，但若今天不使用 Video Pixel Operation 就沒辦法有效率的執行 video 方面的函式，那這些指令應該要列入指令集中。
3. 使用 Video Pixel Operation 的 overhead。如前面所講的，除了考量面積方面的 overhead 之外，能否維護好所需要的 library 更是需要考慮的，否則還是無法使用到這些指令。

隨著無線網路便利性的提高，無線傳輸影像的機會越來越多，影像方面的處理也越重要，要能有效的處理影像，一些有特殊用途的指令也逐漸出現，除了上述的指令之外，關於 interpolation、de-blocking filter 等方面的指令也逐漸被加入指令集中，加入一些高效處理影像的指令，對 DSP 的指令集來說，是有必要的。

第三節 分析 MAC 指令的使用

在表 5.1 中可以看到，Starfish 指令集 Arithmetic 和 Vector 方面的指令使用率，僅高於 Video Pixel 方面的指令，原因在於 Arithmetic 和 Vector 中定義了許多 MAC (Multiply and Accumulate)方面的指令，而 MAC 指令的使用率卻很低，因此在這裡，將針對 MAC 方面的指令做分析，探討 MAC 爲何有那麼多指令，以及減少這些指令是否有意義。

3.1 MAC 指令的簡介

Starfish 是一個 DSP(Digital Signal Processor)，而爲了加速數學方面的運算，除了基本的加法、乘法等指令之外，還定義了許多乘加(Multiply and Accumulate, MAC)方面的指令；所謂的 MAC，是在乘法器之後再接一個加法器，使得 DSP 可以在一個 cycle 之內結束乘加的運算；Starfish 指令集中，MAC 的指令有許多選項(option)可供選擇，也因而定義了許多 MAC 指令；從表 5.13 中可以看到，MAC 方面的指令佔了 Arithmetic 及 Vector 指令的 62%左右，但只使用了 3%，因而使得 Arithmetic 和 Vector 指令的使用率低下。

表 5.13 H.264 中 MAC 及其它 Arithmetic/Vector 指令數與指令使用率

	Defined Instruction Num	Used Instruction Num	Usage
MAC instruction	272	7	2.57%
Without MAC Instr	168	40	23.81%
Total Arithmetic/Vector	440	47	10.68%

MAC 指令是對運算元先後進行乘法、加法的運算，再將結果依照指令的選項(option)做處理，選項大致上可以分成有無負數(sign/unsigned)、整數或小數(integer/fraction)、四捨五入進位或無條件捨去(round/truncation)等不同的運算方式，說明如表 5.14；在 H.264 的例子中，較常使用的有 IS、IU、FU 以及預設(default)的選項，這是因爲 H.264 的乘法運算，並不需要特殊的處理，諸如 truncation、scaling 等這類的選項都沒有使用到；H.264 中較常使用到 MAC 指令的是在運算 DCT(inverse Discrete Cosine Transform)時所做的量化(Quantization)，其它諸如 motion prediction、motion compensation 等計算影像的部份，大多是使用 shift 方面的指令，也因此在 H.264 中，MAC 指令的使用率並不高。

表 5.14 MAC 指令的各種選項

選項	選項說明
default	以 Sign Fraction 將運算元相乘，結果取四捨五入
FU	以 Unsigned Fraction 將運算元相乘，結果取四捨五入
IS	以 Sign Integer 將運算元相乘，取低位的 16 個 bit 作爲結果
IU	以 Unsigned Integer 將運算元相乘，取低位的 16 個 bit 作爲結果
T	以 Sign Fraction 將運算元相乘，結果無條件捨去
TFU	以 Unsigned Fraction 將運算元相乘，結果無條件捨去
S2RND	以 Sign Fraction 將運算元相乘，結果向左 shift 一位後四捨五入
ISS2	以 Sign Integer 將運算元相乘，結果向左 shift 一位後四捨五入
IH	以 Sign Integer 將運算元相乘，取高位的 16 個 bit 作爲結果
W32	以 Sign Fraction 將運算元相乘後，取 32-bit 的 saturation，結果四捨五入
M	Mix 模式，可以將兩個分別是 Sign 和 Unsign 的運算元相乘

