成蹊大学工学研究報告
J. Fac. Eng., Seikei Univ.
Vol.41 No.1 (2004) pp.7-13

SMT 機構を備えた EPIC プロセッサアーキテクチャに関する基礎的検討

飯塚　肇[*1]，中島洋幸[*2]，緑川博子[*1]

A Preliminary Study of an EPIC Processor Architecture with SMT Mechanism

Hajime IIZUKA, Hiroyuki NAKAJIMA and Hiroko MIDORIKAWA
{jim, nakaji, midori}@sirius.is.seikei.ac.jp

ABSTRACT: A design of a new processor architecture and its evaluation are described. The processor takes advantages of both instruction-level and thread-level parallelism. Instructions are scheduled by software and instruction-level parallelism is explicitly indicated in each instruction. In addition, instructions to fork and control a secondary thread is added to the standard MIPS instruction set. The main and the secondary threads are executed concurrently using shared execution units as SMT (simultaneous multithreading). Preliminary simulation study shows 15-40% speed-up compared with a single thread case.

KEYWORDS: EPIC, Simultaneous multithreading, Processor architecture, ILP

（Received　March 22，2004）

１．はじめに

1980 年代以降の驚異的ともいえるプロセッサの高速化はその多くを半導体の進歩によっているが，プロセッサアーキテクチャの進歩によるものも少なくない[1]。アーキテクチャ的な技法は，主として命令レベルの並列性（ILP）を利用した複数命令の並列実行とメモリ参照の局所性を利用したキャッシュによるメモリアクセスの高速化に依っている。しかし，キャッシュは命令の並列実行に必要なデータを十分高速に提供するための技術なので，本質的なのは前者の命令の並列実行といってよいだろう。ソフトウェアにとっては１個のスレッドと見える命令ストリームの並列実行は，ユーザ（コンパイラ）が並列性を管理する必要がなく，過去のプログラムでも高速に動作するので広く受け入れられた。その結果，各クロックサイクルで最大１個の命令を発行する単純な固定パイプラインマシンから始まって，動的パイプラインマシン，１サイクルで複数命令を発行するマルチイシューアーキテクチャへと進化し，並列に実行される命令数が増加して，プロセッサの高速化に大いに寄与した。

しかし，プロセッサの計算資源が多くなって並列に実行可能な命令数が増加すると，命令間の依存関係をチェックして並列に実行できる命令を探し出すには，より先の方の命令まで調べなくてはならなくなる。その結果，命令発行ロジックは急速に複雑になり，分岐予測の精度も高めなければならないから，そのためのハードウェア量が増加してしまい，投資したハードウェア量に対する高速化の効果も殆ど得られなくなってしまう。

この状況に対する方策としては，ソフトウェアで命令のスケジューリングを行う VLIW や EPIC 方式が一部のマシンで用いられている[2]。これは，スケジューリング用のハードウェアの複雑化防止に役立つとはいえ，同時に実行できる命令数を大きく増加できるわけではないから，現在のマルチイシュープロセッサに比べて大きな性能向上を期待できるわけではない。

よって，更なる高速化にはスレッドレベルの並列性の利用が不可欠になるが，ユーザがスレッドを利用した並列プログラムを書かねばならないのでは，利用範囲が限定されるであろう。

*1 工学研究科情報処理専攻
*2 同学生［現：日本電気システム建設(株)］

本稿では，１スレッドを処理する EPIC プロセッサに同時マルチスレッド機構（Simultaneous multi-threading: SMT）を導入し，CPU の余剰資源を利用して低コストでの高速化を期待できる新たなプロセッサアーキテクチャを提案し，その初期的評価結果を述べる。

２．EPIC と SMT

EPIC プロセッサでは, 各命令が同時に実行できる命令に関する情報を含んでいる。同時に実行できる命令数は変化するから，ハードウェアのスケジューラがその数と利用できる演算器の個数の少ない方の個数分だけの命令を発行し，命令数が演算器数より多ければ，発行できなかった命令は次のサイクルで発行することになる．従って，存在する ILP を最大限に生かすには，十分な数の演算器を用意しなければならないが，通常，ILP の平均値は最大値よりかなり小さいから最大値に対応するような資源を用意すれば，大部分のサイクルで多くの資源がアイドル状態になることは明らかである。

