細粒度通信に基づく並列計算機アーキテクチャに関する研究

平成14年 12月

児玉 祐悦

概要

大規模シミュレーションや大規模サーバなどにおいて、計算機に対する高性能化の要求は今後もますます増大していくと考えられる。高性能化のためには、プロセッサ単体の性能を向上させるとともに、並列化を行うことが不可欠である。しかし、単体性能にプロセッサ数をかけて得られるピーク性能に比べて、実際に並列計算機で実行した場合の実効性能は低いものとなっている。一般的に、この並列化オーバヘッドはプロセッサ数が大きくなると増大する傾向にある。これまでは、この並列化オーバヘッドを軽減するために、通信のスループットを増大させるとともに、計算時間と通信時間のオーバラップによる通信時間の隠蔽を図ってきた。しかし、十分な通信スループットおよび通信時間の隠蔽を実現するためには、一回の通信量を大きくするようにプログラムを書き変える必要がある。そのため、そのような変更が可能な粗粒度並列アプリケーションしか十分な並列化の効果が得られなかった。より容易に、かつより広範囲に並列オーバヘッドを軽減し、並列効率を高めるためには、複数のスレッドを切り変えてプロセッサの実行効率を高めるマルチスレッド技術が有効である。このマルチスレッド実行の効果を高めるためには、スレッド切り替えおよび通信セットアップにかかる時間を軽減して、より多くの時間を本来のスレッド実行に割り当てられるようにすることが重要である。

マルチスレッド技術には、1) 逐次的に実行されるスレッドを適切に切り替えながら実行するシンプルマルチスレッド方式、2) サイクル毎に異なるスレッドからの命令を実行する細粒度マルチスレッド方式、3) 複数の演算ユニットを持ち、複数のスレッドからの命令を同時に実行する同時マルチスレッド方式等がある。シンプルマルチスレッド方式は、スレッド切り替えのタイミングがより細かな単位 (数十から数百クロックサイクル) を想定していることが多く、スレッド切り替えやスレッドスケジューリング、スレッド間の通信や同期などにハードウェアサポートが必要である。細粒度マルチスレッド方式は古くは HEP というマシンで提案され、現在 TERA 社 (現 CRAY 社) により MTA というスーパーコンピュータとして実用化されている。同時マルチスレッドは比較的新しい方式で、スーパースカラ方式では演算ユニットを増加させても命令レベル並列性を十分には引き出せないことに対する解決策として提案された。

このようなマルチスレッド計算機を考える場合に、スレッドの生成をプログラム実行中に可能とする動的スレッド生成を基本としスレッド数に制限を設けないようにするか、プログラム開始時に静的に一定個数のスレッドを生成する方式とするかが大きな設計方針となる。スレッドコンテキストをハードウェアで保持する細粒度マルチスレッド方式や同時マルチスレッド方式では、スレッド数がハードウェアリソースによって制限されるため、スレッド数が増大する可能性のある動的スレッド生成を採用することは難しい。しかし、本論文では粗粒度並列処理ではうまく並列化できないようなプログラムを並列化して高速化することを目指しており、効率的な動的スレッド生成を可能とするとともに、スレッド数へは特に制限を設けないことを目指している。このため、スレッドコンテキストをメモリ上に持つシンプルマルチスレッド方式を基本として設計指針の検討を行う。また、本論文では千台規模の並列計算機を想定している。この場合、各ノードは 1 チップ+メモリ程度に集約する必要がある。本研究を開始した当時の 1 チップのゲート規模からは、複数スレッドコンテキストをハードウェア的に保持することは困難であった。このようにシングルチップにネットワーク機構とプロセッサ機構を登載するという制約から、各機構をシンプルな構成に限る必要がある。このハードウェアを頻繁に利用する機能に限り、それ以外の機能は最適化したプログラムにより処理するという考えは、RISC プロセッサアーキテクチャに通じるものであり、本アーキテクチャ設計でも重要な指針となっている。我々はこのような設計指針に基づき、世界に先駆けてシングルチッププロセッサによるシンプルマルチスレッド方式の高並列計算機 EM-4 を先に開発した。本論文はこの EM-4 の評価に基づき、さらにスレッド実効性能を高めるとともに、共有メモリアクセスの実行性能を飛躍的に向上させたアーキテクチャについての提案とその評価についてまとめたものである。

本論文では、レジスタの内容から直接パケットを生成する細粒度通信機構を用いて通信オーバヘッド

を削減するとともに、パケットの到着に基づきハードウェアで直接スレッドを起動することによりスレッド起動オーバヘッドを軽減したマルチスレッドアーキテクチャを提案する。本アーキテクチャでは、直接リモートメモリアクセス機構、局所同期機構などをハードウェアでサポートすることにより、メッセージ通信型プログラミングから共有メモリ型プログラミングまで柔軟なプログラミングが可能である。その際、ハードウェアとコンパイラの協調による処理の最適化に着目した実装を行っている。例えば、スレッドの切り替えタイミングを通信レイテンシが起きる時点としており、コンパイラがその切り替えタイミングを自動的に検出し、スレッド切り替え時に本当に必要なレジスタのみを待避/復帰するという最適化を行っている。これにより、スレッド切り替えのオーバヘッドを削減している。また、基本的なスレッドライブラリなどはハードウェアのサポートにより数命令で実現できるため、インライン展開を適用することにより、関数呼び出しのオーバヘッドを削減している。

本提案の細粒度通信に基づくマルチスレッドアーキテクチャを実装した80プロセッサからなるプロトタイプ計算機EM-Xを構築した。この要素プロセッサとして、ネットワーク機構とプロセッサ機構を含めてシングルチップ化したプロセッサEMC-YをASICにより開発した。本プロセッサを5個登載するプロセッサボードを16枚接続して80プロセッサシステムを構築している。この他、ホスト計算機との接続を行うインタフェースボード、ビデオ信号の入力や計算結果の出力を行うフレームバッファボードを開発した。これらのボードにはパケット処理用にEMC-Yが登載されており、プロセッサボード間を接続するケーブルの間に挿入する形で容易にシステムの拡張が可能である。また、それらとの通信は、プロセッサ間の通信と同様に細粒度パケットを用いた低レイテンシかつ高スループットな通信が可能である。本プロトタイプ計算機では独自のプロセッサを用いているため、コンパイラを始めとするソフトウェア環境も独自に開発を行った。ブロック転送、バリア同期／リダクション処理、ブロードキャスト、実行トレースなどのライブラリを専用に開発することにより、プログラムからハードウェア機構を有効に利用可能である。

提案したアーキテクチャの有効性を示すために、開発したプロトタイプ計算機を用いて各種ベンチマークによる性能評価を行った。最初に、並列プリミティブの評価を行い、リモートメモリアクセスの静的レイテンシが平均1.3マイクロ秒、2点間スループットが1Kバイト程度の小さなブロックで35Mバイト/秒と、低レイテンシと高スループットを両立していることを確認した。実行時レイテンシの評価では、全プロセッサが動作しネットワーク上に200Mバイト/秒のデータが流れている状況でも、直接リモートメモリアクセスのレイテンシが2マイクロ秒程度であり、実行時レイテンシが極めて低く抑えられていることを確認した。バリア同期の評価では、局所同期を組み合わせたソフトウェアによる実装であるにもかかわらず、80プロセッサのバリア同期が13マイクロ秒で行えることを確認した。また、カーネルベンチマークとして、行列乗算による評価では、小さい配列サイズから高い並列性能を達成できることを示すとともに、ナップサック問題や三角方程式の評価では、これまで並列化が難しいと考えられていた問題に関しても並列性能を引き出せることを示した。さらに、より大きなマクロベンチマークとして粒子シミュレーションMP3Dとradixソート用いてEM-Xの全体性能について評価を行い、マルチスレッド処理によるレイテンシ隠蔽の有効性や、細粒度通信によるネットワーク負荷の平均化の有効性などを示した。

従来、並列処理おいて性能向上を図るために通信粒度を大きくして通信スループットを向上させる手法が多かったが、本研究の成果によれば、シンプルであるが命令実行パイプラインと密接に融合した適切なハードウェアサポートにより、細粒度な通信のままでもそのレイテンシを削減/隠蔽することにより並列処理性能を向上させることが可能であることを示した。細粒度通信では、わざわざ通信を粗粒度にまとめることが不要であり、より細粒度な処理でも並列処理効果が期待されるため、並列処理の適用範囲を拡大することが可能となる。また、粗粒度通信では各プロセッサが一斉に通信を行うために通信の衝突による性能低下が引き起こされるが、細粒度通信では通信が平均的に散らばるためにネットワークの負荷が平均化され通信の衝突の影響が軽減されるという利点も見られた。

Abstract

The demand for high performance computing will increase more and more in the future in a large scale simulation and a large-scale server, etc. Parallelization of computer and improvement of the performance of the single processor are indispensable to achieve high performance. But actual performance on parallel computer is lower than the peak performance, that is the multiplication of the number of processors and the performance of the single processor. In general, the parallel overhead tends to increase when the number of processors increases. To reduce the overhead of communication, conventional parallel computer improves the communication throughput and hides the communication time by overlapping with computation time. However, it is necessary to rewrite the program so that one communication may grow to achieve enough throughput and to hide the communication time. And the effect of parallelization is limited to the application that can use the coarse-grained communication. Multithreading is effective in order to reduce a parallel overhead more easily and more widely. In multithreading, threads are switched in order to improve the efficiency of the processor. It is important to reduce the overhead of switching thread and the setup time of communication in order to allocate more time to the native thread execution.

Multithreading is mainly divided to three categories; 1) simple multithreading is that a thread is sequentially executed at a time, and multiple threads are switched one after another in appropriate timing. 2) fine-grain multithreading is that the instruction from a different thread is issued in every cycle. and 3) simultaneous multithreading is that it has multiple execution units and the instructions from multiple threads are executed at the same time. Simple multithreading is a similar image as multitasking or multithreading on operating system. But the grain size of the thread is very fine, and it runs about from tens to hundreds of clock cycles at a time. Therefore, the hardware support is necessary for communicating, switching, scheduling and synchronizing threads. Fine-grain multithreading was proposed with the machine named HEP about 20 years ago. It has been commercialized as a supercomputer named MTA by the TERA Computer Co. (present CRAY Inc.) now. Simultaneous multithreading is a comparatively new method. It was proposed as a solution for limitation of instruction-level parallelism, that is the performance is not scalable to the increase of computing unit in super-scalar processor. It was adopted in a commercial processor as Hyper-Thread by Intel Co. recently.

Thread generation policy is main design decision in multithread computer. Static generation policy assumes that it statically generates a constant number of threads at the beginning of the program. Dynamic generation policy assumes that there is no limitation in the number of threads and it generates threads dynamically in program execution. Fine-grain multithreading and simultaneous multithreading are difficult to adopt the dynamic thread generation with the possibility that the number of threads increases because the number of threads is limited by the hardware resource in their methods. Since we aimed to parallelize the programs, that could not parallelize well by the coarse-grain parallel processing, and to speed up them, we assumed that the efficient, dynamic thread generation was enabled, and the limitation was not installed in the number of threads especially. Therefore, the design in this paper is based on the simple multithreading with thread context in the memory. Moreover, the number of nodes on parallel computer in this theses assumed over one thousand. In this case, one node must be integrated as simple as the single-chip and memory. The number of gates was not enough to hold multiple thread context by hardware at the beginning of this research. It was also necessary to keep each component simple from the restriction of integrating the network and the processor in

a single chip. Only the functions to often be used are implemented by hardware and other functions are processed by the optimized program. It is the design decision of RISC architecture. And it was extended to the parallel architecture in this proposed architecture. We previously developed a highly parallel computer EM-4 by the single chip processor in advance of the world based on the design decision. This thesis proposes the architecture that improves thread processing and enables shared memory access based on the evaluation of the EM-4, and summarizes its evaluation.

This thesis proposes the multithread architecture that reduces the overhead of communication by using the fine-grain communication mechanism which generates the packet directly from the content of registers. Moreover, this architecture reduces the overhead of thread invocation by starting the thread directly with hardware based on the arrival of the packet. This architecture supports flexible programming from the message passing programming to the shared memory programming by the support of a direct remote memory access mechanism and a local synchronous mechanism with hardware. In the programming execution environment, it is implemented by the cooperation of hardware and the compiler from the viewpoint of optimizing the process. For example, the compiler automatically detects the switch timing of the thread where the communication latency occurs. It also detects really necessary registers to be stored to the thread context on the memory when the thread is switched in order to reduce the overhead of switching thread. Moreover, basic thread libraries consist of several instructions by the support of hardware, and the overhead of the library call is reduced by expanding the function to in-line.

The prototype computer EM-X was developed based on the fine-grain communication of this proposal. It consists of 80 processors and supports the multithread architecture. The processor named EMC-Y was designed as the processing element of the prototype using ASIC. It includes the network mechanism and the processor mechanism in a single chip. The prototype with 80 processor consists of 16 processor boards, that includes five EMC-Y processors. The prototype also includes the interface board that is connected with the host computer and the frame buffer board that inputs the video signal as the application input and outputs it as the calculation results. These boards includes a EMC-Y for the packet processing, and the system can be expanded easily by inserting other I/O board between cables which connect between processor boards. And the fine-grain packet communication, that is low latency and high throughput, is also used for the communication between I/O boards as well as the communication between processors. The programming environment includes the compiler and operating system also originally developed because this prototype uses an original processor. The libraries for exclusive use are developed so that hardware function can be effectively used. They includes block transfer, barrier synchronization, reduction, broadcast, and execution trace, etc.

To show the effectiveness of the proposed architecture, we evaluated its performance using various benchmarks on the prototype. First, parallel primitiveness were evaluated. The results showed that the static latency of a remote memory access was 1.3 micro-seconds on the average, and the throughput between two processors was 35MB/seconds in 1KByte data transfer. It achieved both low latency and high throughput. The evaluation of dynamic access latency showed that the direct remote memory access was efficient under the situation where all processors communicated with each other and the total network traffic was over 200MB/seconds. The evaluation of barrier synchronization showed that it takes only 13 micro-seconds in the case of 80 processors, while it was the software library using the combination of local synchronizations. Second, the kernel benchmarks were evaluated. The evaluation of matrix multiplication benchmark showed its efficient parallel performance even if the array size was small. In the evaluation of knapsack benchmark and triangle equation benchmark, the proposed architecture could achieve good parallel performance with the problem that was previously difficult

to be parallelized. Last, the MP3D particle simulation benchmark and radix sorting benchmark were evaluated as application benchmarks to show the total performance of EM-X. The results showed that multithreading could hide the communication latency effectively. They also showed that the fine-grain communication made the network load flat and reduced the network hot-spot.

The communication grain size was enlarged to achieve the performance improvement in the previous parallel processing using the high communication throughput. However, this research showed that the fine-grain communication could reduce and hide the communication latency, and improve the parallel performance using simple hardware support that closely fused with the instruction execution pipeline. Since the fine-grain communication don't have to bring together the communication to coarse grain and it can parallelize the program even if they are fine-grain processing, it can enlarge the parallel application field. Moreover, the performance was degraded because of the collision of the communication in the coarse-grain communication, since each processor may communicate all together. The fine-grain communication can keep the network load flat and the influence of the collision of the communication is reduced.

目 次

図目次

表 目 次

第1章　はじめに

ＶＬＳＩ技術・システム技術の進歩により近年の計算機の性能向上は著しいが、さらなる高性能化への期待はますます大きくなっている。科学技術計算における大規模シミュレーションでは、より現実的な問題に適用するためには現在よりも問題規模を 10 倍程度にする必要があり、その場合に必要な計算規模は現在の $10^3 - 10^4$ 倍と膨大な量となる。また、Web アクセスなどに代表される分散環境においても、利用者/利用分野/利用環境が拡大するにつれて、サーバに対する負荷が増大しており、それに対応するために計算機性能に対する要求は増大している。

現在、高性能計算機を構築するためには、プロセッサ単体の性能を向上させるとともに、並列化による台数効果を得ることが必要不可欠である。これらは、一方のみでは不十分である。なぜなら、プロセッサ単体の性能はデバイス技術の向上やアーキテクチャ技術の工夫により、18ヶ月から 24ヶ月で 2 倍という着実な進歩を果たしてきているが、計算機に求められる性能はそれを大きく越えており、並列化は不可欠の技術である。それに対し、並列化によれば、単体性能にプロセッサ数をかけて得られるピーク性能を数百倍にすることは比較的容易である。しかし、実際に並列計算機で実行した場合の実効性能をピーク性能に近づけることはプロセッサ数が大きくなるにつれて難しくなる。このため、できるだけ少ないプロセッサ数で構成することが望ましく、個々の要素プロセッサの高速化が不可欠である。さらに近年では単位容積あたりの消費電力の限界がささやかれており、単純に高速なプロセッサを多数接続すれば高速化されるというほど単純ではなくなっている。したがって、高性能なプロセッサを高並列化することが目指す方向ではあるが、その実効性能をピーク性能にいかに近づけられるかが根本的な問題となっている。

実効性能を低下させる原因としては、以下のような原因が考えられる。

1. キャッシュミスペナルティの増大
2. 命令レベル並列の限界
3. 並列処理オーバヘッド

1. と 2. はプロセッサ単体での問題であり、3. は並列計算機における問題と言える。近年ではシングルチップに複数のプロセッサが搭載されるようになってきており、共有キャッシュメモリの問題など、単純にプロセッサ単体と並列システムとに分離できない問題も多くなってきているが、本論文では特に 3. の並列処理オーバヘッドによる実効性能の低下について焦点をあてて考察していく。

計算機の高速化のためには、並列化が不可欠であるのに対し、プロセッサ側の並列化に対するサポートはまだまだ限られたものである。汎用プロセッサでは、SMP (Synmetric MultiProcessor, 対称型マルチプロセッサ) 向けに共有キャッシュバスプロトコルを採用した程度のものがほとんどである。最近、小規模 SMP 向けの通信路をプロセッサチップに持たせたプロセッサがいくつか出てきているが、それらも SMP 向けのメモリプロトコルに限られているもので、高速な I/O という範疇にとどまる。SMP では一様なメモリ参照を提供するために、メモリは共有バス (あるいはネットワーク) の先に接続される。シングルプロセッサでもメモリウォールと呼ばれて、プロセッサとメモリとの性能差が大きくなっていることが全体の性能向上の妨げになっている、と指摘されている。SMP ではその差がより大きくなってしまう。そのため、大規模な SMP は限られた用途にのみ利用され、多くは 2-4 プロセッサ程度の小規模な SMP となっている。

それに対し、各プロセッサがローカルなメモリを持つことによりプロセッサ単体のメモリ性能を維持しつつ、他のプロセッサからのメモリアクセス要求にも答える方式として DSM(Distributed Shared Memory, 分散共有メモリ) がある。DSM でもより多くのプロセッサを接続して性能を向上させるためのスケーラビリティが重要である。スケーラビリティを確保する上で、システム構成の柔軟性と、高い処理性能という相反する要求を満足させる必要がある。当初は SMP と同じく、メモリコンシステンシの維持をできるだけローレベルで実現しようとして、ディレクトリベースのメモリコンシステンシのためのハードウェア機構を持つ並列計算機が多く開発された。しかし、あらゆるアプリケーションのメモ

リアクセスパターンで高性能を維持するのは困難であった。メモリコンシステンシの維持により共有メモリプログラムを移植することは容易であったが、メモリ階層の違いを考慮した性能チューニングを共有メモリプログラミングのまま行うことは困難であり、性能を追求するとメッセージ通信ライブラリを陽に利用するということが行われた。また、メモリコンシステンシ維持のハードウェアは複雑なものとなり、開発期間が長くなり性能への要望を満たすことが困難となってきたということもある。

大規模な並列性を考えた場合にはメモリコンシステンシにこだわらずに、メモリは各プロセッサローカルなものとして、各プロセッサ間は通信というよりハードウェアのイメージに近い形でデータのやり取りを行うメッセージ通信パラダイムに基づいた並列計算機も数多く開発された。DSM に比べてハードウェアも簡単であるため、並列計算機開発の当初はこの方式のものが多かった。しかし、当初はメッセージ通信の方式が各並列計算機毎に異なり、プログラムの開発容易性/移植性の面でなかなか一般化しなかった。その後、MPI 通信ライブラリがメッセージ通信方式のデファクトスタンダードとなることにより、並列プログラムの移植性の問題は改善された。共有メモリパラダイムとメッセージ通信パラダイムは、それぞれの得失が当初いろいろ議論されたが、ハードウェアレベルでは相反するものではなく、両者を統合した並列計算機を開発し、ソフトウェアレベルでアプリケーションに応じてプログラミングパラダイムを使い分ける、という考えも広がっている。この場合、ハードウェアレベルでは、リモートメモリ DMA と呼ばれる異なるプロセッサのメモリ間でネットワークを介した DMA によるブロック転送機能を基本としていることが多い。この機能を基に、メッセージ通信や共有メモリをソフトウェアレベル (OS あるいはユーザプログラム) で実現している。さらに、従来は専用のプロセッサボートと専用のネットワークを用いることが多かったが、汎用の単体あるいは小規模 SMP のプロセッサボードと汎用のネットワークを用いたクラスタ型計算機も、近年の単体プロセッサおよびネットワークインタフェースの高速化に伴い広く使われてきている。

並列化というものが性能向上に不可欠なものであり、それに要求される性能が他の I/O 性能とは異なり、メモリ性能と比べるべきものとするならば、より積極的にプロセッサアーキテクチャレベル、すなわち命令実行パイプラインレベルでの並列化のサポートが必要であると考える。ここで、並列化のサポートには、通信処理およびそれに伴うレイテンシの隠蔽等が考えられる。これまで通信処理は低速な I/O 処理として実現されており、その低速性をカバーするためにリモートメモリ DMA を用いたスループットの向上が主に行われてきた。そのため、いったん DMA が起動されれば、メモリ速度に応じた高速化が行われているが、その起動までの時間はほとんど改善されず、十分なスループットを出すためには、通信オーバヘッドを相対的に小さくするために、一度に転送するデータサイズを大きくする必要があった。このため、効率のよい並列化が行えるのは、粗粒度的なデータ並列性を持つアプリケーションに限られていた。

通信時間は $L+T\times S$ で表わさせる。ここで、L は通信のセットアップにかかる時間、T は単位データあたりにかかる時間 (スループット)、S は転送するデータサイズである。これまでは、通信を高速化するためにスループット T を向上させてきた。L が余り改良されないとすると、相対的に L のオーバヘッドが大きくなるため、それを補うためにより大きな S が必要となる。ここでは、S を大きくするのではなく、L を削減するとともに、通信時間を他の処理とオーバラップさせて隠蔽する方式について考える。

