計算機アーキテクチャ 119−11
（1996. 8. 27）

Loop を並列実行するアーキテクチャ

玉造潤史　松本尚　平木敬

コンパイラによって並列化されていない逐次ループのバイナリ実行形式を動的に解析し、並列実行するアーキテクチャを提案する。プロセッサが並列化によってループ間依存が排除されていないループを動的に並列実行するにはループ間の依存を動的に解消する機構を持たねばならない。本アーキテクチャでは、レジスタの生成の依存関係を動的に解析することでメモリアクセスの依存が投機実行の可否を検出する。また、分岐における制御依存は、ループレベルの大きな構造の投機的実行によって並列実行する。本稿では、命令レベルの並列実行を行なう要素プロセッサを用いて、動的なループレベルの並列実行のために必要なレジスタの依存関係を解決する動的依存解析機構と、メモリアクセスと制御依存の複数の投機実行をElastic Barrier を用い資源を次々と解放することで、小さなプロセッサ資源で実現する多重投機実行機構を提案する。また、これらの機能を付加したパイロットモデル OCHA-Pro(On-CHip MIMD Architecture Processor) を述べ、実行性能のシミュレーション評価を示す。

An Architecture for Speculative Parallel Execution with Loop Analyzer

JUNJI TAMATSUKURI , TAKASHI MATSUMOTO and KEI HIRAKI

We propose a new architecture which dynamically analyzes a binary code of a preparallelized sequential loop and executes in parallel. To execute dynamicaly a sequential loop with depedencies among iterations in parallel , the architecture should have a dependency resolving mechanism in processor. In this architecture, the possibility of speculative memory access can be detected by analyzing register production dependencies. And on control dependencies on branch instructions, parallel execution is realized by loop-lebel speculative execution. In this paper, we propose the dynamic dependency analyzing mechanism needed by dynamic loop level parallel execution and the multiple speculative execution mechanism which realize some supeculative execution on memory accesses and control instructions without increasing processor resources which loop analyzer release by Elastic Barrier. We describe about our pilot model OCHA-Pro(On-CHip MIMD Architecture Processor) appending these mechanism and show an execution performance by simulation.

1. はじめに

現在、スーパースカラ [2] アーキテクチャが高性能マイクロプロセッサアーキテクチャの主流となっている。その理由は、スーパースカラアーキテクチャが従来のプロセッサとの命令セットレベルでのバイナリコンパチビリティを維持しながら命令レベルの並列性を抽出し、高速化することができるからである。スーパースカラアーキテクチャではその実現のためプロセッサ内の計算資源を並列に用いることを動的に実現する方式として score board [3] や Tomasulo アルゴリズム [4] が提案されてきた。これらのマイクロプロセッサの出現以前の計算機で用いられてきた技術と大容量の register file [5] [6] [7] や命令の並列実行数を増加させるために reorder buffer [8] を用い out of order 実行を行なうといったマイクロプロセッサ上で初めて実現された技術を融合させたプロセッサアーキテクチャが主流である。これは、常に進歩し続けるプロセス技術によって生み出された資源を既に確立されていた技術を実現するために用いることでマイクロプロセッサは高速化を続けてきた結果である。使用可能なチップ内資源はバイナリレベルのコンパチビリティを保つことに使われている。

しかし、これらの命令レベルの並列性を活用するアーキテクチャの限界は以前から多くの指摘がなされてきた。その一つに、reorder buffer は、fetch 可能な命令の window サイズを大きくとっても得られる並列性の限界 [9] を持っている。

また、並列プロセッサの on-chip 化 は、要素プロセッサ間の結合の高速性と大きな自由度を実現する。この高速で柔軟なプロセッサ間結合を活用するアーキテクチャとしてプログラム上の制御構造をプロセッサ間結合に mapping し、並列動作する Multiscalar [10] が提

† 東京大学 大学院 理学系研究科 情報科学専攻
Department of Information Science,Faculty of Science
University of Tokyo

案されている。また、従来、並列アーキテクチャとして基板上に実現されていた MultiThread を並列マルチプロセッサとして実現した M-Machine [11] などがその例である。

