設計與評估一個 X86 複雜指令與精簡指令的轉換架構
Design and Evaluation of an X86 CISC/RISC Translation Architecture

林經鴻
Lin Gin-Hong
交通大學資訊工程學系
Department of Computer Science and Information Engineering
National Chiao-Tung University, Hsin-Chu
ginhin@csie.nctu.edu.tw

陳昌居
Chen Chang-Jiu
交通大學資訊工程學系
Department of Computer Science and Information Engineering
National Chiao-Tung University, Hsin-Chu
cjchen@csie.nctu.edu.tw

摘 要

本文提出一種設計解碼器的架構，此種架構包含解碼與轉換 X86 複雜指令集成爲類似精簡指令集。使用這種解碼器的架構可增加處理器在 X86 指令上的平行度。此解碼器組成的內容包括預提起、預解碼、預解碼指令緩衝器與包含監視性執行指令的轉換器。

關鍵字：精簡指令集,複雜指令集,預提起,預解碼,監視性執行

Abstract

This paper proposes a method to design the decoder architecture, for decoding and translating X86 instructions to RISC-like instructions. The decoder architecture augments X86 instruction level parallelism to the processor. This decoder contains a prefetcher, a predecoder, a predecoded instruction buffer and instruction converters with guarded-execution instructions.

Keywords:RISC, CISC, prefetch, predecode, guarded execution

一.前言

對於執行 X86 複雜指令集的微處理器，我們相信將會以內部執行類似精簡指令的指令集，而外部依舊提取 X86 複雜指令的混合形式繼續發展，在此種混合(Hybrids)微處理器架構中，令人關注的焦點，則在於如何有效的將 X86 複雜指令轉換成爲適合一般性 RISC 核心架構，或者更先進 RISC 架構可執行的指令。由於 X86 指令長度不一，須循序一一將指令做分割，劃出邊界後再解碼，使得同一時脈中得以被解碼且發送到執行單元的指令數目受限制，爲加快此解碼階段的速度，在本文中將設計一預先解碼的機制，使得接下來 X86 指令在轉換成所謂類似精簡指令(RISC-like)時得到分工。

當程式發生有條件式的分支指令，且此條件受前面運算指令尚未完成的限制，沒辦法事先得知真正的控制流程時，多數的微處理器架構是採用分支預測的處理方法。本文將嘗試在 CISC/RISC 的解碼與轉換過程中，先一步處理所遇及的分支指令，此分支指令不單純只作解碼與轉換的工作，可爲預先處理帶來更多的好處，且依不同的分支形式某些區塊的指令可被轉換成另一種帶有條件式的執行指令，期望能得到較順暢的執行流程。

解碼單元在整個微處理器架構中所扮演的角色有其重要性。從不少文獻中[1,3,10,14]了解到同爲執行 X86 複雜指令的微處理器，因其內部都有不盡相同的解碼架構，所以使用不同的方法就會得到不同的效益。現於吾人微處理器架構研究中，將以改良解碼器的架構爲重心，希望作爲專門轉換兩類的指令外，能增加解碼器其他附屬的功能，進而對於整個執行效能有所助益。另外關於解碼單元中的細部架構之取捨，除了有無的分析外，也將以量的分析提出足以影響特性的因素，以作爲對於整個解碼單元的評估依據。

下面的文章中，包括設計的介紹，將提出吾人的做法。接著是對設計的架構做評估，如何作測試，有哪些可比較出特性上的依據。此結果數據將提供作分析參考。最後爲總結說明。

二.轉換架構之設計

在轉換 X86 複雜指令成爲類似精簡指令的架構上，在此將其分兩個主要的部份來進行設計。前面爲預提起(Prefetch)與預解碼(predecode)單元，配合指令緩衝器(Instruction buffer)儲存預解碼過的指令。接著就是可將一 X86 指令解碼成一個或數個類似精簡指令的轉換(Translate)單元。圖 2.1 爲針對 X86 指令執行過程中，吾人所提出之從提取到解碼 X86 指令的結構方塊圖。前半段爲依序處理的預提起與預解碼單元，後半段則是將 X86 複雜指令轉換成類似精簡指令的操作。在此，吾人將比照 Intel Pentium Pro，將其稱之爲微操作(Micro-operation, uop)。

