行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※
※　　　　　　　　　　　　　　　　　　　　　　　※
※　　具資料流引擎之 x86 微處理機設計(II)　　※
※　　　　　　　　　　　　　　　　　　　　　　　※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：☑個別型計畫　□整合型計畫

計畫編號：NSC89-2213-E-009-221

執行期間： 89 年 8 月 1 日至 90 年 7 月 31 日

計畫主持人：單智君 博士

共同主持人：鍾崇斌 博士

本成果報告包括以下應繳交之附件：

□赴國外出差或研習心得報告一份

□赴大陸地區出差或研習心得報告一份

□出席國際學術會議心得報告及發表之論文各一份

□國際合作研究計畫國外研究報告書一份

執行單位：國立交通大學資訊工程學系

中　華　民　國　90 年　10 月　29 日

具資料流引擎之 x86 微處理機設計(II)

Design of an x86 microprocessor with data-flow engine (II)

記畫編號：NSC89-2213-E-009-221

執行期限：89 **年** 8 **月** 1 **日**~90 **年** 7 **月** 31 **日**

主持人：單智君

共同主持人：鍾崇斌

計畫參與人員：邱日清、徐日明、劉嘉修、林育得

E-MAIL ：jjshann@csie.nctu.edu.tw

一、中文摘要

本計畫的目標在於完成具資料流引擎之 x86 微處理器（DF86）結構設計相關問題的研究與探討。我們擬於三年內研究以資料流引擎為核心的微架構，以模擬及實作驗證設計的可行性，並提出一個測試原型。第二年的研究方向則將延伸第一年的研究，進行 DF86 微架構的效率化的機構研究。

超純量微處理器是目前最被廣泛使用的高效能計算機架構。為了達到提高效能的目的，指令抓取率在此類微處理器的設計中是相當重要之一環。但由於複雜指令集（如：x86）的指令長度並不固定，使得指令抓取效率受到了嚴重的限制，進而限制了 CISC 超純量微處理器的效能。由於迴圈的特殊行為模式，在過去有許多的機制試圖透過迴圈指令之收集以提升指令之抓取效率，例如 Loop Cache 與 Loop Buffer 等。但是此類的技術對指令平行度的提升並無直接之好處。而針對迴圈指令平行度提升的技術中最重要的為 Loop Unrolling。此技術是透過編譯器之協助對迴圈內的指令重新排序以開發迴圈中的指令平行度，但對於指令之抓取效率則無幫助。因此，本期計劃提出了一個處理迴圈的新機制，運用資料流觀念達成迴圈中多個疊代並行執行之目的，稱為多疊代資料流執行機制 (Multi-iteration Dataflow Execution for Loops)。希望藉由動態地收集迴圈中的指令以減低 x86 指令抓取的困難，進而開發迴圈中指令之平行度。設計的想法主要是動態地將迴圈的控制資訊與迴圈本體分離，並將迴圈本體內的指令轉成資料流圖。之後，再利用迴圈的控制資訊安排迴圈中多個疊代以資料流的方式並行執行。根據模擬結果，此機制在並行三個疊代時，其執行迴圈所得到之指令平行度已相當於理想的超純量微處理器所能達到之飽和值。

關鍵詞：x86 指令集、微處理機、資料流架構、超純量架構、資料流圖、高頻寬資料存取

Abstract

The objective of this project is to study and design an x86 microprocessor with data-flow engine – DF86. We will design the micro-architecture of the microprocessor using a data-flow kernel, and verify the design by simulation and emulation. Finally, we will build a prototype using FPGAs. In the secondary year, we will extend approach our

project further to advance the DF86 performance.

Superscalar is the mainstream of the contemporary processors for improving performance. Fetching many instructions in a single cycle is an important ability of superscalar processors. However, in a CISC architecture, the variable length of instructions makes fetching multiple instructions in a cycle very difficult. The limited fetch rate may limit the performance of a CISC superscalar processor. Because of the special behavior of loops, many techniques have been proposed to increase the fetch rate for loops, such as Loop Cache and Loop Buffer. These techniques are proposed to save the instruction fetch energy, but not to exploit instruction parallelism. On the contrary, loop unrolling is a compiler technique that exploits the instruction parallelism by instruction scheduling, but cannot increase the fetch rate. In this project, we propose a mechanism, called *Multi-iteration Dataflow Execution for Loops* for loop. This mechanism is designed both to increase the fetch rate and to exploit the instruction parallelism for simple loops. It splits a loop into two parts, the loop control statement and the loop body, and translates the loop body into dataflow graph. Then, it tries to execute multiple iterations in parallel. According to our simulation results, the issue rate of simple loops achieves the ideal performance of superscalar processors when we execute three iterations in parallel.

