計算機アーキテクチャ 101－2
（１９９３．８．１９）

超並列計算機 RWC-1 における同期機構

岡本 一晃† 松岡 浩司† 廣野 英雄† 横田 隆史*
堀 敦史† 児玉 祐悦‡ 佐藤 三久‡ 坂井 修一†

†技術研究組合 新情報処理開発機構 つくば研究センタ
* 技術研究組合 新情報処理開発機構 超並列三菱研究室
‡電子技術総合研究所

超並列計算機のための高速な同期機構について考える。RICA(Reduced Interprocessor-Communication Architecture) に基づいた高速なマイクロ同期機構を紹介し、これをソフト的に組み合わせることで種々の同期処理を高速かつ簡単に実現できることを示した。さらに、実際のマイクロ同期機構の実現方法について検討した。その結果、ハードウェア化された同期機構を特殊命令によって起動する方法が、もっとも合理的であることが明らかになった。また、この結果を踏まえて現在開発している、超並列計算機 RWC-1 のマイクロ同期機構について述べる。

Synchronization Mechanisms on the Massively Parallel Computer RWC-1

Kazuaki OKAMOTO† Hiroshi MATSUOKA† Hideo HIRONO† Takashi YOKOTA*
Atsushi HORI† Yuetsu KODAMA‡ Mitsuhisa SATO‡ Shuichi SAKAI†

†Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono,
Tsukuba-shi, Ibaraki 305, Japan
*Massively Parallel Systems Mitsibishi Laboratory,
Real World Computing Partnership
‡Electrotechnical Laboratory

This paper describes the efficient synchronization scheme for massively parallel computers. At first, we propose the micro-synchronization scheme based on RICA(Reduced Interprocessor-Communication Architecture). It is shawn that the series of the micro-synchronizations integrated by software can easily and efficiently realize various kinds of synchronizations. Then we discuss the implementation methods of the real micro-synchronization mechanism for the parallel computer. In this paper, a set of special instructions managing the synchronization hardware turns out to be rational, which will be implemented in massively parallel computer RWC-1.

1 はじめに

我々は新情報処理開発機構(RWCP)において、超並列計算機の研究・開発を行っている。本研究の目的は、1) 超並列計算機のハードウェア・ソフトウェア技術を確立し将来あるべき姿への指針を示す、2) リアルワールドコンピューティング(RWC)研究計画において他の研究グループが開発する並列処理ソフトウェアの実行母体となる、の2点を満たすような汎用超並列計算機を構築することである。本研究の第一段階として、要素プロセッサ数1000規模の超並列計算機RWC-1をプロトタイプシステムとして開発し、その開発を通して同期、記憶構造、通信、入出力などの並列アーキテクチャの検証を行う。

RWC-1のアーキテクチャについてはその基本構想をすでに述べているが[1]、その中で中心となる2本の柱が、(a) 演算処理と通信とを融合し単純化した新アーキテクチャRICA(Reduced Interprocessor - Communication Architecture)と、(b) スレッド長やスレッド分配などの最適化を行うスーパスレッディング[2]である。このうちRICAは、処理を単純化して高い効率をあげるRISCの概念を通信にまで拡張し並列処理アーキテクチャに適用したものであり、同じ概念の中で同期処理のためのマイクロ同期機構を規定している。

一般に並列計算機においては、並列に動作するプロセッサ間の同期を如何にしてとるかということが重要な問題である。特に汎用を指向する細粒度並列処理計算機では、同期にかかるコストが全体の性能に大きく影響するため、同期のオーバーヘッドを如何に低く抑えるかが重要なポイントになる。ここで並列計算機の同期処理を大別すると、(1) 命令や小規模の命令ブロックを単位とするミクロな同期と(2) バリアのようなマクロな同期との2種類に分けられる。前者については、生産者-消費者間のデータ依存関係を保持するための同期処理がその代表例としてあげられる他、ベクトル演算のような定型的な繰り返し演算におけるベクタの要素同士の対応付けなどもこれに含まれる。一方後者は、例えば並列に処理されるようにスケジューリングされた複数のタスクにおいて、全体で処理の歩調を合わせるために待ち合わせを行うようなものがこれに該当する。またバリア同期のバリエーションとしては、これまでにFuzzy Barrier[3]、Elastic Barrier[4]、重複可バリア[5]など多数が提案されている。