預提起預解碼單元

爲了使得 X86 複雜指令所組成的程式碼得到數量多、且較快速的提起並解碼，茲採用 AMD K5、K6 與 Digital 21264[14,18]的解碼機制，在指令快取上增加預提起與預解碼單元，吾人把從指令快取上送來的指令流總寬度設定爲 128 bits(16 bytes)，此即是在一次預提起最多被提起的位元數量。因爲在 X86 複雜指令集中，其指令長度最長爲 15 位元組，若預解碼單元想要一次至少解碼一個 X86 複雜指令，則需要傳送如此

長度的指令碼到預提起佇列中，等待預解碼器解碼。至於預提起的指令位址將會參考提起單元或者分支預測單元所送來的資訊。

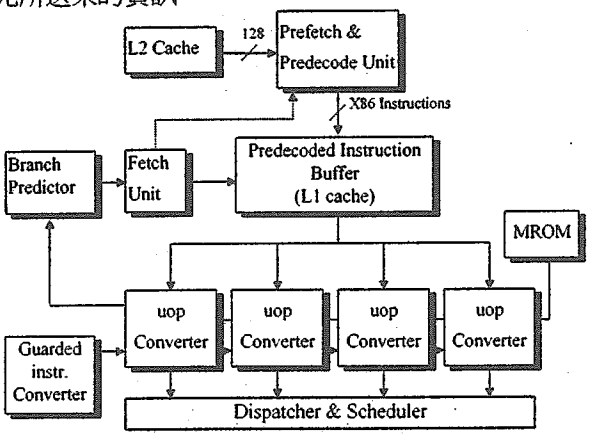


圖 2.1 轉換 X86 指令成為精簡指令的架構方塊圖

預解碼指令緩衝器

預解碼後輸出將可得到每一個 X86 指令的大小，與此指令粗略的運算碼與相關訊息。因爲被分離出來的運算碼大多包含其他運算操作的資訊，所以須進一步結合其他資訊作解碼，故吾人稱之爲粗略的運算碼。在一個預解碼指令緩衝器單元上儲存指令的格式，如圖 2.2 中所示。

Address tag	Prefix bits	Rough Opcode	Mod (reg) R/M	s-i-b	Address Displacement	Immediate Data
1 byte		2 bytes	1 byte	1 byte	4 bytes	4 bytes

圖 2.2 預解碼後存放在指令緩衝器內的 X86 指令形式

一個被預解碼處理過後的 X86 指令，儲存於緩衝器上，可以擁有大小同爲 13 位元組的長度，而指令長度的一致將有助於下一步轉換成 uop 的工作。由於當 X86 指令爲最短的一個位元時，也會被預解碼器解碼而存入此緩衝器中，而緩衝器對於這最短的 X86 指令也須花費 13 位元組儲存，即令實際有用的只有前面 3 個位元組。反之，當有一最長的 X86 指令(15 位元組)，經解碼器解碼後，亦可存放於緩衝器的 13 位元組中，將節省 2 個位元組存放在緩衝器內。所以此結構若用於預解碼器後的緩衝器，對於程式碼帶有多數短 X86 指令的程式碼而言，將造成較多硬體線路的浪費。

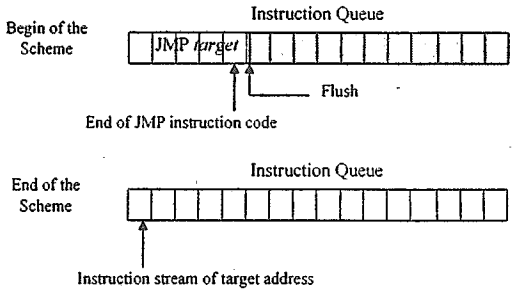
從預提起的指令佇列中來作預解碼的工作，必須一個接著一個位元組依序作預解碼。由於簡單的指令在單一個位元組即可解碼得知，而較長的指令卻必須歷經 4 個步驟的分析才可完成預解碼的工作，所以在此吾人設定之預解碼的架構，至少必須具有 4 階段指令碼的分析能力。當然此 4 階段可能預解碼出單一的 X86 指令，或者各自預解碼出 4 個單一指令。此處的預解碼能力與其後面連接的轉換單元的轉換能力完全不同。