一方, SMT は通常のマルチイシュープロセッサにおいて各サイクルで複数のスレッドからの命令を実行できるようにする機構[3]で，レジスタ等のスレッドのコンテクスト情報を記憶する資源と，同じサイクルで複数スレッドからの命令を発行するロジックを追加するだけで簡単に複数スレッドを実行できる。この方式は，ソフトウェアからはマルチプロセッサと同じに見えるが，1 CPU で追加する資源が少なくてすむので，低コストのマルチスレッドマシンとして intel 社の Xeon[4]等で実用化されている。

しかし，この機構によっても１スレッドの実行が高速化されるわけではなく，複数スレッドのプログラムでなければあまり効果はない。また，元々 1 CPU の余剰資源を用いるので複数スレッドの場合でもマルチプロセッサのように性能向上するわけではない。

そこで，１スレッドの一部をマルチスレッド化して SMT 実行する方が本格的マルチスレッドマシンのような大きな性能向上は得られなくても，必要なコストが大きくないので現実的であると考えられる。本稿で提案するプロセッサアーキテクチャ（以下 EPAS）はこうした観点で設計されている．即ち，EPAS は次のような特性を有する。

(1) 各命令はそれと同時に実行できる命令がその先の命令列のどこまで存在するかを示すフィールドを有する。
(2) スレッドを生成／待合せをする命令を持つ。但し，メインスレッドから生成されたサブスレッドは更にスレッドを生成することはできない。
(3) レジスタセットは２セットがハードウェア的に用意され，各スレッドに対応させられる。
(4) 演算器はスレッド間で共用されるが，各サイクルでサブスレッドにはメインスレッドが使わないものだけを割当てられる。

以下の各節では, EPAS アーキテクチャの EPIC と SMT 機構について詳述するとともに，シミュレータによる評価結果を述べる。

３．EPIC 機構

EPAS の EPIC 機能は，当面 MIPS の命令セット[5]に依存フィールドを追加する形式になっている。本来，新たな命令形式を設定すべきではあるが，現時点では可能性の評価段階なので, 詳細な設計には入っていない。当面，本稿で提案するアーキテクチャの評価を行うにはそれで十分と考えられるからである。

新らたに用意した“依存フィールド”は各命令ごとに設定されていて，それぞれがその命令と依存関係のない後続命令数を示している。例を図１に示す。このプログラムでは１行目の命令（`lui`）は，`$t0` レジスタに関するデータハザードのために３行目のロード命令(`lw`)と同時に実行することはできない。間にある２行目のみが命令１と依存関係にないので，命令１の依存フィールドの値は１になる。同様に。２行目の加算命令(`addi`)は８行目の減算命令(`sub`)と`$t6` レジスタに関するデータハザードがあるからその間にある３～７行に対応する５が命令２の依存フィールドの値となる。また，`$zero` は特殊なレジスタで常に０が入っていて，０以外の値を代入しても無視されるので，`$zero` を使っている命令が後続命令中にあってもハザードは起きない。

また，９行目の加算命令(`add`)のデスティネーションレジスタ`$t5`は12行目の即値加算命令(`addi`)のデスティネーションでもあって WAW (Write-after-write)のデータハザードがある。この場合，同時に発行された命令パケット内の命令の順序を管理すれば両命令間に依存無しとすることも可能であるが，ここでは一般性を保つために依存があると判断して依存フィールドには２を入れている。

13 行目にあるブランチやジャンプのようにプログラムカウンタを変更する命令（後述の `mst` などのスレッド系の命令も含む）は同時に実行すると制御ハザードを起こすので必ず依存フィールドが０になる。また．最後の命令以降に命令はないことから。最後の命令に対応する依存フィールドは必ず０になる。それ以前の命令も依存

関係がなかったとしても最後の命令までをカウントする．

コンパイラがこの様な規則に従って。全ての命令の依存フィールドを埋めることを前提として，EPIC の制御ハードウェアは動作しする。即ち，制御フィールドの値を命令とともにフェッチユニットが読み込んで同時に実行可能な命令数を判別して実行を行う。