粗粒度データ転送でも、計算期間と通信期間のオーバラップによる通信時間の隠蔽は高性能化では不可欠の技術である。しかし、そのような通信時間の隠蔽を実現するには、十分なスループットを得るときと同じように、一回の通信サイズを大きくするようにプログラムを書き変える必要があった。そのため、そのような変更が可能な粗粒度並列アプリケーションでしか計算と通信のオーバラップの効果が得られなかった。より容易に、かつより広範囲に並列オーバヘッドを軽減し並列効率を高める方式として、複数のスレッドを切り変えてプロセッサの実行効率を高めるマルチスレッド技術がある。このマルチスレッドの効果を高めるためには、スレッド切り替えおよび通信セットアップにかかる時間を軽減して、より多くの時間を本来のスレッド実行に割り当てられるようにすることが重要である。

本論文では、コンパクトで単純化したアーキテクチャによって高い実行性能を持つマルチスレッド計算機を提案する。特に、全てをハードウェア化して高速化するのではなく、適切なハードウェアとソフトウェアの協調のバランスポイントを探り、少ないハードウェア規模で十分な並列効率を得られるようなアーキテクチャを考える。

第2章では、これまでの並列アーキテクチャ、特に通信時間の隠蔽を目的としたアーキテクチャについて概観し、それらの問題点について考察する。ここで、本論文で提案するアーキテクチャを発表してから今日までやや時間がたっているため、提案時以前、プロトタイプ開発中、その後に分けて考察を行う。第3章で、細粒度通信に基づくマルチスレッドアーキテクチャを提案するとともに、その特徴である1) マルチスレッド機構、2) 細粒度通信機構、3) 直接メモリアクセス機構について述べる。第4章で、提案するアーキテクチャに基づいて開発したプロトタイプ計算機EM-Xについて、そのハードウェアおよびソフトウェアの実装について述べる。特に本プロトタイプのためにASICにより開発したシングルチッププロセッサEMC-Yについてその詳細を述べるとともに、EM-X向けに開発された各種ライブラリについて、その特徴や工夫について述べる。第5章で、そのプロトタイプ実機を用いた評価を行い、アーキテクチャの有効性について考察を行う。評価としては、アーキテクチャの個々の特性を評価するためのマイクロベンチマーク、その個々の特性がどのように並列性能に貢献するかを見るために比較的小さなプログラムを用いてその実行状況の詳細を評価するカーネルベンチマーク、そして、プログラム全体の性能を評価するマクロベンチマークの3種類に分けて述べる。第6章で、ベンチマークによる評価をもとに、提案するアーキテクチャの性能のバランスや、他の計算機への適用可能性などについて考察を行う。第7章で、それまでの議論をまとめ、本研究の結論と展望について述べる。

第2章　レイテンシ隠蔽の各種方式

本章では、これまでの並列アーキテクチャ、特に通信時間の隠蔽を実現しているアーキテクチャを概観し、それらの問題点について考察する。ここで、本論文で提案するアーキテクチャを発表してから今日までやや時間たっているため、提案時以前 (1990 年以前)、プロトタイプ開発中 (1990 年代中盤)、その後から現在 (1990 年代後半以降) に分けて考察を行う。

2.1 1990 年以前の通信時間隠蔽の方式

オペランドがそろった命令から動的にスケジューリングすることにより、命令レベルの並列性を自動的に抽出できるアーキテクチャとしてデータフローアーキテクチャ[Den75, Den83, Gur85] がある。豊富な並列性を動的スケジューリングすることにより、通信時間の隠蔽を実現している。データフロー計算機は並列処理に向いたアーキテクチャとして古くから研究されてきたが、実際に大規模なプロトタイプとして実現したのは 1980 年代後半の SIGMA-1[Shi86, Hir87, Hir88, 島田 93, 平木 95, 平木 02] や Monsoon[Arv88, Pap90, Pap91] である。

SIGMA-1 は電子技術総合研究所 (電総研、現産業技術総合研究所) で開発された科学技術計算向けデータフロースーパコンピュータで、1983 年に開発が開始され、1988 年に 128 プロセッサのシステムが動作を開始した。SIGMA-1 はタグつきトークンを用いる動的データフローモデルに基づく命令レベルデータフロー計算機である。構成としては、プロセッサユニット 4 台と構造メモリユニット 4 台がクロスバースイッチにより接続されており、それがオメガ網で接続されるという階層型ネットワークとなっている。データフロー計算機では命令実行に必要なオペランドが揃ったかどうかをチェックするマッチングユニットが大きな特徴となるが、それ以前のデータフロー計算機では、連想記憶メモリを用いてマッチング処理を行っており、あまり大規模なシステムを構成できなかった。SIGMA-1 ではマッチングユニットにハッシュチェイン方式を採用し、大規模なシステムを可能にしている。グローバルネットワークに自動負荷分散機能を持たせるとともに、プログラム言語として DFC[島田 88] という単一代入規則を待たせた C 言語を開発し、関数レベルおよび命令レベルの並列性を自動的に抽出させている。

Monsoon はマサチュセッツ工科大学 (MIT) で開発された命令レベルデータフロー計算機であり、1988 年に 1 プロセッサ版のプロトタイプが動作し、最大で 16 プロセッサ版が動作し、各種評価が行われた。Monsoon も動的データフローモデルに基づく命令レベルデータフロー計算機であり、16 ノードが結合網により接続されている。Monsoon では Explict Token Store(ETS)[Pap90] と呼ぶ連想記憶やハッシュ機構を用いないシンプルなマッチング方式を採用している。これはあとで説明する電総研で提案した直接待ち合わせと同様に、通常のメモリ上に待ち合わせフラグを設け、そのフラグの状況により処理を切り替える方式であり、特殊なメモリは不要である。1 プロセッサ版のプロトタイプでは全てのオペランドがパケットマッチングにより供給されなければならなかったが、性能のクリティカルパスとなる逐次処理を高速化する目的で、16 ノード版では各プロセッサに 8 セット ×3 レジスタのテンポラリレジスタを追加している [Pap91]。

これらの命令レベルデータフロー計算機は、命令レベルおよび関数レベルの並列性の抽出の点では非常に効果的であったが、逐次処理におけるパケット循環パイプラインのオーバヘッド、および待ち合わせによるパイプラインバブルの発生による演算ユニットの実行効率の低下等が確認された。このような命令レベルデータフロー計算機の問題点を解決する方法として、データフローと逐次実行とを融合する方式 [Ian88, Nik89] が提案され、それを実現したアーキテクチャとして EM-4[12, 13, 14, 17, 38, 44, 45] や*T[Nik92, Pap93] が提案された。

EM-4 は電総研で開発されたデータフロー計算機である。SIGMA-1 の経験を生かし、逐次処理部分を先行制御型のパイプラインで高速に実行するとともに、マッチング処理を直接待ち合わせ [16] と呼ぶ一般メモリ上の full/empty ビットを用いた方式としてハードウェアの簡略化を実現し、ASIC によるシングルチッププロセッサとして実装を行った。1990 年に 80 プロセッサからなるプロトタイプが動作を開

始している。本論文で提案するアーキテクチャはEM-4の評価[1, 2, 3, 4, 6, 21, 24, 26, 28, 39, 41]を基にしたものである。命令レベルおよびパイプラインレベルの改良等を行っているが、単に各部を改良しただけではない。EM-4ではメモリアドレス空間が特殊目的にセグメント化されていて逐次処理に各種の制約がつけられていたものを、一般化し汎用プロセッサとのスムーズな融合を図るとともに、逐次性能の向上を実現している。また、新たなリモートメモリアクセス向けのハードウェア機構の追加や、待ち合わせ処理の重畳化により、並列処理性能も向上させている。

2.2 1990年代中盤の通信時間隠蔽の方式

本論文で提案する方式と前後して、スレッドレベルの並列性に注目したアーキテクチャがいくつか提案された。

*TはMITでMonsoonの後継として提案されたマルチスレッド並列計算機である。EM-4と同様に、命令の逐次実行シーケンス(スレッド)を並列処理単位とすることにより、動的データフロー計算機とフォンノイマン型逐次処理計算機との融合を図っている。概念的にはリモートメモリアクセスのサービスを行うコプロセッサ、同期処理を行うコプロセッサ、およびスレッド実行を行うプロセッサの組からなり、ローカルメモリを共有する。リモートメモリのサービスについては提案する方式と似ているが、独立したコプロセッサとしていることとプログラマブルになっている点が異る。また、同期処理は同期カウンタ方式を用いており、MonsoonやEM-4よりも簡略化されているが、スレッド実行プロセッサと重畳化することにより同期ミス時のオーバヘッドを隠蔽する点は、提案する方式と似ている。提案時には、実装イメージについては固まっていなかったが、その後、StarT-NG[Chi95], StarT-Voyager[Ang98], StarT-X[Hoe98]等具体的なアーキテクチャが提案されるにつれて、コプロセッサ型からネットワークインタフェースユニットへと変っていった。これはプロセッサ自体に手を加えずに機能を追加する方式として採用されていると思われるが、そのために当初提案していた方向とは離れていった。

Alewife[Aga90, Aga95]はMITで開発された分散共有メモリ型並列計算機である。LimitLESSと呼ぶディレクトリベースのキャッシュコヒーレンス方式を提案している。一方、DMAを用いたメッセージ通信もサポートしており、両者のプログラミングを可能としている。さらに、レジスタウィンドウを用いた高速なコンテキストスイッチを用いたマルチスレッドによりキャッシュミス時のレイテンシ隠蔽を可能にしている。full/emptyビットによる細粒度同期機構も備えるが、ハードウェアによる実装ではリードミス時にトラップを引き起こすため、オーバヘッドが大きい。また、ハードウェアサポートは強力であるが、そのためにプロセッサを除いた機能だけで1チップとなりゲート規模も10万ゲートおよび12Kバイトのメモリが必要となっていた。現在のプロセス技術からみると、このゲート規模は許容できる範囲であるかもしれないが、以下に示すようなチップマルチプロセッサ技術などを考えると、ソフトウェアとの協調などにより、もっとシンプルなハードウェア構成を検討する必要があると思われる。

2.3 1990年代後半以降の通信時間隠蔽の方式

スレッドレベルの並列性に注目した商用計算機が出現している。

TERA社(現Cray社)のMTA[Alv90, Sna98, Bri98, Boi98, Car99]は、複数のスレッドから命令を順に実行する細粒度マルチスレッド方式によるレイテンシに強いアーキテクチャを目指した商用スーパコンピュータである。MTAは通常の計算機のような階層メモリを持たず、プロセッサからDRAMユニットへネットワークにより接続されている。そのため、メモリアクセスレイテンシは140-160サイクルと大きいが、独立したスレッドからの命令実行とハードウェアルックアヘッド機構の組合わせにより、そのような大きいレイテンシに対しても適応できるアーキテクチャとなっている。具体的には、MTAは128スレッドまでのコンテキスト(ストリームと呼ぶ)をハードウェア的に保持している。このストリー

ムにはプログラムカウンタ (PC) と 32 個のレジスタが含まれる。プロセッサはストリームからの命令をラウンドロビンにより公平に実行する。ストリームは 21 サイクルごとに命令を実行できる。すなわち、命令パイプラインの長さが 21 段であり、プロセッサを 100%動作させるためには少なくとも 21 個のストリームが必要である。このため、シングルスレッドの性能は高くない。また、これだけでは 128 サイクル以上あるメモリレイテンシを隠蔽することはできないが、各ストリームは 8 個までのメモリ参照をノンブロッキングで実行することができ、その結果を必要とする命令がいくつ先の命令であるかがルックアヘッド数として指定されている。メモリ参照が完了するまで、その指定された命令は実行が待たされる。この命令レベル並列を利用することにより、パイプライン段数以上のメモリレイテンシに適用可能である。

MTA の方式は、古くは HEP[Smi78] で提案された方式であり、種々の改良を行っているとは言えそれほど画期的なものではない。しかも、MTA の開発が発表されたのは 1990 年代始めであったのに対し、実際に製品が発表されたのは 1990 年代後半であった。この間アーキテクチャの開発以上に時間をかけ、苦労したのは、この様なアーキテクチャに対応する自動並列化技術であった。MTA では高性能コンパイラにより、自動的に逐次プログラムを並列化し、スレッドに分割する。コンパイラはプロセッサを十分満たすのに必要なスレッド数をそのコードの並列度から静的に見積もり、その数のスレッドに分割する。自動並列化の結果は満足のいくレベルである。しかし、更なる性能チューンを行うために、プログラマによって、スレッドへの分割数等を指示でき、それにより性能が向上するときもある。また、MTA の命令は 3 命令発行の VLIW であるが、そのうちメモリ参照は 1 命令しか発行できないことが性能低下の原因となることが多い。MTA のメモリ参照単位は 64 ビットであるので、64 ビットより小さいビットしか必要としない複数のデータがある場合には、それらを 1 ワードにパックするというビットパッキングという方法をループアンローリングと組み合わせることが有効なことがある。

同時マルチスレッド (SMT, Simultaneous Multithreading)[Tul95, Tul96, Tul99, Tul01, Luk01, Pre02] は最近では Intel 社の Xeon に Hyper-Threading[Mar02, Wan02, Bur02, Tia02, Che02, Mag02] として採用されている他、開発が中止になった Alpha EV8[Pre02] に採用される予定であった方式である。これは、1 つのスーパスカラプロセッサの演算ユニットに、複数の独立したスレッドからの命令を同時に実行させる方式である。近年のプロセッサは一度に複数命令の発行を行うスーパスカラプロセッサが主流であり、性能を向上させようと一度に発行する命令数を増加させる傾向にある。しかし、一度に発火可能な命令数を 2 倍にしたからといって、性能は 2 倍にはならない。これはデータ依存や制御依存、あるいは演算リソースの衝突などが影響している。そこで、そのような影響のない、独立した複数のスレッドから命令を発行することにより、演算利用率を上げようというのが SMT の基本的アイデアである。また、一方がキャッシュミスによるメモリアクセスレイテンシ等のためにアイドルになっていても、他方のスレッド実行によりプロセッサ実行効率の低下が抑えられるという効果も狙っている。

また、シングルチップ上に多数のスレッドプロセッサを登載したチップマルチプロセッサも提案されている。例えば IBM 社が研究発表した Cyclops[Cas02] は、シングルチップにスレッドユニットを 128 個登載している。スレッドユニットには PowerPC を簡略化した 1 命令実行の RISC コアであり、レジスタファイルや整数ユニットが含まれる。これらの 4 つのスレッドユニットが浮動小数点演算ユニット (FPU) とデータキャッシュを共有し、これを Quad ユニットと呼ぶ。2 つの Quad ユニットが命令キャッシュを共有する。スレッド数はかなり大規模であるが、演算ユニットをパーティション化することによる、SMT とチップマルチプロセッサの中間的なアーキテクチャといえる。これだけのゲート規模がシングルチップに搭載可能になるときには興味深いトレードオフであると思われる。

Cyclops ではオンチップメモリの利用にも工夫が見られる。各データキャッシュは 2K バイトのブロック 8 個からなり、8-way のキャッシュとして利用することもできるし、ブロック単位でアドレス指定可能な高速メモリとして利用することもできる。また、チップには 16 バンクに分けられたそれぞれ 512K バイトの DRAM が内蔵されており、各キャッシュとメモリスイッチにより接続されている。また、データキャッシュ同士もキャッシュスイッチにより別途接続されており、全スレッドユニットからアクセス可

能である。このデータキャッシュ間のコヒーレンス管理はハードウェアではサポートされないが、参照するキャッシュの範囲を指定することにより、全データキャッシュを1つの512Kバイトのキャッシュとしてスレッド間で共有することも可能であるし、各スレッドが自Quadユニット内のキャッシュのみを参照し、結果としてコヒーレントなしの個別キャッシュとして扱うことも可能である。どのような範囲でキャッシュをアクセスするかはプログラム次第となる。

2.4 本章のまとめ

1980年代に命令レベルデータフロー計算機が並列処理向きのアーキテクチャとして研究され、その細粒度同期などの動的スケジューリングとスレッドレベルの実行制御を融合させたアーキテクチャの有効性が指摘された。しかし、これらを単につなげただけでは、それぞれに必要なハードウェア量を加えただけのハードウェアが必要となったり、それぞれに特有な専用回路が必要になってしまう。そうではなく、両者の特徴を生かしつつ、ハードウェアを簡略化/汎用化して処理速度を向上させるとともに、そのシンプルなハードウェアとコンパイラ等のソフトウェア最適化をうまく協調させることが重要である。メッセージ通信と共有メモリの融合についても同様のことが言える。

第3章　細粒度通信機構に基づくマルチスレッドアーキテクチャの方式

本章では、本稿で提案する細粒度通信機構に基づくマルチスレッドアーキテクチャの方式について述べる。最初に、本アーキテクチャの設計指針について述べ、それから提案するアーキテクチャの特徴を述べる。

3.1 設計指針

第一章で述べたように、大規模な並列処理においてより高い並列効率を達成するため、およびこれまでに並列処理に向いていないと思われていたアプリケーションを並列化するためには、これまでの粗粒度並列処理で行われてきたような、通信をできるだけ1つにまとめて一度に大量のデータをやり取りするという通信スループットに頼った並列化だけではなく、通信のレイテンシを削減するとともに、レイテンシを隠蔽する技術が重要となる。そのような技術として、第二章で述べたような複数の命令実行列(スレッド)を切り替えてプロセッサの実行効率を高めるマルチスレッド方式が有望である。このマルチスレッド実行の効果を高めるためには、スレッド切り替えおよび通信セットアップにかかる時間を軽減して、より多くの時間を本来のスレッド実行に割り当てられるようにすることが重要であり、そのためのアーキテクチャがいくつか提案されている。

第二章で述べたように、マルチスレッド技術には、1) 逐次的に実行されるスレッドを適切に切り替えながら実行するシンプルマルチスレッド方式、2) サイクル毎に異なるスレッドからの命令を実行する細粒度マルチスレッド方式、3) 複数の演算ユニットを持ち、複数のスレッドからの命令を同時に実行する同時マルチスレッド方式等がある。シンプルマルチスレッド方式は、逐次プロセッサにおいて時間多重によるマルチタスク処理と同様なイメージであるが、スレッド切り替えのタイミングがより細かな単位(数十から数百クロックサイクル)を想定していることが多く、スレッド切り替えやスレッドスケジューリング、スレッド間の通信や同期などにハードウェアサポートが必要となる。細粒度マルチスレッド方式は古くはHEPというマシンで提案され、現在TERA社(現CRAY社)によりMTAというスーパーコンピュータとして実用化されている。十分な性能を引き出すためには数十個のスレッドをプログラムから抽出する必要があり、そのためのコンパイラ技術などが開発されている。同時マルチスレッドは比較的新しい方式である。現在のプロセッサでは、1つのスレッドからの命令を一度に複数実行するスーパースカラ方式が一般的である。この一度に実行する命令数を向上させようとする試みが行われているが、単に演算ユニットを増加させて実行命令数を増加させようとしても、命令間の依存関係などで十分な命令レベル並列性を引き出せないことが指摘されている。この命令レベル並列の限界に対する解決策として同時マルチスレッドは提案された。最近ではインテル社によりHyper-Threadとして商用プロセッサにも組込まれている。

このようなマルチスレッド計算機を考える場合に、スレッドの生成をプログラム実行中に可能とする動的スレッド生成を基本としスレッド数に制限を設けないようにするか、プログラム開始時に静的に一定個数のスレッドを生成する方式とするかが大きな設計の選択となる。スレッドコンテキストをハードウェアで保持する細粒度マルチスレッド方式や同時マルチスレッド方式では、スレッド数がハードウェアリソースによって制限されるため、スレッド数が増大する可能性のある動的スレッド生成を採用することは難しい。しかし、本論文では粗粒度並列処理ではうまく並列化できないようなプログラムを並列化して高速化することを目指しており、効率的な動的スレッド生成を可能とするとともに、スレッド数へは特に制限を設けないこととする。このため、スレッドコンテキストをメモリ上に持つシンプルマルチスレッド方式を基本として、設計指針の検討を行う。また、本論文では千台規模の並列計算機を想定している。この場合、1ノードは1チップ+メモリ程度に集約する必要がある。本研究を開始した当時の1チップのゲート規模からは、複数スレッドコンテキストを保持することは困難であった。また、要素プロセッサをシングルチップに集約する、すなわちシングルチップにネットワーク機構とプロセッサ機構を登載するという制約から、各機構をシンプルな構成に限る必要がある。このハードウェアをよく利

用する機能に限り、それ以外の機能は最適化したプログラムにより処理するという考えは、RISC プロセッサアーキテクチャに通じるものであり、本アーキテクチャ設計でもこの指針を並列アーキテクチャに拡張して用いる。

3.2 アーキテクチャの特徴と概要

DMA 転送を用いた大きなメッセージ長の転送によりスループット性能を向上させようという従来のメッセージ通信型並列計算機方式と異なり、レイテンシの削減/隠蔽により並列性能を向上させる並列アーキテクチャを考える。そのためには、プロセッサ間通信のオーバヘッドを低減するために、ネットワーク、プロセッサおよび両者のインターフェースを総合的にとらえた並列アーキテクチャが必要である。すなわち、プロセッサ間通信を I/O としてではなく、メモリ参照と同等のリソース処理ととらえ、通信と命令実行の融合を積極的に押し進める必要がある。この融合の鍵となるのが、細粒度通信とマルチスレッド処理である。さらに、共有メモリプログラミングで重要となるリモートメモリアクセスレイテンシの削減の機構である直接リモートメモリ参照機構を含めた 3 つからなるアーキテクチャが本稿で提案する方式である。前者の 2 つの特徴は、すでに電子技術総合研究所で開発された EM-4 から受け継いでいるが、命令レベルおよびパイプラインレベルで改良を行い性能向上を図るともに、それらをより一般化し使い勝手の向上を図っている。この提案するアーキテクチャを具現化したものが EM-X である。

EM-X では、プロセッサ間通信の単位を細粒度なワード単位のパケットに限ることにより、その機能を命令実行パイプラインと融合し、低レイテンシと高スループットを両立させている。また、並列プログラムを逐次命令列からなるスレッドという単位に分割し、スレッド内を RISC プロセッサによる高速な実行を可能にしている。それとともに、レイテンシ隠蔽が必要な場合には、スレッドを他のアクティブなスレッドに切り替えることにより、そのレイテンシを隠蔽するマルチスレッド実行を効率良くサポートしている。共有メモリ型のプログラミングにおいても、分散配置した配列データをワード単位で不規則にアクセスしたり、各プロセッサで同期をとるような状況はしばしば生じるが、そのような状況に対しても細粒度通信、マルチスレッド実行、直接リモートメモリ機構により効率良く適応できるため、並列処理の適用分野を広げていくことが期待できる。