本稿では、on-chip 化による特性を活かした次世代の並列マイクロプロセッサアーキテクチャとして、要素プロセッサのアーキテクチャを選ばず、バイナリコンパチビリティを維持しながらループを動的に検出し並列実行することで高速化するアーキテクチャを提案する。特徴は、スーパースカラ、VLIW が局所的な並列動作および投機実行のアーキテクチャであるのに対して、要素プロセッサを構成単位とした MIMD 動作が可能であり、ループレベルでの投機実行 [12] をそれらのアーキテクチャと同様に dynamic 動作として実現している点である。

投機実行では、レジスタの生存期間が長くなり [13]、多くのアプリケーションは renaming 出来るレジスタ数が増大すると並列に動作できる [14] ことや、投機実行によって並列実行数が増加する [12] ことが以前から指摘されている。これらの性質を活かし投機実行の可否を解析する動的依存解析機構と renaming によって投機実行を行なう多重投機実行機構を提案する。

動的依存解析では、並列化していないバイナリで、並列化によって本来は解決されているべき依存を解決しなければならない。その依存中からメモリアクセスと制御命令の依存を検出し、投機実行の可否を解析する。

多重投機実行では、解析された投機実行可能なループの load と制御命令の実行を多重に投機的に行なうことを可能にする。投機実行開始時に、巻き戻しのために実行中の context を renaming によって保存する。多重化のために renaming table を用意し、他プロセッサの実行時に投機の成功が判明すると、同期的に全プロセッサの投機実行の context を解放する。この多重投機実行機構を Elastic Barrier [1] を用いて、多重でかつ少ないプロセッサ資源での投機実行機構として実現する。

本稿で述べる OCHA-Pro(On-CHip MIMD Architecture Processor) は上記アーキテクチャのパイロットモデルである。on-chip MIMD アーキテクチャとして多重投機実行機構を実現するのは各要素プロセッサが柔軟な投機実行を行なうにはそれぞれが独立して動作する必要があるからである。ループの並列動作に必要な機能とその実例を示し、投機実行機構を付加した共有メモリ型マイクロプロセッサパイロットモデル OCHA-Pro への実装法を述べ、例題を用いた性能評価を行なう。

2. 動的依存解析

逐次ループの動的並列実行のためには、バイナリコードからの

(1) ループの検出

(2) ループ内の投機実行要素・制御構造の解析

(3) ループの含むレジスタの依存の解析

が、必要である。提案する方式では投機実行要素の解析はレジスタの生成関係から解析する。したがってレジスタの解析が重要となる。

2.1 レジスタ解析

ループ内のレジスタの生成依存 [15] は次のように分類される。

flow 依存 ループ内で source レジスタとしてアクセスされる。iteration 間での通信となり、次々に依存を渡していく。

出力依存 ループ内で destination レジスタだけになっている。ループの実行終了後の実行に必要となる。

逆依存 ループ内で生成されたレジスタ。ループ内で用いられるので、依存とはならない。

レジスタの生成フローグラフの依存を、次のようにレジスタへのアクセスに関する関係から分かる依存として再定義する。

(1) **連接依存** 初め source operand になり、その後 destination operand になる。ループ間に生成関係を生じる依存。

(2) **終端依存** 初め destination operand としてアクセスされる。ループの並列実行の最後に必要となる依存。

(3) **全体依存** source operand としてだけ、アクセスされる。すべてのループが必要とする依存。

連接依存と全体依存はループ間の通信であり、ループ間でレジスタの内容を渡さなければならない。終端依存はループの実行結果であり、結果をループの並列実行の終了時に回収しなければならない。

2.2 投機実行要素の解析

ループ内で投機実行要素となるのは load と分岐である。load は投機実行されるが load したデータが次の iteration に引き渡される場合は並列実行ができない。検出には load したレジスタにマークをし、マークしたレジスタを source にして生成したレジスタにもマークする。マークとは別にレジスタ依存からこのマークのあるレジスタが連接依存となる時は doacross ループであり並列実行ができない。ループ内の分岐も条件によって生成されるレジスタがあり、投機実行要素となる。分岐の実行は分岐予測により行なわれるので、予測が外れた時は投機失敗となる。分岐条件の結果は loop analyzer が通知する。

レジスタの依存関係と投機実行要素の依存関係を命令のアクセスの履歴として生成する機構を loop analyzer に register analyzer として付加する。

3. Loop Analyzer

loop analyzer 内部は 2 個の機能ユニットから成っている。loop detector はループの検出を行ない、register analyzer は ループの検出後、ループ内の依存を動