預解碼對分支指令的策略

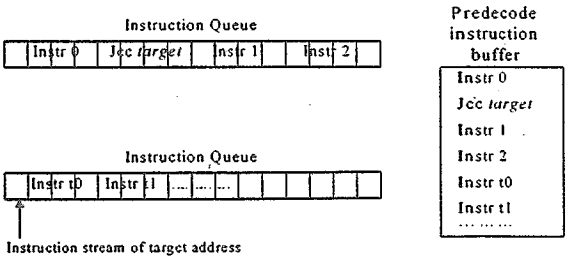
在此吾人將配合預提起與預解碼的設計，使執行碼中包含有分支時，得以提前一步作處理，但此處並不採一般使用複雜線路的作法，而是改以一些最簡單的規則代替，但其後仍然保留一般性的分支預測單元，作較爲精細的分析。如此看來，吾人設計的架構中將多了一次分支判斷的能力。

下面文章中就決策規則的內容作說明，首先，對無條件分支指令所採取的因應規則，在於立即停止預解碼器對下一指令的預解碼工作，直接指示預提起器到目的位址，提起包含這位址的一指令佇列之指令來，之後再繼續作解碼的工作。若在之前的預解碼指令緩衝器存有充足的 X86 指令，則將不因預提起與預解碼單元的臨時停頓而受影響，如圖 2.3(a)所示。

再來，對於有條件分支指令發生時，其後緊接著被預解碼單元取出的位址偏移量，也是判斷如何處置的依據。位址偏移量的正負值，表示分支的目的位址在目前指令所在位置的後面或前面，但在真正條件尚未決定之前，條件分支的目的位址都只是作爲參考而已，所以同爲一指令佇列其後的指令碼，若尚未接受預解碼的，將繼續作預解碼分析，直到此一佇列與解碼完成爲止，如圖 2.3(b)中的情形。緊接著以不同的目的位址偏移量與方向，來作不同的因應之道。如遇到偏移量甚小的條件分支指令時，不管其分支的方向爲向前，抑或是向後，都不作預提起與預解碼指令佇列的更動。若是一向後的分支將由吾人設計的指令轉換器來作進一步的監視性執行指令的轉換，使得執行微操作指令碼時更爲順暢。雖然對於偏移量小的條件分支指令在此不作預提起的工作，但爲了通知接下來的轉換單元，讓它知道有些分支指令接著將作監視性執行指令的轉換，在預解碼時會將此需求附加在預解碼指令當中。



(a) 預提起與預解碼單元對無條件分支指令的策略



(b) 有條件分支指令在長距離目的分支位址的策略

圖 2.3 預提起與預解碼單元對分支指令的策略轉換器單元

當前一單元送來的有條件分支指令中，一類具有順向且短距離分支指令將在此轉換單元作特別的轉換，這些指令在預解碼時會被特別標示，以作爲區別。在作監視性執行指令的轉換時，受監視的條件與狀況，以及開始轉換的位址與結束位址，都將透過一專屬的線路負責。

條件分支指令轉換成監視性執行指令

分支指令中，以條件分支指令的發生頻率最高[20]，所以在前面的預提起與預解碼單元已經對條件分支指令作適當的處理，另外將條件分支指令中具有順向且短距離分支的指令特別標示，以提供在轉換單元作監視性執行指令的轉換處理。在此將一般程式碼轉換成具有監視性執行指令的程式碼，使其基本區塊增大，但並不考慮迴路帶來的條件分支指令，因爲在轉換每個 X86 指令時並不會向前參考，而只是一種更改發生條件分支指令後的轉換指令的形式。吾人採取的架構中，還是保有分支預測的機制，但將針對向前發生條件分支，且分支距離比較短的這一類，作監視性執行指令的轉換。