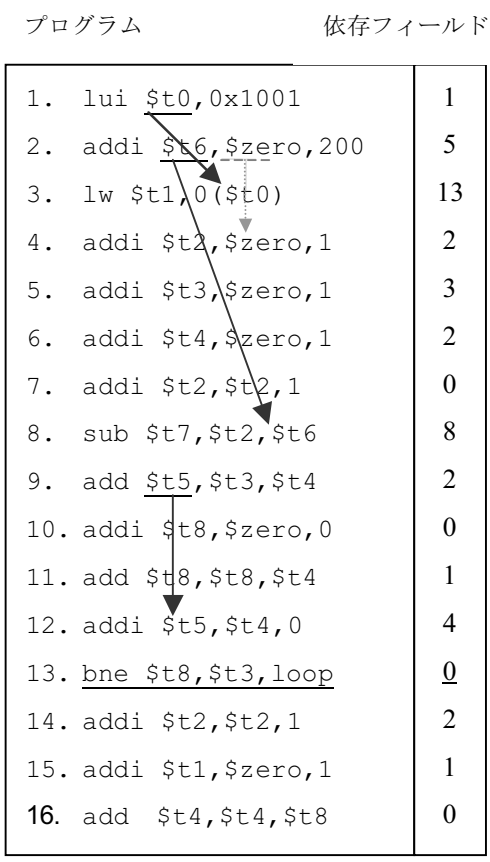

図 1 依存フィールド

4．SMT 機構と命令セット

4．1　基本設計

2で述べたように，EPAS ではスレッドは1又は 2 スレッドで実行を行う。プログラムは基本的には1つのメインスレッド（M スレッド）で実行され，依存関係のない部分を第2のサブスレッド(S スレッド)で実行する。このとき，2スレッドで実行中のスレッドモードを“ツインモード”，M スレッドのみの時を“シングルモード”と呼ぶ．ツインモードで実行中は，演算器は M スレッドが優先的に使用し，余っている演算器のみを S スレッドが使用する。もちろん，M スレッドが待ち状態に入った場合には S スレッドが全ての演算器を使用することが出来る。

従って，この仕組みを実現するためには，2 つのプログラムカウンタとそれぞれのスレッドに別のレジスタセット（32 個の整数レジスタと乗除算用 hi と lo レジスタ，浮動小数点レジスタなど）が必要になる。

ここで，スレッドを2に限定した理由は，ソフトウェア的には1個のスレッドからコンパイラが容易に取り出せる明らかな依存関係のない部分はあまり多くないと考えられる上，仮に3以上のスレッドを生成できたとしても SMT 機構を利用して演算器を共用するという基本的考え方からしてスレッド数に比例して多くなるレジスタ等のコンテクスト記憶に対する投資効果が期待できないからである。

4．2　レジスタモード

M スレッドと S スレッドは通常のスレッド以上に密接に関連を持っているので，多くの情報がレジスタ経由で通信されることになる．即ち，ツインモードになる前の S スレッドレジスタの初期値のセットや，ツインモードでの結果をシングルモードで使うときのために，M スレッドレジスタから S スレッドレジスタ，或はその逆のレジスタ間コピーが必要になるので，当然のことながらそのための命令が用意されている。但し，この命令は依存関係の問題からシングルモードでしか正しい結果を保証されない。

シングルモードからツインモードに移行する際，S スレッドは，初期値として M スレッドレジスタ内のいくつかの情報を必要とする場合が多い。この場合，S スレッドを立ち上げる命令を実行する前に，必要な値をコピーするためにデータの個数だけそれぞれのレジスタ間でのコピー命令が必要になって，終了するまでに実行する命令数が少ないことも多いと想定される S スレッドでは，相対的にオーバヘッドが大きくなる可能性がある。

これを避ける方法として，EPAS ではシングルモード時には M スレッドの命令が M レジスタと S レジスタの両方に書込みを行うモードを用意した．このモードを用いれば値をコピーする命令を実行しなくてすむようになる．実際は，同時書込みをしたくない場合もあると考えられるので，図2に示すように，シングルモードでの実行時に M スレッドレジスタにのみ書込む“分離”モードと両方のレジスタに書込む“シャドウ”モードをプログラムで選択できるようにしてある。

ここで，ツインモードで S スレッドが書込んだレジスタは，シングルモードに戻った後，M スレッドが書込みを行わない限り異なった値のままであるからそのまま再

度ツインモードになっても同じ値にはならない．即ち，たとえ“シャドウモード”に設定されていてもツインモード移行時に全てのレジスタが同じ値を持つことが保証されるわけではないことに注意すべきである。