的に解析する。register analyzer は並列実行中のレジスタの依存関係を renaming 機構に対して通知する。

loop analyzer は、レジスタが制御による生成依存を持たない場合、プロセッサではまず iteration の状態の生成に依存を持つ命令を実行し iteration の実行開始状態を作り出す。制御依存を持つことが register analyzer から通知された時は、命令の実行は (分岐) 予測に従って実行する。準備動作で、前 iteration によって作り出される状態を同一のプロセッサ上で行なうことにより iteration 間の通信をなくすことが可能となる。準備動作は、他の要素プロセッサで行なわれている前 iteration とオーバーラップして行なわれる。

以上の機能をもった loop analyzer を各要素プロセッサに付加する。

3.1 Loop detector

ループ検出器 (loop detector) は、実行中に最内ループを発見し、loop analyzer に通知する。

目的とするループ構造の loop body は、backward branch あるいは backward jump である。制御命令の実行時のアドレスと制御が移った先のアドレスが loop body の先頭と最後のアドレスになっている。つまり、アドレスが減少する方向への制御命令の制御の移る前のアドレスと制御が移る先のアドレスを記録しておく。判別された次の iteration の実行時に、同じアドレスで制御命令を起こしそのアドレスが記録してあるものと同じであればループの実行中であることを検出できる。したがって、ループの検出に 2 iteration 必要となる。求めるループの範囲は記録した二つのアドレスの間である。

以上の機能を持った buffer を一段用意することでループを検出する。

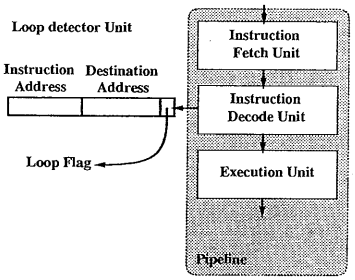

図 1 loop detector

3.2 Register analyzer

先に述べたレジスタの動的依存解析を実現する動的レジスタ依存解析機構 (register analyzer) の構成を下に示す。register analyzer はループ内の命令の source/destination field の decoding の結果からレジスタへのアクセスの仕方により振舞いを記録する。全ての命令セット上のレジスタに対して 6 ビットずつのエントリを持つ table である。それぞれのビットが、次の状況により書き込まれる。

bit	機能
s	初め source operand になる。s ビットを持つレジスタは、全体依存 である。
s → d	s ビットが立ってから、destination operand になる。s → d ビットを持つレジスタは、隣接依存となる。
d	初め destination operand になる。生成後、終端依存となる。ループの投機実行の際に、iteration 準備で生成する必要がない。
d → s	d ビットが立ってから、source operand になる。d → s ビットを持つレジスタは、iteration の実行のために生成する必要がある。
p	制御依存 (分岐の body 内にある) のため、生成されるかされないか分からない。
l	load されたデータのレジスタ load されたデータが iteration 間を跨る場合は並列実行不可

この解析結果に従って、loop analyzer は s ビットを持つレジスタは並列実行の初めに転送を行なう。d ビットを持つレジスタは、並列実行の終りに回収しそれぞれビットを持つレジスタは iteration 準備で生成を行ない、終ると並列実行に移る。

p ビットを持つレジスタは、そのレジスタの生成が投機的に行なわれることを示している。

l ビットを持つレジスタを source として用いて生成されたレジスタも l ビットを立てる。そのレジスタが s→d ビットを持つ場合は並列実行ができない。

reg	s	s→d	d→s	d	p	l
$r1	1	1				
$r2	1					
$r3				1		
$r4				1		1
⋮						
$r64				1		1

図 2 Register Analyzer

4. 多重投機実行機構

高集積のマイクロプロセッサでは命令セットで定義されている数よりも多数のレジスタを持っている。そのレジスタを用いての投機実行を可能とする投機実行機構を register renaming で実現する。投機実行の多重化のために register renaming table を用いる。renaming table には、現在のレジスタの renaming の状態を記録しておく。投機的な実行は renaming されたレジスタ上で行なわれる。新たな投機実行の前に renaming table を更新する。

この機能の実現には、大きなループでは、非常に大きなプロセッサ資源が必要とされる。多重投機実行機構では、Elastic Barrier を用いて次々と成功した投機実行