二、緣由與目的

To achieve high performance, contemporary processors rely on exploiting ILP (instruction level parallelism). For superscalar processors, they exploit ILP through executing many instructions in a single cycle. However, the performance of a superscalar CISC (Complex Instruction Set Computing) processor is limited by the fetch bandwidth because of the variable length of x86 instructions. To fetch more x86 instructions in, many techniques have been proposed. For example, AMD series products implement instruction identification by using pre-decode information in the instruction cache [1][2]. However, the simulation result shows that the upper bound of the fetching rate is about 4.5 x86 instructions per cycle [3]. It is obvious that if the fetcher of a processor can only supplies 4 or 5 x86 instructions per cycle, the performance of the processor is also bounded to this number.

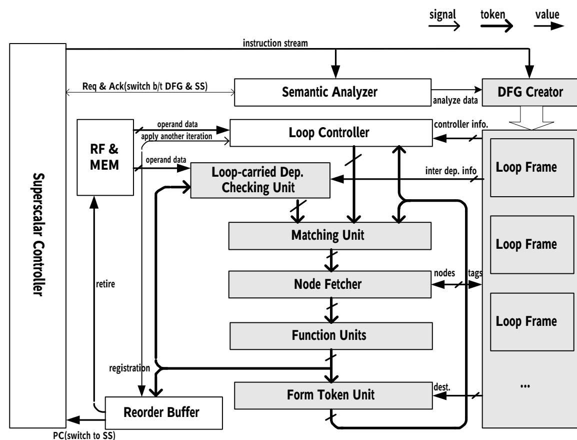
Although many techniques have been proposed to deal with loops, yet most of them did not consider both of the ILP exploiting and the fetch bandwidth increasing. For examples, loop unrolling is a compiler technique to exploit the ILP in loops, but it does not work to increase the fetch bandwidth; loop buffer and loop cache are hardware techniques that proposed to increase the fetch bandwidth for the instructions in loops, but the object of these techniques is not to exploit the ILP in programs. We propose a dataflow mechanism for an x86 processor. It is designed to improve the fetch bandwidth and to exploit the ILP for simple loops. A simple loop is a loop that has no function call and any other loop in it. There are two execution modes in our design, dataflow and superscalar execution mode. When a program is executed in superscalar mode, the proposed mechanism analyzes the loops dynamically and tries to translate simple loops into dataflow graphs. Once any simple loop is detected and the dataflow graph is created, it switches the superscalar mode to the dataflow mode until the end of the loop execution. We also store the created dataflow graphs, because loops may be re-executed. If we detect the entry of a simple loop, this loop can be executed in dataflow immediately.

三、結果與討論

In this report, we present the design of multi-iteration execution for loops in CISC programs by applying dataflow concept. After the extraction of a loop, we need a mechanism to check the validity of each iteration execution. Therefore, there is a Loop Controller to hold the information of the loop control statement and to determine if the execution of iterations is legal. Moreover, the controller also initiates and rolls the multi-iteration execution for loops. When loops are executed with the proposed mechanism, all needed instructions are from the dataflow graphs stored in Loop Frames.

Figure 3-1 shows the architecture of the processor that has two execution modes, superscalar and dataflow execution mode.

Figure 3-1 shows the flowchart for the



operation of this two-mode architecture.

Figure 3-2 shows the flowchart for the operation of this two-mode architecture.

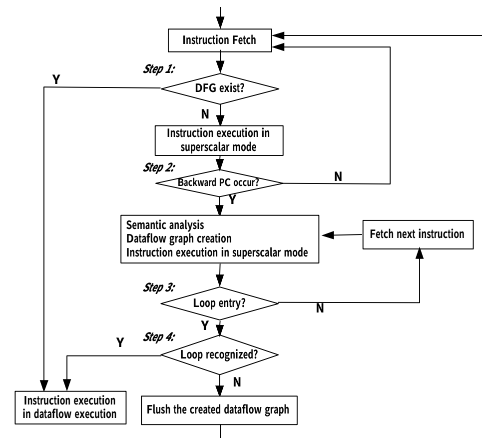


Figure 3- 2 Flowchart for the operation of the two-mode architecture.

Step 1: When instructions are fetched, we compare their program counter with the loop entries that stored in Loop Frames. This comparison determines if it is possible a loop that we stored before. If any matching pair exists, the dataflow graph is loaded and the system is switched to the dataflow execution for the loop. Otherwise, these instructions are still executed in the superscalar mode.

Step 2: After the instruction execution, if it generates a backward NPC (next program counter), Semantic Analyzer assumes that it is a loop entry and we start the dataflow graph creation.

Step 3: After the instruction execution, if the NPC is not the entry of the loop, it represents that the semantic analysis and the dataflow graph creation are not finished yet.

Step 4: If the condition of Step 3 is true and Semantic Analyzer recognizes it is a loop by semantic analyzing, the program will switch to the dataflow execution mode. Otherwise, it means that it is not a loop and the created dataflow graph is flushed.