こうしたさまざまな同期において、それぞれの処理に伴うオーバヘッドを低減し高速な同期処理を実現するために、専用の同期機構をハードウェアで用意することが多い[6]。実際、現存する並列計算機の多くには、同期のための何らかのハードウェア機構が装備されている。RWC-1においても、RICAアーキテクチャで規定しているとおりマイクロ同期機構を内蔵するが、RICAの概念ではマイクロ同期機構そのものの実現方法までは定義していない。すべてハードワイヤードに実現するものからソフトウェアだけで実現するものまでいろいろな実現方法が考えられる。

本稿では、まず並列計算機の同期機構に求められる要件にはどのようなものがあるかを述べ、その実現方法について検討する。そしてRWC-1上でのさまざまな同期処理のインプリメントについて述べる。

2 マイクロ同期機構の実現

RICAでは、演算処理と通信とを融合し単純化することで高効率な通信機構を実現している。同様の枠組の中で同期機構についても規定することができる。すなわち、単純化されたマイクロ同期機構を利用することで、さまざまな同期処理を高速に実行することが可能になる。本節では、RICAにおけるマイクロ同期機構の実現方法について考察する。

2.1 マイクロ同期機構の有効性

2.1.1 RICAにおけるマイクロ同期機構

超並列計算機において、通信のオーバヘッドの軽減は最も重要な問題の一つである。このためにRICAではいくつかの技術を提案しているが、その中の一つに、高速の網インタフェースによるメッセージ処理オーバヘッドの低減があげられている。ここでは、網から取り込まれたメッセージが数クロック程度で処理されるよう、メッセージ処理部をつくらなければならないと考えている。その実現方法は次のようなものである。

- メッセージにはデータとともに、命令番地、作業セグメント番地等が含まれていて、メッセージ処理機構はこれらをそれぞれのレジスタ類へパイプライン式に順次格納していく。これらはすべてハードウェアの機能によって行われる。
- データの到着をトリガとして、命令実行が開始される。
- 数クロックで一つのメッセージを生成するメッセージ生成パイプラインを別置し、演算パイプラインと非同期に動作させることにより、命令実行とメッセージ生成処理を重畳化する。

RICAではこのようにして高速な網インタフェースを実現し、またメッセージ処理・命令実行・メッセージ生成の各パイプラインを融合させることで通信と命令実行を一体化して、高速の通信を可能にしている。

一方、このように通信オーバヘッドを低減しているため、必然的にその通信形態としてはデータをブロックで送るようなものだけでなく、細かなメッセージを

多数搬送するような粒度の細かいものも多用される。したがって、演算を行う一対のデータ同士が待ち合わせるような、細かいレベルの同期が頻出することが考えられる。このような待ち合わせに対し、RICA ではマイクロ同期機構を設けることで対処している。

RICA におけるマイクロ同期機構は、循環パイプライン上のメッセージ処理部と命令実行部との間に設置され、同期が終了したメッセージから順に命令実行部に送られてスレッドが起動される。したがって高速なマイクロ同期機構が用意されれば、スレッドの起動が高速化される。

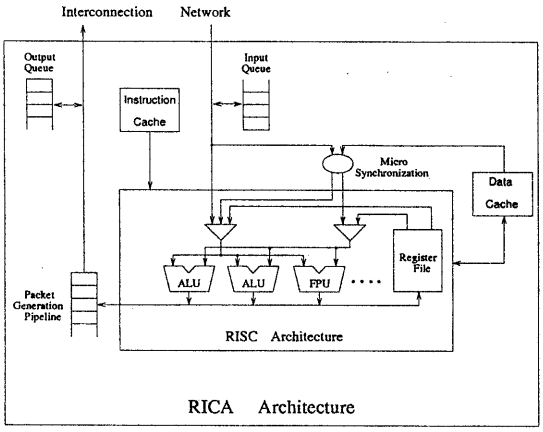


図 1: RICA の構成図

2.1.2 有効性と限界

マイクロ同期機構は、2 つのデータを待ち合わせるだけの単純な同期機構である。これを操作してマイクロ同期を行うことをソフトウェア側から陽に指定することができれば、プログラムでマイクロ同期を組み合わせることにより、さまざまな同期処理を実現することが可能になる。