在前面預提起與預解碼單元中，就曾經對於分支指令作不同的分類處理，此目的除了使預提起與預解碼單元事先對於程式分支作較理想的規劃外，也作爲通知轉換器是否將一條件分支指令以及其後的相關指令轉成監視性執行的指令。其通知的方式爲在預解碼過的指令格式上添加訊息，再存入預解碼指令緩衝器內。

爲了正確與有效地作監視性執行指令的轉換，在吾人設計的 4 個轉換器之外，再增加一專門負責此項監視性執行指令轉換工作的線路。此線路包含：監視性執行指令開始的形成，與通知各轉換器作監視性執行指令的轉換，以及結束此種轉換還原成一般 X86 指令轉換的工作。在圖 2.4 中將此專門處理監視性執行指令的轉換器與一般轉換器之關係以簡單的信號線表示。信號中，首先是來自一般轉換器，4 個一般轉換器中，某一轉換器檢測出一條件分支指令與緊接著一大小已知的區塊需要作監視性執行指令轉換，此通知監視性執行指令轉換器作監督，監督的過程包括把條件(X86 微處理器架構中的旗號)、狀態送到各個一般轉換器上，且檢查目前被轉換的指令位址，是否結束監視性執行指令的轉換。

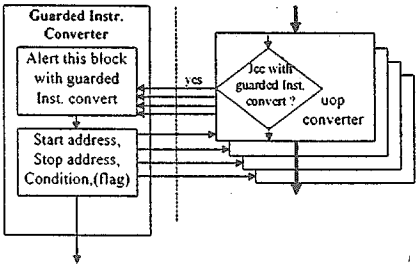


圖 2.4 監視性執行指令轉換器與一般指令轉換器的關係

三.系統模擬

整個模擬的工作主要可分成三方面進行，首先爲模擬環境的建立，包括分析 X86 指令集以建立資料庫與撰寫 X86 指令分析與解碼程式。再來則是取得有效的程式當作輸入，此輸入的程式爲一連串 X86 指令碼，以實際的應用程式爲主。最後是系統模擬，在此將設定相關環境與模擬的步驟，找出相同與相異的特性。

模擬環境的建立

一連串的 X86 指令碼將作爲測試解碼架構的依據，而建立 X86 指令集的資料庫則需要花費不少工夫。建構此資料庫主要是以操作碼爲索引，同一種操作中會有不同的運算元定址模式，將以尋求有規則的資料結構來對映，另一項與轉換有關的資訊爲此操作會被轉成幾個 uop 的基本數量。如圖 3.1 所示。

部分的條件分支指令其隨後指令將會被轉成監視性執行指令，即是在 uop 指令碼中加上條件的監視。在此也用上 Pentium Pro 新的指令 ccMOV 來取代監視性執行指令的轉換，比較兩者將提供給設計轉換單元上的一些建議。圖 3.2 顯示整個模擬系統的架構。

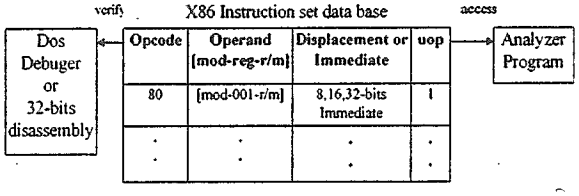


圖 3.1 X86 指令集資料庫管理

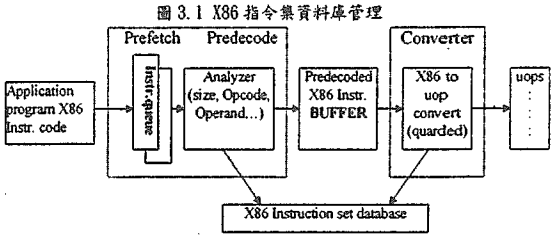


圖 3.2 系統模擬的主要架構

受測試程式

在本論文中計畫利用以下的程式指令碼，進行多次模擬：DOS 下 mem.exe 程式，find.exe 程式，setver.exe 程式，Windows 95 下 netterm 程式，excel 程式，winplay 程式。在蒐集 DOS 下的程式執行軌跡時，並不包含呼叫中斷處理，與重複性指令下的追蹤，因此所獲得的追蹤指令較短。而 Windows 95 的應用程式皆爲程式碼的片斷，也是原始程式的形式。上述中各個測試在篩選時，先作的指令碼分析如表 3.1 爲平均指令長度。