以下では本方式を構成するマルチスレッド実行、細粒度通信、直接リモートメモリ参照のそれぞれについてその特徴を述べる。

3.2.1 マルチスレッド実行

EM-X では、並列プログラムを逐次命令列からなるスレッドという単位に分割し、通信レイテンシが発生してプロセッサがアイドルとなる場合には、このスレッドを他のアクティブなスレッドに切り替えてプロセッサの実行効率を落さないようにするマルチスレッド実行を行う。ここで、通信レイテンシとは、リモートメモリ参照やリモート関数呼び出しの結果待ちのような遠隔処理を行う際に現れる遅延のことである。この通信遅延は、ネットワーク上でのパケット衝突や各要素プロセッサ (PE) における負荷の変動のため、静的に見積もることは一般に難しい。このため、このような通信をコンパイラで静的に最適化するのではなく、そのレイテンシを実行時に他の実行可能なスレッドで埋めてプロセッサを常に稼働状態として、全体の実効性能を維持させようというのが、マルチスレッド実行のねらいである。ここで、各スレッドの実行で、スレッドを実行 (再開) してから他のスレッドに切り替えられるまでの実行単位をサブスレッドと呼ぶことにする。

スレッド切り替えに時間がかかってしまうと、せっかくスレッド切り替えによりレイテンシ隠蔽しようとしても、そのオーバヘッドによって効果が軽減されてしまう。スレッド切り替えのオーバヘッドがサブスレッドの長さに依存せずにほぼ一定であるとすると、サブスレッドの長さを大きくすればそれだけ

相対的なオーバヘッドは小さくなる。しかし、他の実行可能なスレッドに切り替えるためには、そのようなスレッドの並列性を抽出できなければいけないし、サブスレッドの間には通信レイテンシを発生するような処理を置くことはできないので、やたらとサブスレッドを大きくとることはできない。そのため、実効性能を高く維持したままマルチスレッド実行を行うためは、スレッド切り替えをハードウェアとソフトウェア (コンパイラとランタイムの両方を含む) の協調により、スレッド切り替えのオーバヘッドを削減することが重要となる。

逐次プログラムからスレッドを自動生成する自動並列化コンパイラは非常に興味深い問題であり、また実用性の点からも重要な問題である。しかし、現在のところ EM-X では自動並列化コンパイラは開発されていない。第 2 章で取り上げた TERA 社の MTA ではこのような自動並列化コンパイラが MTA 向けに開発されている。EM-X のプログラム開発言語である EM-C[23, 25, 50] では、fork(), remote_read() 等のスレッドライブラリをプログラマが明示的に利用するスレッドプログラミングを基本としている。ただし、これらのスレッドライブラリは、その多くがハードウェアと密接に連携しており数命令で実行可能なものである。そのため、コンパイラによりインライン展開を行ったコード生成が行われており、関数呼び出しのオーバヘッドを削除して効率的な実装を行っている。EM-X におけるスレッドプログラミングおよびライブラリの詳細については 4.2 で述べる。

EM-X では、スレッドを関数呼び出しおよびその他の通信レイテンシを発生する個所で、サブスレッドに自動的に分割している。また、スレッドの切り替えが起きるのは、このサブスレッド単位に限定している。これにより、スレッド切り替え時に必要となるリソース管理、すなわちワークレジスタの待避/復帰のタイミングをコンパイラが認識して、必要最小限のレジスタのみを待避/復帰するように最適化している。平均的なサブスレッドの長さは 10-100 命令程度である。また、スレッド切り替えに必要なオーバヘッドは、起動されるスレッドの種類や待避/ 復帰が必要なレジスタ数に依存する。もしレジスタ待避が必要なければ、起動されるスレッドの種類に応じて 0-2 サイクルでスレッド切り替えが可能である。待避/復帰が必要なレジスタがあればそれぞれにつき 1 サイクルのオーバヘッドがかかる。ラフに見積もってもスレッド切り替えのオーバヘッドは短いスレッド長の場合で 2 割程度、長めのスレッドでは数%程度である。もし、割り込みなどでスレッド切り替えが起きる方式を仮定すると、このような最適化は行えないため、全てのレジスタを待避/復帰する必要がある。この場合、それだけで 100 サイクル程度のオーバヘッドがかかってしまう。また、スレッド実行中は、自分自身がスレッドを中断しない限り、エラー例外ハンドラを除いて、他のスレッドにより割り込まれることはない。そのため、メモリアクセスをそのスレッドが排他的に行うことが可能であり、リソース管理を容易にしている。ただし、以下で説明する直接リモートメモリアクセスだけは例外で、スレッド実行とは独立にメモリアクセスが行われるため、同一領域へアクセスする場合には、明示的な制御が必要である。

スレッド実行の詳細

各スレッドの実行は RISC パイプラインにより高速に実行される。各スレッドの実行状況を表すには、実行しようとする命令の開始アドレスと、その実行に必要な引数や途中結果などを格納する実行環境のベースアドレスのペアがあればよい。EM-X ではこれを continuation と呼ぶ 1 ワードで表わしている。EM-X では後者の実行環境をオペランドセグメントと呼び、512 バイトの固定長のブロックを割り当てている。オペランドセグメントはフリーリストにより管理されている。このオペランドセグメントの先頭に、スレッドの命令列の先頭へのポインタが置かれている。これにより、オペランドセグメントへのポインタとスレッド命令列への変位を 1 ワードにパックしている。詳細は 4.3.4 直接待ち合わせを参照。

パケットがプロセッサに到着すると、いったんパケットバッファである FIFO に入れられる。そしてスレッドの実行が中断される度に、その FIFO から一つづつパケットが取り出されて、パケットのアドレス部を continuation として、それに対応したスレッドがハードウェアにより自動的に起動される。こ

のようにEM-Xのパケットは、単にプロセッサ間で通信されるデータではなく、パケット自身が実行命令列(スレッド)を直接起動するデータ駆動の原理が取り入れられている。同様の考え方にアクティブメッセージ[Cul91]があるが、我々はこれらの考えの元となったデータ駆動計算機[Den83]から研究を進め、いち早くハードウェアの開発[Hir87, 12]を行ってきた。各スレッド間をcontinuationパケットで結合することにより並列プログラムの実行が行われる。

入力バッファ部におけるプロセッサへの到着順によるパケットのFIFO制御、パケットによるスレッドの直接起動、および、スレッド実行の自律排他的制御により、EM-Xの実行モデルは非常に簡略化され、また、マルチスレッドに適したモデルとなっている。すなわち、他PEへの関数呼び出しやメモリ参照などでスレッドの実行がアイドルとなるような場合は、単にスレッドを中断するだけで、入力バッファの先頭のパケットにより次のスレッドが起動される。この起動も待ち合わせ処理部での先行制御により、0-2クロックサイクルで行うことが可能であり、スレッド切替のオーバヘッドは非常に小さく抑えられている。他PEへ処理を要求する際に、スレッド再開のためのcontinuationを一緒に送っておき、リモート側で処理が終った際には、そのcontinuationをヘッダ部として結果を出力することにより、スレッドの再起動が行われる。continuationは1ワードであり、また、この生成も1命令で行うことができるため、オーバヘッドは非常に小さい。このように、プログラムはいくつかのスレッドにより並列処理が明示され、各スレッドは通信レイテンシを挟んでサブスレッドに分割されている。これらのサブスレッドがパケットcontinuationによりつながれており、サブスレッド単位で動的にスケジューリングされる。このようなマルチスレッド機構により、他のプロセッサとの通信などによるレイテンシを隠蔽し、並列処理性能を向上させることが可能となっている。

3.2.2 細粒度通信

細粒度通信とは、2ワード程度からなる細粒度パケットを通信の単位とすることである。EM-Xでは、continuationを含めてすべてのプロセッサ間通信をこの細粒度通信を用いて行っている。これにより、パケットの送信、転送、受信の各ステージを簡略化・高速化することを容易にするとともに、ステージ間のインターフェースのオーバヘッドを低減することが可能となる。例えば、パケット長を2ワードと限っているため、パケット出力命令を<$send\ reg_{data}, reg_{addr}$>のように2つのレジスタを指定するだけで実行できる。このように2入力オペランドの通常のレジスタ演算命令と同様に実装することができ、命令実行パイプラインとの融合が容易となる。これは、通常のRISCパイプラインでは入力オペランドは2つまでであり、これに命令内のデータを加えた限られたリソースから生成できる程度にパケットを簡略化しているためである。このような単純な*send*命令を備えることにより、DMA転送用の各種レジスタ設定などの余分なセットアップ時間を必要とせず、他の演算ユニットと同様に、パケット出力は1クロックで終了する。他の並列計算機では、通信レイテンシの多くの時間を通信のためのセットアップやパケット変換などに費やしているのに比べ、EM-Xではパイプラインレベルでパケットを生成しているため、ほとんどハード的なネットワーク転送時間だけでパケットを転送することが可能であり、低レイテンシを実現している。

*send*命令ではパケットアドレスとして、PE番号(10ビット)とPE内局所アドレス(22ビット)からなる1ワード(32ビット)のグローバルアドレスを用いることができ、局所アドレスのポインタ参照と同様にリモートアドレスのポインタ参照を効率良く行える。また、*send*命令でのアドレス指定にはメモリ参照命令と同様に、$reg_{base} + imm_{disp}$というベースアドレッシングを指定できるため、柔軟なリモートメモリアクセスが可能となり、高スループットを実現している。このようにEM-Xでは細粒度通信を用いることにより、低レイテンシと高スループットを両立させている。

ネットワーク上のパケット転送は、EM-Xではネットワークトポロジーとしてサーキュラーオメガ網[18, 22, 71]を用いており、PE間距離がPE台数のlogオーダに抑えられている。本ネットワークでは、

ストアアンドフォワードデッドロックを防ぐため、3バンクのバッファを各入力ポートに備えているが、細粒度パケットを用いることにより、バッファ量を抑え、パケットスイッチ機構をプロセッサに内蔵することを可能にしている。さらにパケット内の宛先PE番号のみにより最短ホップ数を保証した経路を選択できるセルフルーティング方式[5]や、レイテンシを抑えたバーチャルカットスルー転送の制御も容易になっている。

プロセッサとネットワーク間のパケット送受信は、RISCパイプラインによる命令実行とは独立に処理できるように、送信・受信の双方に8パケット程度のバッファをプロセッサ内部に備えている。このバッファはFIFO(FirstInFirstOut)により制御されているが、細粒度の固定長パケットにより制御は容易である。出力バッファ(OBU)がいっぱいになった時のみ、パケット出力命令の実行パイプラインがストールする。入力バッファがいっぱいになった時には、ネットワークの飽和を抑えるため、溢れたパケットをメモリ上のパケットバッファに直ちに退避する。入力バッファに空きが生じると、到着順序を保ちつつ自動的に復帰される。このパケットバッファの制御は、完全にハードウェアにより制御され、命令実行パイプラインが乱されることはない。このため、仮想的な大容量バッファが存在すると考えることができる。

3.2.3 直接リモートメモリアクセス

EM-Xではネットワークからやってきたパケットは通常プロセッサ内部の入力パケットバッファへ格納され、順番に処理されていく。パケットのFIFO管理は前節で述べたとおり、ハードウェアで行われており、パケットの受信によりスレッド実行が割り込みなどで中断されることはない。

しかし、この方式ではリモートメモリ読み出し要求パケットがやってきても、現在実行中のスレッドが終了しない限りその処理は待たされる。上記のマルチスレッド実行により、リクエスト側でのレイテンシを一時的に隠蔽することは可能であるが、このアクセスレイテンシがクリティカルパスとなっている場合には、それによる遅延はマルチスレッドによっては解決されない。また、サービス側から見ると、マルチスレッド実行であっても現在実行中のスレッドが終了しない限りやってきたパケットの処理は行われないのは同じである。さらに、リモートメモリアクセスリクエストは1、2命令程度の非常に小さなスレッドを起動するため、スレッド起動/切り替えオーバヘッドも相対的に大きなものとなる。共有メモリ型プログラミングを分散メモリで実現する場合は、分散配置された配列へのリモートアクセス遅延が重要であり、これが大きいと性能に大きな影響を与える。さらに、マルチスレッドなどによりレイテンシ隠蔽を行おうとすると、さらに動的レイテンシを増大させてしまう場合もある。すなわち、スレッド切替により新たに起動されたスレッドは、そのプロセッサに対するcontinuationパケットの動的遅延を増大させてしまう。また、その起動されたスレッドがさらに他のプロセッサに対しリモート要求を行う場合には、ネットワーク使用率および目的プロセッサにおける他の処理に対する動的レイテンシを増大させてしまう。

EM-Xでは次の三つの方法で動的レイテンシの低減を図っている。

- 待ち合わせ処理の先行処理による待ち合わせミス時のオーバヘッドの低減
- パケットによるスレッド起動の優先度処理
- リモートメモリ参照パケットに対する特別のサポート

最初の待ち合わせ処理の先行処理とは、待ち合わせ処理を命令実行とは独立に、パケット入力部で行うことを指している。EM-Xでは直接待ち合わせという高速な同期機構により2入力の待ち合わせをサポートしている。しかし、待ち合わせ処理を実行パイプラインで行うと、2つの入力のうち最初に到着するパケットは必ず待ち合わせミスとなるため、実行パイプラインの効率を低下させてしまう。また、

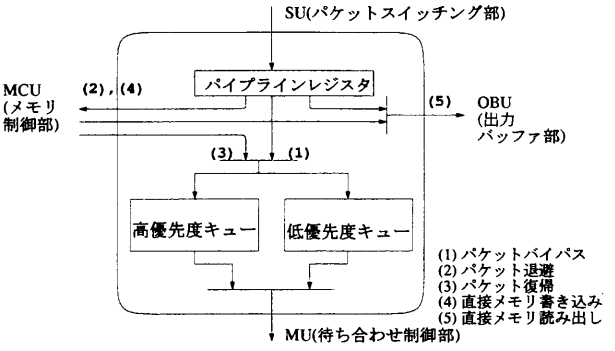

図 3.1: 入力パケットバッファ(IBU) 部

待ち合わせミスを引き起こすパケットが入力パケットバッファに格納されてしまうと、待ち合わせミス処理自体が動的レイテンシを増大させる要因となる。そこで、パケットを入力バッファへ格納する前に、待ち合わせ相手が存在するかどうかを確認し、

- 待ち合わせ相手が存在しなければ、当該パケットを待ち合わせアドレスに格納
- 待ち合わせ相手が存在すれば、当該パケットを入力バッファへ格納

という処理を行う。この待ち合わせ処理に必要となるメモリバンド幅としてメモリパケットバッファへ対するアクセスバンド幅を用いることにより、命令実行と並行して行うことが可能である。このため、他のスレッドの処理が行われている場合には、待ち合わせミスのオーバヘッドを隠蔽することができ、動的レイテンシの低減に役立つ。

2 番目の優先度処理では、パケットに 1 ビットの優先度を付加して、それぞれの優先度パケットを図 3.1 のように個別の入力バッファで管理することにより、優先度の高いパケットの動的レイテンシを低減するものである。低優先度バッファ中のパケットは、高優先度処理バッファにパケットが存在しない時のみ、スレッドの起動が行える。各入力バッファ内では FIFO 制御が行われている。この優先度はパケット出力時に付加するものであり、パケット命令に静的に記述されている。2 種類の優先度しかないため、高優先度のパケットは主にシステム処理的なもののうち特に動的レイテンシを下げたいパケットに使用する。例えば、動的負荷分散のための他のプロセッサの負荷の取得であるとか、以下のメモリ参照機構では対応できない同期つきのリモートメモリ操作などが相当する。

パケットの優先度処理により、入力バッファ中のパケットに起因する動的レイテンシは低減することができるが、実行中のスレッドの終了待ちに起因する動的レイテンシには対応できない。これは EM-X ではスレッドの実行は、他のスレッドによっては割り込まれない排他性を保証しているためである。例えば、全対全通信のように各プロセッサがデータの送り手であると同時に受け手でもあるような場合、到達したパケットが送信中のスレッドにブロックされて処理されず、動的レイテンシが増大する。これを解決するためには、命令実行と並行してパケット処理を行う機構が必要となる。

EM-X ではリモートメモリの書き込みおよび読み出しに対してのみ、スレッド実行とは独立に処理できる機構 (直接リモートメモリアクセス機構) を備えている。これはすべてのパケットに対してサポートするためには、よりきめ細かな優先度処理と、割り込みによるスレッド切替が必要となるが、それでは制御が複雑になる上、スレッド切替えオーバヘッドが大きくなってしまうためである。一方、メモリ参照パケットの処理は以下のような特徴があり、特別な最適化が有効であると考えられる。

- スレッド実行の単位が 1 命令ないしは 2 命令程度と他のスレッド実行に比べ非常に細かい。
- リモートメモリ参照はリモート側の処理と並行して実行したい場合が多い。
- リモートメモリ参照の応答性能が他のスレッドおよび全体の実行性能に影響を与える場合が多い。

直接リモートメモリ参照機構を含めた種々のリモートメモリ参照方式について次にまとめる。EM-X は分散メモリ型の並列計算機であり、前節でも述べた通りプロセッサ (PE) 間の通信は細粒度パケットを用いて行なわれる。EM-X のパケットは、2 ワード構成の非常に短いパケットであり、他のパケット通信型並列計算機のような多大なセットアップ時間を必要としない。このため、パケットの生成・出力は、局所メモリへの参照と同様に命令実行パイプラインレベルで行なうことが可能である。この細粒度パケットを用いて EM-X では種々の方法でリモートメモリへのアクセスが可能となっている。

関数呼び出しの利用

リモートメモリに対してアクセスする最も一般的な方法は、そのメモリの存在するプロセッサに対してリモート関数呼び出しを行い、その関数内で局所メモリに対してアクセスする方法である。EM-X では一つの関数インスタンスに対してオペランドセグメントと呼ぶ関数フレームを割り当てる必要がある。このオペランドセグメントは引数の受渡し、局所変数の退避、待ち合わせメモリなどに使用される。リモート関数呼び出しを行う際には、まずリモート PE に対してオペランドセグメントの割当を要求する。それを受けとると次に、それを用いて引数をオペランドセグメント内の引数領域に書き込む。最後に関数呼び出し後に再開するスレッドの continuation をデータとしてリモート関数を起動する。このように、一度オペランドセグメントの取得を行う必要があるため、2 往復分のレイテンシおよびスレッド切り替えが必要となる。

特殊パケットの利用

上記の方法では毎回オペランドセグメントの割当を要求しなくてはならないため、細かなリモートメモリアクセスを繰り返すような場合には適さない。そのため、EM-X ではオペランドセグメントを割り当てなくても起動できる特殊パケットハンドラを 64 個までプログラミングすることができる。良く使われるハンドラはシステムハンドラとして提供されており、リモートメモリ参照用のハンドラもそのうちの一つである。このハンドラを起動するためには図 3.2 に示すような USRRD パケットを用いる。EM-X のパケットはアドレスワードとデータワードからなり、アドレス部はパケットタイプ (PT)、宛先プロセッサ番号 (PE)、アドレスなどからなる。通常のパケットは、PT が 0 となっており、アドレスが continuation を示している。USRRD パケットは、PT でリモートメモリアクセス用のハンドラを起動することを、アドレスで参照すべき局所メモリアドレス (MA) を示している。EM-X では PE 番号と局所メモリアドレスの組み合わせで、システム内の任意のメモリアドレスを指定することができる。この PE 番号と局所メモリアドレスの組み合わせをグローバルメモリアドレスと呼ぶ。データ部には読み出したデータを処理するスレッドへの continuation を指定する。

USRRD パケットによるリモートメモリアクセスのレイテンシは、他のパケットとの衝突・競合がない場合で平均 23 クロックサイクルである。平均というのは EM-X ではネットワークとして直接網を採用しており、宛先 PE によってその距離が異なるためである。レイテンシの内訳は、USRRD パケット出力 (2 サイクル)、ネットワーク転送 (往復各 7 サイクル)、USRRD ハンドラ (4 サイクル)、結果パケットによるスレッド再開 (3 サイクル) である。このレイテンシの 23 サイクルのうち、図 3.2 に示すように実際に実行パイプラインを占有するのは 7 サイクルに過ぎず、残りのクロックサイクルはマルチスレッド処理により有効利用が可能である。

USRRD パケットと同様に、USRWR パケットというリモートメモリ書き込みハンドラも用意されている。また、同期つきメモリ参照として知られる I-structure[Arv89] や読み出し/書き込み要求をそれぞれキューとして管理できる Q-structure[50] もこの特殊パケットハンドラを用いてサポートされる。この他、やってきたパケットの合計を計算するハンドラや、最大値を保持するハンドラなど自由に設定する

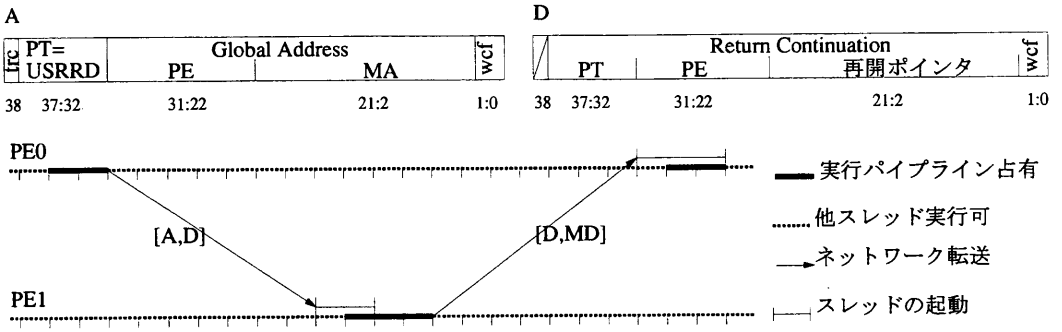

図 3.2: リモートメモリ読み出し (USRRD) パケット

ことが可能である。

優先処理パケットの利用

上記の USRRD パケットのレイテンシは、宛先 PE に他のパケットおよびスレッドが全く無いとした場合の値である。しかし、実際にプログラムを実行している場合には、宛先 PE の入力バッファにパケットが溜っていることが普通である。EM-X ではバッファを FIFO 制御しているため、後からやってきた USRRD パケットは、先に到着した他のパケットの処理がすべて済んだ後に処理が始まるため、実際のレイテンシはより大きな値となる。

バッファの FIFO 制御は、同一処理を行うパケットに対しては制御が簡単で公平なスケジューリングが行えるが、異なる処理を行うパケットに対しては優先度処理が行えないため、十分とはいえない。このため、EM-X では先に述べたようにパケット内の 1 ビットの優先度情報により使用するバッファを切替えることにより優先度処理を行っている。パケットの優先度処理により、入力バッファ中のパケットに起因するレイテンシを低減することができる。

直接リモートメモリ参照の利用

EM-X ではスレッドの実行は、他のスレッドによっては割り込まれない排他性を保証しているため、実行中のスレッドの終了待ちに起因するレイテンシには対応できない。これでは、例えば、全対全通信のように各プロセッサがデータの送り手であると同時に受け手でもあるような場合、到達したパケットが送信中のスレッドにブロックされて処理されず、レイテンシが増大する。これを解決するため、EM-X ではリモートメモリ読み出しパケット (SYSRD) およびリモートメモリ書き込みパケット (SYSWR) に限り、入力パケットバッファ上のパケットおよび現在実行中のスレッドにブロックされずにパケット処理を行う直接リモートメモリ参照を可能にしている。