例えば生産者 - 消費者間の同期は、マイクロ同期をキュー操作と組み合わせてソフト的に同期付構造体を形成することにより、簡単で効率的な実現ができる。またバリア同期については、マイクロ同期をツリー状に展開すればよい。さらにこの場合、ツリーを形成するのがすべてソフトウェアで行われるため、全同期をとるような単純なバリアだけでなく、例えばプロセスごとに同期をとるような比較的複雑なバリアについても簡単に実現できる。

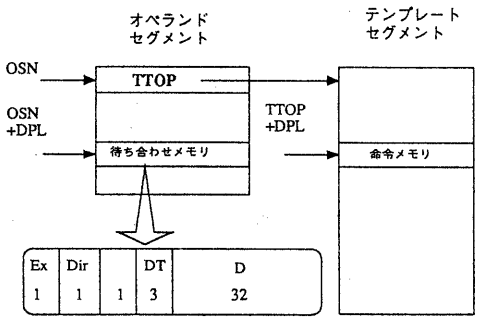
これらのさまざまな同期処理は、マイクロ同期を組み合わせる部分だけをソフトウェアで実現しているため、核となるマイクロ同期そのものがハードによって高速に実行されれば、全体的に高速に処理することが可能である。

ただし、ここで示したバリア同期はあくまでソフト的に実現されたものであるため、クロックレベルの厳密な同期をとることは不可能である。こうした厳密な同期が必要になる場合は、別系統の同期手段を考える必要がある。

2.2 ハードウェアによる実現方法

上述の通り、単純化された高速のマイクロ同期機構を持つことで 2 入力待ち合わせを短時間で処理できるならば、これを応用して多くの同期処理を効率よく実現することが可能である。マイクロ同期を最も高速に処理するには、マイクロ同期機構をすべてハードウェア化するのが最良であると考えられる。実際、代表的な細粒度並列計算機であり、2 入力待ち合わせがほぼ命令ごとに生じる命令レベルデータ駆動計算機などにおいては、待ち合わせを行うためのハードウェアを装備しているのが普通である [7]。

一般に細粒度並列計算機では、マイクロ同期のための特別な制御機構を用いてメモリ上で 2 入力待ち合わせを行い、これを命令記憶と対応付けることで高速のマイクロ同期を実現している。例えば RICA を部分的に実現した最初の実装例である高並列計算機 EM-4 では、スレッド間の通信をパケット交換で行っているが、各々のパケットには同期を行うメモリのアドレスが書かれており、ハードウェアがこれを命令記憶と対応付けて処理する [8]。EM-4 のマイクロ同期機構を図 2 に示す。



OSN： オペランドセグメント番号
DPL： 待ち合わせ・命令オフセット
DT： データタイプ
D： データ
TTOP：テンプレートセグメントベースアドレス
Ex： 待ち合わせデータ存在フラグ
Dir： 待ち合わせデータの左右オペランドフラグ

図 2: EM-4 のマイクロ同期機構

EM-4 ではメモリはセグメント単位に分けて使われ、各パケットは作業セグメント番号と、セグメント

内のオフセットを持っている。一方命令記憶も同様にセグメント化されており、各命令セグメントは作業セグメントと対応付けて使われる。具体的には、作業セグメントの先頭番地に、対応する命令セグメントの先頭番地が保持されている。これにより、両セグメントで同じオフセットのもの同士を対応付けて、待ち合わせと命令の対応付けを行うことによりマイクロ同期を実現している。このようにEM-4ではマイクロ同期メカニズムを完全にハード化しているので、一回のマイクロ同期に要する時間が3クロックと、非常に短い時間で処理できる。

一方、EM-4では待ち合わせを行うメモリと命令記憶がセグメント単位で対応付けられているため、処理効率と融通性に若干の問題があることを指摘されてきた。そこで、パケットに待ち合わせアドレスと命令アドレスの両方を持たせることによりこの問題の解決を図ったのが、EM-5方式である[9]。ただしEM-5方式の場合、待ち合わせと命令の両アドレスを保持するための広い領域がパケットに必要になるため、現実にはこれまでインプリメントされた例はない。