の資源を解放することで、資源をあまり大きくせずに実現する。

- 現在使用中のレジスタは書き込みを禁止して、destination として用いる時は renaming する。この動作によって context を保持する。
- 投機が失敗した場合、 renaming table を巻き戻す。この動作で投機実行前の状態に戻る。
- 制御依存命令の投機実行の成功が分かった時には Elastic Barrier の PREQ で投機成功情報を出してプロセッサは iteration の実行を続ける。
- iteration の実行が終ると Elastic Barrier で PREQ を出して投機に用いた資源の解放を通知し実行を続ける。

したがって、 table の数までの多重投機実行が可能である。投機実行が成功し必要が無くなった renaming table は、 flush (全てを消す) する。同期的な flush は Elastic Barrier により高速に起動される。

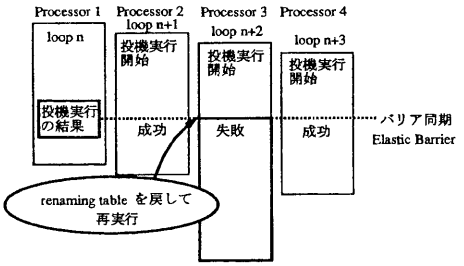

図 3　同期による投機実行の終了

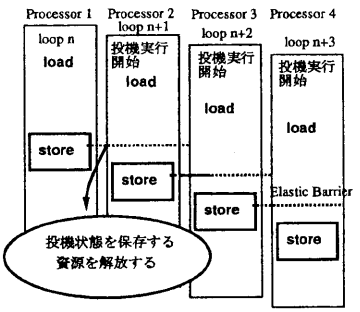

図 4　同期による資源解放

4.1 Renaming Table

renaming table には、どの命令セット上のレジスタが現在プロセッサ上のどのレジスタに割り当ててあるかが記録される。要素プロセッサ内の renaming 機構は、命令フィールドの decoding を table によって参照すべきレジスタと書き込みを行なうレジスタを判定して、必要があれば従来の資源のスケジュールとしての score board を用いて renaming を行なう。投機実行が始まると renaming table は更新され、以前の renaming table の context を保持するように renaming 動作する。 loop analyzer が資源解放を他の processor から Elastic Barrier で通知されると解放すべき renaming table を解放する。

5. メモリアクセスの依存解決

ループ間の依存としてメモリアクセスの順序が保たれなければならない。メモリアクセスを解析するためには、アドレスの生成を解析出来なければならない。アドレスの解析には、メモリアクセスのストライド (距離) から、解析することが可能 [16] である。 OCHA-Pro では、メモリアクセスの依存を解析し、メモリアクセスが後続 iteration の実行制御に影響を与えない場合に、投機的に実行を行なう。ループの同一 iteration でのメモリアクセスの順序は要素プロセッサ内で保証される。 iteration 間のメモリアクセスは、 cache coherent protocol の update protocol に類似の Memory access Ordering Protection protocol (MOP update) で依存関係を維持する。 load は実行されると load buffer 上の tag にループの番号を書き込む (図 5)。 tag は、逐次ループの順番 (index) を表す。

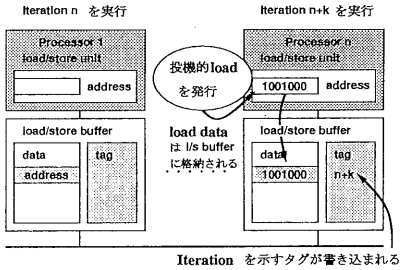

図 5　投機的 load の実行

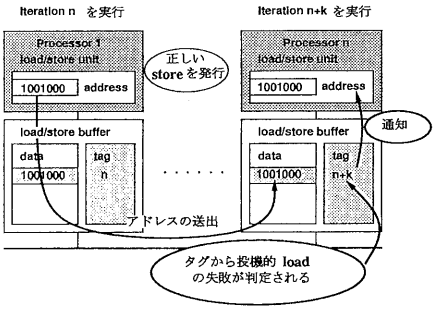

図 6　投機的 load の失敗

store 命令は発行されると store buffer に格納され、アドレスが snoop bus に送出される。受けとった load はそのアドレスに対応する tag のループ番号が store したプロセッサが送出したループ番号よりも大きい場合は、 tag から、投機実行した load の失敗が判別される。 (図 6) そして、 load すべき data の update を行なう。 load buffer は、投機的 load の失敗を検出すると、プロセッサの投機実行を行なう loop analyzer に