EM-X では入力バッファが溢れた時には、新たにやってきたパケットはメモリ上の入力バッファへ自動的に退避され、入力バッファに空きが生じると退避した順序を保って自動的に復帰される。このメモリ入力バッファを参照するためのメモリバンド幅は、データフェッチのためのメモリバンド幅と共有されているが、命令フェッチのためのメモリバンド幅とは独立して確保されている。命令実行パイプラインによるメモリ参照が優先されるが、それ以外の命令を実行している場合には、パケット転送を待たせることなくメモリバッファへ格納できる。直接リモートメモリ参照にはこのメモリバンド幅を利用する。すなわち、SYSWR パケットが PE に到達すると、パケットを入力パケットバッファに格納するのではなく、パケットのアドレス部で示されるメモリアドレスにデータ部を直接書き込む。SYSWR パケットは入力バッファを経由せずに処理されるため、バッファ内で待っているパケットの影響を受けず、また、

このためのメモリバンド幅は命令フェッチとは独立なため、現在実行中のスレッドの影響も非常に小さく抑えることができる。さらに、SYSWR パケットはバッファ領域を使用せず直接相手先アドレスに格納するため、大規模データ転送時においてもバッファ溢れを抑制する効果が期待できる。同様に、SYSRD パケットが到着すると、パケットのアドレス部で示されるメモリを読み出し、そのメモリデータをパケットデータ部に格納されている continuation を用いてネットワークに転送する。ただし、デッドロックを防ぐため、出力バッファがいっぱいの時は通常の優先処理パケットとして扱われる。

このようにリモートメモリ参照処理を、スレッド実行と並行して行えるのは、入力パケットバッファへのアクセスを命令パイプラインと並行して行うことを可能としており、そのメモリバンド幅を利用できるからである。また、このリモートメモリ処理は高々1 回のメモリ操作で済むため、パケットパイプラインに対してもストレスを与えない。このように優先処理される SYSWR、SYSRD パケットを用いることにより、全対全通信のような場合でもパケットの送受信をオーバラップさせて行うことが可能であり、ネットワークにパケットが溜ることがないため、デッドロックやそれを回避するための命令パイプラインの中断を行う必要がない。

一方、SYSWR および SYSRD の処理はスレッド実行とは独立に行なわれるため、同一メモリに対して SYSRD/SYSWR によるアクセスと通常のスレッドによるアクセスを混在させる場合には注意が必要である。通常 SYSWR/SYSRD を用いる場合には、参照領域の確保や開放を、明示的な同期を用いて制御する必要がある。

第4章　細粒度通信機構に基づくマルチスレッドアーキテクチャの実装

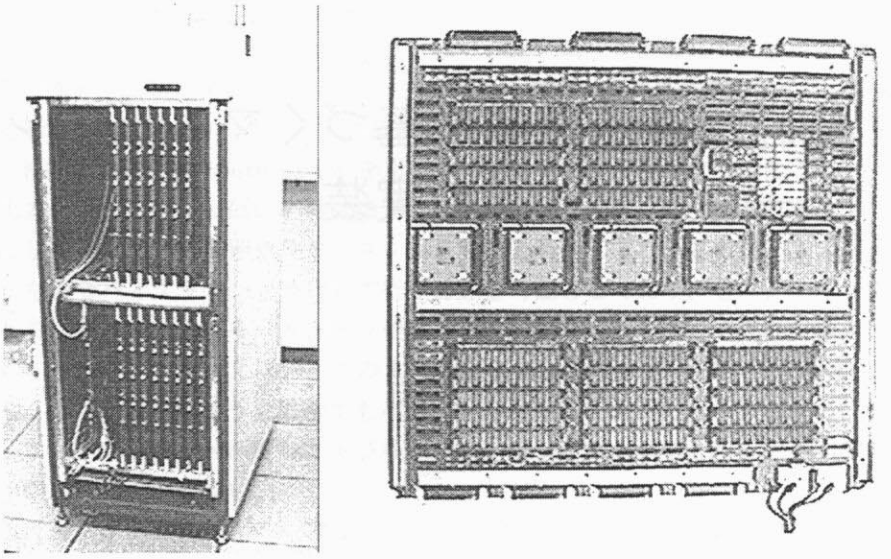

図 4.1: EM-X 80 プロセッサシステムとプロセッサボード

本章では、前章で提案した細粒度通信機構に基づくマルチスレッドアーキテクチャを実装した 80 要素プロセッサからなるプロトタイプ計算機 EM-X について述べ、そのプログラム開発環境ならびに、本プロトタイプのために開発したプロセッサの詳細について述べる。

4.1 EM-X プロトタイプ

前章の細粒度通信機構に基づくマルチスレッドアーキテクチャを、1992 年から開発を開始し、1995 年に 80 プロセッサからなる EM-X プロトタイプが完成した。図 4.1 にその全体写真とプロセッサボードの写真を示す。プロトタイプはプロセッサボード 16 枚とホスト計算機との接続を行うインターフェースボード、計算結果を表示するためのフレームバッファボードが 1 つの筐体に納められている。筐体はさらに上下 2 つのサブ筐体に分かれており、プロセッサボードは各サブ筐体に 8 枚ずつ格納され、それぞれバックプレーンボードで接続されるとともに、上下のプロセッサボードはケーブルで接続される。図 4.1 ではケーブルははずされた状態である。ホスト計算機として Sun Microsystems 社の SparcStation2 を用いている。この計算機はピザボックス型の筐体であり、EM-X の筐体の上に置かれている。内部の S-Bus スロットに専用の I/O カードを登載し、EM-X のインタフェースボードとケーブルにより接続されている。筐体内には、この他、3 種類の電源がある。1 つはインターフェースボード、バックプレーン、および空冷用のファンのための電源で AC 単相 100V の入力で最大出力 5V20A である。あとの 2 つが、各上下サブ筐体内のプロセッサボード用であり、AC 三相 200V の入力で、最大出力 5V300A である。本体の冷却は空冷で、筐体底部より吸気して、筐体裏面上部より排気する。筐体底部には塵を除去するためのフィルターを装備し、ファンはサブ筐体 1 の底部、サブ筐体 2 の底部と上部、筐体裏面上部に計 21 台実装されている。また、筐体内天井部には風向きを上向き後ろ向きに変えるフィンが備えられ、排気を円滑に行っている。

4.1.1 プロセッサボード

図 4.2 にプロセッサボードの構成図を示す。図のようにボードには 5 つの要素プロセッサが登載されており、コンパクトな筐体を実現している。基板サイズは $469.9mm \times 508.0mm$、基板厚 2.5mm(ガイ

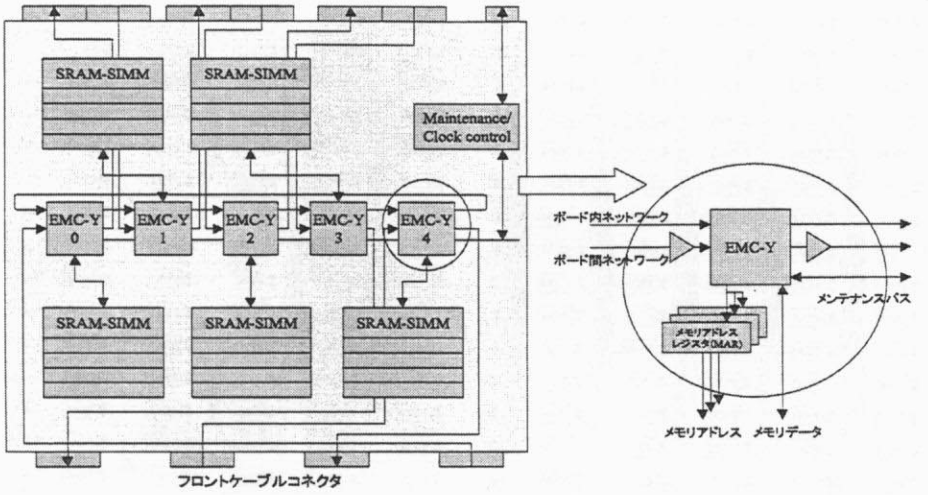

図 4.2: プロセッサボード構成図

ドレール部 1.6mm)、PE 基板上はマルチワイヤーによる配線を用いている。要素プロセッサ部はネットワークスイッチを含めてシングルチップ化したプロセッサチップ EMC-Y と、オフチップの SRAM メモリの他若干のドライバなどからなる。要素プロセッサは 1.0 ミクロンルールのゲートアレイで開発されたオリジナルのプロセッサである。図 4.2 の右側に、プロセッサチップの周辺を拡大した図を示している。EMC-Y は 2 つのネットワーク入力ポートと 2 つの出力ポートを持つ。それぞれ 1 つはボード内ネットワークのポートであり、それらは直接ボード上の配線で接続される。一方、ボード間ネットワークポートには入力ドライバおよび出力ドライバを介してコネクタに接続されている。ネットワークの詳細は以下で別途述べる。

また、PE 基板上には各 PE の初期化やモニタをホストから行うためのメンテナンス制御回路があり、これは PAL により構成している。各プロセッサはこのメンテナンス回路とバス接続されている。この他、クロックはインタフェースボードで生成され、長さの等しい同軸ケーブルにより各プロセッサボードに分配されているが、このクロックから他の制御信号を生成したりタイミング調整したりする回路がクロック制御回路に含まれる。クロック周波数は 20MHz として設計されているが、オフチップのメモリはこの倍の 40MHz でアクセスを行う。

EMC-Y はオフチップメモリとして大規模 SRAM を用いている。これはメモリバスを簡略化するためである。このメモリとしてはデータ幅 1 ビット、サイクルタイム 20ns の 1M ビット SRAM を 40 個用いており、1 プロセッサあたり容量は 5M バイトである。これはプロセッサチップ EMC-Y の接続可能メモリの最大でもある。データ幅 40 ビットというは、データ 32 ビット、データタグ 8 ビット、パリティ2 ビットである。このような構成をしているため、メモリアドレスは 40 個のメモリチップを駆動しなければいけない。これらを分散する意味もあり、メモリアドレスレジスタはプロセッサチップの外に 3 セット並列に用意されている。さらに、メモリはボード上の実装面積を軽減するために、8 チップからなる独自 SIMM(シングルインラインメモリモジュール) を作成した。これをバックプレーン上のボード間隔を小さくするために斜めに装着するコネクタで 5 個並列に接続することにより実装している。

EM-X はネットワークレイテンシを軽減しつつ、スループットを確保するため，ネットワークの各ポートのデータ幅は 38 ビットと比較的大きく設計されている。ネットワークトポロジーはサーキュラオメガ網 [22] である。これは間接網として良く知られているオメガ網の両端を接続し、各ネットワークノードにプロセッサを接続したネットワークである。ネットワークスイッチはプロセッサチップに内蔵されており、各ネットワークスイッチは 3 入力 3 出力のクロスバースイッチの構成をとる。そのうちの 1 対の入出力ポートがプロセッサコアに接続されており、残りの 2 対の入出力ポートでサーキュラオメガ網を

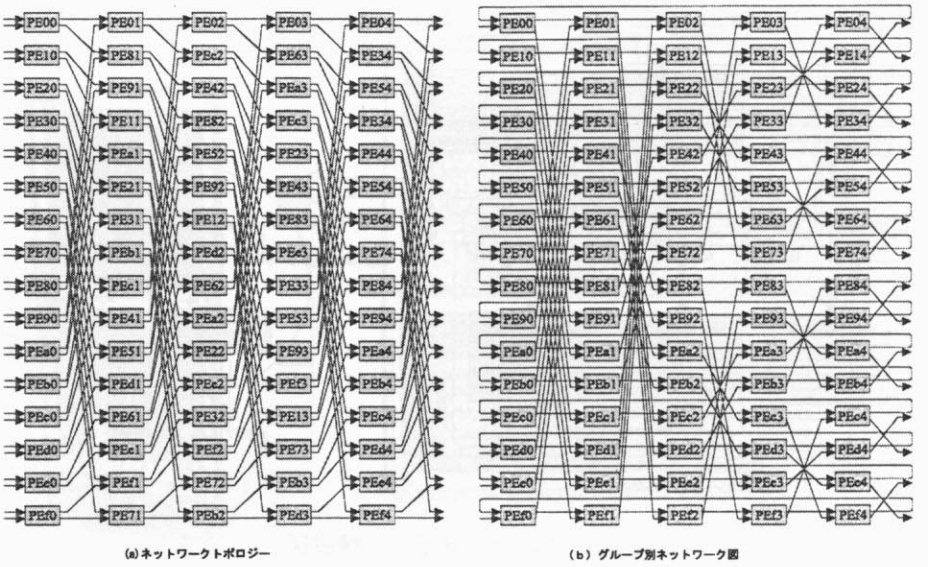

図 4.3: サーキュラーオメガネットワーク

構成している。80 プロセッサの場合のネットワークトポロジーを図 4.3(a) に示す。トポロジーの制約で $2^n \times (n+1)$ の構成を基本としており、80 プロセッサの時には $n = 4$ で $16 \times 5 = 80$ という構成になる。矢印で示されているとおり、各ネットワークは 1 方向の接続である。

各ノードからは 2 本の出力ポートが出ているが、このポートを出て $n+1$ ホップして帰ってくる閉路がそれぞれのポートに存在する。そのうちの 1 つをグルーピングに利用した場合のネットワーク接続図を 4.3(b) に示す。EM-X プロトタイプでは基本的に 4.3(b) をもとにネットワーク接続を実装しているが、さらにこれをグループ番号の偶数/奇数によって 2 つに分割し、PEx0-PEx1 間、PEx1-PEx2 間、PEx2-PEx3 間 をそれぞれのバックプレーンにより実装し、残りの PEx3-PEx4 間, PEx4-PEx0 間をケーブルにより接続している (ただし x は 0, 1, ..., e(14), f(15))。この接続図を 4.4 に示す。この図で青の線がプロセッサボード内で実装されているリング接続のネットワークを、黒の線がバックプレーンで実装されているネットワークを、赤の線がケーブルにより実装されているネットワークをそれぞれ表わしている。

16 枚のプロセッサボードの他に、ホスト計算機と接続するためのインタフェースボード (IFSW)、および、計算結果を直接表示するためのフレームバッファボード (FB) が搭載されている。これらの基板には、ネットワークスイッチのために EMC-Y チップが登載されており、上記のネットワークのケーブル接続の途中に自由に追加可能である。現在の実装では PE03 と PE34 の間のネットワークにインターフェーススイッチボードが、PE04 と PE10 の間にフレームバッファボードが接続されている。各ボードのプロセッサに適切なプロセッサ番号を割り振ることにより通常のパケットルーティングで各ボードのプロセッサへパケット転送することが可能である。また、ホスト計算機宛のパケットはヘッダ部で識別され、そのためには特別なルーティングが行われる。詳しくは 4.3.1 を参照。

4.1.2 インターフェースボード (IFSW)

インターフェースボードは EM-X のプロセッサ群とホスト計算機との通信を行うためのボードである。このボードは以下の機能を行う。

- クロック制御機構
- ホスト計算機と EM-X プロセッサ群との間でのパケット通信機構

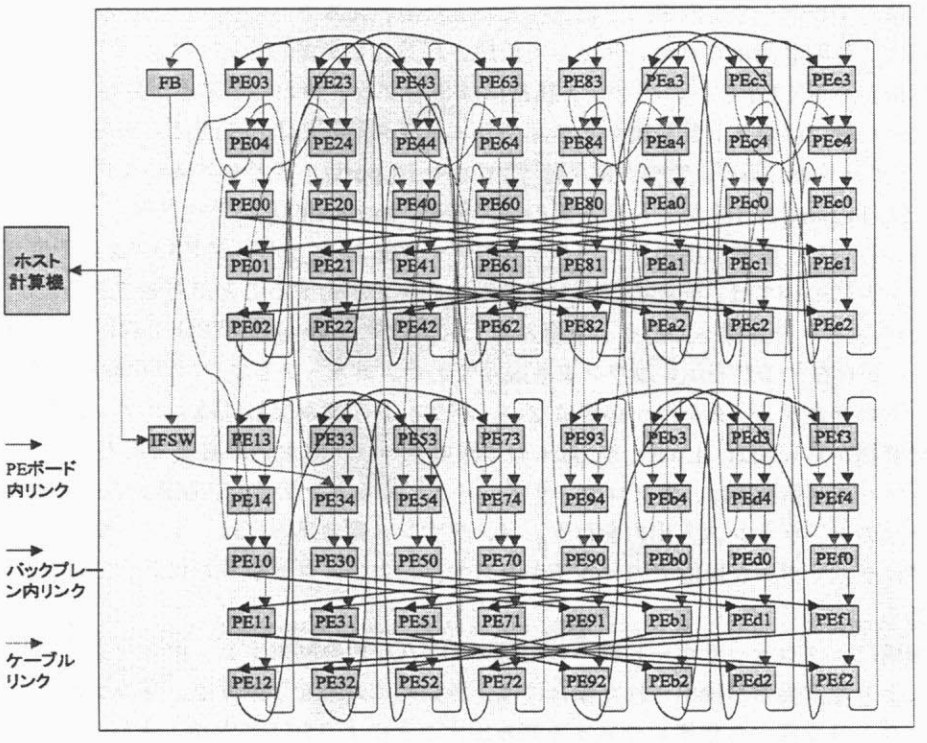

図 4.4: ネットワークの実配線

- メンテナンスバスの制御機構
- プロセッサボード群への電源制御および異常温度監視機構

クロック制御機構は、20MHz, 10MHz, 5MHz の任意の周波数のシステムクロックをプロセッサボードに供給することができる。通常は20MHz が用いられる。また、カウンタ (16 ビット) に設定した個数のクロックを供給した時点でクロックの供給を停止することができる。これは EM-X は完全スタティックな回路のみで構成されており、クロックをとめても状態を保持できることによる。これを利用したデバッグ機能が用意されている。クロックの状況については、ホスト計算機から S-Bus I/O カードを通じて状態レジスタをアクセスすることにより知ることができる。さらにプロセッサでエラー時に生成されるトラップ信号を検出し、数クロック以内にクロックの供給を停止し、その時点でのクロックカウンタの値を読み出すことができる。メンテナンスバスのバスクロックであるメンテナンスクロックは、システムクロックとは独立に20MHz, 10MHz, 5MHz, 2.5MHz の任意の周波数から選択できる。通常 10MHz が用いられる。

インターフェースボードをホスト計算機から制御するための基板がSBus インタフェース (以下、SBus32) である。SBus32 は、EM-X のホスト計算機であるSparcStation2 のS-Bus スロットに装着される。SBus32 は、シングルワイドのSBus 基板であり、ホスト計算機のSBus スロットを1 スロット占有する。SBus32 とインタフェースボードの間は、80 芯のケーブルによって接続される。SBus32 はSBus スロットのアドレス空間をインタフェースボードのアドレス空間に対応付けるための非同期インターフェイスであり、インタフェースボード上の機能はホスト計算機のメモリ空間にマップされる形でアクセス可能となる。この非同期インターフェイスは、最高 1M アクセス/秒のスループットを持つ。SBus32 によって提供される非同期インターフェイスはアドレス 21 ビット幅、データ 32 ビット幅のアドレス空間 (ワードアドレス空間) へのアクセスを提供する。

インタフェースボードには、EM-X のネットワークと接続するために EMC-Y プロセッサ (以下 EXT-PE) を 1 個登載し、パケットのルータとして利用する。この EXT-PE は、メモリやメンテナンスバス

への接続など他のプロセッサと同様に実装されているため、システム処理を行うプロセッサとして利用できる。実際に、時間計測用プロセッサとして使用されることが多い。

ホスト計算機から EXT-PE へのパケット経路には 1 パケット分のレジスタが設けてある。このレジスタの設定は、一般にはホスト計算機から 3 回のライトアクセスによって可能である。すなわち、アドレス部下位 32 ビット (a_part)、データ部下位 32 ビット (d_part)、アドレス部およびデータ部のタグ部 (hi_part, ただし hi_part[22:16] がアドレス部のタグ部を、hi_part[6:0] がデータ部のタグ部を示す) の 3 ワードである。しかし、直前に出力したパケットのデータ部 32 ビットとアドレス部の下位 22 ビットのみが異るパケットについては、一回のライトアクセスのみで出力することが可能である。

EXT-PE からホスト計算機へのパケット経路は、1024 パケット分の FIFO を設け、さらに FIFO に何個のパケットが存在するかを示すカウンタを設けてある。ホストからカウンタを参照することにより、個々のパケットの存在チェックを行う必要はない。パケットの読み出しには、パケット書き込みと同様に、ホスト計算機から a_part, d_part, hi_part の 3 回のリードアクセスが必要である。

メンテナンスバス制御機構は、各プロセッサボード上にあるメンテナンス制御回路との通信によって、全プロセッサとのメンテナンス入出力を行う。メンテナンス書き込みには、全プロセッサへのブロードキャスト、プロセッサボード単位でのマルチキャスト、およびプロセッサを指定してのシングルキャストが可能である。

電源制御機構は、プロセッサボード群への電源の ON/OFF を制御する。筐体および電源に設置した温度センサにより温度異常が検出された場合には、自動的に電源を切断する。電源が投入されているかどうかは、ホスト計算機から状態レジスタを読み出すことにより知ることができる。

ホスト計算機からアクセスできるインタフェース上のレジスタ一覧は付録 A を参照。

4.1.3 フレームバッファボード

EM-X の処理能力を視覚的にかつ効果的に示すことを目的として、フレームバッファボードを開発した。例えば、膨大な科学技術計算の結果をシミュレーションと同時に可視化処理してディスプレイ上に表示したり、ビデオカメラから入力した動画像をリアルタイムでフィルタリングして表示する。これにより、EM-X の処理能力の直感的把握や、計算機応用分野の拡大を目指している。

本フレームバッファ無しでも画像情報のホスト計算機を通じて入出力を行うことが可能である。実際ホスト計算機を通じて X ウィンドウシステムに画面を表示するライブラリを開発した。しかし、ホスト計算機の計算能力や EM-X とホスト計算機とのデータ転送能力が十分でないため、簡単な結果の表示やユーザとのやり取りの GUI 等にしか利用できず、リアルタイムの画像情報を扱うのには不十分であった。

フレームバッファボードは、EM-X の要素プロセッサである EMC-Y を利用して EM-X のネットワークと直接接続することにより、並列計算機側との十分な入出力性能を持つことができる。また、フレームメモリはボード上の EMC-Y プロセッサのメモリとして見えるため、EM-X 側からは通常のリモートメモリと同様に高速かつ柔軟にアクセス可能である。ビデオデータの入出力ボード自体としても、入出力画像メモリプレーンの分離、各画像メモリのマルチバンク化、マルチバッファリングによるスムーズな画像表示等、高い性能を持つ。フレームバッファボードの機能は以下の通り。また、フレームバッファの構成図を図 4.5 に示す。

- NTSC カラー画像信号 ($768 \times 480 \times 24$ ビット) を 30 フレーム/ 秒で連続入力可能。
- 高解像度フルカラー動画像 ($1024 \times 768 \times 24$ ビット) を表示可能。
- EM-X のプロセッサ間相互結合網に直接挿入することによりネットワーク経由で最大 10M ピクセル/秒の描画能力を持つ。