3.2 指令集中加入 MAC 指令的成本

在指令集中加入任何指令時，都有其需求；每加入一組指令，便需要相關的硬體來執行運算，也需要 op code 來對指令做編碼，但 op code 是有限的，而面積越大生產成本也跟著增加，所以定義指令集時需要審慎評估是否需要這些指令，才能在執行效率與成本之間取得平衡。

3.2.1 硬體方面的成本

從硬體資源來看，MAC 指令需要乘法器、加法器，以及控制各類選項的硬體；從圖 4.1 上可以看到這些硬體資源的分布，其中 others 的部份包括 MAC 模組中使用的 non-combinational 電路、解碼器等其他基本的硬體資源，而由於控制選項的硬體(option control)只佔了一小部份，因此可以預期的是除了基本的乘加指令之外，新增各種選項的乘加指令所需耗費的硬體資源不多。

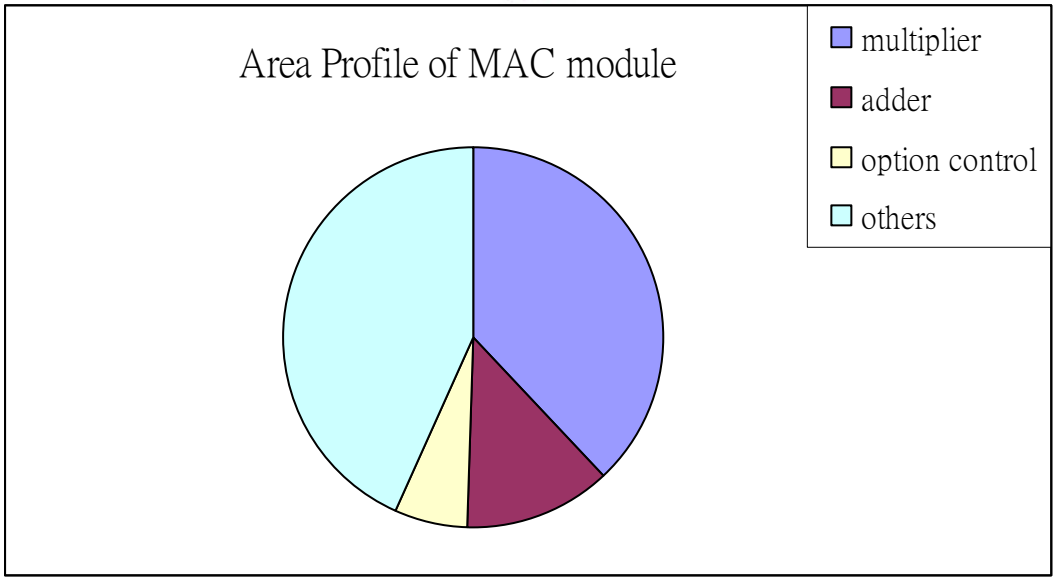


圖 5.4 MAC 模組的硬體資源分布圖

更細部的將各類指令做面積的累加表的話，可以得到表 5.15；因爲 M 選項都是和其他選項一起使用，故在不考慮 M 選項的情況下，將各類選項分成四群：

Basic: 基本的 option，包含 default、FU、IS、IU 四種 option。

Tru: 和 truncate 有關的 option，包括 T 和 TFU。

Scal2: 和 scaling 有關的 option，包括 ISS2 與 S2RND。

Others: 扣除上述所剩下的 option，有 IH 和 W32。

在表 5.15 中可以看到，若只使用 Basic 方面的指令，所使用的硬體面積是原本 MAC 模組的 96.5%，而加入 truncation 和 scaling 方面的指令，也才增加了 3%左右的面積，可以了解到 MAC 指令選項的多寡，影響的硬體面積不會超過原本 MAC 模組面積的 4%。

表 5.15 MAC 各類選項使用面積累進表

Option Type	Area
Basic	0.9650
Basic+Tru	0.9769
Basic+Tru+Scal2	0.9994
Basic+Tru+Scal2+Others	1

3.3.2 Op code 方面的成本

選項越多，就需要越多的 bit 來標注不同的選項，例如整數和小數的分別，就需要 1 bit 來區分；因此和硬體方面的成本比較起來，MAC 指令更需要考慮 op code 是否足夠控制如此多的選項，下列將依照各個選項，來列出 MAC 指令在編碼方面所需要付出的成本。