通知する。

6. OCHA-Pro の概要

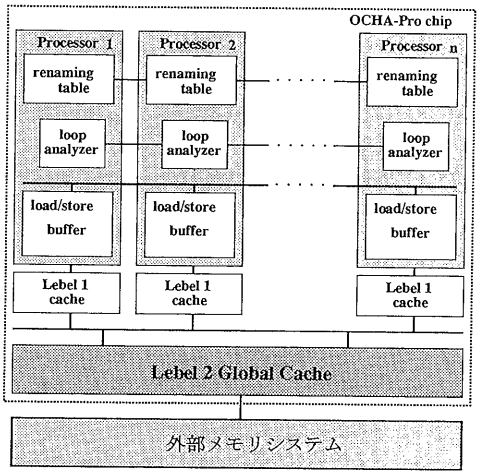

図 7 OCHA-Pro の構造

OCHA-Pro は次に示す特徴を持つ On-Chip MIMD プロセッサである。

(1) 各要素プロセッサが独立のスレッドを実行できる MIMD プロセッサ、要素プロセッサ数は当初 4 ～ 8 を予定。

(2) 要素プロセッサ毎に private な 1 次 cache と chip 内で 共有される 2 次 cache を持つ。

(3) ループの検出とレジスタ依存を解析する loop analyzer

(4) Elastic Barrier によって同期して更新される renaming table

7. Loop の投機実行の例

ループレベルの投機実行の例として、メモリアクセスの投機実行の例を示す。MIPS の R4000 [17] 用の C compiler が生成した逐次実行形式のループを動的に並列化し、実行する。実験で用いたプロセッサのパイプライン構成も R4000 と同様と仮定する。

投機実行が全く失敗しない例を挙げる。投機実行が必ず成功する状況は多くあり、

- ループ内に iteration 間の制御依存を生成する分岐が無い。
- iteration 間のメモリアクセスに依存が無い。

場合である。

OCHA-Pro は最内ループを loop detector で検出すると、1 iteration を用いて register analyzer により投機実行可能ループであるかとレジスタの依存関係を解析する。次の iteration を用いて解析で判明したループ実行に必要なレジスタを転送し、各プロセッサは独立した並列実行に移る。ループの iteration counter は各プロセッサで生成される。投機実行は register analyzer の内容に従い、各プロセッサで独立して行なわれ loop analyzer は 1 iteration 終了すると他のプロセッサに投機の成功を通知する。投機実行の失敗を検出すると loop analyzer は後続の iteration に通知し、通知されたプロセッサの loop analyzer は失敗した投機実行を破棄し、renaming table を巻き戻す。

投機実行が成功した時の OCHA-Pro の並列動作による Speed Up は

$$speedup = \frac{\text{ループの}clock\text{数}}{(\text{準備} + \text{ループの}clock\text{数})} \times processor\text{数}$$

となり、Speed Up は ループ回数に依存しない。

7.1 Tomcatv Loop

投機実行が必ず成功する例として SPEC benchmark にある Tomcatv の I,J-loop を扱う。I,J-loop は、iteration 間のメモリアクセスでのアドレスの衝突はない。

このループ間依存のないループを、要素プロセッサが整数 unit × 2、浮動小数点 unit× 1、ロードのレイテンシが 5 clock、整数 unit は 1 clock 1 命令を実行可能とした場合について実験を行なった。

従来のプロセッサで、逐次的に実行した場合ループの 1 iteration に 338 clock 必要である OCHA-Pro では、Tomcatv I,J-loop の並列実行には iteration の準備に 25 clock 必要となる。

先の関係式から求められる並列動作時の Speed Up にループの解析フェーズを入れた Speed Up は次のグラフになる。

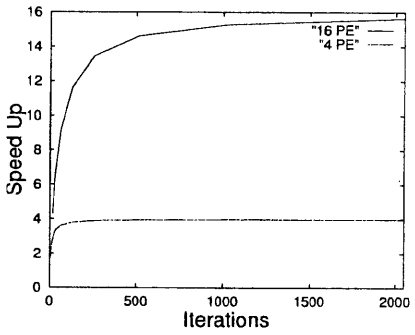

図 8 Tomcatv I,J-Loop

7.2 Livermore Loop

次に、投機的な実行の例として Livermore kernel loop No.14 を挙げる。この kernel loop は 2 重ループになっており、中に for ループが 3 個ある。3 個の for ループの内、一つ目の for ループの load は、アクセスにデータ依存性がある。ただし、このアクセスでは、load のみを行なっているので、投機的な load とならない。