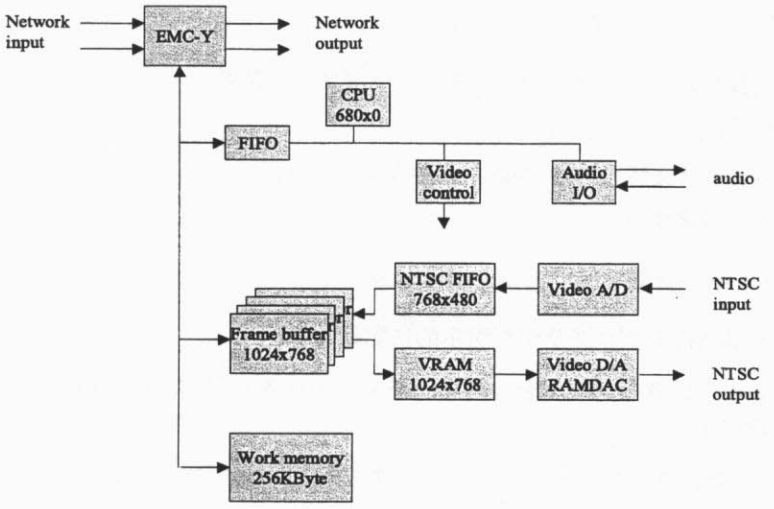

図 4.5: フレームバッファボードの構成図

- フレームバッファメモリを 4 画面分持つほか、入力画像バッファメモリと出力画像バッファメモリを独立に 1 画面分ずつ持つ。
- プロセッサ EMC-Y からアドレス 0x00000 - 0x0fffff にアクセスすると、常にワークメモリがアクセスできる。ただし 0x0fff40 - 0x0fffff は制御用 CPU の制御レジスタとしてマップされている。
- プロセッサ EMC-Y からアドレス 0x100000 - 0x3fffff にアクセスすると、現在マップされているフレームバッファメモリにアクセスできる。
- フレームバッファメモリは 1 ピクセル 32 ビット。16 進表現で xxBBGGRR で各 RGB の値が各 8 ビットで並んでいる。x はビデオ信号としては利用しておらずソフトウェア側で利用可能。
- プロセッサ EMC-Y からアクセスできるメモリへは EM-X から他のプロセッサのメモリと同様にリモートアクセスが可能。
- プロセッサ側にマップされていないフレームバッファメモリへの入力画像バッファの転送や、そこから出力画像バッファへ転送が画像の入出力とは並行して行える。

4.2 プログラム開発環境

EM-X でのプログラム開発言語は C 言語に独自の拡張を行った EM-C である。EM-C は、並列処理計算機における高性能な並列プログラミングを可能にするために、並列性と局所性の記述を C 言語に拡張した言語である。EM-C はグローバルアドレス空間を用いたリモートメモリアクセスをサポートするとともに、スレッドレベルの並列性の利用やリモート処理のレイテンシ隠蔽のための新たな言語機能を提供している。また、汎用のスレッドライブラリの他に、EM-X 向けの専用ライブラリとして、並列実行やトレースなど種々のライブラリを作成している。以下では、スレッドプログラミングおよび各ライブラリについて述べる。

スレッドプログラミング

EM-C の各機能は以下の通り。

- スレッド処理をベースとしたC言語の並列拡張。
  - スレッド操作はコンパイラ組み込みで、ライブラリ関数のインライン展開等により効率的に実装される。
  - スレッド切り替え時の待避/復帰が必要なレジスタはコンパイラにより必要なレジスタのみが選択され、最適化される。
- グローバルアドレス空間
  - 並列性や局所性をプログラマが明示的に記述する。
  - システム内の任意のメモリを、グローバルアドレス (プロセッサ番号 + 局所アドレス) を用いて指定可能。
  - PE 間に分散配置されるグローバルデータを指定可能。
  - グローバルデータへのポインタを global 修飾子で表わすことが可能。
  - グローバルデータへのリモートアクセスはコンパイラが自動生成する。
- 分散構造体操作
  - I-structure: スレッド間での 1 対 1 の同期をサポート。
  - Q-structure: スレッド間で共有されるキューをサポート。
- 並列性記述のための拡張: Task Blocks
  - 動的なスレッドの生成と割り当て、マルチスレッドによるレイテンシ隠蔽が可能。

汎用ライブラリは、FreeBSD の libc.a を移植し、システムコールも基本的なものはサポートしている。ファイル I/O 等は EM-X 内にはハードディスクは装備されていないので、ホスト計算機に要求を投げて、ホスト計算機側のローカルファイルシステムにアクセスするように実装している。

4.2.1 スレッドライブラリ

スレッドライブラリとして用意されている主な関数を以下に示す。これらを用いる場合には、< *thd/thread.h* > をインクルードファイルとして指定することが必要。また、関数として呼び出す他に、< *thd/thd_built_in.h* > をインクルードファイルとして指定することにより、インライン関数として展開させることも可能であり、関数呼び出しのオーバヘッドを軽減できる。

remote_call remote_call(pe_addr, func, argc, arg1, ..., argn)

pe_addr で指定したプロセッサ上で、func で指定した関数を起動し、その実行終了を待つ関数。関数呼び出しの引数は可変長で、その個数を argc で、各引数の値を arg1, ..., argn で指定する。SMP 向けの pthread 等と異なりスレッドを起動するプロセッサ番号を指定する。remote_call() の終了を待っている間は他のスレッドが実行可能である。

EM-C ではローカルな関数呼び出しでもパケットによる起動、およびスレッドの切り替えを行う。これは、同一の関数をローカルにも、リモートからも統一的に呼び出すことを可能にするためである。ただし、remote_call() とは異なり、オペランドセグメントの取得および引数設定はスレッドを切り替えずに自 PE の専用レジスタ ftop を直接参照して行い、関数起動のパケットを発行して始めてスレッドを切り替えるため、スレッド切り替えオーバヘッドは 1 度のみである。

fork fork_call(pe_addr, func, argc, arg1, ..., argn)

remote_call() と同様であるが、呼び出した関数の結果を待たずに直ちに終了する関数。呼び出した関数との同期をとるには、別途、以下に示す局所同期などを利用する必要がある。また、remote_call(), fork() ともにスレッド間の陽な通信はサポートしていない。通信が必要な場合は局所同期やブロック通信などの通信用ライブラリを別途呼び出すことが必要。

self_pe self_pe()

現在スレッドが起動されているプロセッサの番号を返す関数。以下の GLOBAL_ADDR() と組み合わせて、ローカル変数をグローバル変数に変換する場合等に利用。

PE_ADDR PE_ADDR(ga, ca)

プロセッサのグループ番号およびグループ内番号を指定してプロセッサ番号を生成するマクロ。以下の GLOBAL_ADDR() と組み合わせて、任意のグローバルアドレスを生成することができる。

GLOBAL_ADDR GLOBAL_ADDR(pe_addr, local_addr)

プロセッサ番号と局所アドレスを指定してグローバルアドレスを生成するマクロ。グローバルアドレスは 32 ビットで表わされており、上位 10 ビットがプロセッサ番号、下位 22 ビットが局所アドレスを示している。

PE_ADDR_OF PE_ADDR_OF(global_addr)

グローバルアドレスからプロセッサ番号を取り出すマクロ。

resched resched()

現在のスレッドを中断して、他のアクティブなスレッドに切り替える。EM-C ではスレッドはレイテンシを生じるようなリモート呼び出しを実行すると、スレッドを自動的に切り替えるが、逆にそのような実行を行わなければスレッドを中断することはない。例えば、以下のようなループを実行すると、lock の値が他の PE により SYSWR により書き換えられない限り実行を継続し、その間、他のスレッドは実行されない。

```
while (lock != 0) ;
```

一方、次のように resched() を挿入すると、lock の値をチェックして 0 で無いときには、スレッドを中断して他のアクティブなスレッドの実行が開始される。このため、lock の条件を満たすまでの時間を他の処理に活用できる。

```
while (lock != 0) resched();
```

ただし、このようなスピンロックの方式では何度も lock をチェックしに行くため無駄が多い。このような場合には、以下に示す局所同期を用いることによりチェックのオーバヘッドを軽減できる。

mem_write mem_write(g_addr, data)

グローバルアドレス g_addr で指定されるリモートアドレスに SYSWR を用いてデータ data を書き込む。EM-X ではリモートメモリ書き込みは一方向の処理でありレイテンシは生じないので、inline 展開された場合にはスレッド切り替えは起きない。グローバルアドレスおよびデータがレジスタ上にあれば send の 1 命令に変換される。int data; int global *g_addr; *g_addr = data; と同じ。

mem_read mem_read(g_addr)

グローバルアドレス g_addr で指定されるリモートアドレスから SYSRD を用いてデータを読み出

す。リモートアクセスレイテンシ隠蔽のためにスレッド切り替えを行う。グローバルアドレスがレジスタ上にあれば lpa および send の 2 命令に変換される。詳細は 3.2.3 を参照。int data; int global *g_addr; data = *g_addr; と同じ。

I_write I_write(g_addr, data)

グローバルアドレス g_addr にデータ data を同期書込みする。すなわち、すでに g_addr に同期読み出し要求が存在すれば、data をその要求に渡して、同期フラグをクリアする。もし、同期読み出し要求が存在しなければ、同期フラグをセットして data を g_addr に書き込む。もし、同期書き込み要求がすでに存在していれば、対応するトラップハンドラを起動する。

I_read I_raed(g_addr)

グローバルアドレス g_addr からデータを同期読み出しする。すなわち、すでに g_addr に同期書き込み要求が存在すれば、メモリデータを読み出して結果として返すとともに、同期フラグをクリアする。もし、同期書き込み要求が存在しなければ、同期フラグをセットして返りアドレスを書き込む。もし、同期読み出し要求がすでに存在していれば、対応するトラップハンドラを起動する。

lock lock(pe_addr)

pe_addr で指定されるプロセッサにロック要求を送る。もし他のロック要求がすでに存在していれば、そのプロセッサのキューに追加され、ロックが解除されるのを待つ。もし他のロック要求がなければロック要求を受付け、アクノリッジを返す。PE0 へ lock 要求を送る lock0() や自 PE へロック要求を送る lockl() もある。

unlock unlock(pe_addr)

pe_addr で指定されるプロセッサにロック解除要求を送る。もし他のロック要求がキューに存在していれば、その先頭の要求について受付ける。lock0(), lockl() に対応する unlock0(), unlockl() もある。

FLOAT_ARG FLOAT_ARG(f)

float 型変数を float 型のまま引数とすることをコンパイラに指示するためのマクロ。C では float 型の引数はいったん double 型に変換されてしまうが、EM-X では double 型をハードウェアでサポートしていないため、これは大きなオーバヘッドとなる。この変換を防ぐためのマクロ。このマクロを用いた、リモートに float 型データを書き込むための関数 float_write() もある。

FLOAT_PARAM FLOAT_PARAM(f)

float 型を結果とする関数の場合、C ではいったん double 型で結果を返すことになっているが、これを float 型として返すようにコンパイラに指示するためのマクロ。このマクロを用いた、リモートの float 型データを読み出すための関数 float_read() もある。あるいは、float fdata; float global *g_addr; fdata = *g_addr; というように float 型グローバルポインタを用いれば、このような変換をせずに float 型の変数にアクセスすることができる。

4.2.2 EM-X 専用ライブラリ

EM-X のハードウェア機構を利用した種々のライブラリが提供されており、高速なデータ通信や豊富なデバッグ機能を容易に利用できる。

ブロック転送

mem_copyout(int *src, int global *dst, size) はローカルアドレス src から始まるメモリの内容をグローバルアドレス dst に size ワード分だけコピーする関数である。EM-X ではワード単位の通信のみをハードウェアでサポートしているため、基本的には以下のようなループを実行することになる。

```
int *soc;
int global *dest;
for (i=0;i<size;i++) {
    dest[i] = soc[i];
}
```

```
L22: ld       r10,0x0,r8
     send1.pt r8,r13,0,0,0x24 ; mem_write
     add      r10,4,r10
     add      r13,4,r13
     blt      r10,r12,L22
     nop                      ; delayed
```

現在の EM-C の生成するコードは、右に示す通りループあたり 6 命令である。これに対し、アセンブラレベルの最適化を施すとともに、16 ワード (64 バイト) 以上のデータ転送に対して 16 ループのアンローリングを行って更なる高速化を行ったライブラリが mem_copyout() である。まず、16 ワード未満の部分に対するループ処理部分を示す。ここでは遅延スロットを埋めるとともに、ld.a 命令を用いてループあたり 4 命令で処理されている。ld.a および send3 命令については以下で説明する。

```
L0:
    ld.a       ap,4,r9
    send3      r9,r6,0,0        ; mem_write
    blt        ap, r5, L0
    add        r6, 4, r6        ; delayed slot
```

次に、ループをアンロールしたアセンブラ命令の例を以下に示す。以下では簡単のため 16 ワード分をアンロールする代りに 4 ワード分をアンロールした例を示す。コードから分かる通り 4 ワード転送が 10 命令で実行できる。16 ループアンロールのコードでは 16 ワード転送が 34 命令で実行できる。

```
L1: ld.a ap,4,r9
    send3 r9,r6,0,0 ; mem_write
    ld.a ap,4,r9
    send3 r9,r6,0x4,0 ; mem_write
    ld.a ap,4,r9
    send3 r9,r6,0x8,0 ; mem_write
    ld.a ap,4,r9
    send3 r9,r6,0xc,0 ; mem_write
    blt ap,r4,L1
    add r6,0x10,r6
```

以下に、適用した最適化について説明する。

- ld.a という実効メモリアドレスを自動更新するロード命令を用いる。ロード命令に.a 属性を付加すると、その命令の実効メモリアドレスを特殊レジスタ ap に格納する。例えば、ld.a ap,4,r9 という命令は ap+4 を実効メモリアドレスとしてメモリを読み出し、結果を r9 に格納するとともに、ap+4 を ap に格納する。そのあと再度 ld.a ap,4,r9 を実行するとその更新された ap を利用するため、アドレス更新のオーバヘッドが無くなる。ループアンローリング時にはベースアドレス相対のアドレッシングを用いることでロード毎の更新は不要であるが、それでもループ毎の更新を省

く効果はある。また、16 ワード以下のループの場合は、この命令によりループあたり 1 命令が省略できる。

- send3 という命令は ld 命令のベースアドレス相対のアドレッシングモードのように、パケット転送先のアドレスをベースアドレス相対で指定するパケット送信命令である。例えば、send3 r9,r6,0x4,0 という命令は r6+4 を転送先アドレス、r9 をデータとしてパケットを送信する命令である。この時生成されるパケットのタイプはアドレス指定レジスタのデータタグで指定される。ループに入る前に転送先アドレスをレジスタに設定する際にデータタグを SYSWR に設定しておけば、この命令に与える変位を変えるだけで、順次宛先を更新する SYSWR パケットを生成できる。これにより転送先アドレスを更新するオーバヘッドはループ内で一度になる。
- 分岐遅延スロットに転送先アドレスの更新命令をいれることによりループオーバヘッドを削減できる。

この他、以下のようなライブラリ関数がある。

msg_alloc msg_alloc(pe_addr, size)
mem_copyout() で書き込むための領域をプロセッサ pe_addr に確保する。remote_call(pe_addr, malloc, 1, size*sizeof(int)) と同じ。

mem_copyin mem_copyin(dst, src, size)
グローバルアドレス dst からローカルアドレス src に size ワード分だけコピーする。コピーには SYSRD を利用する。通常、SYSRD を発行するとスレッド切り替えを起こすが、このライブラリでは mem_copyout() と同様にループアンローリングを行い、連続的に SYSRD を発行する。また、SYSRD の結果パケット (continuation) のパケットタイプを SYSWR とすることにより SYSRD の発行と結果の格納を同時に行うことを可能としている。size 分 SYSRD を発効後に、一度だけ USRRD パケットを dst プロセッサに発行することにより、全ての SYSRD 要求が到着済であることを保証できる。mem_copuout() では SYSWR を用いており一方向の転送で済むのに対し、mem_copyin() では SYSRD を使うため往復の転送が必要となりネットワークトラフィックは 2 倍となるが、データ転送はパイプライン的に行われるためレイテンシはそれほど問題とならない。転送終了の確認をほぼ caller 側のみで行えるため、直ちに受け取ったデータの処理に取り掛かれるなど、データのハンドシェイクが容易になる場合がある。

mem_copyin0 mem_copyin0(dst, src, size)
グローバルアドレス dst で指定されるプロセッサに mem_copyout() を remote_call() する。結果は mem_copyin() と同じであるが、mem_copyout() を用いるためネットワークトラフィックは半分となる。ネットワークトラフィックが大きい同時転送時にはこちらの関数を用いたほうがネットワーク負荷が軽減されて高速になる場合もある。remote_call(PE_ADDR_OF(dst), mem_copyout, 3, dst, GLOBAL_ADDR(self_pe(), src), size) と同じ。

バリア同期/リダクション処理

EM-X ではバリア同期を直接サポートするハードウェア機構は無いが、局所同期を組み合わせてバリア同期を実現している。このライブラリでは、図 4.6 に示すような scan スタイル [21] の実装を行っている。この場合、パケット送信パターンはバタフライネットワークとなり、バリアに参加するプロセッサ数を n とすると、必要なステージ数は ceil(log n) となる。同じ構造を用いて各種リダクション処理を行うことが可能である。EM-X の専用ライブラリとしては、リダクション関数のみを提供している。最も

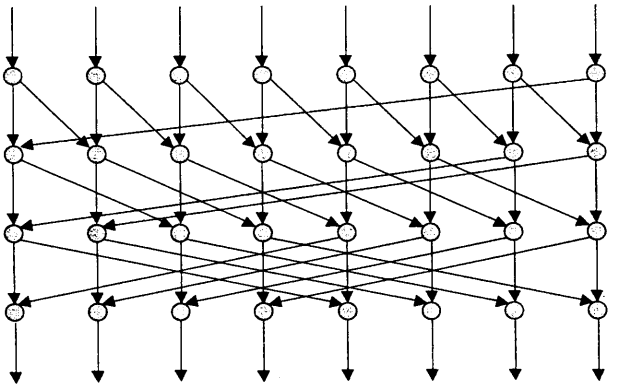

図 4.6: バリア同期の実装

基本的なリダクション関数である各プロセッサのデータの総和を求める barrier_adds() をバリア同期関数として用いる。

各プロセッサでは他プロセッサとの同期が必要であるが、ライブラリではこの同期に I_structure[Arv89] による細粒度同期を用いている。I-structure では、書き込みがまだ行われていないメモリに対して読み出し要求を行うと、書き込みが行われるまで待ってから、その値を返す。これによりメモリデータ単位での局所同期が実現される。I-structure による局所同期はハードウェアでサポートされており、同期ミス時の処理はスレッド実行とは独立に行われる。

barrier_adds() のコアの部分のソースを以下に示す。実際にはノード数が 2 の巾乗で無いときの処理などが加わる。barrier_var 構造体はバリアを実現するための作業領域で、バリアの初期化時に生成される。基本的にはバリアに参加するノード数を n としたときに log(n) 個の局所同期のための作業配列 sync と、各ステップでどこの作業領域を用いて局所同期を行うかをあらかじめ計算しておいたテーブル dest からなる。あとは、このテーブルを用いて図 4.6 にあるとおり、順に局所同期をとっていけば全体の同期/リダクション処理が行える。構造体は (実装の容易化のために) バリアに参加するノードで同じ局所アドレスに配置されることが必要であるが、それ以外には制限はなく、異なる構成のバリア同期を同時に複数実行することなども可能である。

```
barrier_adds(n, barvar)
int n;
struct barrier_var *barvar;
{
    int i;

    for (i=0; i<barvar->stage; i++) {
        I_write(barvar->dest[i+1], n);
        n += I_read(GLOBAL_ADDR(self_pe(), &barvar->sync[i+1]));
    }
    return(n);
}
```

また、各ノードのデータと I_read() で受け取ったデータを適宜処理することにより、他のリダクション処理も容易に実現できる。以下にライブラリとして提供しているリダクション関数を示す。

init_barriers init_barriers(pes, petbl, barvar)

バリア関数用の作業領域となる barvar を初期設定する。pes はバリアに参加するプロセッサ数。

petbl はそのプロセッサ番号からなる配列。barvar は初期化する構造体。ここで pes のみが必須で、petbl, barvar として 0 を指定すると、システムの持っているプロセッサテーブルおよびバリア用作業領域が利用される。

barrier_adds barrier_adds(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の総和を計算する。barvar として 0 を指定すると、default の構造体が用いられる。この関数のみアセンブラレベルでの最適化がなされている。

barrier_ands barrier_ands(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の論理積を計算する。

barrier_ors barrier_ors(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の論理和を計算する。

barrier_eors barrier_eors(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の排他的論理和を計算する。

barrier_maxs barrier_ors(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の最大値を計算する。

barrier_mins barrier_mins(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の最小値を計算する。

barrier_mins barrier_mins(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n の最小値を計算する。

barrier_muls barrier_muls(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n を全て掛け合わせた値を計算する。

barrier_addf barrier_addf(n, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、引数 n を単精度浮動小数点数として全て足し合わせた値を計算する。

barrier_funcs barrier_funcs(n, func, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、各プロセッサの引数 n を 2 引数関数 func() で処理した値を計算する。

barrier_addv barrier_addv(soc, res, size, barvar)

barvar 構造体にて指定されたプロセッサ間でバリア同期をとるとともに、配列 soc の各インデックスの総和を配列 res に返す。

scan_adds scan_adds(n, barvar)

barvar のプロセッサテーブルにより指定されるプロセッサ 0 から自プロセッサの直前までの引数 n の部分和を計算する。

scan_addv scan_addv(soc, res, size, barvar)

barvar のプロセッサテーブルにより指定されるプロセッサ 0 から自プロセッサの直前までの部分和を配列 soc の各インデックスについて計算し配列 res に返す。

ブロードキャスト

ブロック転送やバリア同期のように、1 対全のブロードキャストも EM-X ではワード単位の通信や局所同期等を組み合わせて実現している。しかし、ワード単位の通信を単純に組み合わせただけでは、n プロセッサに m ワードのデータをマルチキャストしようとすると、$n \times m$ に比例した時間がかかってしまう。ここではある程度大きなデータを多くのプロセッサにブロードキャストすることを前提に、データをパイプライン的に転送する方式を考える。これはプログラムのロードなどで利用している以下の特殊パケットハンドラを利用する。

```
DISTI:                              ; ptype = 0x2b
        st      fp,0,pr0
        ld      zr,NEXT0diff,r0
        add     r0,fp,r0
        send2   pr0,r0,0,0
        .break
```