首先整數和小數、有無負數的區別，各需要 1 bit 來控制，而 truncation、scaling、IH/W32 和預設的 round 的區別，需要 2 bit 來控制，其中 IH 和 W32 因爲分別是整數和小數時特殊的選項，所以可以由其他 op code 來區分，在此可視爲同一組；M 選項也需要 1 bit 來控制，所以總共至少需要 5 bit 來控制所有的選項；MAC 指令總共 32 bit，扣除 5 bit 用來控制選項之外，其餘的 27 bit 需要控制哪兩個 register 相乘、結果存到哪個 register、是否是 vector 的指令等等需要控制的事項；在考量還有其它 32 bit 指令需要區隔之下，幾乎沒有可以保留的 op code，若要再增加選項，勢必會壓縮定義新指令的空間。

3.3 結論—關於 MAC 指令

在不壓縮定義其他 32 bit 指令空間的情況下，將 MAC 指令定義各類選項所增加的硬體成本不到原本 MAC 模組面積的 4%，所以雖然在 H.264 的應用上，使用 MAC 指令的機會並不高，但這不影響指令集定義如此多的指令；然而若需要增加 op code 來定義其它指令的話，是可以適當的減少某些 MAC 指令選項。

第六章

總結

在編譯器對指令的影響分析中，清楚的看到在編譯器中使用越多 library 時，指令執行的效率越高，然而這之中又產生許多問題：所編寫的 library 能否廣泛的使用在各種應用程式上？指令集所定義的指令能否讓使用者輕鬆的寫成 library？編寫 library 的成本受到應用程式以及指令集的不同而有所變化，但基本上編寫 library 所需的時間是可觀的，如何減少軟體方面的負擔也是值得研究的課題。

而在 Video Pixel 指令方面，可以了解依照應用程式的不同，所需要的指令格式也不一樣，當初在較早的影像格式如 MPEG 2 也許有所幫助的指令，在較新的影像格式如 H.264 中可能已不適用，或是需要新的改變；隨著應用程式的不同，所需要的指令也不一樣。在 MAC 指令的分析上，知道 MAC 指令在硬體及 op code 的成本有所不同，增加新的選項不會耗費多少硬體資源，而所消耗的 op code 則要看指令集是否還有空間可以提供，這給了使用者很大的彈性來定義與使用 MAC 指令。

參考文獻

[1] Spec of Sony Ericsson cell phone, July 2008

[2] ADI BF53X Instruction Set Reference, Rev 3.0, June 2004

[3] Yu-Chien Kao, “Development of A Main Profile H.264/AVC Encoder on A Multimedia SOC Platform”, *Thesis of National Tsing Hua University*, July 2005

[4] NTHU Design Technology Center, *Starfish Development and Implementation*, July 2007

[5] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVTG050, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 ISO/IEC 14496 10 AVC), March 2003

[6] Forward Concept, *Wireless/DSP Market Bulletin*, Feb 4, 2008 <http://www.fwdconcepts.com/DSPBulletin_2408.pdf>

[7] Berkeley Design Technology, *Pocket Guide to Processing Engines for DSP*, Oct 2007 <http://www.bdti.com/pocket/pocket.htm>

[8] Berkeley Design Technology, A company to insight, analysis, and advice on Signal Processing Technology <http://www.bdti.com/index.html>

[9] Yo-Ray Lee, “Instruction Set Extension for Deblocking Filter”, *Thesis of National Tsing Hua University*, July 2006

[10] Yu-Ru Yang, “Instruction Set Extension for Interpolation”, *Thesis of National Tsing Hua University*, July 2006

[11] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi, “Video coding with H.264/AVC: tools, performance, and complexity”, *Circuits and Systems Magazine, IEEE*, pp. 7-28, 2004

[12] J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron and M. Ware, “Trends in compilable DSP architecture”, *IEEE Workshop on Signal Processing Systems*, pp. 181-199, 2000

[13] K.H. Bang, N.H. Jeong, J.S. Kim, Y.C. Park and D.H. Youn, “Design and VLSI implementation of a digital audio-specific DSP core for MP3/AAC”, *IEEE Transactions on Consumer Electronics*, pp. 790-795, 2002

[14] H. Fujiwara, M.L. Liou, M.T. Sun, K.M. Yang, M. Maruyama, K. Shomura and K. Ohyama, “An all-ASIC implementation of a low bit-rate video codec”, *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 123-134, 1992