そこで、load/store の依存関係を持つように Livermore loop No.14 を変形したのが、次の例である。

```
for  k  k<n  k  {
    vxk
    xxk
    ixk  long grdk
    xik  double ixk
    exk  ex ixk -
    dexk  dex ixk -
}
```

データ依存を持つループの逐次バイナリを並列実行する。配列 grd にデータ依存するため、この配列の内容は動的にしか解析できない。図 9 は、実験として、iteration の 約 10% に、投機的 load が失敗するように、ランダムに配列 grd を構成したデータを与えたものである。

要素プロセッサは先の例と同じになっている。ただし、投機的 load の失敗のコストは検出に 2 clock 、pipeline の巻き戻しに 4 clock 、投機実行の成否が分かるまでの clock 数がかかるとしてある。

配列の要素数 n が十分大きい場合、4PE で 約 2.6 倍、8PE で約 3.7 倍 Speed Up する。

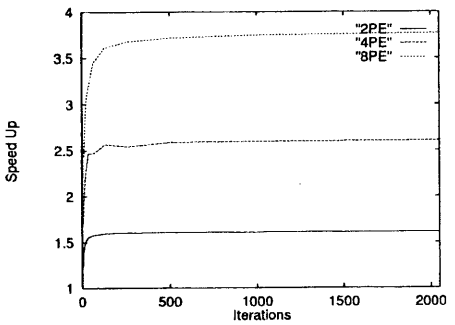

図 9 Livermore kernel Loop No.14

8. おわりに

今後は、本稿にある多重投機実行機能を持ったマイクロプロセッサを実装する際のコストの評価と実装を行なう予定である。

参考文献

1) 松本尚: Elastic Barrier: 一般化されたバリア型同期機構, 情処学会論文誌, Vol. 32, No. 7, pp. 886–896 (1991).
2) M.Johnson: *Superscalar Microprocessor Design*, Prentice Hall (1990).
3) J.E.Thorton: Parallel operation in the Control Data 6600, *In proceedings of the Fall Joint Computer Conference* (1964).
4) R.M.Tomasulo: An efficient algorithm for exploring multiple arithmetic units, *IBM Journal of Research and Development*, Vol. 11, No. 1, pp. 25–33 (1967).
5) SPARC INC: *The SPARC Architecture Manual version 9* (1995).
6) Silicon Graphics International: *R10000 Users Manual version 1.0* (1995).
7) Digital Equipment Corporation: *DECchip 21164A Hardware Reference Manual* (1995).
8) D.Hunt: Advanced Performance Features of the 64bit PA-8000, *In proceedings of COMPCON'95*, pp. 442–451 (1995).
9) M.D.Smith, M.Johnson and M.A.Horowitz: Limits on Multiple Instruction Issue, *In proceedings of the 3rd International Conference on Architectural Support for Programing Languages and Operating Systems*, pp. 290–302 (1989).
10) G.Sohi, S.Breach and T.Vijaykumar: Multiscalar Processors, *proceedings of the 22nd Annual International Symposium on Computer Architecture* (1995).
11) M.Fillo, S.W.Keckler, W.J.Dally, N.P.Carter and A.Chang: The M-Machine multicomputer, *In proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture* (1995).
12) Monica.S.Lam and Robert.P.Wilson: Limits of Control Flow on Parallelism, *proceedings of the 19th Annual International Symposium on Computer Architecture* (1992).
13) S.Damianakis, K.Li and A.Rogers: An Analysis of a Combined Hardware-software Mechanism for Speculative Loads, Technical Report 455, Princeton University (1994).
14) D.W.Wall: Limits of Instruction-Level Parallelism, *In proceedings of the 4th International Conference on Architectural Support for Programing Languages and Operating Systems*, pp. 176–188 (1991).
15) D.J.Kuck, R.H.Kuhn, D.A.Pauda, B.Leasure and M.Wolfe.: Dependence Graphs And Compiler Optimizations, *In proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 207–218 (1981).
16) R.Eickemeyer and S.Vassiliadis: A load-instruction unit for pipelined processors, *IBM Journal of Research and Development*, Vol. 37, No. 4, pp. 547–564 (1993).
17) MIPS Computer Systems Inc: *MIPS R4000 User's Manual* (1992).