		Average of X86 inst. length
DOS	mem.exe	1.923
	find.exe	1.885
	Setver.exe	2.157
Windows 95	netterm	3.507
	excel	3.440
	winplay	3.064

表 3.1 測試程式指令碼的平均長度

模擬相關的事項

被預解碼後的 X86 指令就存放在預解碼指令緩衝器內，此緩衝器除了再被分割成兩段，以作爲多路徑分支預測外，太大的空間將造成資源的浪費。相反地，若取用太小將失去緩衝的效果，因此緩衝器的數量被在模擬之時被暫定爲 1024 個，而再以 512 個爲一組分開使用，以達成基本預解碼到指令轉換間的緩衝目的。

接下來爲 X86 指令轉換單元，4 個相同轉換器可同時解碼 4 個置於緩衝器上的指令，這些指令中包含有部分指令將作監視性執行指令的轉換。轉換器的輸出爲類似精簡指令的 uop，之後這些 uop 將到達一超純量執行核心，有暫存器更名(Register renaming)的動作會消除運算元中某些具有資料的相關，使得 uop 指令可不按次序執行。

在模擬時將以上述的環境與條件爲基礎，以分開幾個步驟來模擬，每個步驟中會有不同的測量數據將提供分析參考用：

步驟 A，首先對預解碼單元作模擬，將受測試的程式視爲一連串的指令碼，不管其中是否有任何的分支路徑，就是把指令碼大小與相關的資訊分解開來。在此將計算出單一個預解碼週期內可獲得的預解碼指令輸出數量。

步驟 B，預定測量預提起與預解碼單元在應付分支指令時以不同的對策其相對的結果變化情形。兩種形式將作爲比較的對象，分別爲一般的方法，與附有監視性執行指令轉換機制的預解碼策略。

步驟 C，再延續步驟 B 中對第二種方法的模擬，但此將選擇不同的分支距離做標準，距離的變化從 4 到 12 爲止。

步驟 D，作轉換單元的模擬，接著預解碼後的工作爲轉換 X86 指令爲 uop 指令，由前面預解碼送過來的指令將可同時送到轉換器中，。同時，將以 Intel Pentium Pro(P6)與 AMD K5 解碼單元爲比較的對象。

最後，步驟 E，模擬了 P6、K5 原先的解碼轉換架構下，與其經過改良後的架構，以及吾人所設計的解碼與轉換的架構，若以同樣的執行速度，計算平均每一週期的平均輸出。以及對預解碼指令緩衝器的空間使用情形，將以實際的 6 個程式碼作模擬，對於使用率的表現。

四.模擬結果與分析

模擬結果

步驟 A 的結果

在基本預提起與預解碼單元架構中，以一連串 X86 指令碼形式當輸入，在單一預解碼週期內平均預解碼數量。其中以最少的預解碼步驟 S4 爲模擬的底限，當單一預解碼週期可解得的位元組數量增加時，相對的預解碼 X86 指令的平均數量越多，S16 爲整個指令佇列同時當作一次預解碼週期。

Applic.program	netterm	excel	wplay3	mem	find	setver
S4	1.349	1.284	1.396	1.944	2.035	1.778
S8	2.455	2.458	2.715	3.789	4.014	3.544
S9	2.695	2.742	2.970	4.163	4.240	3.704
S10	3.049	2.991	3.372	4.163	4.247	3.757
S11	3.310	3.241	3.575	4.163	4.251	3.838
S16	4.561	4.602	5.257	7.610	8.041	7.185