このルーチンは、パケットがやってくると USRWR と同様にアドレス部で指定されたメモリに、データ部の内容を書き込む。次に、あらかじめ設定しておいた次のプロセッサ番号との差分 (NEXT0diff) をアドレス部に足し込むことにより、次のプロセッサへのアドレスを生成して、パケットを転送する。この NEXT0diff にはパケットタイプ DISTI(0x2b) も一緒に格納されている。パイプライン転送の最後から 2 番目のプロセッサの NEXT0diff のパケットタイプを SYSWR(0x23) とすることによりパイプライン転送の最終段ではメモリ書き込みのみが行われ、パイプライン転送が終了する。パイプラインの最終段かどうかの分岐をなくすことにより 1 命令削減できた。このルーチンでは 4 クロックサイクルに 1 ワードのデータ転送を行うことが可能である。

このルーチンを用いたブロードキャスト関数を以下に示す。本関数を用いると、80 プロセッサに 4096 ワードのデータをブロードキャストするのに、928 ミリ秒かかっており 17.7M バイト/秒のスループットである。これは 2 プロセッサ間転送の約 1/2 のスループットであるが、全 PE に転送が完了していることを確認した場合の性能であり、各 PE へ mem_copyout() を行うのに比べると約 40 倍の性能である。

em_broadcast_init em_broadcast_init(petbl, id, pes, v)

ブロードキャスト関数のための初期化を行う。ブロードキャストの送信、受信を行う各プロセッサで一度だけ実行しておくことが必要。petbl はブロードキャストのパイプライン転送に使用するプロセッサ番号の配列。バリア同期の初期化を同じ petbl を使用して事前に行っておくことが必要。id は petbl 内の index。pes はブロードキャストに参加するプロセッサ数。v はブロードキャスト関数が使用するワークエリアを保持する構造体で、0 を指定するとシステム default の構造体を利用する。

em_broadcast_send em_broadcast_send(array, size, v)

ブロードキャストの送信を行う。array は転送すべき配列で size はそのワード数。各プロセッサの

同じローカルアドレスにブロードキャストされる。v はブロードキャスト関数が使用するワークエリアを保持する構造体で、0 を指定するとシステム default の構造体を利用する。構造体内の next プロセッサが正しい値に変更されたことを確認するために、ブロードキャストに参加するプロセッサで一度バリア同期を行ってから、DISTI パケットを用いてブロック転送する。転送が終了すると、next プロセッサに転送完了を通知して、自分はブロードキャスト全体の完了を待たずに終了する。

em_broadcast_receive em_broadcast_receive(sender, v)

ブロードキャストの受信を行う。ブロードキャストの送出先を指定することにより、自分がパイプライン転送の最後から 2 番目かどうかをチェックして、それに応じて構造体内の next プロセッサ変数のパケットタイプを DISTI か SYSWR に設定する。この設定の完了を確認するために、ブロードキャストに参加するプロセッサでバリア同期を行う。あとはブロードキャストの転送完了を直前のプロセッサとの間で局所同期を行い、完了通知を受けとったら、次のプロセッサに転送完了を伝える。

時間計測

EM-X にはタイマ機能はないが、インタフェースボード上のプロセッサでソフトウェアによるタイマ機能を実現している。具体的には、20 クロックサイクル (1 マイクロ秒) に 1 だけ増加するカウンタをソフトウェアにより実現して、パケット起動によりループさせる。他のプロセッサからそのプロセッサに現在のカウンタを問い合わせるパケットを送ると、そのパケットにより起動されるスレッドで、現在のカウンタの値を返すとともに、その処理を含めて 20 クロックサイクルでカウンタを 1 増加させるように調整して、カウンタの動作に影響を与えないようにしている。このような機能を em_utime() というライブラリ関数により提供している。これにより、em_utime() を 2 回呼び出してその間の経過時間をマイクロセカンド単位で得ることができる。この方式ではネットワークトラフィック等により誤差が生じるが、通常は計測する処理の前後のトラフィックが少ない状態で em_utime() を呼び出すので、この方法で十分正確な時間の計測が可能である。逆に、そうでない状況で時間計測を行う場合には十分注意が必要である。

em_init_utime em_init_utime(pe, pes, petbl)

タイマースレッドをプロセッサ pe で起動する。そのスレッドへのオペランドセグメントを petbl 内の pes 個のプロセッサにコピーする。他のプロセッサでは利用しない時には、pes として 0 を指定する。

em_utime em_utime()

タイマースレッドにアクセスして、現在のカウントを得る。カウントは 20 クロックサイクルに 1 増加しているので、1 カウントが 1 マイクロ秒の経過に相当する。

実行トレース

EM-X にはメンテナンス回路が備わっており、ホスト計算機から各プロセッサの状態 (フリップフロップやレジスタファイル、FIFO の内容など) 全てを観測できるとともに、そのうちの一部は変更することもできる。また、EM-X の回路はスタティック回路のみから構成されておりクロックを停めて状態を保持しておくことが可能である。メンテナンス用のクロックはメインクロックとは別系統となっているため、メインクロックを停止した状態で、各プロセッサの状態を取得することができ、非常に精度の高い状態取得が可能である。ただし、クロックの制御に時間がかかるため毎サイクルクロックを停止しての

状態取得には時間がかかる。一方、メインクロックを走らせた状態でも、メンテナンス回路より各プロセッサの状態取得は可能である。ただし、この場合、一度に1つの状態しか取得できないため、各プロセッサ/各フリップフロップの取得時間が異なることに注意が必要である。一つのメンテナンスアドレスを全プロセッサについて取得するには、取得データをディスクに保存する時間も含めて2ミリ秒ほど必要である。したがって、2ミリ秒周期での状態取得には、プログラム実行に全く影響を与えずに行うことが可能である。これらの実行トレースの取得を制御するためのライブラリ、および所得したライブラリを表示するためのツールとして以下のようなものがある。

em_mtrace em_mtrace(char *path, int ctrl, int mode, int pe, int clks)

path はトレースファイルを格納するファイルへのパスでトレース開始時のみ指定。NULL を指定すると default として/tmp/exmt_trace に格納する。ctrl はトレースの制御を行う。指定できるのは以下のいずれか。

MTRACE_START トレース開始

MTRACE_STOP トレース中断

MTRACE_RESUME トレース再開

MTRACE_END トレース終了

mode によりトレースの種類を指定する。以下の指定を組合わせ可能。

MTRACE_PESTAT 80PE のステータスレジスタの取得

MTRACE_IFSTAT インタフェーススイッチ上のプロセッサステータスレジスタの取得

MTRACE_PCTRACE 引数 pe で指定したプロセッサのプログラムカウンタの取得

MTRACE_BINARY トレース結果をバイナリで保存

MTRACE_CLKSTOP 引数 clks でしたクロック刻みでクロックを停めながらトレースを実施

stat2t stat2t trace_file

em_mtrace() で取得したトレースデータを以下のように集計して表示するコマンド。以下は MP3D の実行をトレースした結果を stat2t で表示したものである。EXE はプロセッサが実行中であった回数をプロセッサ毎に表示している。WAIT はプロセッサがパケット出力待ちのためにストールしていた回数をプロセッサ毎に表示している。次に、トレース回数とシステムコール中であった回数を表示している。最後に、1度も実行中でなかったプロセッサは実行が割り当てられていなかったものとして、動作プロセッサ数を表示するとともに、プロセッサが実行中であった割合を全プロセッサの平均、プロセッサのなかでもっとも動作割合の高かったものの割合、低かったものの割合を示している。最後に、パケット出力待ちでストールしていたプロセッサの割合を同様に表示している。

```
EXE
GA0          372       328       317       314       304
GA1          316       336       322       329       315
GA2          335       341       338       314       318
GA3            0         0         0         0         0
GA4          323       326       325       330       311
GA5          329       320       336       330         0
```

```
GA6          333      332      329      335        0
GA7          348      333      324      337        0
GA8          337      334      329      335        0
GA9          339      310      333      313        0
GA10         316      333      304      323        0
GA11         326      324      333      330        0
GA12         326      326      313      336        0
GA13         331      323      346      308        0
GA14         326      331      311      320        0
GA15         324      312      345      326        0
WAIT
GA0           20        2        1        1        0
GA1           26       13        1        2        2
GA2           14       15        2        0        2
GA3            0        0        0        0        0
GA4           19       16        0        2        0
GA5           10        0        2        0        0
GA6           13        4        4        2        0
GA7           15        1        0        1        0
GA8           10        4        1        2        0
GA9            8        0        0        2        0
GA10          12        2        1        0        0
GA11           1        1        3        1        0
GA12           2        2        0        0        0
GA13           8        0        0        1        0
GA14           3        3        2        1        0
GA15           7        1        0        1        0
step 577 + syscall 4
64 PEs run, activity (ave: 56.66%, max: 64.47%, min: 52.69%)
            wait (ave:  0.73%, max:  4.51%, min:  0.17%)
```

stat2x stat2x trace_file

em_mtrace() で取得したトレースデータを図 4.7 のように X 上に表示するコマンド。縦軸が各プロセッサに対応し、横軸が時間経過を表わす。1 ドットが何マイクロ秒に対応するかは、トレースデータのとり方に依存している。1 トレース 1 ドットで表わしている。黒が命令実行中である状態を、赤がパケット出力待ちでストールしている状態を、緑が ls 命令の 2 サイクル目を実行していることを、青がシステムコールを実行中でサンプリングが行えなかったことを、白がそれ以外のアイドル状態を示す。図 4.7 は MP3D を 64 プロセッサで実行した例であるが、PE15-19 や PE79 等が割り当てられていないことが分かる。また、各プロセッサはずっとスレッドを実行しているのではなく、スレッドが切り替えられながら実行していることが分かる。

その他

その他、libemx.a に含まれている関数には以下のよう関数がある。

aputc aputc(char c)

デバッグするときにprintf()を挿入して途中結果をプリントすることがあるが、EM-Xではprintf()はスレッドの中断やホスト計算機との通信を伴うため、それによって実行状況が大きく変化してしまう。aputc()は単に1文字をホスト計算機に送るだけでスレッドを中断すること無く処理が継続されるため、それほど状況を変えずにどこまで進んだかをチェックするのに有用である。ホスト計算機側では送られてきた文字のみを表示する。

em_mkpkt em_packet(addr, data, ptype)

EM-Xでは種々のパケットタイプを指定することができるが、EM-Cでサポートしているのはその一部のみである。任意のパケットを生成するための関数がem_packet()でaddrでグローバルアドレスを、dataで送信するデータを、ptypeでそのパケットのパケットタイプを指定することにより、指定されたパケットを生成して送信する。

em_get em_get(addr, ptype)

em_packet()はSYSWRのような一方向のパケット送信であるのに対し、em_get()はSYSRDのような結果を受け取るような任意のパケット送信を行うための関数である。addrでグローバルアドレスを、ptypeでパケットタイプを指定する。スレッドを継続させるためのcontinuationを作成し、それをデータとしてパケットを生成し送信したあと、スレッドを中断する。

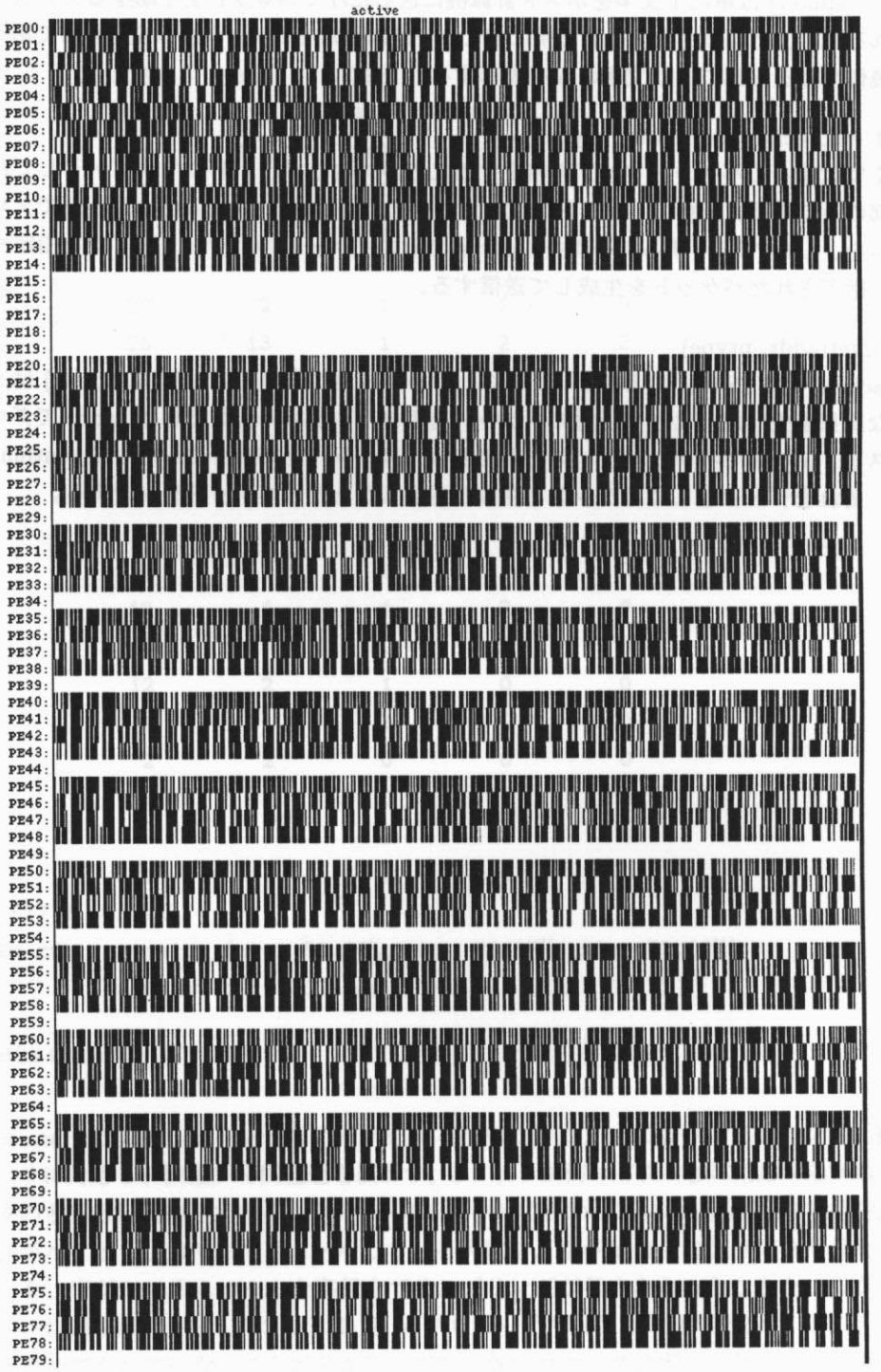

図 4.7: stat2x の表示例

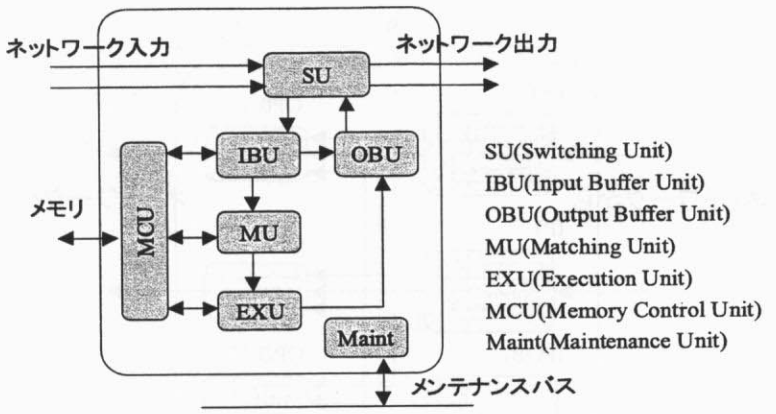

図 4.8: EMC-Y の構成図

4.3 プロセッサアーキテクチャの詳細

ここでは、EM-X の要素プロセッサ EMC-Y の詳細について述べる。図 4.8 に EMC-Y の構成図を示す。EMC-Y は図に示すように、スイッチングユニット (Switching Unit, SU)、入力バッファユニット (Input Buffer Unit, IBU)、出力バッファユニット (Output Buffer Unit, OBU)、待ち合わせユニット (Mathcing Unit, MU)、実行ユニット (Execution Unit, EXU)、メモリ制御ユニット (Memory Control Unit, MCU)、メンテナンスユニット (Maintenance Unit, MAINT) からなる。以下に各ユニットについて詳しく述べる。

4.3.1 SU(スイッチングユニット)

SU(スイッチングユニット) は各 PE を接続するネットワーク間のパケットの入出力を行うユニットである。ネットワークからの入力ポートが 2 ポート、プロセッサからの入力ポートが 1 ポートと、入力ポートは計 3 ポートある。また、ネットワークへの出力ポートが 2 ポート、プロセッサへの出力ポートが 1 ポートと、出力ポートは計 3 ポートある。各ネットワークポートは、データ信号が 39 ビット、転送要求信号が 2 ビット (3 バンクへの要求を 2 ビットに符合化)、転送許可信号が 3 ビット (3 バンクそれぞれ 1 ビット) の計 44 ビットの信号からなる。この入出力各 3 ポートがクロスバー的に接続されている。図 4.9 に SU の構成図を示す。SU は次の 6 つのユニットからなる。グループ内ネットワークからの入力ポートの制御を行う IP0。グループ間ネットワークからの入力ポートの制御を行う IP1。IP0 と IP1 は同じ回路である。プロセッサからの入力ポートを制御する IPOBU。グループ内ネットワークへの出力ポートを制御する OP0。グループ間ネットワークへの出力ポートを制御する OP1。OP0 と OP1 は同じ回路である。プロセッサへの出力を制御する OPIBU。

IP0、IP1 にはバンクバッファと呼ばれるパケットバッファがそれぞれ 3 パケット分ある。パケットをネットワークからいったん受け取ってから、ネットワークへ出力するストアアンドフォワード型のプロトコルでは、ネットワーク上にあるパケットの送り先のバッファがすべて塞がってしまうと、デッドロックが生じてしまう。例えば、図 4.10 のように、リング上に接続された 4 つのプロセッサが、同時に 2 つ先のプロセッサにパケットを送ろうとすると、次のクロックサイクルでは、全てのバッファが埋まってしまい、どのプロセッサもパケットを送ることができなくなり、デッドロックしてしまう。これを回避するため、IP0 および IP1 ではネットワークからのパケットが格納されるパケットバッファが 3 バンクからなる構造をとっている。プロセッサから出力されるパケットは、最初、バンク 0 を利用して SU 間を転送される。パケットがネットワーク上のあるステージ (現在の実装では第 0 ステージ) を通過する際に、格納するパケットバッファとして一つ上のバンクを用いるように変更する。この時、高いバンクか

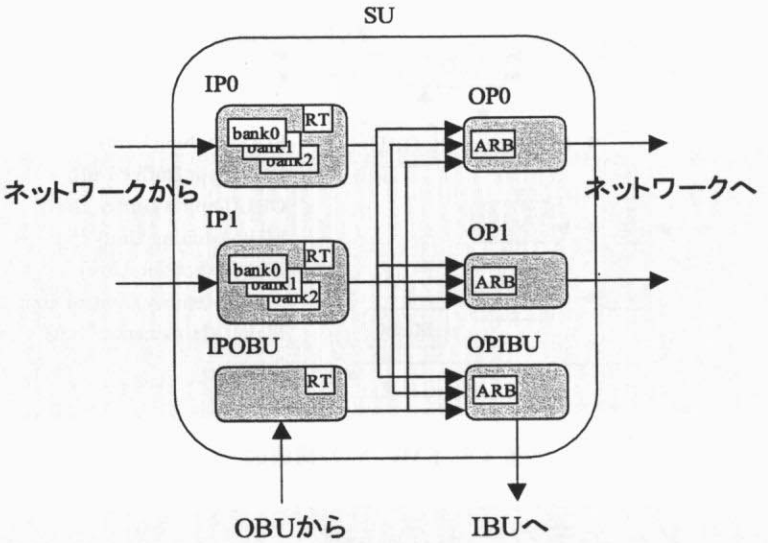

図 4.9: SU の構成図

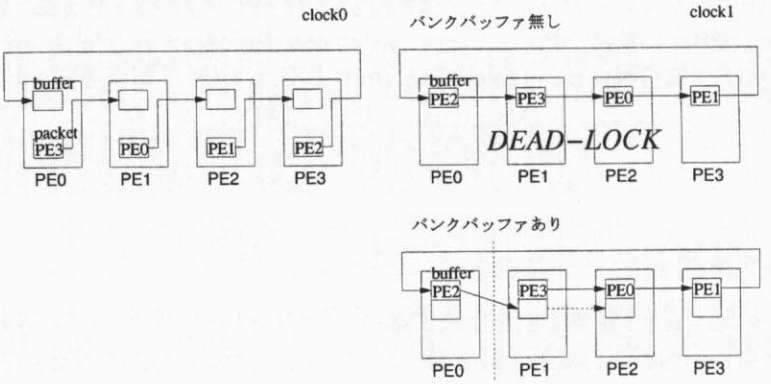

図 4.10: バンクバッファによるデッドロックの回避

らの転送要求を優先的に転送することにより、上記のようなデッドロックを生じなくすることが可能である。例えば、図 4.10 の場合では、PE0 上にある PE2 宛のパケットは PE0 を通過するパケットであるので、PE1 へはバンク 1 への転送要求となり、PE1 のバンク 0 に PE3 宛のパケットがあっても、転送要求は受け付けられる。バンク 1 がバンク 0 よりも優先的に転送されるため、次のクロックサイクルでは、PE1 のバンク 0 にあるパケットではなく、バンク 1 にあるパケットが PE2 のバンク 1 へ転送される。こうしてデッドロックなしにパケット転送が行われる。この方式はバーチャルチャネル [Dal90] を用いたデッドロック回避と同様であるが、バーチャルチャネルを用いた他のネットワークでは宛先プロセッサに到着するまでの最大ホップ数だけのバンクを必要とするのに対し、サーキュラオメガネットワークでは、最適ルーティングにより最大 2 周するまでに宛先プロセッサに到着することが保証されており、プロセッサ数によらずに 3 個のバンク構成でデッドロックを回避できることが知られている [22]。

さらに、宛先プロセッサがシステム中に無いような迷子パケットは、ネットワークを周回する度にバンクが上がるため、第 0 ステージに 3 度到着した時点で迷子パケットであると判定できる。SU ではこのようなパケットを検出した場合、パケットの HST ビットを 1 に変えて、迷子パケットをホスト計算機に送り返すことが自動的に行われる。

パケットのルーティングは、IP0、IP1、IPOBU それぞれの RT 回路で行われる。RT では、パケットに含まれている宛先 PE 番号を元に、各 SU が最短経路で転送されるように出力ポートを選択する。

具体的なルーティングアルゴリズムを以下に示す。通常の最適ルーティングの他に、インターフェーススイッチやフレームバッファのようなグループ構成をとらないプロセッサでのルーティングであるI/Oルーティングや、ホスト計算機宛のパケットのルーティングを行うホストルーティングがあり、後者ほど優先順位が高い。これらのルーティングをパケット内のホスト領域やパケットタイプ領域、およびプロセッサ内のフラグにより切り替えるようになっている。以下の順にチェックをして該当ルーティングを選択する。ただし、パケット宛先PE番号を(ga, ca)、ルーティングを行うSUのPE番号を(GA, CA)とする。