このように、EM-4ではマイクロ同期機構をすべてハードウェアにより実現しており、これを利用することでさまざまな同期処理が非常に高速に実行できることが報告されている。しかしマイクロ同期機構の全てをハード化してしまうと、逆に処理が定型化されてしまい、柔軟性を失うという問題点が生じる。例えばオペレーティングシステムを構築してページング処理などをサポートすることを考えた場合、以下の問題が表面化してくる。すなわち、待ち合わせ領域がページフォルトした時にはページの入れ換えを行わなければいけないが、この処理を全てハードが行うのは困難である。したがってページングに関する処理はOSに依頼しなければならないが、この場合マイクロ同期機構側にOSのインタフェースを組み込まねばならない。これはかなり複雑な機構になることが予想され、あまり現実的ではない。実際、これまで構築が報告されている同種の計算機は、すべてバックエンドで動かすことを前提にしたものばかりであり、OSのサポートまで考慮したものはまだ開発が報告されていないのが現状である。

2.3 ソフトウェアによる実現方法

待ち合わせ処理に柔軟性を持たせ、OSとのインタフェースを複雑にしないようにするために、特殊なハードは用いないで全てソフトウェアだけでマイクロ同期を実現する方法も考えられる。例えば、マイクロ同期をサポートするための最低限のハードとして、スレッド間の排他制御を実現するための TestAndSet 命令だけを持つようなシステムを考える。この場合、ソフトウェアだけでマイクロ同期を実現すると、大体次のようなプログラムになると思われる。

```
start:  TAS   (lock)
        BB1    start
        NOP
        TAS   (flag)
        BB1    fire
        NOP
        STR    regQ, (data)
        CLEAR (lock)
        BREAK
 fire:  MOV    regQ, reg0
        LOAD  (data), reg1
        CLEAR (flag)
        CLEAR (lock)
        CONTINUE
```

図3: 完全ソフトウェアによるマイクロ同期

ここで、括弧でくくったものはメモリアクセスであることを示している。キャッシュへのアクセスは、たとえヒットしても最低2クロックはかかる(スループットは毎クロックに1つ)ので、このようなマイクロ同期処理は少なくとも13～14クロックくらいは要するものと考えられる。さらに、ここではオペランドの左か右かの判定を省略しているが、実際にはこれに左右判定のための数クロックが加わる。したがってこの実現方法では、全てをハードで実現した場合に比べ、5倍近い処理時間がかかることになり、これでは高速な同期処理とは言い難い。

しかしその一方で、すべての処理が命令で起動されているため、ページフォルトなどが生じても簡単にプリエンプションを起こしてOSに処理を引き渡すことが可能である。さらにページ入れ換え後の復帰についても、すべての処理をソフトウェアで自由に操作できるので、簡単に実現することができると思われる。

2.4 特殊命令を用いた実現方法

前述の通り、マイクロ同期機構を実装するにあたり、全てをハードウェア化すると高速な処理が可能になるが処理の柔軟性に欠け、またソフトだけで実現すると柔軟な処理に対応はできるが処理の効率化が図れず遅い同期機構になってしまう。そこで両者の利点を引き出すために、その折衷案として同期操作をハードウェア化し、これを実行する特殊命令を導入することを考えてみる。すなわちハードウェアで実現される処理をすべて特殊命令によって起動し、それらの処理の組み合わせ操作をソフトウェアで記述することにより、高速なマイクロ同期を柔軟に行う。これにより

- 処理の一部をハードウェアがサポートするため、高速な動作が望める。

- すべての同期処理が命令によって起動されるため、プリエンプションやページフォルトが生じた時の処理の移行が比較的簡単に行え、かつ復帰の問題も解決しやすい。

といった利点を得ることが期待できる。

この方法は、どこまでの処理をハードウェアに委ね、どこからをソフトウェアが処理するかのトレードオフが問題となる。

特殊命令の一例として、「待ち合わせフラグのtest&setを行い、さらにその結果によって分岐先を決める」ところまでを一命令で実行する、「Test&Set&Branch」命令を考えてみる。マイクロ同期の基本は、