表 4.1 預解碼對連續指令碼的分析

在表 4.1 中的內容爲預解碼平均輸出量，此數量會因爲預解碼一次所解得的位元組數增加而增加。但預解碼的線路一次增加一個位元組的解碼能力，相對獲得的平均輸出量並沒有那麼明顯，甚至某些時候沒有任何獲益，如表中以 mem.exe 這個測試程式碼預解碼平均輸出的變化來看，從 S9 增加到 S11 的能力，其平均輸出都爲 4.163 個。

步驟 B 的結果

以步驟 B 的形式進行模擬，預解碼週期中一次可對 4 到 16 個位元組作預解碼工作，表 4.2 中爲其輸出 X86 指令預解碼後的數量平均值，與前一表格的內容差別在於增加了對分支指令在預提起與預解碼上的影響，由於分支指令被預解碼時將會影響下一預解碼週期的行爲。

Applic.program	Netterm	excel	wplay3	mem	find	setver
S4	1.322	1.245	1.386	1.932	1.928	1.741
S8	2.299	2.196	2.631	3.612	3.246	3.234
S9	2.435	2.382	2.871	3.884	3.359	3.362
S10	2.656	2.507	3.059	3.934	3.696	3.492
S11	2.773	2.625	3.225	3.984	3.766	3.620
S16	3.365	3.220	4.413	6.585	4.721	5.457

表 4.2 預提起對預解碼的影響

步驟 C 的結果

表 4.3 到表 4.5 爲預解碼單元解碼到不同距離範圍的短距離條件分支指令時採取特別提起策略，而影響到預提起的次數，其變化情形。由於此種短距離的設定關係到預解碼緩衝器的大小與順向分支指令作監視性執行指令的轉換能力，在此設定從距離小於等於4開始測試，到距離小於等於 16 爲止，表中將以此種短距離條件分支指令在所有條件分支的比例與採用此策略後降低預提起次數的比例作詮釋。

D≦4	Condition branch with short distance jump	Prefetch count reduction
Netterm	0%	0%
Excel	0.53%	-0.11%
Wplay3	2.70%	-4.61%
Mem	38.68%	-18.21%
Find	26.88%	-11.31%
Setver	0.09%	-3.07%

表 4.3 短距離條件分支指令與預提起策略的影響 (d≦4)

d≦8	Condition branch with short distance jump	Prefetch count reduction
netterm	7.5%	-1.35%
excel	5.34%	-1.13%
wplay3	69.59%	-16.40%
mem	58.02%	-27.32%
find	29.67%	-12.48%
setver	30.02%	-9.95%

表 4.4 短距離條件分支指令與預提起策略的影響 (d≦8)

d≦16	Condition branch with short distance jump	Prefetch count reduction
netterm	14.48%	-2.21%
excel	11.22%	-2.16%
wplay3	85.13%	-20.06%
mem	58.02%	-27.32%
find	71.86%	-30.177%
setver	48.78%	-16.10%

表 4.5 短距離條件分支指令與預提起策略的影響 (d≦16)

步驟 D 的結果

見表 4.6，P6 具有三個被排列成 G-S-S(一般-簡單-簡單)形式的解碼器單元以對 X86 指令碼作解碼的工作，第三欄爲其平均解碼指令的數量。P6+ 的架構同樣具有 G-S-S 形式的三個轉換器，但在轉換前對分支指令作預提起與預解碼的先前工作，使得轉換器在遇上分支指令時，可以不用理會下一個應被提起的指令碼在何處。此外 P6 轉換器也被設計可作監視性執行指令的轉換能力。

	P6(G-S-S)		P6+	
	instr/time	Average	instr/time	average
netterm	2016/1203	1.675	2016/1009	1.998
excel	2016/1356	1.486	2016/1259	1.601
wplay3	2145/1171	1.831	2145/1144	1.875
mem	1857/859	2.161	1857/763	2.433
find	8210/5643	1.454	8210/4411	1.861
setver	4764/2919	1.632	4764/2751	1.731