- 受け取ったパケットのホストフィールドが1の場合:
  - PE内のHSTRTフラグが0の場合: グループ内出力ポート(OP0)へ出力。
  - PE内のHSTRTフラグが1の場合: グループ外出力ポート(OP1)へ出力。
- PE内のIORTフラグが1の場合:
  - ga==GA
    - ca==CA: チップ内出力ポート(OPIBU)へ出力。
    - ca!=CA: グループ内出力ポート(OP0)へ出力。
  - ga!=GA: グループ外出力ポート(OP1)へ出力。
- それ以外の場合:
  - ga==GA
    - ca==CA: チップ内出力ポート(OPIBU)へ出力。
    - ca!=CA: グループ内出力ポート(OP0)へ出力。
  - ga!=GA, CA==0
    - parity(ga)==parity(GA): グループ内出力ポート(OP0)へ出力。
    - parity(ga)!=parity(GA): グループ外出力ポート(OP1)へ出力。
  - ga!=GA, CA!=0
    - ga[7-CA]==GA[7-CA]: グループ内出力ポート(OP0)へ出力。
    - ga[7-CA]!=GA[7-CA]: グループ外出力ポート(OP1)へ出力。

この他、MLPEと呼ばれる負荷分散用のパケットがある。このパケットは宛先が自PE宛になっており、ネットワーク上の循環路を一周して、その経路でもっとも負荷の低いPEを検出して、元のPEに到着するようになっている。このため、生成したPEでは上記ルーティングではなく、パケットのwcfフィールドにより、次のようなルーティングを行う。ただし、その後のPEにおけるルーティングでは、このような特殊な扱いはせず、通常のルーティングに従う。

- wcf[0]が0の場合: グループ内出力ポート(OP0)へ出力。
- wcf[0]が1の場合: グループ外出力ポート(OP1)へ出力。

OP0、OP1、OPIBUでは、入力ポート制御回路から出力ポートへの転送要求が衝突した場合は、ARB回路により以下の通りに調停する。

- アドレス部の出力が完了している場合は、そのパケットのデータ部転送が最優先
- OBUからの出力はバンク0と考え、バンクが高い方が優先

- 直前に出力した入力ポートを出力ポート毎にLRUに保存しておく(LRU = 0b01 : IP0, LRU = 0b10 : IP1, LRU = 0b00 : OBU, 初期化時は0)
- 3入力が衝突した場合は

  LRU=0b00 の場合 IP0を選択

  LRU=0b01 の場合 IP1を選択

  LRU=0b10 の場合 OBUを選択
- IP0と他のポートが衝突した場合は

  LRU=0b01 の場合 IP1あるいはOBUを選択

  それ以外 IP0を選択
- IP1とOBUが衝突した場合は

  LRU=0b10 の場合 OBUを選択

  それ以外 IP1を選択

4.3.2 IBU(入力バッファユニット)

IBU(入力バッファユニット) はネットワークから当該プロセッサにやってきたパケットをいったんFIFO(ファーストインファーストアウト) バッファに格納するユニットである。

FIFOバッファとしては、プロセッサ内部に小容量(8パケット分) のFIFOのみを用意し、もしこの内部FIFOがあふれた場合には、主メモリ上に確保されたメモリバッファに格納される。内部FIFOに空きが生じると、メモリバッファからFIFO順序を保ったままパケットが内部FIFOに格納される。このメモリバッファへの待避、内部FIFOへの復帰は、すべてハードウェア的に自動的に行われるので、ユーザからは、ほぼ大規模なFIFOがあると考えることができる。

このメモリバッファへのアクセスは、実行中のスレッドを極力妨げないように制御されている。EM-Xでは1サイクルにオフチップの主メモリへ2回アクセスする構成をとっている。すなわち、サイクルの前半フェーズで命令フェッチを行い、後半フェーズでデータの読み出し/書き込みを行う。詳しくはMCU(メモリ制御ユニット) 参照。前半フェーズの命令フェッチはスレッドが実行されている限り毎サイクル実行されるが、後半フェーズのデータアクセスは命令がメモリ参照命令のときのみで毎サイクル実行されるわけではない。そこで、その利用されていないときの後半フェーズを用いることにより、スレッド実行へは影響を与えずに、メモリバッファへのアクセスを行うことが可能になる。IBUがメモリバッファへアクセスしようとしたときに、実行中のスレッドの命令がメモリアクセス要求を行ったときには、命令によるメモリ参照の方が優先される。このために、IBUにはSUから受け取ったパケットを保持するためのレジスタが1パケット分ある。メモリバッファへアクセスするための調停待ちのためだけには1ワード分(1/2パケット分) のレジスタがあれば十分である(実際EM-4では1ワード分のレジスタのみが用意されていた) が、その場合、残りのパケットはSU内に残されることになり、後続パケットは処理されず、ネットワークのホットスポットを引き起こす要因ともなる。そのため、EM-Xではこのレジスタを1パケット分用意することにより、他プロセッサ宛のパケットへの影響を少なくするように考慮されている。

メモリバッファへのアクセスには、内部FIFOからあふれたパケットを格納するための書き込み(ストア処理) と、内部FIFOへ格納するための読み出し(リストア処理) がある。これらの処理は同時に必要となることがあるので、処理間に優先度を設定しておくことが必要である。基本的にはリストア処理

は、内部 FIFO が空になる前に行えれば、実行パイプラインに余分なバブルは生じないので、ストア処理を優先させている。

EM-X では通常のパケットはいったん FIFO に格納されて、到着順に処理される。そのため、到着してパケットが処理されるには、すでに FIFO に格納されている他のすべてのパケットが処理されることが必要となる。この時間はパケットの量や処理内容にもっても変動するため、予測が非常に困難である。また、FIFO による管理だけでは、システム処理など優先的に処理しなければならないような場合に対応できない。さらに、以下に述べる待ち合わせ処理では、一般的なスレッド処理として実行すると、待ち合わせミスの場合には、実行パイプラインにバブルが生じてしまう。そのため、EM-X では以下の 3 点による対応を行った。

2 レベル優先度つき FIFO

優先的に処理するパケット用に FIFO(以下では FIFO-HIGH) を別途用意し、各パケットに優先度を示す 1 ビットのフラグを持たせた。この FIFO も上で述べた FIFO と同じく、内部 FIFO とメモリバッファからなり、完全にハードウェア的にパケットのストア／リストアが制御される。通常優先度の FIFO(以下では FIFO-LOW) 内のパケットは、FIFO-HIGH にパケットが存在しない場合にのみ、処理が行われる。ただし、EM-X ではスレッド処理を他のパケットによっては中断されないので、通常優先度のスレッドが実行されているときに、優先処理パケットが到着してもそのスレッド処理が中断されることはない。

それぞれの内部 FIFO はデュアルポート RAM を用いて実装している。小さな容量の RAM を 2 個用いるよりは、2 倍の大きさの RAM を用いるほうがスペース効率は良い。1 つの RAM を用いることにより、同時に読み書きできる数はそれぞれ一方の FIFO に限られるが、それによる効率の低下はほとんどない。なぜなら、FIFO の読み出しは必ず一方 (FIFO-HIGH が空でなければ FIFO-HIGH からのみ、FIFO-HIGH が空なら FIFO-LOW からのみ) からしか起きないからである。また、書き込みに関しても、ネットワークからの入力は一度には 1 パケットのみであるし、メモリバッファからのリストアも一度には 1 パケットだけである。唯一 2 つの FIFO への同期書き込みが生じるのは、片方にはネットワークからの書き込みが、もう一方にはメモリバッファからの書き込みが必要な場合であるが、この場合、リストア処理を待たせても、実行パイプラインの効率への影響はほとんどない。

そこで、2 つの内部 FIFO は、1 つの RAM をアドレス分割して利用している。すなわち、それぞれの内部 FIFO 用に、書き込みアドレス、読み出しアドレスを保持し、それらのアドレスに、FIFO-LOW には最上位ビットに 0 を、FIFO-HIGH には最上位ビットに 1 を追加したアドレスとして、デュアルポート RAM のアクセスを行う。

また、すでに述べた通り、EM-X では 1 パケット 2 ワードの固定長パケットを用いている。ネットワークやメモリバッファへのアクセスでは信号線の制約などにより一度に 1 ワード分しか転送できないが、チップ内部ではそのような制約は少ないので、一度に 1 パケット分 (2 ワード) を転送する方が効率が良いし、制御も容易になる。この 1 ワード転送と 2 ワード転送の変換をこの IBU で行っている。具体的には、アドレス部用の FIFO とデータ部用の FIFO を別に用意し、書き込み時にはワード単位で行うが、読み出し時には 2 つの FIFO を同時にアクセスして、2 ワード転送を実現している。

直接リモートメモリアクセス

EM-X は分散メモリ型並列計算機であり、各プロセッサはローカルメモリを持っている。通常、分散メモリ型並列計算機では他のプロセッサのメモリを参照することはできないので、プロセッサ間でメッセージ通信を行い、相手からメモリ内容をメッセージとして送ってもらう、あるいはメッセージとして送って相手にメモリへ書き込んでもらう必要がある。しかし、この方法では互いのプロセッサで同期的

に処理を進めることが必要で、同期オーバヘッドが生じるとともに、プログラミングが難しくなる。これに対し、EM-X ではプロセッサ番号とその局所メモリアドレスからなるグローバルアドレスを指定することにより、ワード単位のリモートメモリへの書き込みや読み出しをサポートしている。MPI-2 の PUT/GET といった一方向通信に相当する。ただし、このリモートメモリアクセスを通常のスレッド処理として実現すると、送ったリモートメモリアクセスパケットが相手 PE に到着しても、実行中のスレッドが中断するまで待たされてしまう。リモートメモリ読み出しの場合には、それらすべてがレイテンシとなって現れる。たとえネットワーク自体の遅延やネットワークインタフェースの遅延を小さくしても、この実行時レイテンシを小さくすることはできない。

そこで、EM-X ではスレッド実行中でもリモートメモリアクセス要求をサービスできる直接リモートメモリアクセス機構が IBU に備わっている。例えば、直接リモートメモリ書き込み要求パケット (以下 SYSWR パケット) が到着すると、IBU ではパケットのアドレス部で示されるメモリアドレスに、データ部で示されるデータを書き込む。このためのメモリバンド幅には、上記で示したメモリバッファへのアクセスに使用されるメモリアクセスパスを使用する。したがって、実行中のスレッドがメモリアクセス命令を実行している場合は、SYSWR の処理は待たされるが、それ以外の場合は即座に書き込みが終了する。この時のメモリ保護機構については MCU を参照。パケット転送は 1 パケットに 2 サイクルかかるが、SYSWR の処理には 1 サイクルしかかからないため、連続的に SYSWR パケットがやってくるようなブロック転送の場合でも、メモリアクセスパスの占有率は 1/2 を越えない。スレッド実行も、1 サイクル 1 命令実行の単純な RISC コアで実行されるため、メモリアクセス命令の頻度は 1/2 を越えないと考えられる。このため、スレッド実行と、SYSWR によるブロックリモートメモリ書き込みは重畳して実行可能である。

直接リモートメモリ読み出し要求パケット (以下 SYSRD パケット) の場合も同様である。SYSRD パケットが到着すると、パケットのアドレス部で示されるメモリアドレスを読み出す。そして、SYSRD のデータ部で示される戻り番地をアドレス部とし、読み出したデータをデータ部とするパケットを生成し、出力バッファユニット (OBU) へ転送する。SYSRD も基本的に 1 サイクルで処理は終了する。ただし、OBU が full の場合には、full で無くなるのを待ち続けるとデッドロックを引き起こす可能性があるので、OBU への出力をやめて、SYSRD パケットを IBU 内の FIFO-HIGH へ入れる。この場合 SYSRD パケットは通常の高優先度スレッドとして処理される。その後 SYSRD パケットが到着して OBU が full で無い場合は、FIFO に入れられること無く処理が行われるため、同じプロセッサへの SYSRD パケットの結果パケットの到着順序は、出力した順序とは限らない。

このようにリモートメモリアクセスを SYSRD/SYSWR パケットを用いた直接リモートメモリアクセスとすることの利点は以下の通りである。

- リモートメモリアクセスのサービスが、スレッド実行と重畳化して処理可能となり、見掛け上 SYSRD/SYSWR のサービス時間は 0 となる。
- 直接リモートメモリアクセスの実行時レイテンシは、相手 PE での待ち時間がほぼ 0 となるので、静的レイテンシ (他のスレッドやパケットがない場合のレイテンシで、ネットワーク転送およびネットワークインタフェース部の遅延を加えたもの) により見積もることが可能となる。実際には、これにネットワーク上でのパケット衝突などの実行時遅延が付加されたものとなるが、直接リモートアクセス機構を用いないときに比べると大きく削減することが可能となる。
- 直接リモートメモリアクセスのパケットは FIFO には入れられず、FIFO にはスレッド起動などのパケットのみが入ることになる。大規模通信の多くを占める配列データなどを、この直接リモートメモリアクセス機構により通信することにより、通信に必要となる FIFO 容量を大きく減らすことが可能となる。
- リモートメモリアクセス処理をスレッド処理により行う場合、これらのスレッドは 1、2 命令とい

う非常に短いスレッド長となる。このため、FIFOの最悪アクセス頻度等がかなり高くなり、リストア処理が間に合わなくなる可能性が生じる。しかし、直接リモートメモリアクセスを利用することにより、そのような非常に短いスレッドの割合が減ることにより、平均スレッド長が長くなり、FIFO設計への負荷が軽減する。

2入力待ち合わせ

EM-Xでは2入力待ち合わせを直接待ち合わせという方式でサポートしている。データ駆動計算機などでは連想記憶メモリを用いた待ち合わせ方式が一般的であるが、直接待ち合わせ方式は、そのような連想記憶を用いず、通常のメモリを利用できる。直接待ち合わせでは、待ち合わせパケットのアドレス部で、待ち合わせ用の作業メモリのアドレスを指定し、そのメモリの最上位2ビットをフラグとして用いる。そのうちの1ビットで、待ち合わせパケットがいるかどうかを示し、もう1ビットで、そのパケットが右オペランドか左オペランドかを指定する。

この直接待ち合わせを通常のパケット処理として実行すると、2回に1回は待ち合わせ相手が存在しないため、待ち合わせミスとなる。その場合は、パケットデータを待ち合わせメモリに格納するとともに、待ち合わせフラグを適切にセットしただけで処理が終了し、演算ユニットはアイドル状態となる。これが同期オーバヘッドとして陽に見えてしまう。待ち合わせ相手がいないときの処理は、上記の通り非常に単純な処理であるので、これをIBUで処理することにより、同期オーバヘッドをスレッド実行と重畳化できる。

具体的には、待ち合わせパケットが到着すると、待ち合わせアドレスのメモリデータを読み出す。もし待ち合わせフラグが0であれば、相手はまだ到着していないので、パケットデータを待ち合わせメモリに格納するとともに、待ち合わせフラグを適切にセットして処理を終了する。もし、待ち合わせフラグが1であれば、すでに相手が到着していて待ち合わせ後のスレッド実行が可能なので、当該パケットをIBUのFIFOに格納して、次のパケットの処理に移る。この時、待ち合わせフラグの更新は行わない。待ち合わせフラグの更新は次のMUステージで行われる。待ち合わせミスの場合には2サイクルを要するが、パケット転送サイクルも2サイクルなので、特に問題とはならない。また、待ち合わせが成功したときには、パケットはアドレス部とデータ部がすでに入力レジスタに格納されているので、2ワード同時にFIFOへ格納できるため、通常のFIFOへの格納に比べて遅延はない。これらの処理は、他のパケットバッファリング処理や直接リモートアクセスと同様にスレッド実行と重畳化して実行される。この直接待ち合わせは同期メモリ処理であるI-structureの実行にも利用される。

4.3.3 OBU(出力バッファユニット)

OBU(出力バッファユニット)はEXUあるいはIBUで生成されたパケットをいったんFIFOバッファに格納するユニットである。FIFOバッファとしては、プロセッサ内部に小容量(8パケット分)のFIFOを搭載している。もしこのFIFOがあふれた場合には、それ以上のEXUのパケット生成を停める。もしパケット生成以外の命令を実行している場合は、通常通り命令実行は行われる。パケット生成命令を実行する時のみ、実行パイプラインがストールする。

EXUではパケットが1クロックサイクルで生成される。パケットは2ワードからなっており、ネットワークは1ワードずつ転送される。OBUはこのバンド幅の緩衝にも役立っている。すなわち、OBUのFIFOをアドレス部用のFIFOとデータ部用のFIFOの2つのFIFOから構成し、書き込み時には2つのFIFOに同期に書き込むことにより1クロックサイクルで2ワードの書き込みを実現するとともに、読み出し時には交互に読み出すことによりネットワークバンド幅の1クロックサイクル1ワードに合わせている。

OBU へは EXU の他に、IBU からもパケット転送要求がある。これは、直接リモートメモリ読み出しにより生成された返答パケットの転送である。IBU からの要求を優先的に受付ける。ただし、OBU がいっぱいのときに IBU から要求があるときには、デッドロックを避けるために OBU への転送をやめて、読み出し処理をスレッド起動により行うこととしている。

4.3.4 MU(待ち合わせユニット)

MU(待ち合わせユニット) は、待ち合わせ処理の最終段を含むスレッドの起動を行う。MU には IBU からパケットが渡されるが、そのパケットの種類により以下の処理を行う。待ち合わせメモリへの書き込みはメモリアクセスのデータアクセスフェーズ (クロックの後半フェーズ) を用いるが、それ以外はメモリアクセスの命令アクセスフェーズ (クロックの前半フェーズ) を用いる。そのため、MU の処理はスレッド実行とは重畳化されず、待ち合わせメモリの読み出し、テンプレートトップの読み出しがスレッド起動オーバヘッドとなる。逆にこれらが不要な特殊パケットによるスレッドの起動にはオーバヘッドは 0 である。

待ち合わせパケット

IBU から渡されたパケットが待ち合わせパケット (パケットの WCF[1] フィールドが 1 のパケット) の場合、再度待ち合わせメモリを読み出し、待ち合わせフラグ (以下では MF と略記する) のチェックを行い、その結果に応じた処理が行われる。待ち合わせの成功を確認すると、以下の特殊パケットあるいは通常パケットの起動と同様にスレッドの起動を行う。その際、パケットデータの WCF[0] が 0 であれば、パケットデータを汎用レジスタ r28(あるいは pr0 と呼ぶ)、待ち合わせメモリ内のデータを汎用レジスタ r29(あるいは pr1 と呼ぶ)、パケットアドレス部を汎用レジスタ r30(あるいは fp と呼ぶ) に格納する。WCF[0] が 1 の場合は、パケットデータと待ち合わせデータがそれぞれ r29、r28 に格納される。もし待ち合わせパケットでない場合は、パケットデータが r28、r29 の両方に格納される。

MF==’00’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF をパケットの WCF で更新する。通常は IBU で相手が到着していることをチェックしているので、このような場合はないはずであるが、以下のような Q-structure のサポートでは起こりうる。

MF==’01’ の場合: WCF=’10’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’11’ を書き込み、左マッチングエラールーチンを起動する。

WCF=’11’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’10’ を書き込み、右マッチングエラールーチンを起動する。

MF==’10’ の場合: WCF=’10’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’01’ を書き込み、左マッチングエラールーチンを起動する。

WCF=’11’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’00’ を書き込み、対応するスレッドを起動する。

MF==’11’ の場合: WCF=’10’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’00’ を書き込み、対応するスレッドを起動する。

WCF=’10’ の場合: 待ち合わせメモリに当該パケットデータを書き込むとともに、MF として’01’ を書き込み、右マッチングエラールーチンを起動する。

表 4.1: レジスタ一覧

レジスタ番号	内容
r0 - r23	汎用レジスタ
r24	例外処理時のワークレジスタ
r25(ftop)	オペランドセグメントのフリーリストの先頭を指すポインタ
r26(imr0)	即値レジスタ 0。ワード即値をロードするレジスタの 1 つ
r27(imr1, ap)	即値レジスタ 1、あるいはアドレスポインタ。ワード即値をロードするレジスタの 1 つ。また、insb(バイト挿入命令)、divs(整数割算命令)、swap (レジスタ交換補助命令)、ld.a(実効メモリアドレス格納つきメモリ読み出し命令) などの暗黙のレジスタとして使用される。
r28(pr0)	パケットデータ 0。スレッド発火時にパケットデータ (2 入力待ち合わせ時には左オペランド) が自動的に格納される。
r29(pr1)	パケットデータ 1。スレッド起動時にパケットデータ (2 入力待ち合わせ時には右オペランド) が自動的に格納される。
r30(fp)	フレームポインタ。スレッド起動時にパケットのアドレス部が自動的に格納される。このうち [31:22] の 10 ビットはプロセッサ番号を示すフィールドで、通常のレジスタ書き込みでは変更できない。初期化時にホスト計算機からのメンテナンス操作で設定することが必要。addp などの演算結果をパケット出力する命令では、暗黙の出力先オペランドセグメントとして使用される。
r31(zr)	ゼロレジスタ。常に 0 を返す。書き込みおよびバイパス不可。

特殊パケット

パケットの PTYPE フィールドの下位 5 ビットが 0 で無い場合、$0x8000 + PT \times 0x100$ により指定されるルーチンを起動する。主な特殊パケットルーチンを付録 B に示す。これらは、一部ハードウェアで指定されているもの以外は、全てプログラマブルである。

通常パケット

上記の特殊パケットで無い場合、まずパケットのアドレス部の下位 9 ビットをマスクしたアドレスからデータを読み出す。これが起動するスレッドの命令列のベースアドレス (テンプレートトップ (TTOP)) となる。TTOP の下位 9 ビットは 0 にマスクされ、512 バイトアラインとなる。この TTOP にパケットのアドレス部 9 ビットを加えたアドレスがスレッド開始ポインタとなり、そこからスレッドを起動する。ただし下位 2 ビットは 0 にマスクされ、ワードアラインとなる。

待ち合わせメモリのアドレス、TTOP の読み出しアドレス、あるいはスレッドの先頭アドレスが読み出し禁止領域であるときには、不正パケットアドレス (ILPA) として、上記に示したスレッドではなく、エラートラップルーチンを起動する。メモリ保護機構については MCU を参照。

4.3.5 EXU(実行ユニット)

EXU(実行ユニット) は、通常の RISC コアと同様にスレッドを処理する。ただし、他のプロセッサと異なり、リセット直後はアイドル状態であり、MU からスレッド起動要求を受け取ると、実行状態に移行する。