- 相手がいれば、フラグをクリアして相手をレジスタ上に読み出す。
- 相手がいなければ、フラグをセットして自分をメモリに書き込む。

の2点にあるわけなので、この特殊命令を用いると、

```
     TSB   (lock&flag), yes, no
yes: LOAD  (memory), reg1
     CLEAR (lock&flag)
     break
no : STORE  reg0, (memory)
     CLEAR (lock)
```

で実現される。TSB命令はフラグのtest&setと同時にロックの判定も行い、仮にすでにロックされていれば例外を発生する。このように特殊命令を導入すると、完全ソフトウェアで実現する場合に比べてはるかに命令数が少く実現でき、高速であると言える。しかも、ハードウェアの処理はすべて特殊命令で起動されるため、プリエンプションなどの問題も解決しやすい。ただし、実際のマイクロ同期ではオペランドが左であるか右であるかの判別が必要なので、上述の処理ほど簡単ではない。また、命令実行と同期処理が重畳化できないので、完全にハードウェアで実現する場合ほど効率は良くない。

2.5 実現方法の検討

ここでは、前述した3つの実現方法について、その処理能力と実現容易性の比較検討を行う。

2.5.1 処理能力の比較

マイクロ同期機構を全てハードウェアの回路として実現した場合、例えばEM-4のマイクロ同期機構だと、一回の同期を3クロック程度で処理してしまう。さらに、同期処理と命令実行を重畳化することで[1]、同期にかかる時間は無視できる程度になると思われる。

[1]EM-4ではしていない

一方、特殊命令を用いた実現方法では、例えばTSBのような命令が3～4クロック程度で処理されるようにハードを作り込めば、同期処理全体にかかる時間も5～6クロック程度に納めることができる。ただし、命令によってハードが起動されるため、命令実行と同期処理の重畳化はできない。

完全にソフトウェアで同期を記述する場合は、前述の通り一回の同期処理に十数クロックを要するので、速度的にはかなりの損失となる。高い柔軟性が要求される処理に対しては有効だが、マイクロ同期のように単純な処理をソフトウェアだけで実現するのは、処理能力の点からかなり不利であると言える。

2.5.2 実現容易性について

実現の容易性について考える。第一の問題はハードウェアのコストである。完全にハードウェアで実現した場合、同期系と命令実行系とで別々にメモリ(キャッシュ)インタフェースが必要になる。これはオーバヘッドが大きい。一方、完全にソフトウェアで実現した場合は追加ハードウェアの必要はない。ただしこの場合、前述の通り効率を著しく損なう。特殊命令を用いる場合、フラグのテストや分岐などのためのハードウェアが必要になるが、これは量・複雑さともに小さく、問題にならない。また、メモリ(キャッシュ)インタフェースは単一でよく、最も有利と考えられる。

第二番目に問題点になるのは、OSとのインタフェースである。特に、待ち合わせメモリへのアクセスがページフォルトした場合について考えてみると、処理の手順は、

(1) 待ち合わせメモリへのアクセスをサスペンドする。
(2) ページを入れ換える(I/Oへ要求)
(3) I/O待ちの間、別のスレッドを実行する。
(4) ページインが終了して割り込みが上がる。
(5) 現在実行中のスレッドをサスペンドする。
(6) 待ち合わせメモリへのアクセスを再実行する。
(7) サスペンド中のスレッドを復帰。

となる。ここで、完全にハードウェアで実現した場合、OSの介在なしで(2)～(6)を実現するのはかなり困難であると思われる。ここではRICAの思想より、マイクロ同期機構は単純で実装規模の大きくないものを想定しているので、実質的には不可能と言わざるを得ない。次に同じく完全にハードウェアの場合で、ページ入れ換え、およびスレッド切り替えをOSに委ねる場合を考えてみると、これは先の例に比べればかなり現実性が高い。しかし、仮に命令実行部と同期機構とが独立していて、命令実行と同期処理が重畳化されている場合を考えると、OSの処理としては