表 4.6 P6 的架構中 X86 指令被轉換成 uop 的數量

見表 4.7，K5 擁有 4 個相同的轉換器，除了一部份較複雜的 X86 指令會被轉成超過 4 個以上的微指令由 MROM 負責外，多數都由這 4 個轉換器共同合作，將排列在此轉換器前的 X86 指令轉成相對的微指令。在第三欄位內即是此架構下對於這 6 個應用程式的 X86 指令碼被轉成微指令的平均數量。K5+ 的架構一樣具 4 個轉換器，但在轉換前對分支指令作預提起與預解碼的先前工作，使得轉換器在遇上分支指令時，可以不用理會下一個應被提起的指令碼在何處。此外 K5 轉換器也被設計可作監視性執行指令的轉換能力。

	K5		K5+	
	instr/time	average	instr/time	average
Netterm	2016/1303	1.547	2016/1207	1.670
Excel	2016/1337	1.507	2016/1248	1.615
Wplay3	2145/1014	2.115	2145/931	2.303
Mem	1857/680	2.730	1857/613	3.029
Find	8210/4933	1.644	8210/4005	2.049
Setver	4764/2593	1.837	4764/2311	2.061

表 4.7 K5 的架構中 X86 指令被轉換成 uop 的數量

見表 4.8，在吾人的架構上有 4 個單獨的 X86 指令轉換器，同時前一階段送來的 X86 指令已經是預先解碼過，此還包括預提起機制將分支指令作提起目的碼的預測，而對於順向短距離的條件分支指令則交由轉換器作監視性執行指令的轉換行爲。

	Designed method	
	Instr/time	average
netterm	2016/548	3.678
excel	2016/643	3.135
wplay3	2145/573	3.743
mem	1857/530	3.503
Find	8210/2514	3.265
Setver	4764/1745	2.730

表 4.8 具分支指令預提起與監視性執行指令轉換

步驟 E 的結果

見表 4.9，以五種轉換架構，轉換 X86 指令成 uop 輸出的數量。P6+、K5+爲改良原的架構，對於轉換輸出的 uop 數量也有明顯增加，最後一欄爲吾人設計的架構下轉換輸出的 uop 數量。

	P6	P6+	K5	K5+	Prefetch & Predecode + Translate
Netterm	3.413	4.070	3.151	3.233	7.494
Excel	3.238	3.487	3.284	3.518	6.828
Wplay3	2.830	2.896	3.268	3.559	5.783
Mem	2.880	3.242	3.638	4.035	4.667
Find	2.579	3.300	2.951	3.634	5.790
Setver	3.681	3.906	4.144	4.650	6.158
Average	3.103	3.484	3.406	3.772	6.120

表 4.9 單一週期轉換輸出的 uop 數

在預解碼與轉換器間的緩衝器被設計爲長度爲 13 位元組，在不同的應用程式下所表現的使用率都不盡相同。表 4.10 爲其 X86 指令平均長度，若以此存放在緩衝器，將平均浪費的空間。

	Average of X86 instruction length	Increased length in the buffer
Netterm	3.51	9.49
Excel	3.44	9.56
Wplay3	3.06	9.94
Mem	1.92	11.08
Find	1.885	11.115
Setver	2.158	10.842

表 4.10 預解碼指令在緩衝器增加的位元組數量。

五.結論

在採用上述將整個解碼單元分成預提起與預解碼單元以及轉換指令單元的架構之前，曾關注目前許多微處理器應用了較多的管線階級，之所以如此再分工相信必有其道理。吾人分析 X86 指令集時即發現，要解碼此種複雜的指令可將其分結爲：解碼分析指令的長度與大致的指令格式，再來作轉換成類似精簡指令的微操作指令格式。

在預提起與預解碼單元中，提前作分支指令的處置將有效分擔真正在作解碼與轉換指令時因爲分支路徑的轉變與目的指令碼從新提起等待造成的負擔，且以不必花費太多的線路來完成。對於程式指令碼中常常發生短距離的分支指令，對於時常須提起在快取記憶體中的指令碼，以及接著再從新解碼將是很費時不利的，此將以預解碼緩衝器與監視性執行指令的轉換來改善。但執行單元中仍然保有分支預測單元，這使得微處理器在執行控制相關的程式碼中得較少的障礙。就指令預提起的次數將因條件順向短距離分支指令佔所有條件分支指令的增加而減，轉換 X86 指令的數量也會應此增加許多。