From the simulation result, we know that if we apply the proposed mechanism on a scalar processor, the performance is worse than

that of an ideal superscalar processor. The main reason is that the percentage of the simple loops in the programs is not high enough. However, for the x86 sperscalar processors, it is very difficult to design a fetcher that can fetch three or more x86 instructions per cycle. To reduce the hardware complexity of the instructions fetcher, various fetching rules are defined in the contemporary x86 processors. Unfortunately, the fetching rule will reduce the fetching rate and the performance of the processors. Figure 3-4 shows the performance of some commercial x86 products, and none of their issue rates is greater than 1.5 instructions per cycle.

Figure 3-3 Performance simulation for the execution of the benchmark programs under a scalar processor with the proposed mechanism.

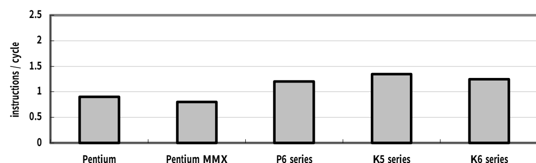


Figure 3-4 The performance of some commercial products.

Recall that, over 80% of the dynamically executed instructions of the benchmark program *compress* are within simple loops. From Figure 3-3, the performance of *compress* may exceed the ideal issue rate of superscalar processors easily. It shows that the proposed mechanism is good for some special applications, such as the mpeg 4 and the scientific computation, in which the percentages of loops in programs are high.

四、計畫成果自評

In this project, we will extend approach our project further to advance the DF86 performance. We have constructed a simulator to evaluate the parameters of our mechanisms, and have excellent results. These show that the DF86 microarchitecture is better than other current microprocessors'. This project is fully matched with the proposal requirements. There are still several researches could be further studied. First, according to the discussions above, we know that the compiler support can reduce the hardware cost and make the proposed mechanism more practical. For compilers, it is much easy to achieve the goals of Semantic Analyzer.

五、參考文獻

[1] AMD Corporation, *AMD-K6-III Processor Datasheet*, 1999

[2] AMD Corporation, *AMD Athlon Processor Technical Brief*, 1999

[3] Jih-Ching Chiu and Chung-Ping Chung, *High-Bandwidth X86 Instruction Fetching Based on Instruction Pointer Table,* IEE 2001

[4] Arthur H. Venn, *Dataflow Machine Architecture*, ACM Computing Surveys, Vol. 18, No. 4, December 1986

[5] Jurij Silc, Borut Robic and Theo Ungerer, *Processor Architecture*, Springer, 1999

[6] Arvind and David E. Culler, *Dataflow Architectures*, Annual Reviews in Computer Science, 1986

[7] J. R. Gurd, C. C. Kirkham, and I. Watson, *The Manchester Prototype Dataflow Computer*, ACM 1985

[8] Toshitsugu Yuba, Toshio Shimada, Kei Hiraki, and Hiroshi Kashiwagi, *SIGAMA-1: A Dataflow Computer for*

Scientific Computations, Computer Physics Communication 1985

[9] Jack W. Davidson and Sanjay Jinturkar, *Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation*, IEEE 1995

[10] J. W. Davidson and S. Jinturkar, *Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler*, Proceeding of Compiler Construction Conference 1996

[11] D. A. Patterson and J. L. Hennessy, *Computer: Architecture A Quantitative Approach,* 2th Ed., Morgan Kaufmann, 1996.

[12] Lea Hwang Lee, Bill Moyer, and John Arends, *Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops*, ACM 1999

[13] Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis, *Energy and Performance Improvements in Microprocessor Design using a Loop Cache*, IEEE 1999

[14] J. E. Thornton, *Design of a Computer: the Control Data 6600*, Glenview, 1970

[15] J. E. Thornton, *Parallel Operation in the Control Data 6600*, Proceeding of the Fall Joint Computers Conference, 1961

[16] Bryan Black, Bohuslav Rychlik, and John Paul Shen, *The Block-based Trace Cache*, IEEE 1999

[17] Eric Rotenberg, Steve Bennett, and James E. Smith, *A Trace Cache Microarchitecture and Evalution*, IEEE 1999

[18] Matt Postiff, Gary Tyson, and Trevor Mudge, *Performance Limits of Trace Caches*, IEEE 1998

[19] Intel Corporation, *The Microarchitecture of Pentium 4 Processor*, Intel Technology Journal 2001

[20] Davis, A. L., *A Data Flow Evaluation System Based on the Concept of Recursive Locality*, Proceedings of National Computing Conference 1979

[21] Arvind and David E. Culler, *Dataflow Architectures*, Annual Reviews in Computer Science, 1986

[22] Arvind and K. P. Gostelow, *The U-Interpreter*, Computer, February 1982