- 現在実行中のスレッドのサスペンド

- 待ち合わせメモリへのアクセスのサスペンド
- I/O へのページ入れ換え依頼

となって、同時に2つの処理をサスペンドしてしまうので、その復帰が複雑になる。これは基本的に解決できない問題ではないが、OS 側にも負担をかけるし、ハード側にもそれなりの回路規模が必要である。もし命令実行と同期処理を重畳化しないならば、このような問題は生じないが、逆にこの場合は全てをハードウェア化した利点が薄れてしまう。

一方、完全にソフトウェアで実現する場合、および特殊命令を用いる場合については、すべて命令で同期処理を起動するので、上述のような問題は生じない。どちらの場合も、OS とのインタフェースは比較的取りやすい。前述のとおり、特殊命令による実現方法では、完全ハードウェアによるものに比べ高速とは言えないが、この処理能力の差を実装規模や実現容易性の差と比べると、容認できる程度のものであると言える。

以上をまとめると、RICA における単純で高速なマイクロ同期機構の実現を考えた場合、完全にハードウェアだけで実現するには処理が複雑過ぎ、また完全にソフトウェアによる実現では効率が悪すぎる。したがって Test&Set&Branch のような特殊命令による実現が、もっとも合理的であると言える。

3 RWC-1 の同期機構

本節では、前節での検討結果を踏まえ、RWC-1 に実装する同期機構について述べる。

RWC-1 では、スレッドの起動はパケットの到着により行われる。網から取り込まれたパケットは、一度パケットキューに貯えられたあと、ハードの処理で直接レジスタ上にロードされ、これと同時にスレッドが起動される。マイクロ同期は、通常スレッドの先頭で行われ、待ち合わせに失敗したメッセージはメモリ上に退避される。待ち合わせ時のアドレシング形式は、待ち合わせアドレスと命令アドレスの両方をパケットが保持している EM-5 方式である。

3.1 RWC-1 のマイクロ同期機構

前節で述べた通り、同期処理が高速に行われかつ OS の操作などにも柔軟に対応できるマイクロ同期機構を実現するには、特殊命令を用いてハードウェアのサポートを受けながら実現するのが最も現実的であると思われる。

3.1.1 パケットサイズと待ち合わせフラグ

メモリ上の待ち合わせによる同期を行う場合、待ち合わせメモリの状態を示すための待ち合わせフラグが必要である。待ち合わせフラグは通常、メモリ上に相手がいるかどうかを示す presence bit、相手が右オペランドか左オペランドかを示す L/R bit、の2ビットあればよい。RWC-1 の場合、後述する同期付構造体をサポートするために、Waiting Queue を作っているかどうかを示す Q bit が必要で、合計3ビットにより構成されている。

また、RWC-1 は多ワードのパケットをサポートしているため、メモリ上のパケットのワード数を示すパケットサイズが必要である。さらに、同期処理を特殊命令で実現する場合に、不可分ブロックを形成するための Lock Bit が必要で、これらが図4に示されるごとくメモリ上に書き込まれている。

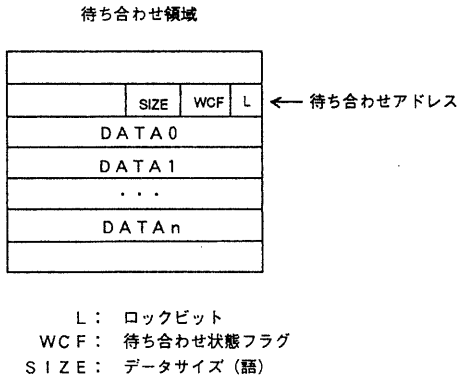


図 4: RWC-1 におけるメモリ上での待ち合わせ

3.1.2 特殊命令

マイクロ同期のために RWC-1 で用意する特殊命令は、以下のとおりである。

(1) BSYNC 命令

2.4 節で述べた Test&Set&Branch 命令を、RWC-1 用に改善した命令である。書式は、

- bsync reg, label

である。動作は、reg が示すレジスタの内容を待ち合わせアドレスとして待ち合わせを行う。最初にロックをかけて、待ち合わせフラグの内容をチェックし、待ち合わせが成立したら label 番地に分岐する。不成功の場合はそのまま。