預解碼在此主要的工作是解決連續性不等長 X86 指令的分解，在測試中以不同的預解碼寬度來求得較佳的設計值，通常預解碼寬度愈長，在預解碼所得的指令輸出愈多，到底所需要的解碼寬度有多大，在對轉換器的模擬中可得到明確的答案，當然在於真正執行時若此解碼寬度不夠則轉換單元必須等待預解碼的輸出；若所設計的預解碼寬度過長，則除了將預解碼的輸出暫時儲存在緩衝器內，若緩衝器已經飽和則只好等待有空的緩衝器使用再繼續預解碼的工作。上述兩種情況都是設計上的缺失，但也是因 X86 複雜指令與生俱來的特性。

參考文獻

[1].Linley Gwennap, "Klamath Extends P6 Family", Microprocessor Report, February 17,1997, pp. 6-8

[2]Brian Case, "New Instruction Sets Are Coming", Microprocessor Report, August 5, 1996, pp. 14-27

[3]Michael Slater, "AMD's K5 Designed to Outrun Pentium Four-Issue Out-of-Order Processor Is First Member of K86 Family" Microprocessor Report, October 24, 1994, pp.5-11

[4]Tom R. Halfhill, "AMD K6 Takes On Intel P6", Byte, January, 1996, pp. 67-72

[5]Linley Gwennap, "NexGen Enters Market with 66-MHz Nx586", Microprocessor Report, March 28, 1994

[6]Brian Case and Michael Slater, "Cyrix Joins X86 Fray with 386/486 Hybrid", Microprocessor Report, April 15, 1992

[7]Michael Slater, "Cyrix Doubles 6x86 Performance with M2", Microprocessor Report, October 28, 1996

[8]Zhendong Su and Min Zhou, "A Comparative Analysis of Branch Prediction Schemes", Computer Science Division, University of California at Berkeley, CA94720

[9]Linley Gwennap, "Intel's P6 Uses Decoupled Superscalar Design", Microprocessor Report, February 16, 1995, pp. 9-15

[10]Intel, "Pentium Pro Family Developer's Manual", Volume 2: Programmer's Reference Manual, 1996, chapter 11

[11]Cyrix, "6x86 Processor Superscalar, Superpipelined, Sixth-generation, x86 Compatible CPU", Technical report, March, 1996

[12]Mike Johnson, "Superscalar Microprocessor Design", Advanced Micro Devices, 1990, pp. 261-272

[13]Don Anderson and Tom Shanley, "Pentium Processor System Architectuhe", PC System Architecture Series Volume 5, pp. 1-4

[14]Michael Slater, "K6 to Boost AMD's Position in 1997", Microprocessor Report, October 28, 1996, pp. 26-27

[15]Wen-Bin Jian, "A Method of Improving Translating Performance in the CISC/RISC Hybrids", Master Thesis of the Institute of Computer Science and Information Engineer, National Chiao Tung University, June, 1996

[16]Dionisions N. Penevmatikatos, "Incorporating Guarded Execution into Existing Instruction Sets", A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy, Computer Sciences, University of Wisconsin-Madison 1996

[17]Brian Case, "ARM Architecture Offers High Code Density", Microprocessor Report, December 18, 1991

[18]Linley Gwennap, "Digital 21264 Sets New Standard", Microprocessor Report, October 28, 1996

[19]John L Hennessy and David A Patterson, "Computer Architecture A Quantitative Approach " Second Edition, Morgan Kaufmann, 1996, pp. 262-277

[20]Ing-Jer Huang and Tzu-Chin Peng and Jer-Ming Shiu, "Analysis of x86 Instruction Usage in DOS/Windows Applications and it's Implication Microarchitecture Design", Institute of Computer and Information Engineering, National Sun Yat-Sen University, 1996

[21]David L. Weaver and Tom Germond, "The SPARC Architecture Manual", Prentice-Hall, 1994, pp. 61-82