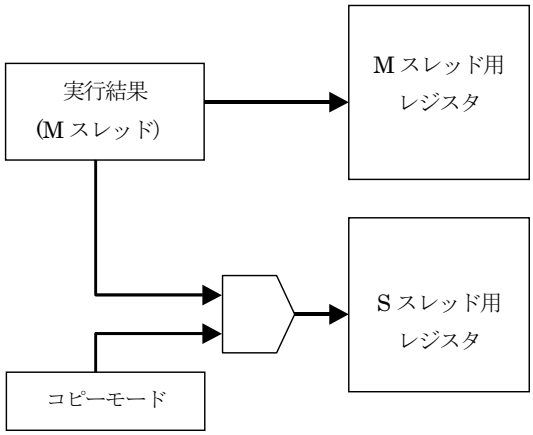

図２　レジスタのコピーモード

４．３　SMT 制御命令

既に述べたように，EPAS ではソフトウェア的には１個のスレッド内の依存関係のない部分をコンパイラが S スレッドとしてコンパイルして（もちろん，ユーザが明示的に示してもよい)，SMT で並列実行を行う．従って，S スレッドの生成/同期のために新たな命令が必要になる。表１にその目的のために設計した命令を示す。

表 1 追加した命令

種類	命令ニモニック
スレッド系命令	mst
	mstr
	jst
	jstal
レジスタ命令	mfst
	mfmt
	srcm
状態確認命令	stm
	stt

この中で特に重要と思われる命令について，以下に説明する。

・mst [address]　　：make sub thread

・mstr [register]

S スレッドを作成する命令．M スレッドのみが実行でき，スレッドモードはツインモードになる。mst 命令では S スレッドは PC 相対指定の address からの命令を実行する。また，mstr では S スレッドの実行開始アドレスは指定レジスタ内の値となる。

・jst　　：join with sub thread

・jstal

スレッドの処理が終了して，もう片方のスレッドの終了を待つときに使われる命令である。名称からは M スレッドしか使用できないように見えるが，いずれのスレッドでも実行できる。これらの命令以降，そのスレッドはウェイト状態になり，演算器は全てもう一方のスレッドを処理するのに費やされる．そして残りのスレッドの終了を確認してから，シングルモードに戻る．また，jstal 命令では，次の命令のアドレスがレジスタ 31 に格納される。

・mfst [M-Reg] [S-Reg]

：move from sub thread

・mfmt [S-Reg] [M-Reg]

：move from main thread

レジスタ間コピーを行う命令．命令によってソースとデスティネーションが逆になる。

・srcm [immediate]

：set register copy mode

シングルモードでのレジスタ間コピーモードを設定するための命令。

・gtm [Reg]　　：get thread mode

指定されたレジスタ [Reg] に現在のスレッドのモードを示す値（）を代入する。

・stt [Reg]　　：test thread type

指定されたレジスタ [Reg] にこの命令を実行したスレッドが M スレッドか S スレッドかを示す値（）を代入する。

５．EPAS のプログラム構造

次に EPAS の S スレッドを利用したプログラムの構造と簡単な例を示す。

５．１　基本構造

プログラムの基本的構造は，図３のようになる。

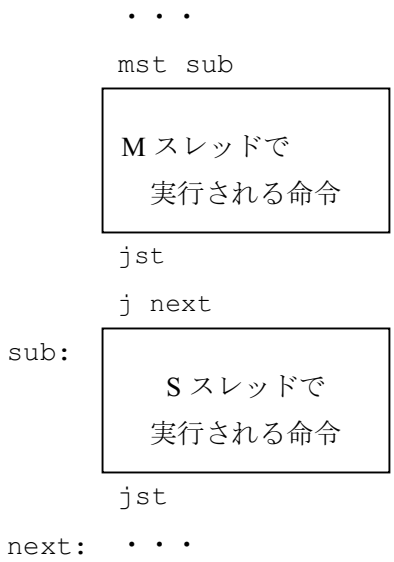

図３ プログラムの基本構造

５．２ mstr 命令と jstal 命令

jstal 命令はジョインの際に次の命令アドレスをセーブするから，一旦 S スレッドをジョインした後，M スレッドが S スレッドを再度ディスパッチして元々実行していたコード部分を継続実行させたい場合に利用できる。図４にその例を示す。