(2) CLEAR_FLAG 命令

待ち合わせフラグを全てクリアする命令。書式は、

- clear_flag reg

reg が示すレジスタの内容を待ち合わせアドレスとみなし、待ち合わせフラグをクリアする。同時にパケットサイズおよびロックビットもクリアする。

(3) STR_MULT 命令

一般に、複数のデータを連続してメモリに格納する命令であるが、パケットを待ち合わせメモリへストアする場合にも用いる。書式は

- str_mult reg

regが示すレジスタの内容を待ち合わせアドレスとして、待ち合わせに失敗したパケットを退避する。この時、パケットが持っている待ち合わせフラグ、パケットサイズも同時に書き込む。

(4) LOAD_MULT 命令

一般に、メモリから複数のデータをレジスタにロードする命令であるが、待ち合わせメモリから、相手パケットをレジスタファイル上にロードする場合にも用いる。書式は

- load_mult reg

regが示すレジスタの内容を待ち合わせアドレスとみなし、待ち合わせの相手をサイズ分だけレジスタファイルにロードしてくる。

(5) その他

不可分ブロックを制御するために、Lock Bitだけのset/resetをするlock/unlock命令が必要である。

3.1.3 マイクロ同期機構の実現

RWC-1のマイクロ同期は、以下のようにして実現される。

到着したパケットは、ハードウェアによってレジスタファイル上に自動ロードされる。この時、パケットが持っていた命令アドレス、待ち合わせアドレス、パケットサイズ、待ち合わせフラグは、それぞれIAレジスタ(プログラムカウンタ)、OAレジスタ(ベースレジスタ)、SIZEレジスタ、WCFレジスタにロードされる。これと同時に、スレッドが起動される。スレッドの先頭には、次のような処理が埋め込まれている。

```
start:  bsync       OA, fire
        str_mult    OA
        unlock      OA
        break
fire:   load_mult   OA
        clear_flag  OA
```

図5: 特殊命令によるマイクロ同期

本命令列により、2入力の同期処理は5～6クロック程度で実行される。

3.2 同期付構造体

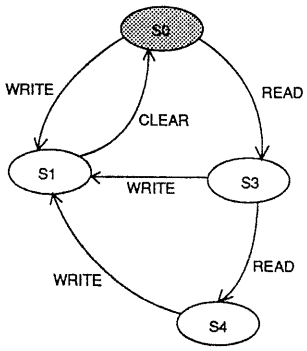
細粒度並列演算において、例えばWavefront問題のように一つの演算結果が次の演算へとどんどん伝搬していくような場合、I-structureと呼ばれる同期付構造体が有効であることが知られている。さらにオブジェクト指向の分野でよく見られるように、生産者-消費者の関係が一対一で対応するような場合、I-structureを拡張したQ-structureと呼ばれる同期付構造体が有効であると言われている[10]。これらの同期付構造体も、RICAに基づくマイクロ同期機構を利用することで実現可能であり、すでにEM-4上にはソフトウェア的に実装されている。

RWC-1でも先に述べたbsync命令を用いて同期付構造体を実現することは可能であるが、実際には処理効率の向上を目指し、専用の特殊命令を追加してハードウェアからのサポートを強化する。

ここで追加する命令は、

- I-access 命令
- Q-access 命令

の2つである。これらはいずれもbsync命令と同様にTest&Set&Branch命令の拡張であるが、bsync命令との違いは待ち合わせフラグの状態による分岐先が複数になっている点である。通常、同期付構造体において左対左(readの連続)、もしくは右対右(writeの連続)のような組合せで入力された場合、キューの操作をしなければならないが、bsync命令ではこうした入力がマッチングエラーとなり例外を発生してしまう。この場合キュー操作ごとに処理がkernel渡るので、処理がかなり重くなる。そこでこれを回避して、キュー操作をユーザレベルのまま行うことを目的とした特殊命令が、I-access、およびQ-accessである。図6に、I-structureの状態遷移図を示す。



S0: 相手なし (Empty)
S1: WRITER在
S3: READER在 (キューなし)
S4: READER在 (キュー有り)