EXU は 2 段のパイプラインを持つ。1 段目の前半フェーズが命令フェッチ、後半フェーズで命令デコー

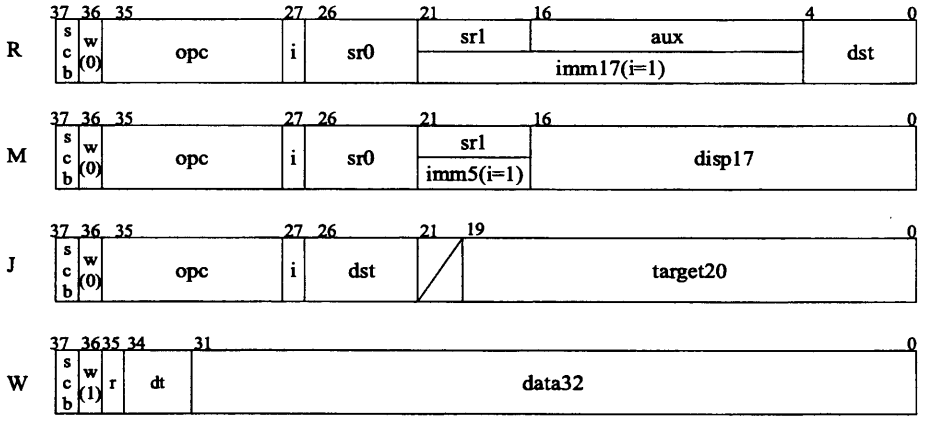

図 4.11: 命令フォーマット

ドおよびレジスタオペランドの読み出しを行う。2 段目で命令実行と結果のレジスタへの格納を行う。命令の結果を次の命令で利用するためのパイパス処理が行われる。

汎用レジスタは、32 個であるが、そのうち 8 個が専用レジスタとなっている。各レジスタの説明を表 reftbl:registers に示す。全てのレジスタが、データ部は 32 ビット、データタグ 6 ビットの計 38 ビットである。浮動小数点レジスタは無く、float 型の演算も同じレジスタを利用する。

EM-X の命令は 38 ビットの 1 ワード固定長であり、メモリの読み書きを同時に必要とする xchg(r) 命令, deq(r) 命令以外は 1 クロックサイクルで終了する。また、EM-X 特有の命令であるパケット出力命令も通常は 1 クロックサイクルで終了するが、出力バッファユニット (OBU) がフルの場合は、パケット出力待ちのためにパイプラインをストールする。メモリロード命令を含めて命令の結果はパイプラインバイパス機構により次命令で利用可能である。分岐命令は 1 命令の分岐遅延スロットを持つ。

命令フォーマットには図 4.11 に示す 4 種類がある。スレッドの継続を示すのが scb 領域で、これが 1 であればスレッドは継続され、次命令のフェッチを行う。0 であれば、スレッドの中断を示し、EXU はアイドル状態になる。それと同時に、新たなスレッドのための命令フェッチが (MU で) 行われる。命令オペコードを示す opc 領域は 8 ビット。ソースオペランド 0 を示す sr0 領域、ソースオペランド 1 を示す sr1 領域、結果格納オペランドを示す dst 領域は、それぞれ 5 ビットで対応するレジスタ番号を示す。

タイプ R レジスタ間演算のための命令フォーマット。opc の他に補助命令オペコードを示す aux が 12 ビットある。i 領域が 0 であれば、sr1 領域がソースオペランド 1 を示すレジスタ番号となる。i 領域が 1 であれば、sr1 と aux の領域を結合して 17 ビットの埋め込み型即値をソースオペランド 1 とする。

タイプ M メモリ演算や条件分岐のための命令フォーマット。disp17 がワード変位を示す即値であり、メモリ演算の場合はベースレジスタに対する変位、条件分岐の時にはプログラムカウンタに対する分岐先の変位となる。i 領域が 0 であれば、sr1 領域がソースオペランド 1 を示すレジスタ番号となる。i 領域が 1 であれば、5 ビットの埋め込み型即値をソースオペランド 1 とする。後者は条件分岐命令に使われる。

タイプ J 分岐命令のための命令フォーマット。target20 が分岐先を示す 20 ビットの値である。

タイプ W ワード即値ロードのための命令フォーマット。w 領域が 1 となる。r 領域でロードするレジスタを imr0 か imr1 かを指定する。dt がデータタグの下位 3 ビットを指定し上位 3 ビットは 0 となる。data32 がデータ部 32 ビットを指定する。

表 4.2: 命令一覧

命令タイプ	命令
整数演算命令	add, sub, addc, subb, mul, div, and, or, xor, xnr, lsl, lsr, lsrr, lslr, asl, asr, rol, sht
浮動小数点演算命令	addf, subf, cvtif, cvtfi, divsf, mulf, maaf, absf
メモリ参照命令	ldr, ld, st, xchg, enq, deq, stb, extb, insb
分岐命令	bcc, jl, jlr, strap, reti
パケット命令	send0, send1, send2, send3, senda, lpa, alup, sendc
その他	swap, lddt, anddt, stdt, chgdt, setmt, ldmt, ldsr, alutst, ldi

命令は、整数演算命令、浮動小数点演算命令、メモリ参照命令、分岐命令、パケット出力命令、その他に分けれられる。一覧を表 4.2 に示す。以下に特徴的な命令について述べるが、各命令の詳細は付録 C を参照。

整数演算命令

整数演算命令には、整数の加減乗除や論理演算、シフト命令が含まれる。add は 32 ビットデータの加算であり、演算結果によりキャリーフラグおよび 2 の補数表現の符号付き整数とみたときのオーバーフローフラグが設定される。シフト命令はバレルシフタを用いており、ソースオペランド 1 により任意桁数のシフト/回転を指定可能である。整数乗算命令 mul は 32 ビットデータの乗算を行い、下位 32 ビットのみを結果として返す。整数除算は整数除算補助命令 div を用いた命令シーケンスで 36 クロックサイクルかかる。現在の C 言語の実装ではソフトウェアトラップを利用して呼び出している。

浮動小数点演算命令

浮動小数点演算命令では、単精度浮動小数点のみをサポートしている。IEEE フォーマットに準拠しているが、デノーマライズ数はサポートしていない。丸め方式も 4 通りから選択できる。addf は浮動小数点同士の加算を行う命令で、1 クロックサイクルで実行される。浮動小数点レジスタはなく、オペランドとしては汎用レジスタを用いる。除算のみ、除算補助命令 divsf を用いた命令シーケンスにより行う。現在の C 言語の実装では、ソフトウェアトラップを利用して呼び出している。初期値として 64 ワードのテーブルを用いてニュートンラプソン法を 4 回反復計算をすることにより 32 クロックで除算を行う。maaf 命令は乗算と加算を同時に行う命令である。直前に実行した乗算の結果を保存するレジスタと、それらを加算していくレジスタの 2 つの特殊レジスタをもっている。n 項からなる内積を n+2 命令 (クロックサイクル) で実行できる。

メモリ参照命令

ldr 命令および ld 命令はともにロード命令であるが、実効メモリアドレスの指定方法が異なる。ld 命令はベースレジスタと、埋め込み型即値によるワード変位により実効メモリアドレスを指定する。ldr はベースレジスタとレジスタ指定によるバイト変位により実効メモリアドレスを指定する。st 命令は ld と同様のアドレッシングモードを持つストア命令である。各命令で auto 領域を 1 に設定すると、実効メモリアドレスを特殊レジスタ ap(r27) に自動的に格納する。例えば、ld.a ap,4; を実行すると、ap には ap+4 が格納されるため、自動ポストインクリメントと同様の効果がある。自動インクリメントでは加

算される値は一定であるが、本方式では、任意の変位を指定できるため、ストライドアクセスなどにも利用できる。メモリ参照はワード単位で行われる。バイト単位およびショートワード単位のアクセスはextb、stb等の命令を組み合わせることにより行う。

この他、EM-X独自のメモリ参照命令がいくつかある。以下の命令は他の命令を組み合わせることにより実行することは可能であるが、命令長が短いスレッドでの性能向上のために専用命令を用意した。

lr lr命令は、ld命令と同様のロード命令であるが、実効メモリアドレスを計算する際に、ベースレジスタの下位11ビットを0にマスクする。これは、オペランドセグメント(関数呼び出しに使われる固定長(128ワード)の関数フレーム)内のデータにアクセスするために、スレッド起動を行ったパケットのアドレス部を保持するfpレジスタをベースレジスタとして利用することを想定した命令である。fpレジスタの値はスレッド開始アドレスのための変位が付加されているため、それをマスクすることが必要である。sr命令は同様の実効メモリアドレス計算を行うストア命令である。

enq オペランドセグメントはフリーリストにより管理されているが、この管理をサポートする命令がenq、deq命令である。enqr命令は現在のオペランドセグメントをフリーリストに追加する命令である。現在のランタイムシステムの実装では、フリーリストの先頭はftop(r25)に格納されており、enqr fp,ftop,ftop;の1命令で、現在のオペランドセグメントの先頭(fpレジスタの値の下位11ビットを0にマスクした値)にftopの値をストアするとともに、その実効メモリアドレスをftopに格納する。enqrを用いなければ3命令必要である。

deq deq命令は、フリーリストからオペランドセグメントを1つ取り出すとともに、そのオペランドセグメントにスレッドの命令列へのベースポインタ(テンプレートトップ)を設定する命令である。テンプレートトップがr0に入っているとすると、deq ftop, r0, ftop;の1命令で、レジスタftopの示すメモリアドレス(フリーリストの先頭を示す。ここには次のオペランドセグメントへのポインタが格納されている)をロードしてftopに格納するとともに、その実効メモリアドレスにr0の値を書き込む。deq命令はロードとストアを行うため、2クロックサイクルかかる。deq命令を使わないと3命令必要である。

分岐命令

bcc命令は条件分岐命令で、条件が満たされると、次命令アドレスに17ビットの符合拡張した値をワード変位として加えたアドレスを次次命令アドレスとする。条件が満たされなければ、分岐せずに実行を継続する。次命令は遅延スロットとしていずれの場合も実行される。ただし、命令内のannulledフラグを1にすることにより、分岐条件が満たされると次命令をフェッチ後にnop命令に置き換えることが可能である。これにより、不要なnop命令を軽減したり、条件分岐を連続的に実行することを可能にしている。

strap命令はソフトウェア例外ハンドラをコールする。システムソフトとして提供されているソフトウェア例外ハンドラの例は付録Bの例外ハンドラ一覧を参照のこと。

パケット出力命令

send命令はパケットを生成して出力する命令であり、生成するパケットフォーマットによってsend0、send1、send2、send3の4種類がある。図4.12にsend命令の命令フォーマットとその生成するパケットフォーマットのアドレス部を図示する。いずれのsend命令もデータ部はsr0により指定されるレジスタの値となる。

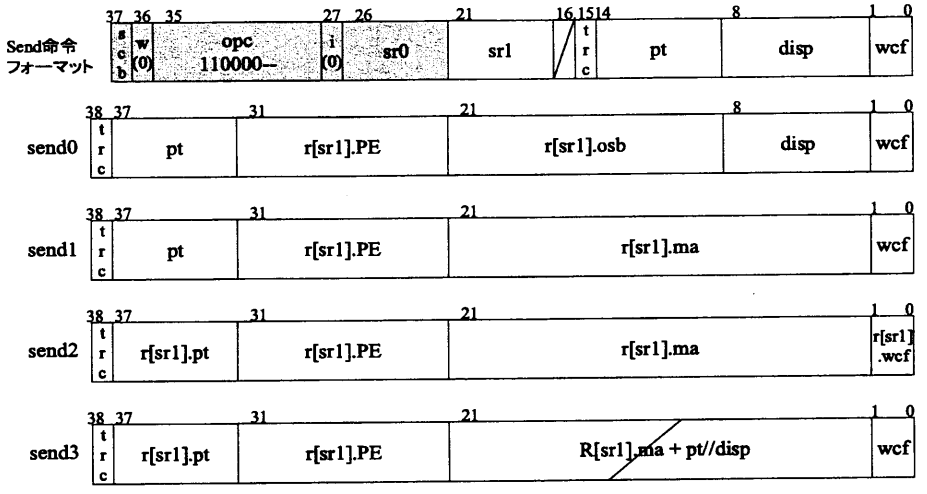

図 4.12: send 命令の生成するパケットアドレスフォーマット

send0 命令 グローバルな (すなわち PE 番号 10 ビットが局所メモリアドレス 22 ビットの上位に付加された) オペランドセグメントアドレスを用いて、それが示す関数フレームへデータ等を出力するのに用いられる命令である。ソースオペランド 1 によりグローバルオペランドセグメントを指定し、埋め込み型即値でトレース (trc)、パケットタイプ (pt)、オペランドセグメント内変位 (disp)、待ち合わせ条件 (wcf) などのパケット領域を指定する。

send1 命令 グローバルアドレスに対するリモートメモリ参照や、待ち合わせ処理等のためのパケット生成に利用される命令である。ソースオペランド 1 によりパケットのグローバルアドレスを指定し、埋め込み型即値で trc、pt、wcf を指定する。

send2 命令 関数の結果を返す際の結果パケットの生成のように、呼び出し側が指定したアドレスにパケットを出力するのに用いられる命令である。ソースオペランド 1 により trc を除く全てのパケットアドレス部を指定する。

send3 命令 グローバルな配列や構造体へのアクセスに利用される命令である。ソースオペランド 1 をグローバルアドレスのベースレジスタとして、埋め込み型即値の pt と disp をつなげた 13 ビットをワード変位として加えたアドレスをパケットのアドレス部とする。パケットタイプはソースオペランド 1 のデータタグ部で指定し、その他の trc、wcf を埋め込み型即値で指定する。このパケットアドレスモードにより、ブロックメモリ転送性能をネットワーク転送ピーク性能にまで高めることができた。EM-X 専用ライブラリのブロック転送を参照。

その他、lpa0、lpa1、lpa2、lpa3 命令は、それぞれ上記 send0、send1、send2、send3 命令で生成するパケットアドレス部を結果レジスタに格納する命令である。例えば、関数呼び出しの戻りアドレスを計算する場合には、fp をオペランド 1 として lpa0 命令でパケットアドレスを生成すれば、それを send 命令のオペランド 0 として利用できる。

その他の命令

その他の命令で主なものを以下に説明する。

データタグ命令 データタグ部を結果レジスタのデータ部に格納する lddt 命令や、ソースオペランド 0 のデータタグ部をオペランド 1 のデータで置き換える stdt 命令などがある。

メンテナンス命令 EMC-Y では、ほぼすべてのフリップフロップや内部メモリにメンテナンスアドレス番号が振られており、その内容を ldmt 命令で読み出すことが可能である。また、そのうちの一部は setmt 命令で内容を書き込むことができる。ホスト計算機からはメンテナンスバスを通じて同様の操作が行える。メンテナンス機構に関してはメンテナンスユニットを参照。

即値ロード命令 35 ビットのデータを特殊レジスタ imr0(r25) か imr1(r26) に格納する ldi 命令である。

4.3.6 MCU(メモリ制御ユニット)

EMC-Y では入力バッファユニット (IBU)、待ち合わせユニット (MU)、実行ユニット (EXU) のそれぞれがメモリへのアクセス要求を行う。メモリ制御ユニット (MCU) はそれらの調停を行って、メモリアクセスを行う。EMC-Y では 1 クロックサイクルで、2 回のメモリアクセスを行う。前半フェーズは命令参照、後半フェーズはデータアクセス等である。EMC-Y のオフチップメモリは、制御を簡単にするために、SRAM が使用されている。クロックサイクルの 1/2 のアクセスサイクルでアクセス可能なようにサイクルタイム 20ns の高速な SRAM が使われている。SRAM としてはフロースルー型の SRAM を用いているが、チップ外にアドレスレジスタを持ち、EMC-Y からはゼロバスターンアラウンド (ZBT) 型の同期 SRAM とみることができる。すなわち、クロックサイクルの 2 倍のサイクルに同期してアドレスを出力し、その次のサイクルでそのアドレスデータの読み出し、あるいはそのアドレスへの書き込みを行える。

前半フェーズは、主に EXU からの命令メモリの読み出しであるが、そのほか、MU からのメモリ参照 (待ち合わせメモリの読み出し、テンプレートトップの読み出し、命令メモリの読み出し) がある。命令メモリの読み出しが優先度が高く、その要求がないとき (すなわち実行中のスレッドが中断したとき)、MU からの要求を受け付ける。読み出すメモリのアドレスはクロックサイクルまでに出力され、そのデータはクロックの中間 (立ち下がりエッジ) で読み出される。

後半フェーズは、主に EXU からのデータメモリの読み出しおよび書き込みである。そのほか、IBU からのメモリ (入力パケットメモリバッファ、直接メモリアクセス、待ち合わせメモリ) 読み出しおよび書き込み、MU からの待ち合わせメモリ書き込みがある。MU からの優先順位が一番高く、次に EXU、IBU の順に要求を受け付ける。MU が一番高いのは、MU のアクセスがあるのは前半フェーズで待ち合わせメモリの読み出しを行ったときで、そのクロックサイクルで同じアドレスへの書き込みを行うためである。

EM-X のメモリマップは表 4.3 の通り。ただし、低優先度入力パケットバッファの先頭アドレスは、メンテナンスアドレス BASE0TOP の値を設定することにより変更可能である。システムプログラムの開始アドレスや、ユーザプログラムの開始アドレス、ユーザスタック, システムスタックのアドレス等は、ソフトウェア的な取り決めであり、メモリサイズに応じて変更が可能である。図 4.3 はメモリをフル実装した場合の典型的な利用例である。

ユーザオペランドセグメントは、現在実装しているシステムプログラムでは、メモリ上位から下位に向かって利用するようになっている。オペランドセグメントはフリーリストにより管理されている。起動時は、フリーリストの先頭を保持している特殊レジスタが 0 に初期化されており、そこに引数やテンプレートトップを書き込もうとすると、メモリアドレス例外による例外ハンドラが起動され、そこでオペランドセグメントの初期化が行われる。現在の実装では、一度アドレス例外起きると 16 セグメントが割り当てられるようになっている。このようなオペランドセグメントの動的割り当ては、ユーザプログラム側のヒープの動的メモリ割り当てと連携して、プログラムの特性に応じたメモリ割り当てを可能にしている。

EMC-Y では非常に単純なメモリアドレス保護を可能にしている。プロセッサ内の特殊フラグ IMAERON を 1 にセットすると、以下のようなときに不正メモリアドレス例外が発生し、対応する例外ハンドラが

表 4.3: メモリマップ

メモリアドレス			内容
0x000000	-	0x007fff	高優先度入力パケットバッファ
0x008000	-	0x00bfff	特殊パケットハンドラ
0x00c000	-	0x00ffff	例外ハンドラ
0x010000	-	0x01ffff	システムプログラム
0x020000	-		ユーザプログラム
	-	0x6fffff	ユーザオペランドセグメント
0x070000	-	0x077fff	システムオペランドセグメント
0x780000	-	0x07ffff	低優先度入力パケットバッファ

起動する。例外ハンドラ内では、すべてのメモリアドレスに対して読み書きができる。

- プロセッサ内特殊レジスタ MEMPRO に設定されたアドレスよりも下位のアドレスへの書き込み
- 高優先度入力パケットバッファ領域の読み出し
- 低優先度入力パケットバッファ領域への読み出し/書き込み

同様に、特殊フラグ PRTEON を 1 にすると、メモリのパリティをチェックし、パリティエラーを検出すると、パリティエラー例外が発生し、例外ハンドラが起動する。

4.3.7 MAINT(メンテナンスユニット)

メンテナンスユニットは、メンテナンスバスを通じて送られてくるホスト計算機からのメンテナンス要求に答えるとともに、メンテナンス命令を実行するユニットである。EMC-Y の全てのフリップフロップ並びにメモリポートは、32 ビット毎にグループ化されて、メンテナンスアドレスと呼ばれる 7 ビットのアドレスが振られており、その値を読み出すことが可能である。また、そのうちの一部は書き換えも可能である。メンテナンスアドレスの一覧を付録 D に示す。

このメンテナンスアドレスの読み出しは、命令実行とは全く独立に行うことができるので、プログラムを実行しながら内部の状況をホスト計算機から読み出すことが可能である。EM-X 専用ライブラリで述べた実行トレースライブラリはこのメンテナンス機能を利用している。

ホスト計算機からメンテナンス機能を利用するには、メンテナンスバスからメンテナンス要求を送ることになる。EMC-Y のメンテナンスバス用の入出力信号としては以下の 10 本の信号がある。

MCLK メンテナンス用クロック。システムクロックの 1/2 のクロックを用いる。

MSTBN メンテナンスシーケンスの開始を指定。ローアクティブな信号である。

MDI 4 ビットのメンテナンス入力データ。

MDO 4 ビットのメンテナンス出力データ。

図 4.13 にメンテナンスバスのタイミングチャートを示す。ここで ma がアクセスしようとしているメンテナンスアドレス。ma[7] は読み出しの場合 0、書き込みの場合 1 とする。また、mtrd はメンテナンスから読み出される 32 ビットのデータ、mtwd はメンテナンスに書き込もうとする 32 ビットのデータである。また、メンテナンスユニットの初期化はメンテナンスクロック投入後、12 クロックサイクル以上 MSTBN をハイに設定することにより行われる。

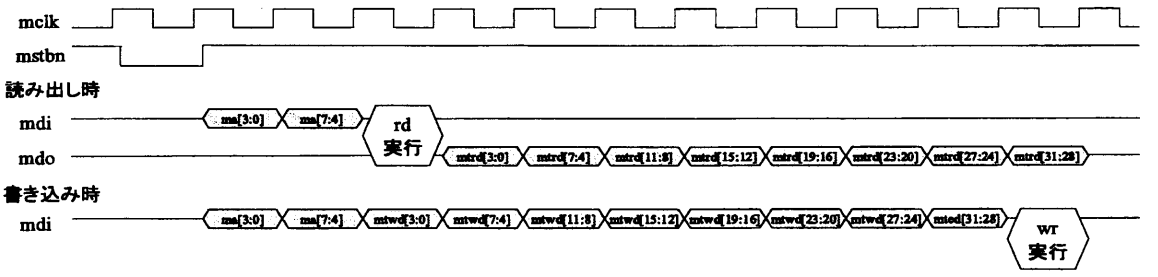

図 4.13: メンテナンスバスのタイミングチャート

EMC-Y 自体のリセットは、メンテナンスユニットから、EMC-Y 内部のリセットアドレスをオン/オフすることにより行われる。また、そのときに以下のような初期化を行う必要がある。

PE 番号 0x47 FP[31:20] に PE 番号を設定する。

ルーティング 0x1f IORT、HSTRT に適切な値を設定する。インタフェースボード、フレームバッファボード上の EMC-Y 以外は通常 IORT は 0。HSTRT は全てのプロセッサからのホスト計算機宛のパケットがきちんとインタフェースボードに到着するように設定する必要がある。

メモリプロテクト メモリプロテクトを有効にするには、まず全メモリをクリアしてパリティなどを正しく設定してから、0x63 MEMPRO に適切な値を設定し、0x62 PREON、MAEON を 1 に設定する。