今，M スレッドは①で同期待ち状態であり，S スレッドは④の命令を実行するところとする。S スレッドは jstal を実行する度に$ra に次の命令のアドレス（④の jstal であれば⑤の命令のアドレス）を代入するから，S スレッドが jstal を実行すると，M スレッドは既に同期待ち状態であるため，シングルモードに移行して②からの命令を実行する。そして②で S スレッドの jstal が$ra に代入した次の命令のアドレスを，M レジスタにコピーする．その値が⑥で代入した 0 でない限り③で $ra をアドレスとして S スレッドを実行させ，M スレッドは再び①に戻り同期待ち状態になる。

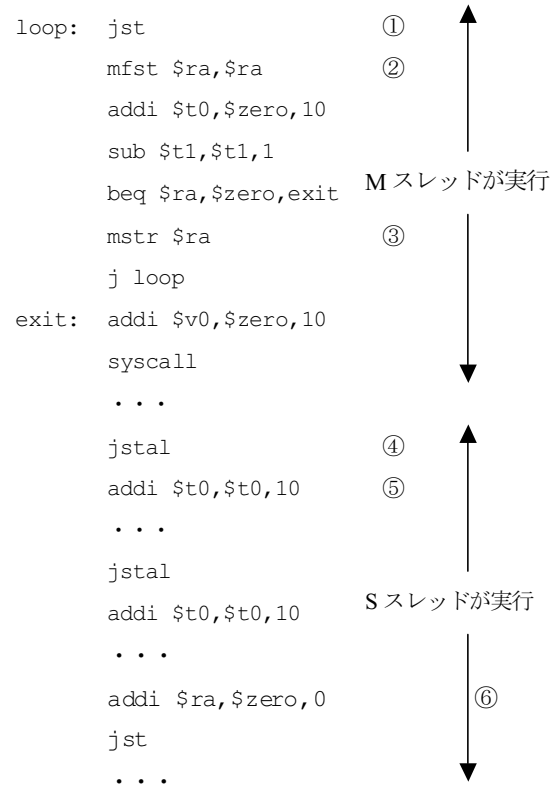

図 ４ mstr と jstal を使ったプログラム例

５．３ 実際のプログラム例

ここで実際の EPAS プログラムの例として，後述のシミュレータによる評価に使用した“フィボナッチ数と平方根を求めるプログラム”を図５に示す。

このプログラムでは明らかに二つの作業に依存関係は無いので M スレッドで与えられた値をフィボナッチ数が超える値を求め，同時に S スレッドで与えられた値の平方根を求めている。

```
        .data
num:    .word  1000

        .text
main:   lw $t0,num($zero)
        addi $t6,$zero,200

        mst sqrt                ①

fib:    addi $t2,$zero,1        ②
        addi $t3,$zero,1
        addi $t4,$zero,1
floop:  addi $t2,$t2,1
        sub $t7,$t2,$t6
        add $t5,$t3,$t4
        addi $t3,$t4,0
        addi $t4,$t5,0
        sub $t5,$t5,$t0
        bgez $t7,fend
        bltz $t5,floop

fend:   addi $t7,$t2,0
        jst                     ③
        j send                  ④

sqrt:   addi $t2,$t0,0
sloop:  addi $t1,$t2,0
        addi $t6,$t6,-1
        div $t0,$t1
        mflo $t2
        add $t2,$t2,$t1
        srl $t2,$t2,1
        bltz $t6,send
        bne $t2,$t1,sloop
        jst                     ⑤

send:   mfst $t2,$t2            ⑥
        addi $v0,$zero,10
        syscall
```


図５ フィボナッチ数と平方根を求めるプログラム

プログラムはラベル main からシングルスレッドで始まる。初期値（ここでは 1000）を num からロードし，無限ループしないために回数として 200 を与える。

①で S スレッドが作られ，S スレッドはラベル sqrt から実行を開始する。M スレッドはそのまま②の命令へ進む.