図6: I-structureの状態遷移図

3.3 バリア同期

バリア同期については、マイクロ同期をツリー状に展開することによって、ソフト的に実現することが可能である。一般に、バリア同期をソフトで実現すると、柔軟であるかわりに低速になるとされているが、十分高速なマイクロ同期機構が用意されていれば、例えばデータパラレルの応用に耐えられるだけの十分な性能が得られることが報告されている[11]。

また、ソフト的にバリア同期を実現する利点は、全同期だけでなく任意の組合せの同期が自由にとれることである。ハード的にこれを実現しようとすると、カウンタなどを含めた特殊な回路が必要になるが、高速なマイクロ同期機構さえ持っていればこうした回路は不要で、ソフトによる組合せだけですむ。これはRICAにおけるマイクロ同期の大きな長所である。

一方、このバリア同期機構では、全体を止めることは可能であるが、全体をまったく同時に再開させるようなことができない。したがって、クロックレベルで厳密に処理を同期させたい場合に、このバリア同期機構ではサポートすることができない。そこで、I/O側の通信網を利用して厳密な全同期をかけることを、現在検討中である。

4 おわりに

本稿では、超並列計算機RWC-1の同期機構について述べた。RICAに基づく高速なマイクロ同期機構が、種々の同期処理の実現を可能にすることを示し、そのマイクロ同期機構の実現方法を検討した。その結果、処理速度と柔軟性、OSのサポートなどの観点から、特殊命令を導入し、ハードウェアの支援を受けながら命令で同期を組み上げていく実現方法が、もっとも現実的であることが明らかになった。

また、RWC-1で採用する特殊命令について述べ、同期付構造体やバリア同期の実現方法についても初期的検討を行った。

謝辞

本研究を遂行するにあたり、有益な御指導、御討論をいただいた島田つくば研究所長、古谷超並列・ニューロ研究部長、石川超並列ソフトウェア研究室長、超並列ソフトウェア研究室員の諸氏、ならびにRWC超並列アーキテクチャWGの諸氏に感謝いたします。

参考文献

[1] 坂井修一、岡本一晃、松岡浩司、廣野英雄、児玉祐悦、佐藤三久、横田隆史、超並列計算機RWC-1の基本構想、並列処理シンポジウムJSPP'93, pp.87-94 (1993).

[2] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama and Mitsuhisa Sato, Super-Threading: Architectural and Software Mechanisms for Optimizing Parallel Computation, to appear in Proceedings of International Conference on Supercomputing (1993).

[3] R.Gupta, The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors, Proc. Third Int. Conf. ASPLOS, pp.54-63 (1989).

[4] 松本、Elastic Barrier: 一般化されたバリア型同期機構、情報処理学会論文誌, Vol.32, No.7, pp.886-896 (1991).

[5] 高木、有田、曽和、細粒度並列実行を支援する種々の静的順序制御方式の定量的評価、並列処理シンポジウムJSPP'91, 269-276 (1991).

[6] Thinking Machines Corporation, Connection Machine CM-5 Technical Summary, November, 1992.

[7] K. Hiraki, S. Sekiguchi and T. Shimada: System Architecture of a Dataflow Supercomputer, Proc. of TENCON87, pp.1044-1049 (1987).

[8] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama and Toshitsugu Yuba, An Architecture of a Datalfow Single Chip Processor, Proceedings of 15th International Symposium on Computer Architecture, pp.46-53 (1989).

[9] Shuichi Sakai, Yuetsu Kodama and Yoshinori Yamaguchi, Architectural Design of a Parallel Supercomputer EM-5, Proceedings of JSPP'91, pp.149-156 (1991).

[10] 佐藤三久、児玉祐悦、坂井修一、山口喜教、並列計算機EM-4における分散データ構造を用いたマルチスレッドプログラミング、情報処理学会計算機アーキテクチャ研究会 92-7 (1992).

[11] Andrew Shaw, Yuetsu Kodama, Mitsuhisa Sato, Shuichi Sakai and Yoshinori Yamaguchi, Data-Parallel Programming on the EM-4 Dataflow Parallel Supercomputer, Proceedings of Frontiers '92, pp.302-309 (1992).