M スレッドは floop から続く命令を，フィボナッチ数が超えるかループの回数を超えるまで繰り返す。そして③の jst 命令で同期待ちを行う。

S スレッドは sloop から続く命令を，平方根が求まるか，ループの回数を超えるまで繰り返す．そして⑤で同期待ちを行ない，M スレッドと S スレッドの両方が同期待ちになると，シングルモードに切り替って M スレッドのみが④から実行を再開し，⑥で S スレッドが求めた値を M スレッドのレジスタにコピーして終了する．

６．シミュレータによる評価

EPAS アーキテクチャの可能性を評価するために，シミュレータを作成していくつかのプログラムについて調べた．以下，作成したシミュレータと初期的評価結果について述べる。

６．１ シミュレータ

EPAS のようなアーキテクチャの評価には，きっちりとした論理設計を行ってクロックレベルでシミュレートすることが必要であるが，そうしたシミュレータの開発には時間がかかるので，今回はかなり簡略化したシミュレータを用いた。従って，得られた結果は真の効果を現すものとはいえないが，第一段階の基本的効果を見ることは可能と考える。次に，作成したシミュレータの特性を示す。

（1）命令セット

MIPS の整数の演算のみをシミュレートし，浮動小数点演算命令は実装していない。浮動小数点演算は演算時間が可変長になるのでハードウェアの詳細まで設計しないとクロックレベルのシミュレーションは難しく，複雑になってしまうし，整数と同じ演算でできるとするのも非現実的なのであきらめた。しかし，より正確な評価には不可欠なので今後対応していくつもりである。

（2）キャッシュは全てヒット

即ち，メモリアクセスは 1 クロックで行われるとしている。しかし，マルチスレッドなので，１スレッドのメモリアクセスミスはその間他のスレッドを走らせられるので，ある程度ミスが生じる状況の方が EPAS の効果としてより大きくなる可能性がある。

（3）パイプライン

単純な５段パイプラインで行っているが，浮動小数点命令への対応と共により現実的なものを設計する必要があろう。

（4）分岐は全て非分岐予測

命令は in-order 実行であるため，分岐した場合は後続命令のうち，同じスレッドの命令は全てフラッシュする。

（5）シミュレータの出力

実行が全て終了すると，“総クロック数”，“総実行命令

数”，“CPI”及び“実行した命令の内訳”を表示する。

６．２　テストプログラム

テストプログラムは，明らかな依存関係のない例として 5.3 で示した“フィボナッチ数と平方根”を求めるプログラムと，一般的なプログラムの例として“クイックソート”をとりあげた。1 スレッドで実行した場合と，S スレッドを使用した場合の平均実行命令数と平均サイクル数を以下に示す。

フィボナッチ数と平方根の結果は表 2 のようになった。単一スレッドの方が命令数が 5 個多いのは，2 スレッド化のために増えた命令の個数である。EPAS ではその機構が無い場合に比べ約 25%サイクル数が減少している。

表 2 フィボナッチ数と平方根

	平均実行命令数	平均サイクル数
1 スレッド	486	522
EPAS	491	387

一方，クイックソートの結果は表 3 のようになった。EPAS の方が命令数が 3 個少なくなっている。これは EPAS 実行のオーバーヘッドよりも，1 スレッドの場合スタックのプッシュとポップが 1 回多くなってしまい，それにかかる命令の方が多いためである。これらの例でも分かるようにプログラムの書き方にもよるので正確なことはいえないが，EPAS の命令数オーバヘッドは少ないと見てよいと思われる。実行サイクル数は，EPAS では約 11%減少している。

表 3 クイックソート

	平均実行命令数	平均サイクル数
1 スレッド	13716	19150
EPAS	13713	17069

６．まとめ

本稿では，命令レベルとスレッドレベルの並列性を併用し，複雑なハードウェアスケジューラを要さない新しいプロセッサアーキテクチャを提案し，その初期的評価結果を述べた。評価は EPAS の価値を検証するには全く不十分なものではあるが，一応の効果が判明した。更なる検討が必要ではあるが，組込み用プロセッサ等低コストのプロセッサアーキテクチャとして EPAS は十分な可能性を持つものと思われる。

参考文献

［1］J.L.Hennessy and D.A.Patterson: Computer Architecture -- A Quantitative approach --, 3rd ed., Morgan Kaufmann, 2003.

［2］J.Huck et al: Introducing The IA-64 Architecture, IEEE Micro, vol.20, no.5, pp.12-23, 2000-9/10.

［3］T.Ungerer, B.Robic and J.Silc: A Survey of Processors with Explicit Multithreading, ACM Computing Surveys, vol.35, no.1, pp.29-63, 2003-03.

［4］D.T.Marr et al: Hyper-Threading Technology architecture and Microarchitecture, Intel Technology Journal Q1, 2002, pp.1-12.

［5］D.Sweetman: See MIPS Run, Morgan Kaufmann, 2002.